

TraceBack: First Fault Diagnosis by Reconstruction of Distributed Control Flow

Andrew Ayers
Richard Schooler
Microsoft Corporation
{andya,schooler}@microsoft.com

Anant Agarwal
Chris Metcalf
MIT CSAIL / VERITAS Software
agarwal@csail.mit.edu
chris.metcalf@veritas.com

Junghwan Rhee
Emmett Witchel
University of Texas at Austin
{jhrhee,witchel}@cs.utexas.edu

ABSTRACT

Faults that occur in production systems are the most important faults to fix, but most production systems lack the debugging facilities present in development environments. TraceBack provides debugging information for production systems by providing execution history data about program problems (such as crashes, hangs, and exceptions). TraceBack supports features commonly found in production environments such as multiple threads, dynamically loaded modules, multiple source languages (e.g., Java applications running with JNI modules written in C++), and distributed execution across multiple computers. TraceBack supports *first fault diagnosis*—discovering what went wrong the first time a fault is encountered. The user can see how the program reached the fault state without having to re-run the computation; in effect enabling a limited form of a debugger in production code.

TraceBack uses static, binary program analysis to inject low-overhead runtime instrumentation at control-flow block granularity. Post-facto reconstruction of the records written by the instrumentation code produces a source-statement trace for user diagnosis. The trace shows the dynamic instruction sequence leading up to the fault state, even when the program took exceptions or terminated abruptly (e.g., kill -9).

We have implemented TraceBack on a variety of architectures and operating systems, and present examples from a variety of platforms. Performance overhead is variable, from 5% for Apache running SPECweb99, to 16%–25% for the Java SPECjbb benchmark, to 60% average for SPECint2000. We show examples of TraceBack’s cross-language and cross-machine abilities, and report its use in diagnosing problems in production software.

Categories and Subject Descriptors D.2.5 [Testing and Debugging]: Debugging aids.

General Terms Performance, Design.

Keywords fault diagnosis, instrumentation

1 Introduction

TraceBack is a first fault diagnosis system. It is a tool for diagnosing faults which occur in production environments, without the developer having to recreate the fault. While relatively few software defects escape a mature development process into actual production deployment, the defects that do escape are typically the most difficult and expensive to find and

fix. Often these problems occur only in production environments, or the cost in time and/or money to reconstruct them by the developer is prohibitive. These problems can be timing dependent (such as deadlocks or race conditions), environment dependent (such as high load levels or mismatched versions of software libraries), or dependent on code that a customer cannot ship off-site even in binary form.

Production bugs are the most important to fix, but they can present the greatest challenge because the developer often does not have enough information or resources to recreate the problem. Sometimes it is impractical to maintain a test environment that fully parallels the production environment, especially for distributed systems. TraceBack benefits the end user by allowing the developer to fix bugs actually experienced by the user, and it benefits the developer by freeing them from spending resources to recreate important bugs.

TraceBack uses static binary translation to do efficient runtime execution tracing. A program is translated by TraceBack into another program that is functionally identical to the original, but which also records information about its own execution history. The instrumented program can usually run in a production environment because the additional execution time and memory overhead of recording the execution history is usually small. If a program fails in the production environment, the execution history information it collects allows an engineer to single step the program back from the fault location to discover why the program terminated abnormally. Using TraceBack is like having an implicit debugger in every program.

By using binary rewriting, TraceBack operates on production program components, without the need for source code. This implementation technique depends on the specifics of the instruction set architecture, operating system interface, object module format, and compiler code generation style. We present an architecture that isolates these platform-specific details to the lowest levels of the system.

TraceBack is the first binary rewriting system to focus on reconstruction of control flow for first-fault diagnosis. As such, it differs from previous binary instrumentation systems [17][20][27] by integrating traces from multiple languages, robustly allowing parts of a program to be not traced, and by recording useful trace information up to the point of a crash, exception, or abrupt termination. It differs from previous systems that capture program control flow [4][18] by supporting multi-threaded applications and by supporting distributed execution. It captures traces in a distributed system, across threads, machines, languages, and runtime systems and correlates these traces into a causal order. It supports C, C++, Java, .NET, and VB6 on Microsoft Windows on IA32; C, C++, and Java on Sun Solaris on SPARC; Java on AIX, HP-UX, and Linux on x86; C and C++ on Linux on x86; and Cobol on IBM OS/390.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI’05, June 12–15, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-056-6/05/0006...\$5.00.

