Top: An Infrastructure for detecting Application-Specific Program Errors by Binary Runtime Instrumentation

by

Prasad Gopal

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Godmar Back, Chair

Stephen Edwards

Naren Ramakrishnan

June 15, 2006

Blacksburg, Virginia

Keywords: Program analysis, debugging, application-specific errors

Top: An Infrastructure for detecting Application-Specific Program Errors by Binary Runtime Instrumentation

Prasad Gopal

(Abstract)

Finding errors in applications has been achieved using a wide variety of techniques. Some tools instrument the application to check for program properties dynamically whereas others analyze the program statically.

We use a technique that analyzes a program's execution by binary runtime instrumentation. Unlike tools that work on a particular language or an intermediate representation of a language, our approach works directly on binaries and hence it is not bound to any language.

In order to instrument binaries, we use a binary instrumentation system called Pin, which provides APIs to instrument the application at runtime. We have built an infrastructure using Pin called Top that allows program entities like variables and events to be traced. Using finite automata we can check if certain events take place during the execution of the program.

Top consists of a *Tracing System* that can trace movement of pointers to memory locations or 32-bit data values and keeps track of all their copies. It also provides an *Event Framework* that reports the occurrence of events such as function calls or returns. Top provides a programming interface which allows querying for particular events. The query is compiled with Top to produce a customized analysis tool, also called *client*. Running the analysis tool with the application, under Pin, results in events of interest being detected and reported.

Using Top, we built a *Memory Checker* that checks for incorrect usage of dynamic memory allocation APIs and semantically incorrect accesses to dynamically allocated memory. Since

we perform fine-grained checking by tracing references, our memory checker found some errors that a popular memory checker called valgrind did not. We have also built an *MPI Checker* which is used to check if programs use MPI's asynchronous communication primitives properly. This checker can detect errors related to illegal data buffer accesses and errors where the programmer inadvertently overwrote a handle needed to finish the processing of a request.

Acknowledgments

I am extremely thankful to my advisor Dr. Godmar Back for all the guidance and support. The thirteen months I have spent working with him have been insightful and memorable. I want to thank him for the time he has taken out of his busy schedule to help complete this work.

I also want to thank my committee members Dr. Stephen Edwards and Dr. Naren Ramakrishnan for giving valuable comments on my thesis.

I want to thank my current roommate Poorna Chandra for proof reading my thesis several times. I want to acknowledge all the help extended to me by my roommates Parag Kshirsagar, Vivek Venugopal, Suraj Menon and Vishnu Vimjam. I also want to thank my sister Sunitha Gopal for all the comments and suggestions on my thesis.

Finally, I want to thank my parents for all the love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Detecting and correcting errors in programs is a never-ending task. A common reason for errors lies in the improper usage of library APIs, which are often not straightforward and easy to use. For instance, the MPI[For94] library defines asynchronous communication primitives that have restrictions on the request handle and the data buffer usage. These APIs require a check for completion of the operation before the handle/buffer can be reused. Consider the MPI asynchronous send primitive, MPI_Isend which is used to send data of a specified size. MPI_Isend returns a handle for every send request. The rules associated with its usage are:

- The request handle must be used to check the completion of the operation using either MPI_Wait or MPI_Test.
- The data buffer must not be overwritten before the operation is complete.
- The request handle must not be overwritten before the completion of the operation.

Such restrictions make it difficult to write correct programs as the restrictions have to be taken care of explicitly by the programmer.

Even if the APIs are straightforward, reasoning about their correct use may be difficult in the presence of pointer aliasing[1]. For example, consider the rules guiding explicit memory management in a language such as C. The rules are simple and straightforward:

- Memory must be obtained by calling a memory allocation routine (e.g. malloc in C).
- Memory thus obtained must be accessed within its bounds.
- Memory must be released by calling a memory deallocation routine (e.g. free in C).
- At least one pointer to the memory block must be present so that the memory can be freed.
- Memory must be deallocated only once.

Despite their apparent simplicity, programmers often violate these rules in the presence of aliasing. For example, let us assume that there are two pointers pointing to the same memory location. If this memory location is freed using one of the pointers, the other pointer points to an invalid memory location. Thus, presence of pointer aliasing makes explicit memory management hard.

Hence we need tools that help in finding such errors. Several tools have been developed for detecting such errors. Most of these tools are written to detect a particular kind of error, as each such tool is generally written with a specific library in mind. Every new library built will require a new error checking tool to be written. For example, memcheck in valgrind is a tool that can detect errors related to dynamically allocated memory. Extending it to detect errors in other libraries is not a trivial task.

[1]Pointer aliasing is present when multiple pointers point to the same address, and the address represents an object in memory.

Consequently, we need a generic infrastructure using which we can build plugins that can detect errors in usage of any library APIs. Library writers could then write specifications in the form of plugins that check the proper use of APIs. The programs that use these APIs could be checked with the specifications so that improper usage of the APIs could be detected.

1.2 Background

There are different approaches to detecting errors. One approach is to analyze the program source statically during the preprocessing, compiling or linking stages. Another approach is to analyze the program at runtime. Based on when the code is analyzed, program analysis techniques can be classified into the following categories (from Nethercote's PhD thesis [Net04])

- *Static Analysis* examines program code (source, intermediate representation or binaries) without executing it [HCXE02, HL03]. Tools that use this technique construct an approximate model of the state of the program [Ern03] and then analyze it. This approximation is necessary because it is impossible to model a program in its entirety if the program has a large number of run-time states. So most tools work on an abstracted representation of the program. Compilers are good examples of static analysis tools; they statically analyze programs to check if the program is syntactically correct; they also analyze programs to find opportunities for code optimizations. Static analysis is mostly sound and conservative [Ern03]. It is sound because the result of the analysis holds true irrespective of the program's input. Fully sound analysis do not report false positives or negatives. But sound techniques do not scale well. Hence tools like xgcc [HCXE02] perform unsound analysis so that they can work with large programs. Static analysis is conservative because it reports "*weaker properties*" (from

Ernst [Ern03]) than what may be actually in the original program. For example, consider a function $f(x) = 2^x$; a "weaker property" [Ern03] for $f(x)$ is that it returns an even number, though $f(x)$ returns only powers of 2. Also, since this approach analyzes the program statically, it is not possible to check for errors that depend on the run-time behavior of the program.

- *Dynamic Analysis* analyzes the program while it is executing. It is "*precise*" [Ern03] as the program's run-time execution state is analyzed and not an abstracted program representation. Profilers are examples of dynamic analysis tools. The drawback of dynamic analysis is that the analysis depends on the code paths executed in the program. Hence the results obtained over one run of the program need not be true over subsequent runs. This property makes dynamic analysis techniques *incomplete*. In order to cover all the execution paths of the program, an additional step of generating test cases is involved. The test cases generated should cover all code paths, and all boundary conditions may need to be tested for as well. This step may require tools for code coverage and generating boundary conditions.

Program analysis can be also classified based on what form of the program is analyzed (from Nethercote's PhD thesis [Net04]).

- *Source Analysis* involves examining the source code of an application. The analysis is not dependent on the machine architecture. Since source analysis depends on the language of the program source, an analysis technique that works with one language may not work with other languages. Source analysis can be performed with or without executing the application. Static source based analysis examines source code without executing it. A compiler is a good example for static source based analysis. Dynamic source based analysis involves instrumenting the program source, building and executing it to check properties at runtime. For example, SLIC [BR01] instruments the

source and checks for the behavior of the program at runtime using finite automata. This approach can perform analysis based on high level type information present in the source. The disadvantage of dynamic source analysis is that we need the source to instrument as well as recompile the source every time we instrument.

- *Binary Analysis* analyzes the object code or the executable of an application. This analysis approach does not have access to the high level type information that is present in the source. Binary analysis is dependent on the architecture for which the binary was built and hence the techniques have to be reimplemented for different architectures. It operates directly on application binaries and does not depend on any language. This approach works with all languages that compile programs into a binary for a given Application Binary Interface (ABI). An ABI describes the low-level interface between an application program and the operating system, between an application and its libraries, or between component parts of the application. A binary analysis tool is written for a specific ABI, so only programs compiled as per that specific ABI can be analyzed by the tool. The biggest advantage of binary analysis is that analysis can be done on the binaries without recompiling them.

From the above discussion we can conclude that dynamic analysis of binaries has unique advantages over other approaches. Dynamic analysis is precise, scales well and it does not suffer from the "over-conservativeness" of static checking. Binary analysis works with binaries compiled from programs in different languages. Also, binary analysis does not need recompilation of applications to run new checkers.

Existing dynamic binary analysis tools are hard to generalize because they are built to solve specific problems. For example, although tools like valgrind analyze binaries dynamically, they are implemented for checking specific APIs, such as the APIs used for explicit memory management. It is difficult to make such tools to work for any other library. Moreover, these

tools do not allow a user to specify error detection strategies in terms of patterns of function calls and data accesses. The dynamic analysis tools PQL[MLL05] and PTQL[GOA05] support the programmatic specification of patterns, but these tools are specific to Java bytecodes. Consequently, the problem of specifying error patterns and detecting errors by instrumenting the binaries at runtime remains unsolved.

This thesis solves the following problem -
"How can errors be detected in application binaries at runtime using error patterns which are specified by a library developer or an application programmer, in a manner that allows new error patterns to be easily added?"

To answer this question, we built an infrastructure, called Top, that can detect and report program events, trace data movement and accesses with the help of a binary instrumentation system. Top clients include the specifications for errors that they detect. These clients specify the error patterns to Top in terms of program events and Top reports the occurrence of these events to the client. The clients then check if the reported sequence of events results in an error and flags them accordingly. Top can run multiple clients simultaneously - unlike valgrind, which can run only a single skin at a time.

1.3 Outline

Chapter 2 discusses how Top fits into the binary instrumentation system called Pin. Chapters 3 and 4 discuss the important features Top provides to simplify the construction of error checking tools.. Chapters 5 and 6 talk about two error detecting clients that were built using the infrastructure. Chapter 7 evaluates Top's performance.

Chapter 2

Architecture

Top [1] is an infrastructure that provides APIs for high level functions that are useful for error checkers. It is built using Pin's APIs. Pin is a system that can instrument binaries at runtime. Pin provides low level APIs to instrument an application and analyze it at the instruction, basic block, routine, trace[2] or image level. An example of a low level function Pin provides is `INS_IsMemoryWrite(INS instruction)`, which returns true if an instruction writes to memory and false otherwise. On the other hand, error checkers are interested in high level events such as function calls or data modifications. If the error checker were to be using the Pin API directly to monitor writes to a particular memory address, then it would have to inspect every instruction to see if the instruction writes to the memory address.

[1] The name Top was derived as we started *tracing over Pin*. Since it is built on *top of Pin*, the recursive acronym was another reason to keep the name.

[2]

> *"A trace is a straight-line sequence of instructions which terminates at one of the conditions: (i) an unconditional control transfer (branch, call, or return), (ii) a pre-defined number of conditional control transfers, or (iii) a pre-defined number of instructions have been fetched."* [LCM+05]

Hence there is a need to bridge the gap from what the error checkers need to what Pin provides, so that error checkers can be easily implemented.
Several error checkers may need to detect the same or similar high-level events. Top facilitates such reuse. Top's APIs have features to query for events in the program. It also has support for tracing data in the program.

2.1 Top

Top's architecture is shown in figure 2.1. It consists of two components that are sandwiched between the error checkers and Pin. The event framework is responsible for registering events of interest to error checkers (also called clients) and delivering those events when they occur. The tracing system helps in tracing pointers and data values. The figure also shows the interaction between Top, Pin and the clients. An *Error Checking Client*, or client in short, is an entity that implements a specific bug detection logic or other analysis functions.

2.2 Pin

Pin [LCM+05] is an easy-to-use and efficient run-time binary instrumentation system for applications. It supports Linux binary executables for Intel IA-32, IA-32E and Itanium processors. It also supports programs in the Cygwin and MacOSX for Intel environments. Pin's software architecture (figure 2.2 from [LCM+05]) shows three programs running in the same address space, i.e., Pin, pintool and the application. Even though Pin, pintool and the application share the same address space, they do not share any libraries between them. Pintools in figure 2.2 corresponds to Top and the error checking clients in figure 2.1.

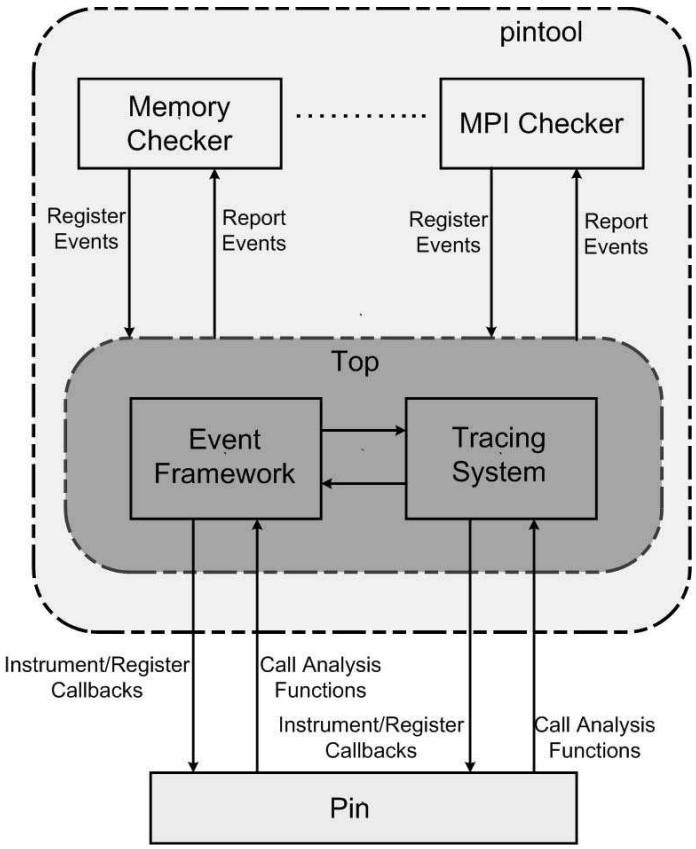

Figure 2.1: Top Architecture

Pintools are instrumentation programs written using Pin's API in C or C++. They can perform detailed analysis of the application being run at instruction, trace, image or routine level using the API. As the pintool runs in the same address space as the application, it has access to the application's data. It can read or write stack, heap or global data of the application.