Our description of TraceBack is divided into the following sections: *Tracing control flow*, for collecting the trace data, *Runtime*, for the code supporting the instrumentation probes, *Reconstruction*, for raising the trace back to source level and displaying it, and *Distributed tracing*, for merging trace data from several sources, including distributed machines. Experimental evaluation, related work, and conclusions follow.

2 Tracing Control Flow

TraceBack instruments applications by statically rewriting the binaries, transforming the original program into a program that performs the same computation as the original, but which also records a history of its control flow.

Binary rewriting begins by separating code from data. Except for Cobol on IBM OS/390 (which is beyond the scope of this paper), TraceBack relies on known techniques to separate code from data [17][20][27]. After separation, code and data are *lifted* to an abstract graph representation, which is independent of machine specifics and which is analyzed, modified and then lowered back to a legal binary representation [26]. TraceBack uses well-known compiler algorithms like liveness analysis to allow instrumentation code to make use of architectural registers.

TraceBack aims to accurately describe the execution history of a program, especially the execution history that directly precedes a crash or program hang. It provides an exact, interprocedural, whole program profile [18]. Section 2.1 describes how TraceBack probes work within a procedure, and then Section 2.2 progresses to interprocedural probes. TraceBack supports multiple code modules which are dynamically loaded, so the next section describes cross-module calling. Section 2.4 discusses how TraceBack makes sure exception records refer to the correct source line, while Section 2.5 discusses support for multiple threads.

2.1 Intraprocedural Control Flow

We first consider the problem of summarizing control flow in a connected graph of code blocks, for instance a procedure. This section addresses control flow in a leaf procedure. The next section considers more complex cases.

The problem of probe placement for procedural flow summary has been studied in detail [4][18]. However, the requirements for TraceBack differ from the usual needs of profiling. TraceBack must be able to capture the exact path of execution, even if a block is partially executed because of an exception. This means that it is generally not possible to omit probes from any execution paths. Furthermore, it is not always possible to recover the register state at the point of an exception, so we are not able to keep instrumentation data in registers across original instructions.

A simple approach to instrumentation is to modify each block to append its address to a trace buffer. While this works, it fails to take advantage of the constrained execution orders imposed by the flow graph. The trace will be accurate, but unnecessarily voluminous at one word per block; this extravagance imposes runtime costs that can be avoided.

Previous systems [4] record the address of an initial block in a run and then a succession of branch outcomes at block ends. Since the average number of control flow successors for a block is approximately 2, each additional block requires about 1 extra bit of information. TraceBack instrumentation splits the probes up into two basic categories: heavyweight probes to start the run, and lightweight probes within a run. The probes write trace records,

whose format is shown in Figure 1. The heavyweight probes record the current execution location in the control flow graph, while the lightweight probes set bits in the trace record indicating the execution path.

To keep the lightweight probes simple, they cannot involve conditional logic; this limits the number of bits in a trace record available for use by lightweight probes. The number of blocks that can be described by a trace record is thus likewise bounded. So the heavyweight probes must be placed in such a way that no path through the graph starting at a heavyweight probe can exceed the length limit.

Blocks that end in unconditional branches do not require lightweight probes. Blocks that end in multiway branches will either require special lightweight probes to record the successor block, or, equivalently, one can just end the trace at this point and force all multiway branch targets to hold heavyweight probes.

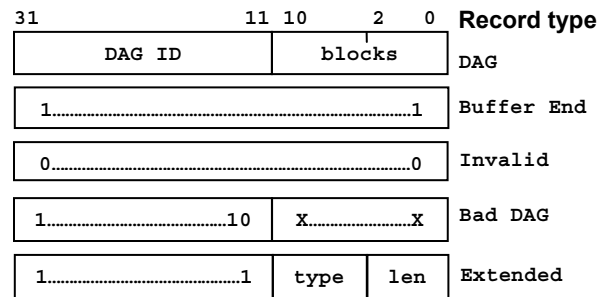


Figure 1 Trace record format. The numbers in bold in top are bit positions (each record is 32 bits). The DAG record includes fields for the DAG ID and which blocks in the DAG were executed. The buffer end record is the sentinel value that the instrumentation code checks for buffer wrap. Zero is an invalid record to support sub-buffering (Section 3.2). The bad DAG ID is used if the DAG ID space is exhausted (Section 2.3). An X denotes a “don’t care” bit which can be 0 or 1. Extended records are used for SYNC records and timestamps. They can span multiple words, so they have a length field.