Pin's APIs are architecture independent. Pintools written using Pin's API can be run on several different architectures. In addition, Pin provides APIs to access architecture specific information. It instruments the application such that its behavior does not change when compared to the uninstrumented version, allowing the pintool to analyze the original application's behavior. Hence the instrumentation is transparent. Pin's API provides access to the

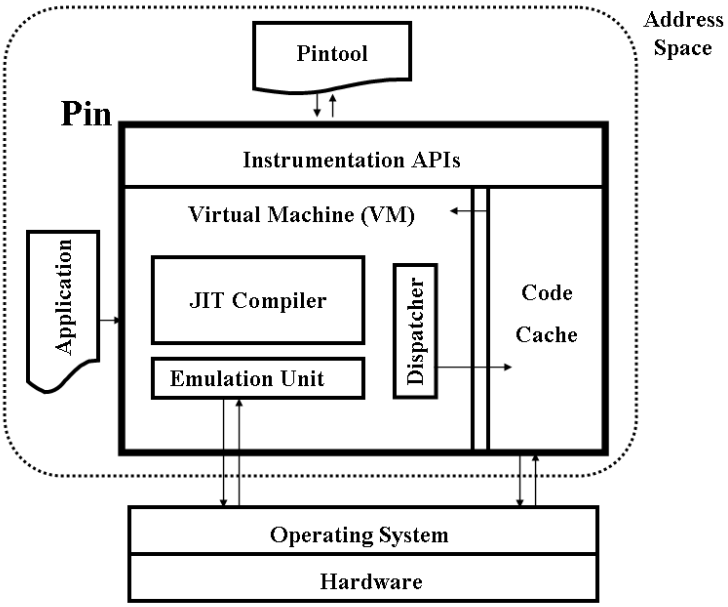

Figure 2.2: Pin's software architecture (from [LCM+05])

complete architectural state of the process including its register state and memory contents. Pin uses techniques such as register allocation inlining, liveness analysis and instruction scheduling to just-in-time (JIT) compile and optimize code. Hence its instrumentation is efficient.

Pin provides APIs to analyze an application at run-time at instruction, trace, image or routine level. The rules that defines how an application should be instrumented are written as instrumentation routines. The instrumentation routines determine points of interest in the application where (typically) small pieces of code need to be inserted. These small pieces of code are called analysis routines and they can be run before or after the points of interest in the application.

Pin's instruction instrumentation APIs allow analysis of the application at the instruction level. Analysis routines can be inserted either before or after executing the instruction. Pin

has support to provide the analysis routine with information. For example, the analysis routine can be passed information about the types (memory/register), number, and size of operands.

Chapter 3

Event Framework

Events are actions that take the program from one state to another. The change of state in the program involves the modification of program data. The event framework allows clients to register for events of interest. A callback mechanism is provided for clients to be notified when these events occur.

Suppose we represent a function event and its context as a 3-tuple

$$\Psi(\eta, \tau, \rho_n)$$

where η is the name of the function event,
τ is the type of the function event and
ρ_n is the set of all the parameter (value, position) pairs, also referred to as the context of the function event.

τ is defined as

$$\tau \in \{\tau_{call}, \tau_{ret}\}$$

where τ_{call} represents function call and τ_{ret} represents function return, and

```
1. int main ()
2. {                       // client registers for malloc return event.
3.   char *cptr = malloc (sizeof(char) * 10);
                           // when malloc returns, client registers for
                           // free with start address of allocated memory.
4.   cptr[0] = 'c';
5.   free (cptr);          // when free is called with a start address
                           // of allocated memory, report back to the client.
6. }
```

Figure 3.1: Program *E1* Source

ρ_n is defined as

$$\rho_n = (\upsilon_1, \upsilon_2, ..., \upsilon_n)$$

where υ_i is the parameter value of parameter i. A parameter which is not of interest is represented using a wildcard character $*$.

Alternatively, we can represent $\Psi(\eta, \tau_{call}, \rho_n)$ as $call(\eta, \rho_n)$ and $\Psi(\eta, \tau_{ret}, \rho_n)$ as $ret(\eta, \rho_n)$ in a concise form.

For Program *E1*, a memory checker client, say 'A', wants to intercept all `malloc` function returns. Client 'A' would register for an event with signature $ret(\text{“}malloc\text{”}, (*))$ which does not restrict the value of malloc's first parameter. Let us assume that the return value of `malloc`, which is the address of the allocated memory block, is 0xbf4e8000. If client 'A' is interested when the block starting at 0xbf4e8000 is freed, it requests for the event $call(\text{“}free\text{”}, (0xbf4e8000))$. Meanwhile, let us also assume that there is another client, 'B', who registered for $call(\text{“}free\text{”}, (*))$. The call to `free` on Line 5 of Program *E1* will result in both 'A' and 'B' being notified of this event.

```
1. #define SOME_TAG 10
2. int main ()
3. {
4.   // Initialize ...
5.   char buf[100];
6.   MPI_Request request;
7.   MPI_Status status;
8.   //...
9.   MPI_Isend (buf, 10, MPI_CHAR, 1, SOME_TAG, MPI_COMM_WORLD, &request);
10.  // client requests for MPI_Isend return event.
11.  MPI_Wait (&request, &status);
12. }
```

Figure 3.2: Program *E2* Source

The event framework permits clients to specify events and any combination of the parameters as the context of the event. For Program *E2*, a client, 'C', is written to check for proper usage of MPI asynchronous communication APIs as discussed in Chapter 1. Client 'C' could register for $ret(\text{“}MPI\_Isend\text{”}, (*, *, *, *, *, *, *))$. When the event takes place on Line 9, 'C' could then register for another event $call(\text{“}MPI\_Wait\text{”}, \rho_{mpiwait\_request})$ where $\rho_{mpiwait\_request}$ is $(request, *)$. When an event with $\eta = \text{”}MPI\_Wait\text{”}$ occurs with the context $\rho_{mpiwait\_actual} = (request, status)$, client 'C' is notified of this event as $\rho_{mpiwait\_actual} \subseteq \rho_{mpiwait\_request}$.

The event framework also supports reporting of memory accesses as events. This feature requires determining every memory address generated in an instruction and checking if the address belongs to any block that is being traced. Clients interested in learning about accesses to a memory address should register for memory access events for a block of memory. The block can be a single memory location or a contiguous set of memory locations. If any address within the block is accessed, the event framework reports a memory access event.

The event framework also supports two other event types with the help of the Tracing System. The dereference event is similar to the memory access event, but with the additional constraint that the memory is accessed through a pointer or a reference that is being traced. It also supports lost reference events which is described in Chapter 4.

3.1 Event Types

The events currently supported by the event framework are function call/return, memory access, dereference and lost reference. Although lost reference and dereference events are triggered by the tracing system, the event framework is responsible for routing these events to the clients who have registered for them. The event framework maintains a request handle from the tracing system, and the reference to the client who requested notification for the event.

Function Call/Return Event - Function calls and returns are an important class of events an error detecting client may be interested in. If a function event is recursive, clients may be interested in only the top level events of such functions or they may be interested in all occurrences of those events. The event framework supports both modes of notification.

Dereference Event - This event is generated when a pointer is dereferenced to read or write memory. It is generated by the tracing system as it involves tracing references to memory. The client specifies the memory block and the reference that needs to be traced. The tracing system traces the movement of the references and creation of aliases to references. It will report when a reference or its alias is dereferenced to access memory.

Memory Access Event - Reading or writing to memory can be another set of events that error-detecting clients may be interested in. This event is different from the Dereference Event as the pointer used to access memory is not taken into consideration. Instead, the memory address that is being read from or written to is used to determine if a Memory Access Event needs to be reported.

Lost Reference Event - Clients can also register for lost reference events. If there are no references to a memory block that was previously requested by the client for tracing, then a Lost Reference event is generated. For example, to detect a memory leak, we must know if there is at least one reference to the dynamically allocated memory block at any point of time.

3.2 Implementation

The event framework encapsulates all data structures and algorithms required for event registration, event lookup and dispatch.

Instrumentation The event framework instruments all functions in the application code as part of the initialization. This instrumentation includes the functions that are present in the shared libraries that the application may use. The event framework uses the `RTN_xxx` APIs of Pin to instrument the application and the shared libraries. When a function call or a return event takes place, the event framework checks to see if any client has registered for this event. If yes, it dispatches the event to the corresponding client.

When clients request to register function events before the application libraries are loaded, the event framework cannot register these requests immediately as the function address is not known. This event request is stored and processed later when the library containing

this function gets loaded. The *Deferred Events Table* stores all the events that need to be handled later. This table stores all the information required to register the event when the image that contains the function is loaded. When an image gets loaded, the image is checked for definitions of functions the client had requested. If they are present in the image, they are registered.

It is also possible, during the course of execution of the program, to replace the definition of a function. For example, when we execute a program that is linked with the GNU C library, there are two different `malloc` functions that execute. One of them is defined in GNU linker (ld-linux.so.2) and is used by the C start up routines to load shared libraries. The other definition, present in the C library (libc.so.6), is used by the application. To handle such cases, all events are added to the deferred events table. When a new image is about to be loaded, it is checked for definitions of all the function events. If a function that matches the event is defined in the image being loaded, the event is registered.

Function Call/Return Event - The event framework implements the function event registration and lookup using the data structures shown in figure 3.3. The data structures consists of three components, i.e., Function Address Map, Parameter Index Map and Multi-level Parameter Value Map.

- **The Function Address Map** is a map where the key is a unique identifier for a function and the value is a reference to a Parameter Index Map that stores details of the parameters a client registered for.

  The function start address is used as the key because the GNU C Compiler allows the declaration of aliases for functions through attribute annotations. The Pin function `RTN_Address (RTN)` gives the address of the required `RTN` function object. A function object is obtained using `RTN_FindByName(img, name)` where *name* is the function name and *img* is the image object containing the function.

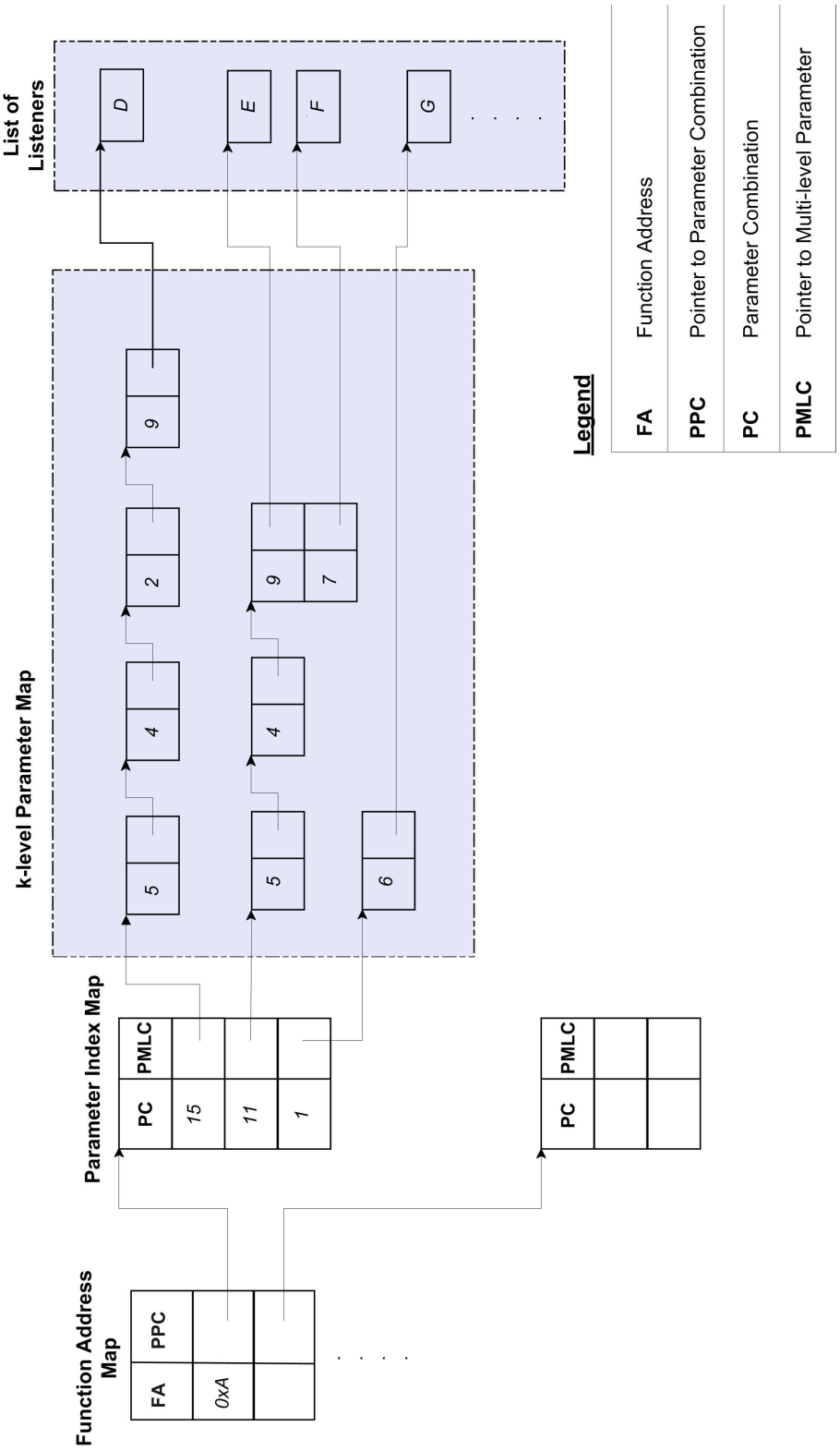

Figure 3.3: Event Framework Data Structures

- **The Parameter Index (PI) Map** has an entry for each parameter combination requested by the client. The key to the map is a function of the parameter positions present in a combination. The value is a reference to a multi-level parameter map which contains the parameter values of interest to clients. The following function is used to compute the key to the PI map: $\sum_{i=1}^{n} 2^{p_i}$. This function maps each parameter combination into a unique value.