The limit on run lengths also implies that each loop will contain at least one heavyweight probe (one can do better if the loop trip count is known, but in general it is not known). Also, a heavyweight probe is required at each external entry point to the graph. Because of this, the presence of the heavyweight probes effectively tiles the control flow graph into a set of directed, acyclic subgraphs (DAGs), each headed by a heavyweight probe. Hence, we call the process of identifying the placement for heavyweight probes *DAG tiling*. An example of DAG tiling is seen in Figure 2.

To keep trace records compact, the DAG ID and the path bits are stored in the same machine word. Each heavyweight probe writes a fixed DAG ID in the upper bits of a trace record. The lower bits are then reserved for lightweight probes. To simplify matters, each block that contains a lightweight probe is assigned a particular bit in the lower portion; when the block is executed, that bit is set. For instance in Figure 2, DAG 1, block 1 sets bit 0x1 when it is executed.

To support later reconstruction of the control flow, the instrumentation process needs to build a table to translate block addresses to DAG Ids, and a table to map DAG bits to successor

blocks. This information is saved out alongside the instrumented executable in a file called the *mapfile*.

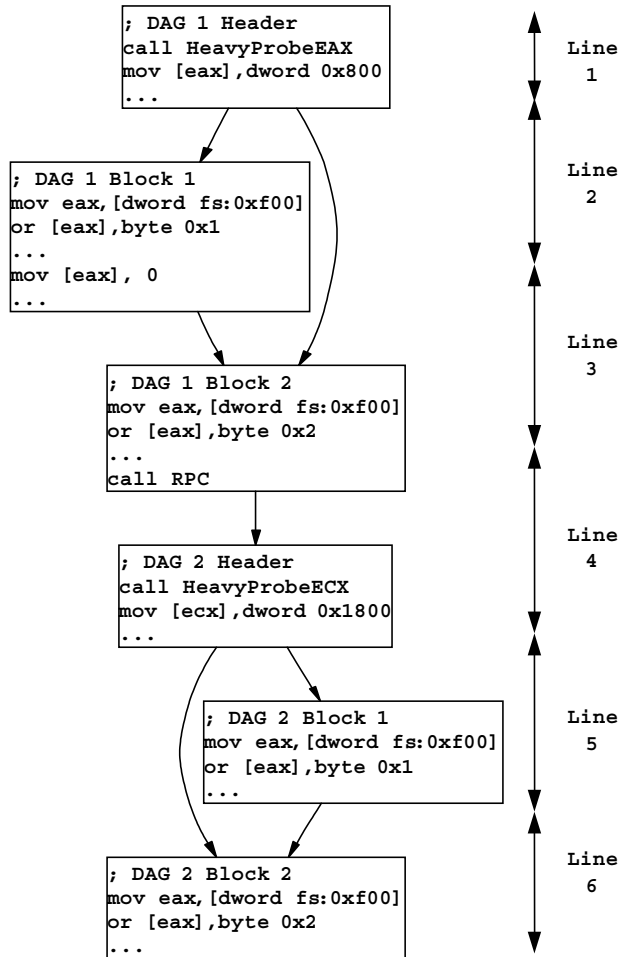


Figure 2 A sample control flow graph showing the placement of heavyweight and lightweight probes. The RPC call forces TraceBack to tile this graph with two DAGs. The line numbers and ranges show how code in the control flow graph corresponds to lines (this particular mapping is illustrative, not realistic).

The runtime support library (see section 3) provides a trace buffer to hold successive trace words. Each thread maintains a pointer to the last-written record in the buffer. The heavyweight probes load, increment, and then dereference this pointer to check if free space remains in the buffer. If so, the preshifted DAG ID is written to the buffer and the updated pointer is saved; if not, the probe calls into the runtime to free up space. The heavyweight probe then writes the pre-shifted, DAG ID into the trace buffer. Because these probes include conditional logic, TraceBack calls them as subroutines. To avoid the overhead of an inter-module call, TraceBack statically adds these subroutines into every instrumented module.