- **The Multi-level Parameter Value Map** is used to store the parameter values of interest to clients. For example, let a client be interested in some event call("func", (5, 4, 2, 9)). Figure 3.3 shows the four levels of the parameter value map when this event is registered by the event framework. Each level of the parameter value map is indexed by the parameter value at position 'i' in the request. If any parameter is a don't-care (indicated by a '*'), it is ignored. The number of levels in the multi level map can range from zero to the number of parameters in the function prototype. The PI map key indicates the parameter combination present in the multi-level map structure. It is used to determine the position of the parameter in the function call.

  The state of the data structures for another event request, call("func", (5, 4, *, 9)) is shown in figure 3.3. Note that the parameter at position three in the request is not of interest to the client and hence does not appear in the multi-level map (shown in the figure). We see that an entirely new branch of maps is created as this parameter combination is different from the previous one. Similarly, the multi-level maps for event requests call("func", (5, 4, *, 7)) and call("func", (6, *, *, *)) are shown in the figure.

Run Time Event Stack At times clients may want to express their interest in a function call and its corresponding return event. To enable this, there is a provision for the clients, while registering for a function call, to specify their interest in the function's return as well.

This is also useful in cases where clients want to observe the return of a 'void' returning function. Specifying a 'void'returning function using the function return registration API would result in all the return events of that function to be reported, even though the client may be interested in a specific return event corresponding to a specific function call. This feature saves the client from making more registration calls and some unnecessary code to check if the return event is of interest. To support this feature, the event framework maintains a *Run Time Event Stack* (RTES) of events that take place in the application. When the function call occurs, the framework checks to see if any of the clients also expressed interest in the corresponding return. All such clients are appended to a list of *default return listeners* and a reference to them is maintained on the RTES. When any function returns before the event is popped off the RTES, the *default return listeners* (if any) are triggered.
Events can be recursive or non-recursive. The framework allows clients to request for recursive or non-recursive occurrence of events. This functionality is also implemented using the RTES. The RTES maintains a count of how often every function is contained in the stack. If a function is called more than once, its count will be greater than one. When an event occurs, the RTES determines if the event is recursive or not and dispatches it accordingly.

Memory Read/Write Event Clients can register for events that track reads and writes to memory locations. Clients need to specify the memory address(es) they are interested in. A single level map whose key is the address of a memory location is used to maintain the event registration information. Since we need to check for reads and writes, before and after they have occurred, we use four maps to register for events *Before Read*, *After Read*, *Before Write*, *After Write*. This feature is implemented by instrumenting all memory reads and writes. When a memory is accessed at a particular address, the event framework checks to see if any client had requested notification for a memory access event at that address. If so, the event is dispatched to the client.

Lost Data Reference Event Event Clients may be interested in knowing whether there are references to data at any point of time during the execution of the application. This may be because the client logic is interested in knowing if certain events take place when there are references to certain data or not. For example, a memory leak detection tool would be interested in knowing if references to a particular memory block exist. The event framework provides this feature in conjunction with the tracing system, which is described in Chapter 4.

Complexity of Event Framework Operations

Event Registration for a function event involves walking a k-level map structure, where k is the number of parameters of the event. This operation involves adding an entry in the map if the parameter does not already exist. This operation can also involve creating new maps to create the k-level map of parameters.

Event Lookup and Dispatch When an event occurs, the event framework is notified of the occurrence of the event by Pin. The event framework then does a lookup in the corresponding data structures and triggers the listeners that had registered for the event. When a function event occurs, Top uses the function address as an index into the first level map. As explained in the beginning of this chapter, clients can narrow their interest in a function call event based on the value of any combination of parameters. Clients could register events with various combination of parameters. The event framework checks if a particular combination of parameters is of interest to any client and dispatches the event accordingly.

The time required to dispatch an event is different for function events and other events. Memory access and lost data references events take constant time. The time for dispatching

function events to a state machine depends on k, in the k-level map. The value k is known at compile time of the client and is different for each type of event. For the same function call/return, we could have maps of different depth depending on the number of parameters the client is interested in. In this fashion, the time taken to dispatch any event to one or more state machines is O(k), which is a constant, since k remains constant for any function event.

Some Interesting Cases

Handling Position Independent Code. Since the event framework instruments the application using the `RTN_InsertCall` API of Pin, it is necessary to understand how Pin instruments a function. When `RTN_InsertCall(rtn, IPOINT_BEFORE,...)` is used to instrument a routine `rtn`, Pin calls the analysis routine just before the start of the `rtn`.The `RTN_InsertCall(rtn, IPOINT_AFTER,...)` call runs through the instructions of `rtn` and instruments all `RET` instructions. However, code which is compiled as Position Independent Code (using -fPIC option), uses a technique where it makes a call to a nested subroutine and returns from there. The nested subroutine copies the return address present on the stack to a register and returns. This technique is used to copy the current instruction pointer into a register so that the program can access the contents of its data segment via this register, because the data segment is located at a known offset from the current instruction pointer. These call instructions are not seen by Pin's `RTN_InsertCall` APIs. But since `RTN_Insert(rtn, IPOINT_AFTER,...)` instruments all `RET`s, Pin will report a spurious function return event, even though it did not report the corresponding call. We handle this anomaly with the help of the RTES. When a call event takes place, it is pushed on the RTES. When the corresponding return event takes place, it is popped off the stack. When a return event takes place and the top of the stack does not have the function's corresponding call event, such a

return event is simply discarded and the anomaly is logged.

Chapter 4

Tracing System

Error checking clients may need to trace data for a variety of reasons. For instance, clients may be interested in knowing if there is at least one reference to a block of memory or if a block of memory is being read or written through some reference while the application is executing. Clients may be also interested in knowing if certain locations retain their original values, or are copied to other memory locations. The Tracing System reports such events of interest to the client.

The Tracing System is a component of the infrastructure that allows us to trace references to 32-bit values or a block of memory. Tracing a 32-bit value is called Value Tracing and tracing references to a block of memory is termed as Reference Tracing. Value Tracing involves keeping track of all memory locations or registers storing a particular 32-bit value. The 32-bit value was either requested to be traced by a client, in which case it is called the *seed*, or was copied from the seed directly or indirectly. It is not possible for the Tracing System to start tracing a TraceItem without a seed. The information about the seed has to be determined by the client who requests to trace a TraceItem. Reference Tracing involves knowing all references to a block of memory at any point during the execution of an application.

A *TraceItem* is defined as either a block of memory or a 32-bit value that is being traced by the Tracing system. Hereafter, any mention of the term *TraceItem* can be considered either as a block of memory or a 32-bit value, which will be evident from the context if not explicitly specified.

TraceItem that are blocks of memory can have references not only to the start of the block but to any data within the block. So a memory location or register containing a reference to a valid address range of a TraceItem is considered a valid reference.

The Tracing System instruments the application's instructions with analysis code with the help of Pin. The analysis code analyzes each instruction at runtime to see if the execution of the instruction modifies the state of any TraceItem/s. If the instruction destroys an existing TraceItem, the data structures to track the TraceItem are updated to reflect the change. If the execution of an instruction results in an event of interest being generated, such events are delivered to the client.

4.1 Value Tracing

Value tracing involves tracing the movement of 32-bit values and knowing all locations where copies of a particular 32-bit value are present. We trace the copying of the value from a single start location to various other memory locations and registers. Instructions that modify a location that holds a copy of a value that is currently traced will result in the removal of that location from the tracing set.

4.2 Reference Tracing

The different aspects of reference tracing are explained using a sample Program P , shown in figure 4.1. Lines 1 through 5 declare global and automatic variables. Lines 6 through 12 in figure 4.1 show the location sets maintained while tracing the references used in this program. Figure 4.2 shows the corresponding assembly code. Figure 4.3 shows the state of TraceItems after executing line 6 of Program P. In the figure, the *Memory Alias Set* is the set of all the memory locations that are references to TraceItems we are currently tracing and the *Register Alias Set* is the set of all registers that are references to TraceItems.

Let the bottom of the activation record for `main` be located on the stack at address 0xbfe2e788, which is also the value of the Base Pointer (EBP). Let the local variables `ptr, pptr, localptr` and the global `globalptr` be allocated on stack at addresses 0xbfe2e77c, 0xbfe2e780, 0xbfe2e784 and 0x80495a0. Suppose a client wants to trace all the references to the memory block returned by `malloc`, on line 6 in Program P.

A new TraceItem A is created for the address block (say, address range 0x821b008-0x821b030) returned by the `malloc` call. Since the return value of a function is passed in register EAX (as per IA32 Application Binary Interface [1]), EAX is the first reference to TraceItem A. In this case, EAX is the seed for TraceItem A. The MOV instruction shown on Line 4 in figure 4.2 moves the TraceItem reference in EAX to variable `ptr` at address 0xbfe2e77c, creating a second reference to TraceItem A.

Line 7 of Program P is another `malloc` call that returns a block of 40 bytes (say, address range 0x821b038-0x821b060), creating a new TraceItem B. This `malloc` call overwrites the EAX register with the start address of the newly allocated block. This start address is copied

[1] The IA32 Application Binary Interface defines the system interface for compiled application programs [ia3]. It is a standard that defines the binary interface for application programs so that object code compiled using different compilers for the same platform may be linked together on the Intel IA32 architecture.

```
1. int **globalptr;
2. int main ()
3. {
4.   int *ptr, **pptr;
5.   int **localptr;
6.   ptr = malloc (sizeof(int) * 10); // start tracing TraceItem A,
                                      // location set = {(A, ([ptr,0]))}
7.   pptr = malloc (sizeof(int*)*10); // start tracing TraceItem B,
                                      // location set = {(A, ([ptr,0])), (B, ([pptr,0]))}
8.   localptr = pptr;                 // create local alias for B
                                      // location set = {(A, ([ptr,0])),
                                                        (B, ([pptr,0], [localptr,0]))}
9.   globalptr = pptr;                // create global alias for B
                                      // location set = {(A, ([ptr,0])),
                                                        (B, ([pptr,0], [localptr,0], [globalptr,0]))}
10.  *localptr = ptr;                 // create heap alias for A
                                      // location set = {(A, ([ptr,0], [*localptr,0])),
                                                        (B, ([pptr,0], [localptr,0], [globalptr,0]))}
11.  pptr = pptr + 1;                 // pointer modification
                                      // location set = {(A, ([ptr,0], [*localptr,0])),
                                                        (B, ([pptr,1], [localptr,0], [globalptr,0]))}
12.  ptr = NULL;                      // destroy reference to A
                                      // location set = {(A, ([*localptr,0])),
                                                        (B, ([pptr,1], [localptr,0], [globalptr,0]))}
13.  (*localptr) = NULL;              // destroy reference to A, lost reference to A
                                      // location set = {(A, (empty),
                                                        (B, ([pptr,1], [localptr,0], [globalptr,0]))}
14.  return 0;
15.}
```

Figure 4.1: Program P Source

```
1.  push   $0x28
2.  call   80482a8 <malloc@plt>  % Line 6: call malloc
3.  add    $0x10,%esp
4.  mov    %eax,0xffffffff4(%ebp) % malloc returns address in EAX,
                                  % copy into variable 'ptr'
5.  sub    $0xc,%esp
6.  push   $0x28
7.  call   80482a8 <malloc@plt>  % Line 7:  call malloc
8.  add    $0x10,%esp
9.  mov    %eax,0xffffffff8(%ebp) % malloc returns address in EAX,
                                  % copy into variable 'pptr'
10. mov    0xffffffff8(%ebp),%eax % Line 8:  copy into variable 'localptr'
11. mov    %eax,0xffffffffc(%ebp)
12. mov    0xffffffff8(%ebp),%eax % Line 9:  copy into variable 'globalptr'
13. mov    %eax,0x80495a0
14. mov    0xffffffffc(%ebp),%edx % Line 10: copy 'ptr' into '*localptr'
15. mov    0xffffffff4(%ebp),%eax
16. mov    %eax,(%edx)
17. lea    0xffffffff8(%ebp),%eax % Line 11: add one to 'pptr'
18. addl   $0x4,(%eax)
19. movl   $0x0,0xffffffff4(%ebp) % Line 12: set 'ptr' to 0
20. mov    0xffffffffc(%ebp),%eax % Line 13: set '*localptr' to 0
21. movl   $0x0,(%eax)
```

Figure 4.2: Program P Assembly (Excerpt)

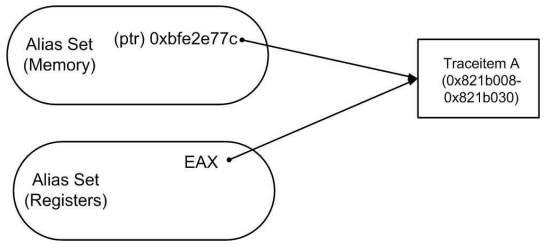

Figure 4.3: Reference Tracing. State after executing Line 6 of Program P

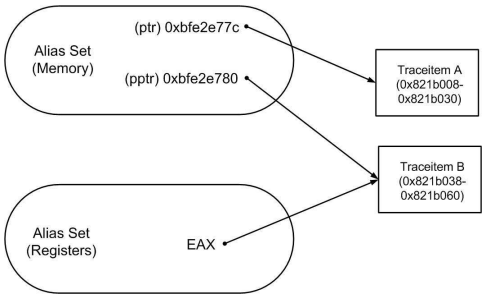

Figure 4.3: Reference Tracing - State after executing Line 7 of Program P

to variable `pptr` at address 0xbfe2e780, as shown in figure 4.3.

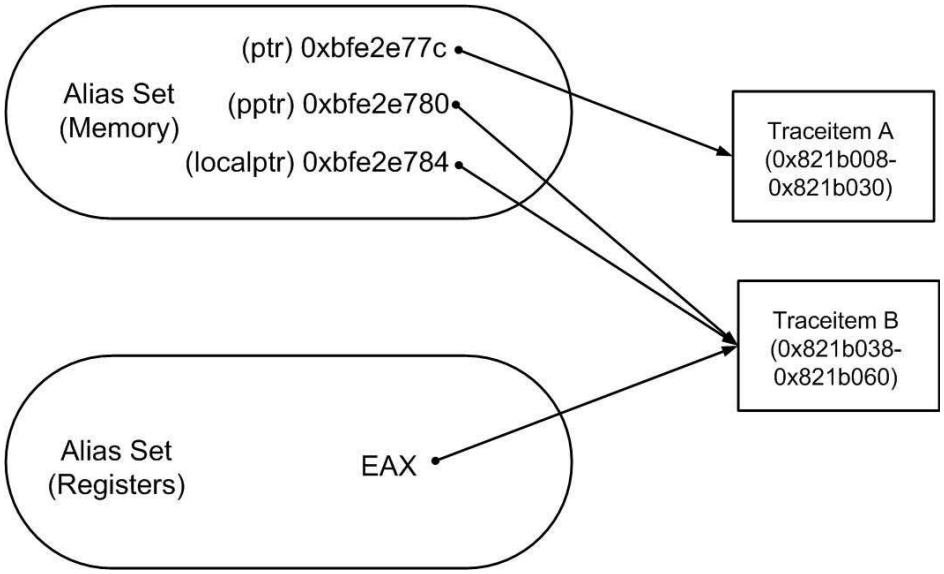

Figure 4.4: Reference Tracing - State after executing Line 8 of Program P

Figure 4.4 shows the state of the tracing system after executing line 8 of Program P. This statement creates a local alias to TraceItem B in variable `localptr` at address 0xbfe2e784.

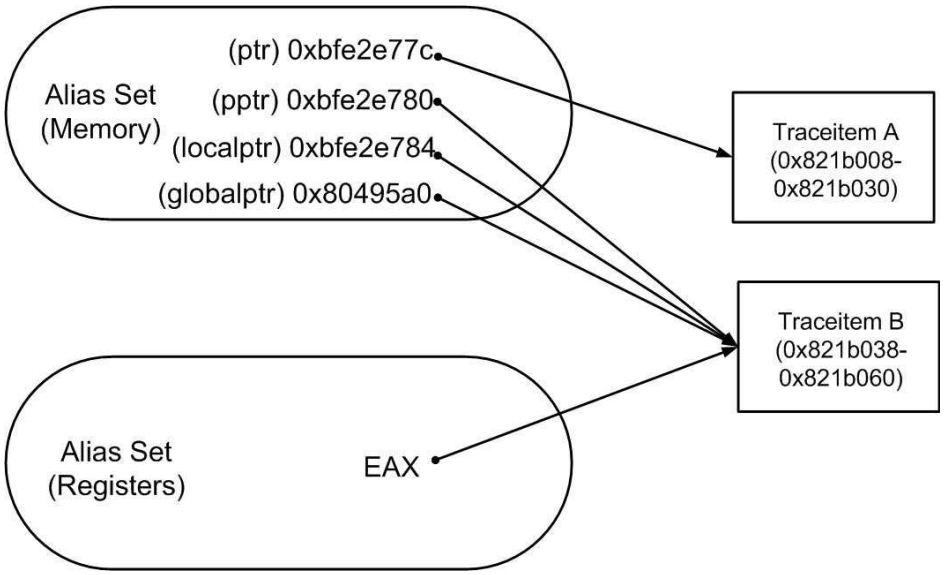

Figure 4.5: Reference Tracing - State after executing Line 9 of Program P

Figure 4.5 shows the state of the tracing system after executing line 9 of Program P. This statement creates a global alias to TraceItem B in variable `globalptr` at address 0x80495a0.

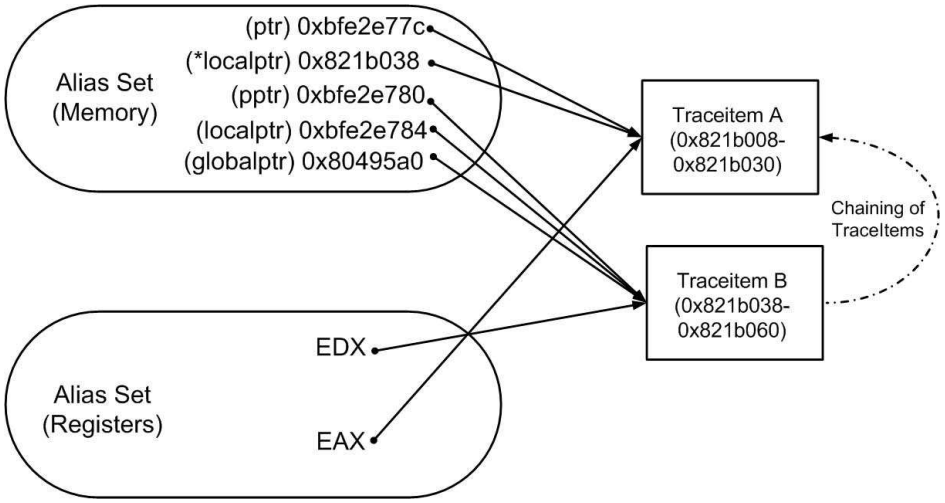

Figure 4.6: Reference Tracing - State after executing Line 10 of Program P

Figure 4.6 shows the state after executing line 10. This statement creates a new reference to TraceItem A. Since this reference is a memory location within TraceItem B, we introduce chaining within TraceItems, where one TraceItem holds references to other TraceItems.

The next statement on line 11, increments `pptr` by 4. It is clear that even after incrementing `pptr` it is a reference to TraceItem B. References to TraceItems may not always point to the beginning of a block - for instance, if the user program uses pointer arithmetic on an array. A reference to any address within a memory block is considered a valid reference because any address within the block can be obtained by adding or subtracting from that reference. Hence, every reference to a memory block also stores the offset from the start of the block. Figure 4.7 shows TraceItems and offsets for the state of the tracing system after executing line 10.

Every time a client asks the tracing system to trace a seed, a new TraceItem is created. Consequently, a particular memory location or register may reference multiple TraceItems.

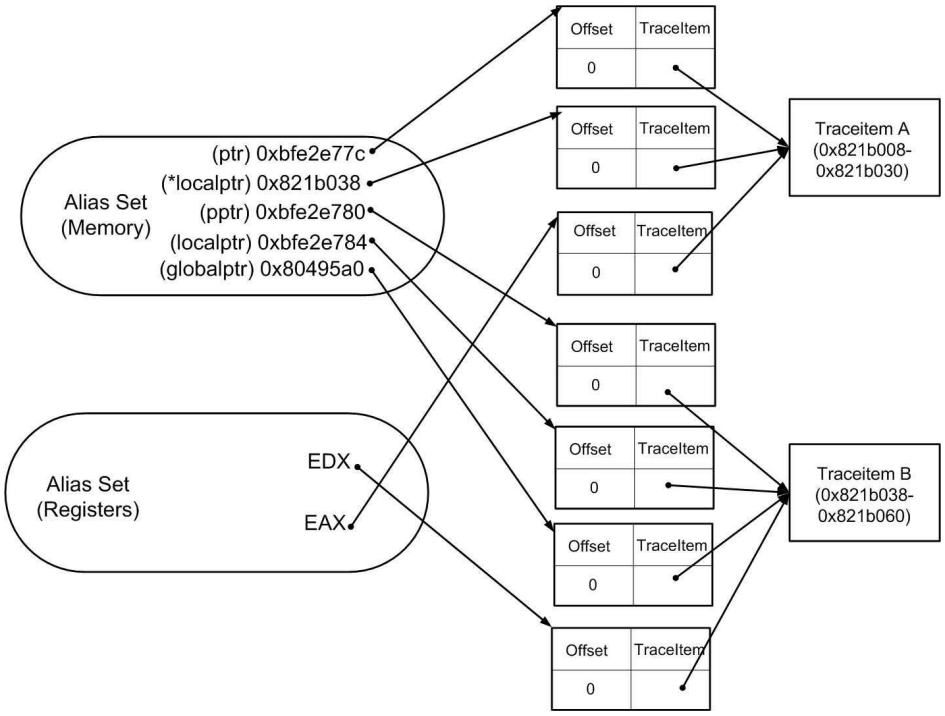

Figure 4.7: Reference Tracing (Modified) - State after executing Line 10 of Program P

If the TraceItems refer to blocks in memory, their addresses may be partially or fully overlapping. To handle this case, a reference needs to point to multiple TraceItems. The modified data structures that are thus needed include a list of (offset, pointer to TraceItem) pair for every memory or register location. Using the data structure, a reference can point to an overlapping TraceItem by adding a new entry in the (offset, pointer to TraceItem) list.

After executing line 11, the state is as shown in figure 4.8.

In line 12, the reference `ptr` to TraceItem A is killed. This state is shown in figure 4.9.

Overwriting `*localptr` results in both reference sets of TraceItem A becoming empty. The seed is no longer a reference to TraceItem A. Also all the references that were created as a result of moving the address from the seed to a memory location or a register, directly or indirectly, are no longer references to TraceItem A. Hence TraceItem A is lost in the context requested by the client and we report the occurrence of *Lost Reference Event* to the client.

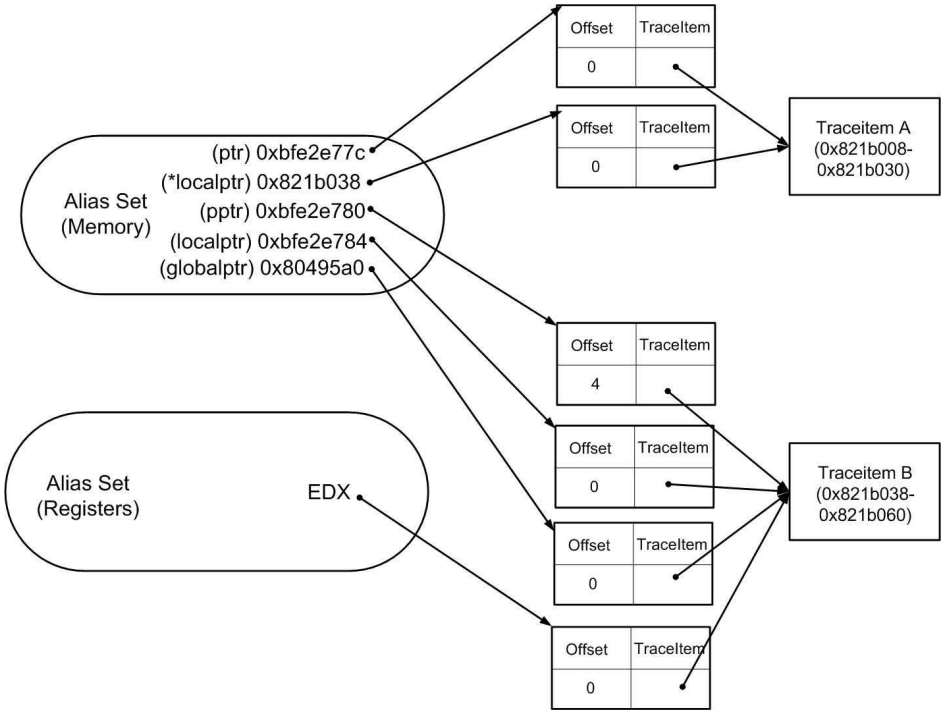

Figure 4.8: Reference Tracing (Modified) - State after executing Line 11 of Program P

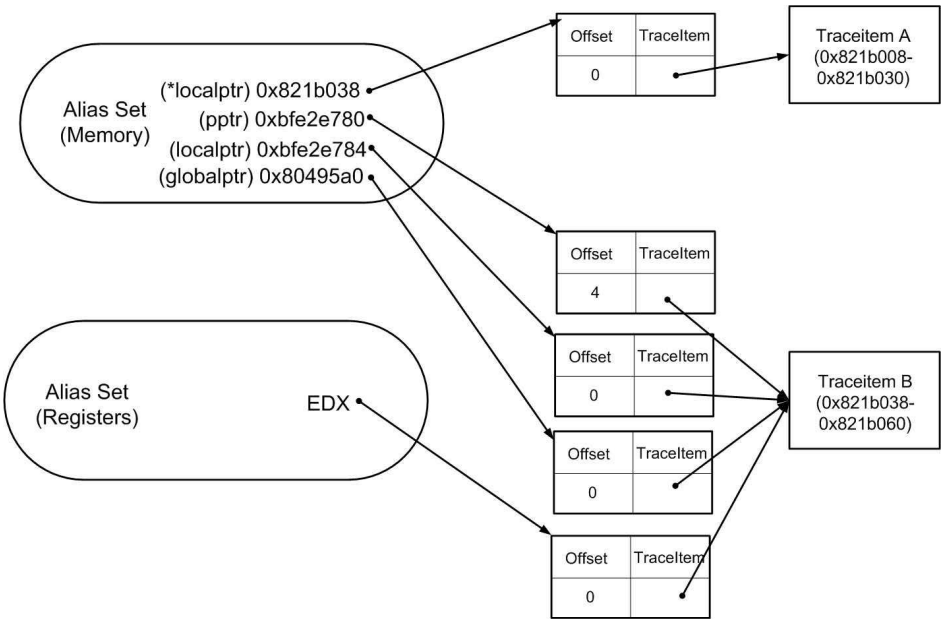

Figure 4.9: Reference Tracing (Modified) - State after executing Line 12 of Program P