The lightweight probes simply OR their assigned bit as they are executed. For Windows NT binaries on x86, our probes are implemented as shown below. Heavyweight probes are 8 instructions, with two reads (the buffer pointer, and the old next

record) and two writes (the updated buffer pointer and the new record):

```

// Heavyweight probe: DAG ID is 0x800
1010 call near 0x7000
1015 mov [eax], dword 0x800

// Helper subroutine for heavyweight
// probe, use/return via EAX
7000 mov eax, [dword fs:0xf00]
7006 add eax, byte 0x4
7009 cmp dword [eax], byte -0x1
700c jnz short 0x7014
// invokes runtime
700e call [dword 0x51b4] // buffer_wrap
7014 mov [dword fs:0xf00], eax
701a ret

```

Lightweight probes are two instructions: a read to get the buffer pointer, and a read/write to update the bit.

```

// Lightweight probe
1048 mov eax, [dword fs:0xf00]
104e or [eax], byte 0x2

```

This probe architecture yields roughly one line of source code per byte of trace buffer. With a typical buffer of 64Kbytes per thread, TraceBack is able to display tens of thousands of source lines back in time. Furthermore, trace buffers are themselves readily compressible by a factor of 10 or more for ease of archiving or transmission.

2.2 Interprocedural Control Flow

If the control flow graph has a call, the return from that call establishes another entry point into the procedure-level flow graph. Thus, calls are handled by placing a heavyweight probe immediately after the call return point (e.g., the RPC call in Figure 2 breaks a DAG).

In practice, breaking DAGs at calls is a limiting factor for path length, and therefore limits the performance of instrumented code. Heavyweight probes require more instructions and memory references, so eliminating them increases performance. But placing heavyweight probes at procedure returns solves problems related to cross-module control flow and refining the exception address, both of which are explained in the following two sections. Because TraceBack collects interprocedural traces and must provide accurate traces for programs that terminate abruptly, it is less efficient at runtime than path profiling systems [4][5][15].

2.3 Cross-Module Control Flow

Most systems allow separately linked modules to interact within a running process. Trace records must have enough information to reconstruct module crossings. If the architecture restricts module entry points so that they are all known at instrumentation time, the instrumentation package can simply add special “module entry” records to each entry points.

But most architectures allow arbitrary procedure addresses to be saved in data structures as callbacks, so in general it is not possible to know the set of module entry points at instrumentation time. Thus, each module must use a distinct set of DAG Ids to reconstruct cross-module traces instead of using module entry point records.

Unfortunately, because users may separately or independently instrument modules, there is no general scheme for ensuring that all the different modules in a process use distinct ranges of DAG Ids. To address this problem, TraceBack uses DAG rebasing, similar to Windows DLL rebasing. When a module is loaded, the TraceBack runtime library is notified. Every module is compiled with a default DAG ID range. The runtime checks to see whether if the default range conflicts with any existing module. If there is a conflict, the runtime uses an instrumentation-produced fixup table within the module to rewrite all DAG ID references in the module, so the inlined probe instructions end up using a distinct range of Ids.

The DAG ID space is finite—TraceBack uses 21 bits for the DAG ID field—and in a heavily loaded process it is possible (though unlikely) that the runtime could run out of available DAG Ids. The runtime reserves one DAG ID value as a “bad DAG id.” If the runtime is unable to find a distinct ID range for a module, it rewrites all DAGs within the module to use this Id. In this case TraceBack does not recover a trace within the bad DAG ID module, but the module will continue to execute properly, and TraceBack can recover traces from other modules.

To support DAG ID collision detection, the runtime must maintain a list of loaded modules. Many architectures support unloading and reloading of a module into a process, and in a long running server a module might be loaded and unloaded repeatedly. To keep from leaking DAG ID space, the runtime tries to assign the same DAG ID range to the module each time it is loaded. When instrumenting a module, TraceBack computes an MD5 checksum of most of it (omitting timestamps and other data that can change easily). The runtime uses the checksum as a key for information related to the module. TraceBack also stores the key value in the mapfile (which was generated at instrumentation time) so that it can properly match up mapfile and trace data during trace reconstruction.