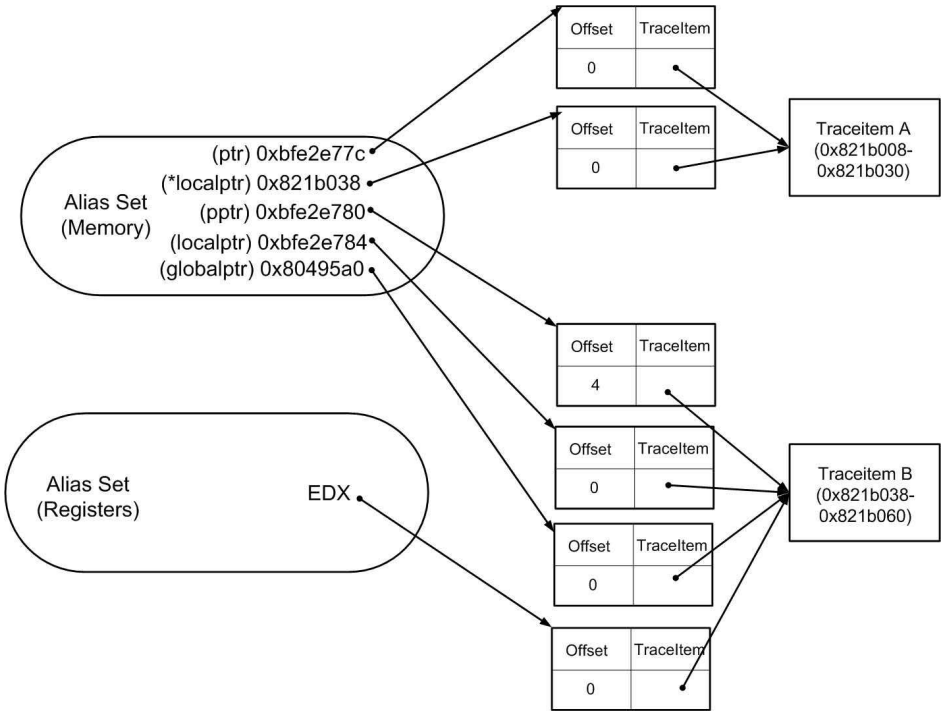

Figure 4.10: Reference Tracing (Modified) - State after executing Line 13 of Program P

If TraceItem A contained references to other TraceItems, it would be necessary to check if these other TraceItems lost all references as a result of losing TraceItem A. This process involves checking the *reachability* of all TraceItems transitively. Lost Reference events are generated for all such TraceItems that have lost all references.

4.3 Events Related to Tracing data

The Tracing System is responsible for reporting events that are related to tracing of data. There are five kinds of events associated with any TraceItem, namely Valid Read, Valid Write, Invalid Read, Invalid Write and Lost Reference.

- Valid Read Event - Reading from valid locations (in-bounds) of a TraceItem using one of the references that is currently being traced is termed as a *Valid Read*. A *Valid Read*

Event occurs when the application program being analyzed performs a Valid Read.

- Valid Write Event - Similarly, a *Valid Write Event* occurs when the application program being analyzed attempts to write to a valid address (in-bounds) in a TraceItem through a reference that is currently being traced.

- Invalid Read Event - An *Invalid Read Event* occurs when the application program being analyzed attempts to read from an invalid address (out-of-bounds) in a TraceItem through a reference that is currently being traced.

- Invalid Write Event - An *Invalid Write Event* occurs when the application program being analyzed attempts to write to an invalid address (out-of-bounds) in a TraceItem through a reference that is currently being traced.

- Lost Reference Event - A *Lost Reference Event* occurs when all references to a TraceItem are lost.

4.4 Implementation

This section describes the implementation details of the Tracing System. We discuss the important data structures, algorithms and techniques that are necessary to understand the working of this system.

TraceItem A TraceItem is represented as an instance of `class TraceItem`. Every TraceItem has the following members.

- `type` - There are two types of TraceItems currently supported, they are Reference TraceItem and Value TraceItem. Accordingly the value of this member is `VALUE_TYPE` and `ADDRESS_TYPE`.

- `item` - For a reference TraceItem, it is the start address of the memory block and for a value TraceItem it is the value at the 'seed location'.
- `size` - Size of the memory block for a reference TraceItem.
- `locations` - It is the set of all locations that hold valid references to this TraceItem.
- `registers` - It is the set of all registers that hold valid references to this TraceItem.
- `state` - The tracing system designates a state to all TraceItems. TraceItems are either `ACTIVE` or `LEAKED`. A TraceItem starts in the `ACTIVE` state. If all the references (locations and registers) are lost at any point during the execution of the program, they are moved to the `LEAKED` state. TraceItem cannot transition from a `LEAKED` state to an `ACTIVE` state.
- `strace` - As part of every TraceItem, we also have the stacktrace of the program state when the TraceItem was created. If the TraceItem transitions to the `LEAKED` state, then the stacktrace will contain the state when it leaked.

AddressTraceItemMap and RegisterTraceItemMap The AddressTraceItemMap is a map where the key is an address of a reference and the value is a pointer to a list of (Offset, TraceItem) pairs. The RegisterTraceItemMap is a map where the key is a register object and the value is a pointer to a list of (Offset, TraceItem) pairs as shown in figure 4.11.

Instrumentation Top instruments data movement and data modification instructions using Pin to facilitate the tracing of data. Pin provides architecture independent APIs to instrument programs. Since instructions are specific to a particular architecture, all references to instructions in the following sub-sections, unless explicitly specified, refer to the IA-32 architecture.

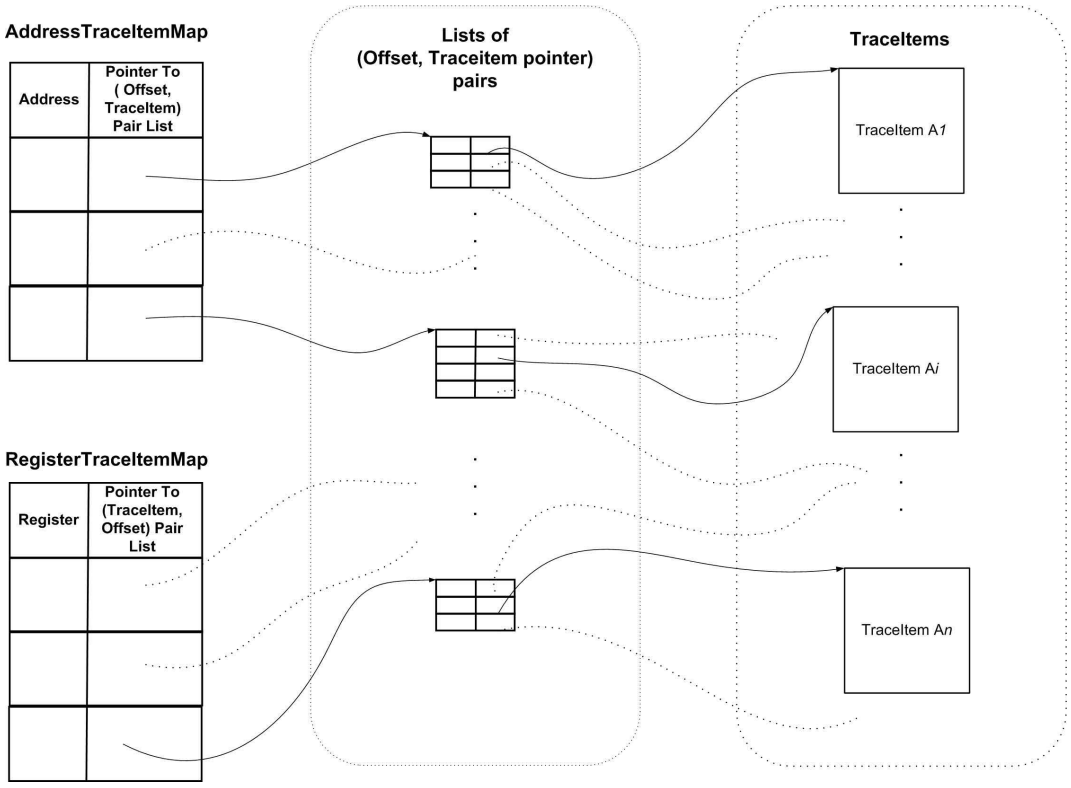

Figure 4.11: Tracing System Data Structures

Data movement instructions move data from registers to memory, memory to registers or from memory to memory via registers. Such data movement instructions may move the references or 32-bit values which are being traced. Other instructions, like ADD, SUB, etc, also need to be examined to check if they modify any references or 32-bit values.

- Data Movement Instructions : Move (MOV, MOVS) instructions move data to/from memory and registers. The MOV instruction is defined as `MOV dst, src` where `MOV` is the instruction mnemonic, `dst` and `src` are the destination and source operands respectively. The source and destination operands can be memory or registers with the restriction that the instruction can have a maximum of one memory read operand or one memory write operand.

  When a move instruction reads from memory (or register), the tracing system checks if

the reference being read is present in `AddressTraceItemMap`. If this reference is present in `AddressTraceItemMap` then it is being traced by the tracing system. Since this reference is being copied to another location, the tracing system makes a copy of the list of TraceItems pointed to by the reference, and adds the destination location (memory address or register) to each TraceItem's `locations` or `register` set. Then the reference is added to either `AddressTraceItemMap` or `RegisterTraceItemMap` depending on whether the new reference is a location or a register. If the destination operand is already present as a reference to any TraceItem, the reference is first killed.

In the process of determining if a source or a destination operand is a reference to some existing TraceItem, it is also necessary to check if reading the source or writing to the destination involves dereferencing a pointer to a TraceItem. This step requires checking if the Base Register used to address the source/destination memory operand is a reference that is being traced. If this is true, the instruction is trying to access some TraceItem through a reference that is being traced. Since the offset of the location from the TraceItem being accessed is known to be the difference between the effective address of the source/destination operand and the start of the TraceItem being accessed, it is possible to determine if the access to the memory location is in-bounds or not. If the Base register is being traced then this will result in the generation of one or more of the following events: *Valid Read Event*, *Valid Write Event*, *Invalid Read Event* and *Invalid Write Event*.

When an instruction kills a reference, it is also important to check the state of the TraceItems that were referred to by this killed reference. If killing this reference results in a state where the `location` and `register` set become empty, a *Lost Reference Event* occurs. All the clients interested in these TraceItems will be notified of the occurrence of this event.

- Stack Modifying Instructions : are data movement instructions that modify the Stack

Pointer (ESP) in the process of the movement. PUSH, POP are the instructions which are used to insert or remove data from the stack. Modifying the ESP will result in data being removed or added from the stack. If data is being removed from the stack, then all references to TraceItems that may exist on the area being removed must be removed from the Tracing data structures.

The events generated are similar to the events generated during data movement instructions.

- Other Instructions :

  - Load Effective Address (LEA) : is an instruction that loads the effective address of a memory location into a register. For example, `lea dst, src` loads the effective address of a memory location addressed by src into the dst register. If the instruction loads the address of a 32-bit value or an address of a TraceItem (start address inclusive and end address exclusive), then a new reference to that TraceItem is created. The list of TraceItems for this reference is created from the TraceItems that have the effective address being loaded in their address range. This instruction will result in the creation or destruction of a reference. Hence it can generate only lost reference events.

  - Subtract (SUB) and Add (ADD) : SUB and ADD, among other uses, can be used to modify memory addresses. When a reference that is currently being traced is added to or subtracted from, the operation will result in a new reference. If neither of the operands to ADD or SUB instruction are references being traced currently, the resulting value cannot logically reference any TraceItem. Such values are discarded.

    When these memory addresses belong to the TraceItems we are tracing, we need to trace modifications to these addresses. This case is of interest when references

hold addresses of reference TraceItems. After modification of the reference, for example adding one to it, it still refers to the original TraceItem and we can always get the original reference by subtracting one from it. Similarly the tracing system also checks to see if any increment (INC) or decrement (DEC) instruction modifies a reference that is being traced currently and adds it to the tracing set. References to Value TraceItems are removed from the tracing set when they are modified by any instruction.

- Exchange (XCHG) - XCHG exchanges the contents of a memory location or register with another memory location or register with the restriction that both operands cannot simultaneously be memory locations. The tracing system checks if any operand is a reference being traced. If either one of the operands (not both) is a reference being traced, then the pointer to the list of TraceItems from the valid reference is copied to the other operand. Also the `location` and the `register` set of each TraceItem in the list is modified; the old reference is removed and the new reference is added. If both operands of this instruction are references, then the pointer to the list of TraceItems is exchanged and the TraceItem's `location` and `register` sets are modified accordingly.
- Any other instructions that write to registers simply destroy existing values. If the register/memory location that is overwritten currently holds a reference to a TraceItem, then we remove the register/memory location from the set of references to that TraceItem.

Limitations. Currently, the tracing system supports only the instructions discussed above. It is possible to modify references using instructions like `AND`, `OR`, `XOR`, etc. The tracing system does not handle the effect of these instructions in the tracing process.

Chapter 5

Case Study: A Memory Checker

We have developed a Memory Checker (MC) as a first step to demonstrate the ease with which error detecting clients can be developed using Top. MC is a client that helps in detecting various kinds of errors that occur when the rules of dynamic memory allocation are violated. MC can detect errors such as out of bound memory accesses, accesses to memory already freed (dangling pointers), attempts to free memory more than once and memory leaks.

5.1 Memory Checker State Machine

The correct use of dynamically allocated memory can be represented using a finite state machine as shown in figure 5.1. The state changes of each memory block are tracked using a new instance of the state machine.

When a malloc event occurs, the state machine goes to the Initial State. Calling free releases the memory block; this results in the state machine going to the final state. Any read/write after freeing (Final State), results in the state machine going to the error state (Error State

1). If the object is leaked, while the memory block is still valid, the state machine goes to the error state (Error State 2).