To avoid the module load-time penalty of DAG rebasing, TraceBack allows the user to supply a “DAG base” file that automatically assigns DAG ranges to different modules instrumented from the same source tree. These ranges are used every time the module is rebuilt.

2.4 Exception Records

On exceptions, the runtime places additional information in the trace buffer so that the reconstruction phase can determine how much of the current block actually executed before the exception (exceptions include signals on UNIX). For native code instrumentation, TraceBack needs to capture only the exception address. If the exception is in an uninstrumented callee, or for architectures where instrumentation is done in intermediate code (like Java), TraceBack must sacrifice some performance to get accurate information about exceptions.

If an exception occurs in an uninstrumented callee, the trace should stop at the call which causes the exception. However, determining which function call that might be within straight-line code requires somehow recovering the return address to the instrumented section of code. TraceBack could recover the return address by walking the stack, but the return address might not be in memory (it could be in a register), and reading stack memory after an exception can cause additional problems. Recovering the return point after an exception has happened is difficult, so TraceBack inserts heavyweight probes at the return point of all calls, so even if an exception occurs in uninstrumented code, the

trace accurately depicts which instrumented function call eventually lead to the exception.

For architectures where instrumentation is done in intermediate code there is no easy way to relate the exception address (which is typically a JIT artifact) back to the intermediate code location. For instance, Java’s exception context does not directly indicate the bytecode offset of the faulting instruction. If a single block contains multiple exception-causing bytecodes, TraceBack is not able to say which bytecode actually causes the exception. But users want the exact line number of an exception. In order to compensate for Java’s shortcoming, TraceBack inserts lightweight probes at each source line boundary. These additional probes allow TraceBack’s exception reporting to select the correct line number. Since debug information and the TraceBack GUI operate in source lines, source line accuracy is all that is needed. The exact faulting block is not necessary.

TraceBack will automatically trace any exception unwinding process that has an impact on instrumented code. Each `catch` or `finally` clause is treated just like another procedure entry point, and initiates a DAG header.

2.5 Multiple Threads

TraceBack supports multiple threads of control within a process. For tracing it is desirable to keep track of each thread’s execution separately, for two reasons: first, it avoids the need to synchronize access to the trace buffer (which slows execution and imposes artificial execution constraints on the threads); and second, it provides a per-thread trace.

Most systems with threads also support thread-local storage. TraceBack uses thread-local storage to keep a per-thread buffer pointer, and the runtime tries to assign each thread its own trace buffer. Unfortunately, access to thread local storage is typically fairly slow, and usually requires a library call or equivalent. Since TraceBack needs to access the buffer pointer in the lightweight probe, a library call is out of the question.

For Windows NT, we take advantage of the fact that the first 64 thread-local storage (TLS) indexes can be accessed directly from the thread information block (TIB), which is accessed via the FS segment register. Instrumentation assumes that the runtime will be able to reserve TLS index 60 and all probes are set up to directly access this slot, which is at FS:0xF00. If this TLS index is not available, the runtime rewrites all the TLS indices in the inline probes using a fixup table, in a fashion similar to the DAG rebasing. TraceBack uses a similar technique on Linux, reserving an unused word at the start of the per-thread area referenced by the GS segment register. On Solaris, TraceBack uses a direct memory access to the normal libthread TLS area.

3 Runtime

The instrumentation introduced by TraceBack’s binary rewriting relies on an external runtime library to provide key services, such as trace buffers. Section 3.1 explains the memory allocation issues with providing threads memory buffers to hold their trace records, and how multiple threads are handled efficiently. Section 3.2 explains runtime support for reconstructing traces from threads that terminate abruptly. Section 3.3 describes the two approaches that support multiple source languages in a single process. Section 3.4 shows how TraceBack deals with the limited dynamic code generation done for many web servers. Section 3.5 discusses how the runtime puts records in the trace that the reconstruction layer uses to order events. Section 3.6 explains the runtime support for snapshots

(snaps) of the current execution. Finally, Section 3.7 discusses how the runtime interacts with the underlying operating system.

3.1 Trace Buffers

At startup, the runtime obtains configuration information that specifies how much memory it should allocate for trace buffers, and how many buffers to create. It allocates the desired memory and initializes the buffer structures. All of the main buffers are the same size. Each buffer is managed as a wrap-around, first-in first-out queue of trace records. At the physical end of each buffer, the runtime writes a sentinel record. DAG header probes check for the sentinel value, and if they load it, they call `buffer_wrap`, a function in the runtime, allowing the runtime to periodically gain control of each instrumented thread.

In addition to the main buffers, the runtime creates three special buffers, called the *static*, *probation* and *desperation* buffers. The static buffer is a small statically allocated buffer within the runtime image that the runtime can assign to threads if dynamic allocation requests fail. The runtime dynamically allocates the probation and desperation buffers.

The probation buffer handles threads that are created, but which never run, or which run non-instrumented code. TraceBack initially assigns all threads to the probation buffer, which contains only the sentinel word – thus each thread immediately triggers a `buffer_wrap` the first time it hits instrumented code. Only threads that execute instrumented code are allocated to real buffers, limiting the memory use of the runtime.

If the runtime ever runs out of main trace buffers and cannot allocate more, subsequent threads coming off probation enter a shared desperation buffer. Since many threads may write trace records in an unsynchronized fashion into the desperation buffer, the trace data itself is not recoverable, but threads that are in the main buffers are traced normally. Threads in the desperation buffer periodically call `buffer_wrap` (each thread has its own buffer pointer), so threads can leave the desperation buffer when resources become available.

Buffers reside in memory mapped files, so they can be easily copied (by another process) if the program terminates or becomes unresponsive.

3.1.1 Buffer Assignment

When a thread’s buffer wraps, the runtime has an opportunity to assign it to another buffer. The runtime uses a simple first-come allocation scheme: as each probationary thread hits its first probe, it enters the runtime, which assigns it an unused main trace buffer.

3.1.2 Reusing Buffers

When a thread exits, the runtime writes a thread termination record into its buffer, and the buffer is freed for subsequent reassignment. The buffer maintains the buffer pointer and the trace records left by the old thread; these will gradually be overwritten by the next assigned thread provided that the thread is fairly active. However, it is common to see several threads’ entire lifetimes packed into one buffer.

The runtime may also periodically run a dead-thread scavenging pass to see if any threads have terminated without notifying the runtime (this can happen when threads terminate abruptly). If so, the runtime writes the appropriate thread termination record and frees the buffer for reassignment.

3.2 Sub-Buffering for Exceptions

Certain key pieces of state are lost when threads abruptly terminate; in particular, the current buffer pointer in thread-local storage. In general, the runtime cannot reliably locate the most recent trace record in a full buffer (and thus cannot recreate the flow of control before termination) unless it takes additional steps.

The runtime partitions each main trace buffer into a set of sub-buffers so the reconstruction phase can provide some trace information if a thread terminates abruptly. Each sub-buffer ends with a sentinel, and a count in the buffer header enables the runtime to distinguish a full `buffer_wrap` from a `sub_buffer_wrap`. At each `sub_buffer_wrap` the runtime “commits” the contents of the just-filled sub-buffer by writing the sub-buffer’s index into the overall trace header and zeroing the next sub-buffer so that the thread’s progress through the buffer can be distinguished by looking for the last non-zero entry.

Sub-buffering imposes a runtime penalty because of the more frequent callbacks to the runtime and the clearing of the next sub-buffer before returning back to the probe. Sub-buffering also requires space in the trace buffer for additional sentinels, leaving less space for execution history records.