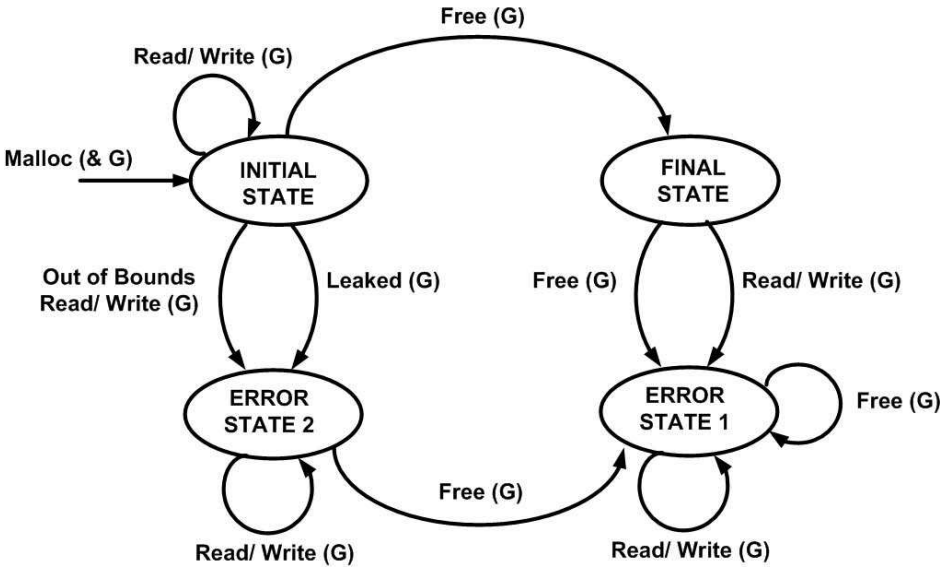

Figure 5.1: Memory Checker State Machine

The events like Malloc, Free, Leaked, etc., specified in figure 5.1 need not be restricted to any specific language, since the usage of dynamically allocated memory in programs that do not use a garbage collector are similar. The events in figure 5.1 are described in table 5.1.

5.2 Design and Implementation

For every `malloc` call by the application, MC creates a new instance of the state machine to record the state of the memory block. A single state machine can only track the changes in state related to a single block of memory. Hence we need one instance of the state machine per block of allocated memory.

A single dispatcher object dispatches events of interest to different instances of state machines. The dispatcher is a special purpose listener that listens to events and dispatches

Event	Description
G = Malloc()	Memory allocation routine. G is a reference to the allocated memory block.
Free(G)	Memory deallocation routine. G is a reference to the memory block that is being deallocated.
Read/Write(G)	Reading or writing to the block of memory referenced by G.
Leaked(G)	All references to the memory block containing the address G is lost.

Table 5.1: Description of Memory Checker Events

```
1  int main () {
2    char *c = malloc (10);
3    free (c);
4  }
```

Figure 5.2: Program R Source

them to the appropriate state machines. The dispatcher object does not have any internal state by itself, instead, it is responsible for registering start events. When the dispatcher receives a start event, it creates a new instance of the state machine and hands over the start event to that instance.

In the case of MC, the dispatcher registers for all malloc function return events. When the dispatcher receives a `malloc` function return event, it creates a new Memory Checker state machine instance and passes the event to this newly created instance. This instance then registers for other events that are related to the memory block returned by the malloc call. It registers for three events, a free event with the start address of the memory block as the

parameter, a memory access event for the memory block returned by malloc, and a lost reference event for the malloc returned blocks. When these events occur, they are directly reported to the Memory Checker state machine instance that had registered for it.

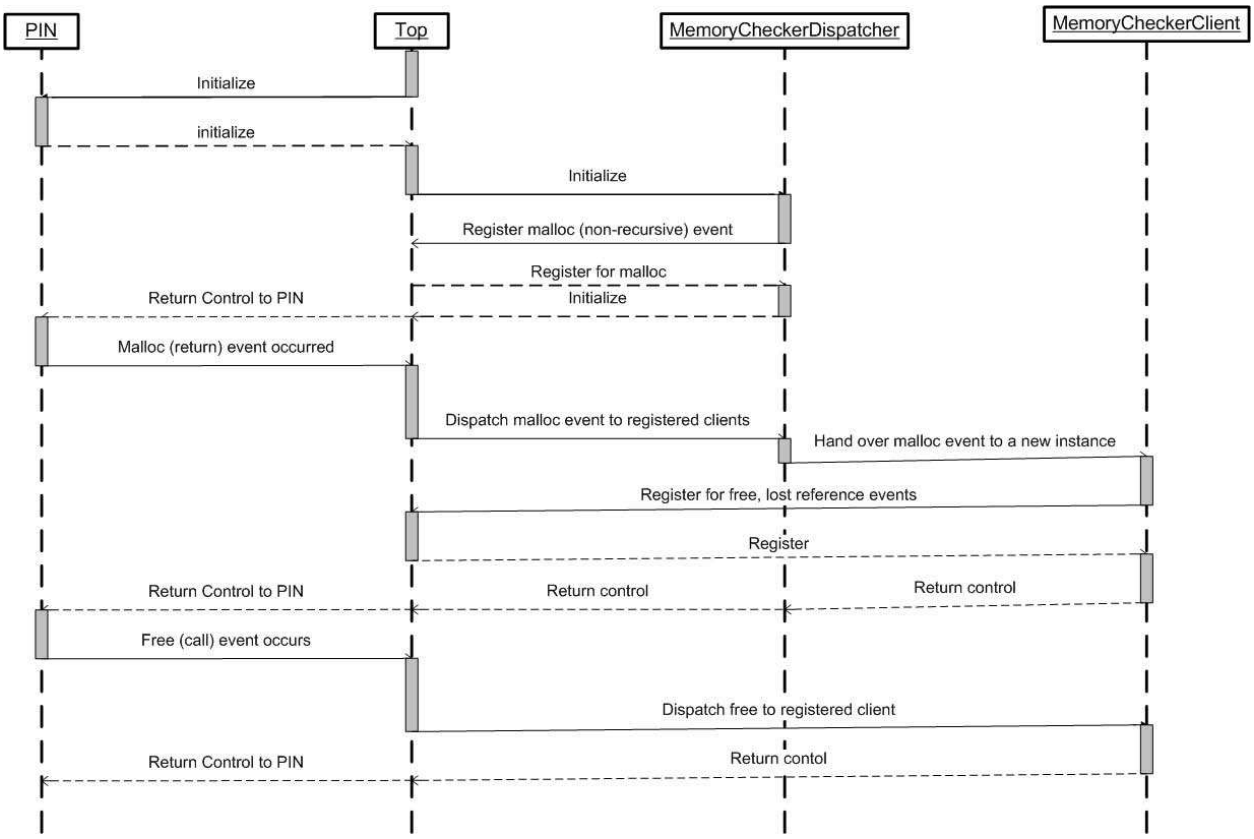

Figure 5.3: Interaction between Pin, Top and the Memory Checker

The MC comprises a listener `class MemoryCheckerDispatcher` that listens for start events and hands them over to a new instance of `class MemoryCheckerClient`. A single instance of `class MemoryCheckerDispatcher` registers for the start event, the `malloc` call, and hands over the event to a new instance of `class MemoryCheckerClient`. The role of the singleton `MemoryCheckerDispatcher` object is to listen for the start event and dispatch the event to a new instance. The new instance registers for other events with Top and Top reports the occurrence of those events directly to the instances. The state transition logic which does the

```
1  void MemoryCheckerClient::stateTransition (MemoryCheckerEvent event) {
2    switch (state) {
3      case STATE_UNINIT:
4        if (event == EVENT_MALLOC)
5          { state = STATE_INIT; }
6        else
7          { state = STATE_ERROR2; }
8         break;
9
10     case STATE_INIT:
11       if (event == EVENT_FREE)
12         { state = STATE_FINAL; }
13       else if (event != EVENT_ACCESS)
14         { state = STATE_ERROR2; }
15       break;
16
17     case STATE_FINAL:
18       if ((event == EVENT_ERROR) || (event == EVENT_FREE))
19         { state = STATE_ERROR1; }
20       break;
21
22     case STATE_ERROR2:
23       if(event == EVENT_FREE)
24         { state = STATE_ERROR1; }
25        break;
26
27     case STATE_ERROR1:
28        break;
29     }
30 }
```

Figure 5.4: Memory Checker Client: State Transition Logic

actual checking of the correct usage of dynamic memory allocation API and the dynamically allocated memory is implemented as part of `class MemoryCheckerClient`.

Figure 5.3 is a sequence diagram that depicts an interaction between Pin, Top and the MC

for a simple application program Program R(figure 5.2). MC comprises a single instance of `class MemoryCheckerDispatcher` and one or more instances of `class MemoryCheckerClient`. For Program R, a single instance of `class MemoryCheckerClient` does the required error checking. Top sets up the required initialization by instrumenting the application and the `MemoryCheckerDispatcher` instance registers for the start events of the MC state machine. When Pin notifies Top of a `malloc` event, Top dispatches the event to the `MemoryCheckerClient` instance. The `MemoryCheckerClient` instance then registers for `free` events. When a `free` event occurs, Top notifies the client of the same.

Figure 5.4 shows the Memory Checker state machine implementation.

5.3 Testing the Memory Checker

We used tests from Luecke et al[LCH+06] (explained in Appendix C) to test the effectiveness of our memory checker client. Luecke et al has a detailed survey of systems for detecting run time errors. Several commercial and non-commercial tools were tested with a suite of tests. The tests include detecting several errors such as out of bounds indexing of statically/dynamically allocated arrays, out of bounds pointer references, memory allocation/deallocation errors and memory leaks. We have re-used the tests related to memory allocation/deallocation, out of bounds indexing of dynamically allocated arrays and out of bounds pointer references to test our memory checker client. MC passed all of Luecke's tests. Several new tests were also added to the test suite to test the effectiveness of the memory checker client and compare its capabilities to tools like valgrind.

We ran several tests on our MC. We have also run some sanity tests to show that the event registration/dispatch and the tracing infrastructure work as expected. MC is implemented using the APIs provided by the infrastructure in under 700 lines of C++ code. It does

not provide all the features provided by valgrind's memcheck skin, but provides stronger guarantees than valgrind's addrcheck. Memcheck provides definedness checking[Net04] which is used to check for the use of uninitialized values with bit-precision which is not present in MC. Definedness checking requires an extra bit of memory for every bit of program memory being checked, doubling the memory required for program data. This feature could be implemented in Top by checking if a memory location is written to before being read. Memcheck is implemented in 7737 lines of C code and 65 lines of assembly code; addrcheck is implemented in 1345 lines of C code.

Chapter 6

Case Study: MPI Checker

Message Passing Interface (MPI) is a library used for communication and synchronization between parallel programs. The MPI Checker checks for proper usage of asynchronous communication primitives in MPI applications.

6.1 Semantics of Asynchronous Communication Primitives in MPI

MPI provides MPI_Isend and MPI_Irecv functions for performing non-blocking send and receive operations. The function prototypes are shown in figure 6.1.

MPI_Isend starts a send operation but does not wait for its completion, i.e., the call returns before the data is copied out of the buffer. Similarly, MPI_Irecv starts a receive operation but returns before the data has been received and copied into the buffer. MPI_Isend and MPI_Irecv allocate a request object and return a pointer to it. This request object must be used later to query the status of the non-blocking send or receive operation using MPI_Test

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request *request);
```

Figure 6.1: MPI_Isend and MPI_Irecv

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

Figure 6.2: MPI_Test and MPI_Wait

or MPI_Wait.

The MPI_Test and MPI_Wait prototypes are shown in figure 6.2. MPI_Test checks whether the non-blocking send or receive operation identified by the request object has finished and returns a flag. If the flag is true, the operation has completed. The MPI_Wait function blocks until the operation identified by the request completes. For proper operation, a call to MPI_Isend or MPI_Irecv must be followed by a call to MPI_Wait or MPI_Test until the send or receive is complete.

6.2 MPI Checker Design and Implementation

The MPI asynchronous communication primitives can be checked using the state machines in figure 6.3 and figure 6.4. Table 6.1 shows the events that are relevant to the MPI Checker.

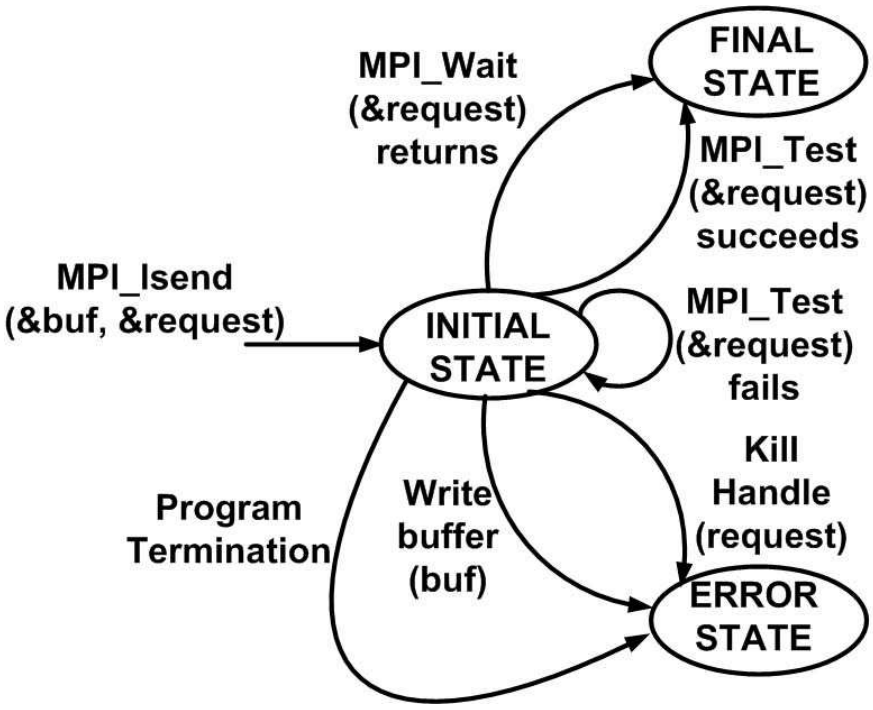

Figure 6.3: MPI Checker Isend State Machine

The MPI Checker comprises a listener `class MPIIDispatcher` that listens for start events and hands them over to a new instance of `class MPIIsendClient` or `class MPIIrecvClient`, depending on the start event. The client classes register for other events (shown in Table 6.1) with Top, and Top reports the occurrence of the events to the instances. The state transition logic which checks for the correct usage of asynchronous MPI function calls is implemented as part of `class MPIIsendClient` and `class MPIIrecvClient`.

6.3 Testing the MPI Checker

We tested this client on test programs that make incorrect usage of these API calls for the following cases,

- Kill Request Handle - This test is used to check if the program calls `MPI_Isend`/`MPI_recv`

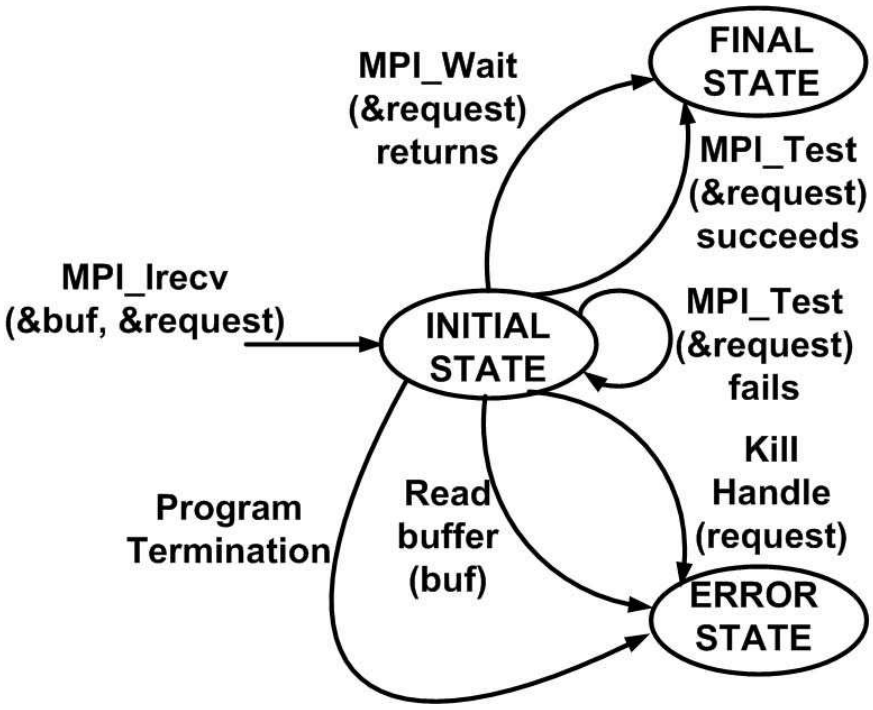

Figure 6.4: MPI Checker Irecv State Machine

and the handle for the operation is killed without waiting for the operation to complete.

- Write to buffer before MPI_Isend is complete - This test is used to check if the program calls `MPI_Isend`, but starts to use the buffer (to write) without waiting for the operation to complete.

- Read From buffer before MPI_Irecv is complete - This test is used to check if the program calls `MPI_Irecv`, but starts to use the buffer (to read) without waiting for the operation to complete.

  MPI Checker was able to detect and report these errors.

Event	**Description**
MPI_Isend(&buf, &request)	Request for asynchronous send operation, returns a request handle
MPI_Irecv(&buf, &request)	Request for asynchronous receive operation, returns a request handle
MPI_Wait(&request)	Wait for asynchronous operation completion
MPI_Test(&request)	Test the status of the asynchronous operation
KillHandle(request)	Delete all copies of the request handle
ReadBuffer(buf)	Read from the buffer
WriteBuffer(buf)	Write to the buffer

Table 6.1: Description of MPI Checker Events

Chapter 7

Evaluation

Top's API can be used to specify high level program events. Thus it is easy to write complex error checkers using only small amounts of code. We have shown its effectiveness in the previous chapters by discussing the Memory & MPI Checkers. In this chapter, we evaluate Top's performance and discuss some of the limitations of our current prototype.

7.1 Performance

The Tracing System instruments every instruction that reads/writes memory and registers. There are at least two analysis routines executed for every such instruction that the application executes. As a result, Top executes a significant amount of analysis code and applications run slower than they would without Top. Since the implementation is not optimized for performance, we expect a significant slowdown.

[Cha06] has done an analysis of Top's performance. Table 7.2 (from [Cha06]) shows the slowdown numbers for the programs Program *1*, Program *2* and Program *3* (from [Cha06]) on different configurations (shown in Table 7.1).

Program *1* calls the `inet_ntoa()` function in a loop, Program *2* is a simple malloc/free program and Program *3* is a matrix multiplication program which multiplies two square matrices of size 10.

Configuration	**Description**
pin	The application is executed with pin, but without any pintool
icount	The application is run with a simple pintool which counts the instructions
MallocChecker + Top + pin	The application is run with the MallocChecker over Top
NullChecker + Top + pin	The application is run over Top with a NullChecker

Table 7.1: Performance Test Configurations

Configuration	Slowdown		
	Program *1*	Program *2*	Program *3*
pin	2.45	2.186	0.75
icount	34.62	28.5	18.93
MallocChecker + Top + pin	3274.39	8201.55	2891.6
NullChecker + Top + pin	4343.87	3268.21	4222.22

Table 7.2: Slowdown of Programs (from [Cha06])

7.2 Limitations

Pin limits the number of parameters that can be passed to an analysis routine. This limitation restricts the number of parameters the client can specify while registering for an event. Currently clients can specify a maximum of seven parameters when registering for events.

```
1. int main ()
2. {
3.   int *arr = malloc (sizeof (int)*10);
4.   int *ptr = arr;       // create local copy of 'arr'
5.   func(arr);            // suppose we are interested in
                           // pointer dereference using 'arr'
                           // or its copies made from this point.
6.   return ptr[0];        // causes a memory access event
7.  }
8. void func (int *lptr)  // copy of arr made
9. {
10.  lptr[0] = 5;          // causes two events.
                           // a. pointer dereference event
                           //    using lptr, copy of arr.
                           // b. memory access event for
                           //    accessing the allocated array.
11.}
```

Figure 7.1: Program *E3* Source

As mentioned in Chapter 4, it is not always possible to register for dereference events. In such cases, clients should register for memory access events. For example, in the Program *E3* (figure 7.1) a new array is created on line 3. A copy of `arr` is made in the local variable `ptr`. Suppose the client is interested in knowing all dereference events and memory access events from this point (Line 5) in the program with `arr` as a seed. For instance, a checker that traces the first argument passed to 'func' might register for such an event. The formal parameter, `lptr`, in `func` is another local copy of `arr`. Hence `arr` and `lptr` are known references to the array. However, although `ptr` is a valid reference to the array, it was made to point to array before the client expressed an interest in knowing about these events. Hence it not considered to be a valid reference for a dereference event to occur. The access to the array on Line 10 in the program will generate both the events. But the access to the array

on Line 6 causes only a memory access event and not a dereference event, as the pointer `ptr` used to access the array is not known to Top as an alias to the array allocated on line 3.

Chapter 8

Related Work

Dynamic Program Analysis. Valgrind [NS03] is a program supervision framework for creating new error checking tools. It is similar to Pin in that it provides an infrastructure that allows the creation of error checking tools, called skins. However, compared to Top, it has the following limitations:

- Valgrind does not provide an easy to use API to analyze the application. Hence it is not very easy to write new skins.
- Skins written on valgrind are very tightly coupled to the valgrind core. Any changes to the valgrind core requires changes to the skins.
- Valgrind uses the LD_PRELOAD technique to get control of the application. When the LD_PRELOAD environment variable is set to a shared library name, the dynamic linker loads that library before loading any other shared library of the application. The LD_PRELOAD technique works only with applications linked with shared libraries. Therefore valgrind cannot work with statically linked binaries.

Pin [LCM+05] on the other hand, provides a simple interface to instrument the application and analyze it.

- It uses `ptrace` system call to get control of the application. `ptrace` allows Pin to get control of the application before executing the first instruction. In the LD_PRELOAD approach, valgrind gets control only after some linker code gets executed. One of the advantages using `ptrace` is that the dynamic linker code can be instrumented, if needed. In addition, Pin [LCM+05] can instrument statically linked binaries as `ptrace` does not have the limitation of the LD_PRELOAD technique.

- Valgrind's memcheck [Net04] replaces the default memory allocator with its own. Hence the original application behavior may be altered. Our memory checker (with the help of Pin) does not alter the application behavior.

- Valgrind's memcheck skin does not have a tracing system and hence it does not keep track of references to dynamic allocated memory. Our memory checker uses the tracing features of Top and can trace references to dynamically allocated memory. The third example in Section C.1.1 shows that Top's memory checker can perform fine grained checking. On the other hand, valgrind outperforms the current Top prototype by about an order of magnitude, making it an excellent tool for more coarse-grained analyses that do not require the fine-grained tracing Top provides.

Querying Traces. The Program Query Language (PQL) [MLL05] is a recent work which defines a query language for programmers to query for application specific errors or security flaws in Java programs. It uses a hybrid approach consisting of both static and dynamic analysis. The static checker uses an advanced static analysis technique [WL04] to find all potential errors as specified by the programmer's query. The dynamic checker instruments the code and finds the error that actually occurred during execution. The advanced static

analysis help in reducing the number of instrumentation points. The query also has a provision to specify dynamic error recovery.
Program Trace Query Language (PTQL) [GOA05] is another recent work which allows programmers to specify relational queries on program traces. The queries are then compiled into the input Java code by the PARTIQLE compiler.
Unlike Top, the queries in both these techniques do not have provisions to match events based on context (parameters, return values, etc).

Automata Based Checking. SLIC (Specification Language for Interface Checking) [BR01] instruments C sources and uses automata to check for properties. A preprocessor reads the query, transforms it to a state machine and combines the state machine implementation with the original source. It instruments the original source so that control can be transferred to the state machine when certain events of interest take place. SLIC also defines a language to specify queries. Top is similar as it instruments the application with error checking code. But the difference is that Top instruments the application binary at runtime with the help of Pin. Also, our solution not only works with C programs but with programs written in other languages as well.

Aspect Oriented Programming. Aspect Oriented Programming [KLM+97] is an approach by which "cross-cutting" concerns or aspects are separated out into components such that each component has distinct features with minimal overlap. It involves adding code at specific points called "join points" in the program. This process of adding aspects to code is called aspect weaving. An *aspect weaver* weaves all separated concerns and the program together to produce a new program. For example, if an application program uses a hashtable, and if accesses to the hashtable need to be mutually exclusive, a lock needs to be used. The lock procedures can be implemented as an aspect and then weaved with the application code

to ensure mutually exclusive access to the hashtable. In this manner, the concerns regarding acquiring mutually exclusive access to the hashtable and the use of the hashtable by the application are separated. Similarly, the error checking clients and the application program could be considered separate concerns. At runtime, Top(with the help of Pin) weaves the error checking code into the application to detect errors.

AspectJ is an aspect-oriented extension to the Java programming language. Trace matching with free variables in AspectJ [AAC+05] is an extension to AspectJ, which provides a history based language feature called *tracematches*. Tracematches allow execution of extra code when a pattern of events matches during execution; matches can also be made based on event types and the values of free variables. Top uses an approach where it matches function parameters and return values based on the values the client is interested in. The tracematches concept could be implemented as part of the Top client, where the client remembers the sequence of events and triggers execution of code when the match happens.

Domain-specific Runtime Approaches. DieHard [BZ06] is a runtime system that provides "probabilistic memory safety." It uses a "randomized memory allocation" [BZ06] technique to give the application the illusion of an infinite-sized heap. It [BZ06] can run multiple replicas of the application to increase the probability of correct execution since it is less probable that every replica shows the same behavior on errors. Memory Checker client does not replace the memory allocator and hence maintains the original application behavior. Our technique does not currently provide any safety, but that is a possible proposition for the future.

MARMOT [KMR05] is a tool used to find programming bugs in MPI applications at runtime. MARMOT is mostly targeted at detecting errors in communication or synchronization

between processes. It uses a client-server approach. Client code intercepts calls to MPI library functions. Any checking that can be performed locally is done by the client. For group communication or synchronization operations, it notifies the server of the events. The server has a global view of the current state of the processes and hence can be used to detect any bugs due to synchronization or communication. Our tool currently does not support detecting communication or synchronization errors in MPI applications.

Chapter 9

Conclusion and Future Work

9.1 Contributions and Conclusion

We have successfully built Top, an infrastructure for writing clients that can be used to detect application specific program errors. We have demonstrated that a client can be written in the form of a finite state machine using the small set of APIs that Top provides. The memory checker client and the MPI Isend clients are simple examples that demonstrate that it is possible to write such clients in relatively little code.

The contributions of this work are:

- A *Tracing System* that can trace the movement of pointers to memory locations. It can also trace the movement of 32-bit data and keeps track of all its copies. Using this system, we can build checkers that can do fine-grained checking.
- An *Event Framework* that accepts function call/return event queries and report their occurrences.
- Two error checking tools, *Memory Checker* and *MPI Checker*, were built using Top.

Using Top, we can build checkers to detect errors in applications. Complex error checkers can be written easily with the help of Top's APIs. Error checkers can be written in less amount of code when compared to valgrind. Multiple error checkers can be executed together and the application can be checked simultaneously for all error types supported by those checkers.

9.2 Future Work

Our current prototype could be improved in both performance and functionality. We outline some ideas below.

- Instrumenting basic blocks - Top uses a technique that instruments all data movement and data modification instructions. There are at least two analysis functions executed for every such instruction. This technique does not scale and is the cause for the considerable slowdown. Instead of instrumenting every instruction, the tracing system could instrument at the basic block level. The analysis routine at the basic block level can examine the effect of all the instructions (in the basic block) on the TraceItems. This approach will eliminate the time spent in analysis function calls and returns for the instructions in the basic block.

- Support for complex queries - Function call and return events are switched based on the values of the parameters. Clients cannot currently query for more complex patterns, such as `p->prev->next`, which involve repeated dereferencing at a predefined offset. For function calls, in addition to being able to key in based on a function parameter's value, it would also be desirable to register for events based on predicates associated with those function parameters. For example, a client might wish to know if a function is called with a certain value even if this value is passed by reference.

- Capability to trace Global and Stack data - The infrastructure currently does not have

any support to name stack allocated or global variables or arrays. More powerful and complex clients could be written if this capability were added to the infrastructure.

- Extending the infrastructure to other architectures - We have implemented the infrastructure for the IA32 architecture. Top could be ported to handle other architectures already supported by Pin, such as IA64 and XScale.

- Query Language and Compiler - Defining a query language and building a compiler forms a very necessary next step. The language and the compiler will make it easier to write clients.

Bibliography

[AAC+05] C. Allan, P. Augustinov, A. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. 2005.

[BR01] Thomas Ball and Sriram K. Rajamani. SLIC: A Specification Language for Interface Checking. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[BZ06] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *PLDI '06: Proceedings of the ACM SIGPLAN 2006 conference on Programming language design and implementation*, New York, NY, USA, 2006. ACM Press.

[Cha06] Bhupesh Chandra. Analysing Top. Unpublished manuscript, 2006.

[Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[GOA05] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN*

conference on Object oriented programming, systems, languages, and applications, pages 385–402, New York, NY, USA, 2005. ACM Press.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *PLDI 2002*. PLDI, December 2002.

[HL03] D. Heine and M. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 168–181, June 2003.

[ia3] *System V Application Binary Interface, Intel386 Architecture Processor Supplement*, fourth edition.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KMR05] Bettina Krammer, Matthias Muller, and Michael Resch. Runtime Checking of MPI Applications with MARMOT. In *Tools Support for Parallel Programming at ParCo 2005*, September 2005.

[LCH+06] Glenn R. Luecke, James Coyle, Jim Hoekstra, Marina Kraeva, Ying Li, Olga Taborskaia, and Yanmei Wang. A survey of systems for detecting serial run-time errors. *Concurrency and Computation: Practice and Experience*, 18(15):1885–1907, 2006.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 365–383, 2005.

[Net04] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, November 2004.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

Appendix A

Client API Reference

A.1 Event Objects

The error detecting clients specify events of interest using an `class Event` object. All specific events are derived from the `Event` class. Clients and Top each use only a subset of the attributes that are present in the event object. Some attributes are valid for the clients while others are for Top's use, and some will be used by both the client and Top.

Function Call/Return Event `FunctionEvent` is derived from `class Event`. `class FunctionCallEvent` and `class FunctionReturn` are derived from `class FunctionEvent` and they represent function call and function return events.

Lost Reference Event `class LostReference` represents events related to losing references to an object being traced. Note that the same interface, i.e. `class LostReference`, is used by clients to express their interest in tracing objects and by Top to report lost reference events.

Dereference Event `class DereferenceEvent` is derived from `class Event`. Top uses this event object to report dereference events.

Memory Access Event `class MemoryAccessEvent` is derived from `class Event`. This event is used by clients to specify their interest in memory access (read/write) events to blocks of memory.

A.2 Event Registration

The following set of APIs is used to register for events.

Function Call/Return Registration Register Event call is used to register for a function call or a function return.

```
EventCoordinator::registerEvent (FunctionEvent& event,
                                 ParamSelection& combination,
                                 Listener* listener,
                                 bool allCalls,
                                 bool defaultReturn);
```

This API is used to register an interest in a Function event. The `FunctionEvent` object is used to describe the function of interest and any necessary parameters. The `ParamSelection` object specifies the combination of the parameters that are of interest. The values of the parameters are specified as part of the `FunctionEvent` object. A reference to the error detecting client that is interested in the event is specified by the pointer to `Listener` object, as stateless clients are inherited from `class Listener` . Clients that maintain state are inherited from `class StateMachine`, which is a derived class of `class Listener`.

The last two parameters are used for specific functionalities. The boolean `allCalls` specifies whether the client is interested in all the function calls or only the non-recursive ones. If not specified, it returns all calls. This parameter is valid for both function call and return events.

The boolean `defaultReturn` flag is valid only for function call events. It is ignored for function return events. `defaultReturn` specifies if the client is interested in the return of the function call. If the flag is set, then the return event is registered. The client need not register for the function return explicitly.

Tracing Registration To trace reference or value TraceItems, clients register for `LostTraceEvent`'s using the `registerEvent` API.

```
EventCoordinator::registerEvent (LostTraceEvent& event,
                                 REG seed,
                                 Listener *listener);

EventCoordinator::registerEvent (LostTraceEvent& event,
                                 ADDRINT seed,
                                 Listener *listener);
```

There are two different API's for the two types of seeds that the TraceItem could have. Clients need not register for events related to reading and writing of TraceItems as they are automatically registered when the LostTraceEvent is registered. If clients are not interested in the read and write events, they can simply ignore them by not implementing the call back `trigger` function for those events.

Memory Access Event Registration Memory access events can be registered by clients using the following API.

```
EventCoordinator::registerEvent (MemoryAccessEvent& event,
                                 Listener *listener);
```

Note that this event is different from the DereferenceEvent described in the previous section. DereferenceEvents are reported when a memory location is accessed using any of the traced aliases, whereas MemoryAccessEvents are reported regardless of whether a memory location was accessed via a traced alias.

Appendix B

Writing a client

A client is a part of the pintool which implements the error detecting logic using the Top API. Most clients fall into two categories, i.e., Stateless and State-based clients. Stateless clients are those that do not have any state information; they register for events and when such events occur, they either log or write the events to a console. Stateless clients are derived from `class Listener`. State-based clients are used for application specific error detection. They are implemented by inheriting from `class StateMachine`.

A client has two important parts. Firstly, every client implements the transition logic that determines what the client is trying to achieve. Secondly, the client implements event callback functions. Event callback functions are functions that are called by Top when events of interest to the client occur. This mechanism is used to notify clients of events. Every event is reported with the context of the event. Client's event callback function may provide some input to their state transition logic and may register in new events as well. Top currently provides four callback functions that clients can implement. It is not necessary that every client implements all callback functions; a client need only implement the ones in which it is interested. Callback functions for FunctionEvent, DereferenceEvent, MemoryAccessEvent

and LostReference events are currently supported. Function call and return events can be distinguished by the `type` member in the `FunctionEvent` object.

There are four steps involved in building a client as listed below:

- Deriving from `class Listener` or `class StateMachine` - All clients need to inherit from either `class Listener` or `class StateMachine`. `class Listener` is the stateless abstract class that declares the four callback functions corresponding to the four types of events supported by the infrastructure. `class StateMachine` derived from `class Listener` is the abstract class that has a state.

- Registering for events - Clients need to register with Top for events of interest. The client registration functions allows them to register for function call/return, pointer dereference or lost trace events.

- Implementing the callback functions - Clients need to implement the four callback functions when they inherit from `class Listener` or `class StateMachine`. They can choose not to implement one or more of these functions depending on whether those events are of interest.

- Implementing the client state transition logic - Clients will need to implement a state transition logic. The events and states need to be defined by the client. The stateTransition is generally triggered from the callback functions, depending on what event takes place.

Appendix C

Tests

C.1 Tests from Luecke et al[LCH+06]

C.1.1 Out of bound pointer references

- Check if writing out of bounds using array subscripts is detected.

```
1  float *A;
2  A = (float*) calloc (5, sizeof(float));
3  for (i = 0; i < 5; i++)
4      A[i] = i;
5  A[5] = 1.0; // out of bounds writing using
6              // subscripts.
```

Figure C.1: Out Of Bound Pointer Reference Test 1

- Check if accessing an array using a pointer is valid.

```
1  float *A, a, *p;
2  int i;
3  A = (float*) calloc (5, sizeof(float));
4  for (i = 0; i < 5; i++)
5      A[i] = i;
6  p = A;
7  for (i = 0; i < 5; i++)
8      p++;
9  a = *p;
```

Figure C.2: Out Of Bound Pointer Reference Test 2

- Check if out of bound array access is detected - Valgrind's memcheck failed to detect the incorrect access of memory on line 5 in figure C.3. The program accesses the memory area pointed by '`b`' using the pointer '`a`'. Though the program is accessing a valid memory block, it is doing so using an incorrect pointer. Valgrind is not able to detect this error because it only checks if the memory address accessed is valid, but does not perform bounds checking on the pointer. In Line 5, due to the padding by valgrind's memory allocator, `a[70]` has the same address as `b[6]`. Even though `a[70]` points to the sixth byte of memory pointed by pointer '`b`', '`a`' is not supposed to be pointing to this area of memory. Top's Memory Checker, unlike valgrind's memcheck skin, maintains the references and knows that '`a`' cannot refer outside the allocated 16 byte area. Hence any memory access through '`a`' outside the block is flagged as an error.

```
int main()
{
  char *a = malloc(sizeof(char) * 16);
  char *b = malloc(sizeof(char) * 16);
  a[70] = ' ';
  free(a);
  free(b);
}
```

Figure C.3: Out Of Bound Pointer Reference Test 3

C.1.2 Memory allocation and deallocation errors

- Check if attempt to free invalid memory is detected.

```
int *a, *b;
a = (int*) calloc (32, sizeof(int));
b = (int*) realloc (a, 64*sizeof(int));
free (a);
free (b);
```

Figure C.4: Memory Allocation And Deallocation Test 1

- Check if attempt to free memory using an address that does not point to start of the allocated memory is detected.

```
1  int *a;
2  a = (int*) calloc (32, sizeof(int));
3  a++;
4  free (a);
```

Figure C.5: Memory Allocation And Deallocation Test 2

C.1.3 Memory Leaks

- Function is called which allocates memory, but does not deallocate on return and the single reference is lost, resulting in a leak.

```
1  void func () {
2      int *p;
3      p = (int*) calloc (5, sizeof(int));
4  }
```

Figure C.6: Memory Leak Test 1

- Memory leak occurs when an exception is being handled.

```
1  class A {
2  public:
3    A () {
4        dummy = new char [100];
5        throw "boom";
6      }
7   ~A () { delete dummy; }
8  };
9  int main () {
10   try {
11       A *pa = new A ();
12     }
13   catch (...) { }
14   return 0;
15 }
```

Figure C.7: Memory Leak Test 2

C.2 Other Tests

We have built our own test suite to test the reference and value tracing component of our infrastructure. Each of the tests listed here test a specific feature of either the Tracing System or the Event Framework.

- Normal Malloc/Free - A simple test of a malloc followed by a free.

```
1  int main () {
2    char *c = malloc(10*sizeof(char));
3    free(c);
4    return 0;
5  }
```

Figure C.8: Normal Malloc/Free Test

- Simple Leak - A test where the address of the malloc'ed buffer is lost.

```
1  void func (void) {
2    char *c = malloc (10);
3  }
4  int main () {
5    func ();
6    return 0;
7  }
```

Figure C.9: Simple Leak Test

- Local Variable Aliasing - A local alias to the pointer is made and the copy is used to release memory. This test confirms that the copying of pointers within the function is traced.

```
1  int main() {
2    void *p = (void*)malloc((sizeof(int)*10));
3    void *q;
4    // local aliasing
5    q = p;
6    free(q);
7    return 0;
8  }
```

Figure C.10: Local Variable Aliasing Test

- Heap Variable Aliasing - This test is similar to the local variable aliasing test except that the copy is made on the heap.

```
1  struct ds { void *x; };
2  int main() {
3    struct ds *ptr = (struct ds*) malloc (sizeof(struct ds));
4    void *p = (void*)malloc((sizeof(int)*10));
5    // create reference to trace item on heap
6    ptr->x = p;
7    free(ptr->x);
8    free(ptr);
9    return 0;
10 }
```

Figure C.11: Heap Variable Aliasing Test

- Heap Variable Aliasing Leak - The single reference (to a memory block) on heap is lost as a result of freeing a memory block containing it.

```
1  struct ds { void *x; };
2  void func() {
3    struct ds *ptr = (struct ds*) malloc (sizeof(struct ds));
4    void *p = (void*)malloc((sizeof(int)*10));
5    // put reference on heap
6    ptr->x = p;
7    // free TraceItem containing reference.
8    free(ptr);
9  }
10 int main() {
11   func();
12   return 0;
13 }
```

Figure C.12: Heap Variable Aliased Leak Test

- Global Variable Aliasing - Create an alias to a memory block in a global variable and use that to free memory.

```
1  int *g;
2  int main() {
3    int *x = (int*)malloc((sizeof(int)*10));
4    g = x;
5    free(g);
6    return 0;
7  }
```

Figure C.13: Global Variable Aliasing Test

- Function Call - Tests aliasing by passing a pointer as function argument.

```
1  void func(void *p) {
2          free(p);
3  }
4  int main() {
5    void *p = (void*)malloc((sizeof(int)*10));
6    func(p);
7    return 0;
8  }
```

Figure C.14: Function Argument Aliasing Test

- Function Return - Tests aliasing by passing a pointer as function return value.

```
1  void* func() {
2    void *p = (void*)malloc((sizeof(int)*10));
3    return p;
4  }
5  int main() {
6    void *p = func();
7    free(p);
8    return 0;
9  }
```

Figure C.15: Function Return Aliasing Test