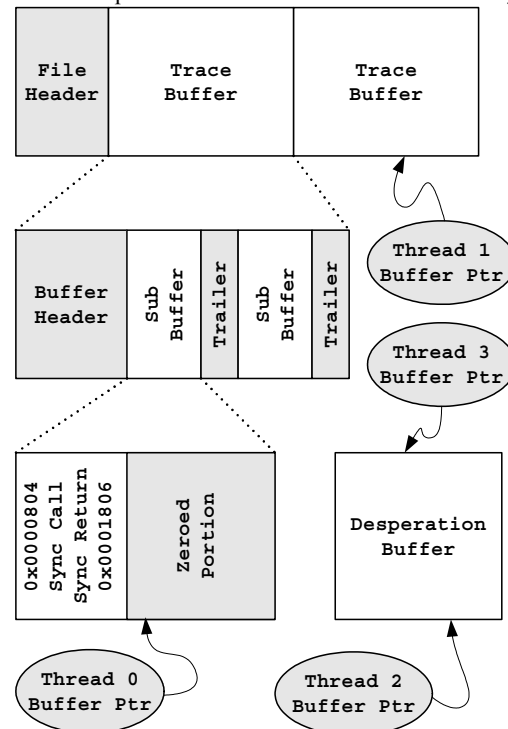


Figure 3: Runtime trace buffers and threads. This example shows the memory mapped file image with two trace buffers and four active threads. Each buffer is further partitioned into two sub buffers. Thread 0 is executing the program fragment from **Figure 2**. Threads 2 and 3 are currently writing their records into the desperation buffer since there are no free buffers.

3.3 Multiple Source Languages

A single process may host instrumented modules created from different language technologies. In particular, Java and native code, or Microsoft intermediate language (MSIL) and native

code, or even all three, can coexist in a process and interact. These mixed technology implementations often pose serious debugging challenges, since one debugger can't provide a simultaneous view of all the languages; having traces which show cross-language interactions provides a natural debugging environment.

TraceBack uses two different approaches to support cross-language tracing. For MSIL/native it takes advantage of the fact that there are efficient techniques for invoking native code snippets via `PInvoke`; that a single module structure houses both kinds of code; that exception handling ultimately relies upon a single runtime mechanism (SEH); and that it is possible to hook MSIL module loads. Within TraceBack the native and MSIL code share the same native runtime, so the integration is seamless.

For Java/native something different is required. The most natural solution might be to add instrumentation into the JIT, but the variety of JVM implementations and the lack of any standard interface for plugging into a JIT mean that any JIT-based approach is not portable. Furthermore, capturing control at exception points would require access to JVM internals. Instead, we treat the coexistence of instrumented Java and native code in one process as a simple form of distributed tracing (see Section 5). Thus the Java code and the native code write their records into different buffers.

3.4 Dynamically Generated Code

The runtime detects certain well-constrained types of dynamic code creation, such as ASP.NET .aspx pages and Java J2EE .jsp pages. To handle these cases, the runtime hooks into the web server environment and monitors for newly-created code. In the case of ASP.NET, for example, the runtime is notified about new DLLs and instruments them before use.

All newly-instrumented modules are stored in a TraceBack-specific on-disk cache. When a module is instrumented it is placed in the cache, and a reference to it is returned to the ASP.NET environment by the TraceBack runtime in lieu of the normal module. Subsequent ASP.NET processes benefit from the on-disk cache by avoiding the performance penalty of re-instrumenting the module. When a module is rebuilt due to changes in the .aspx source, the TraceBack runtime notices a modified MD5 module checksum and re-instruments the module.

3.5 Time and Ordering

In addition to the DAG records written by instrumented modules, the runtime can write timestamp event records into trace buffers to help reconstruction order traces produced by different threads. Trace records for a given thread are inherently ordered by their position within the buffer.

TraceBack makes use of the native high-performance real-time clock on platforms that support it; for example, the `RDTSC` instruction on x86 or the `gethrtime()` routine on Solaris. On other platforms TraceBack uses a simple “logical” clock, which increments on each important event (like a thread start/end, buffer wrap, exception, etc.). The logical clock synchronizes threads within a process effectively but does not support cross-process interleaving of execution.

TraceBack's instrumentation heuristically recognizes language artifacts that relate to synchronization or OS services and will automatically insert timestamp probes into binaries at such points. This allows TraceBack to reconstruct thread interleavings that are relatively correct – for any two trace records A and B in separate threads, TraceBack determines either that A

clearly happened before B, B clearly happened before A, or that there was no apparent constraint on the order of A and B.

3.6 Snaps

Traces of execution history are useless unless they are examined by a user. A TraceBack snapshot (or snap) is a collection of execution histories and metadata from which TraceBack reconstructs program state to display to a user. A user might want to “snap” a process if it were hung, or might want a snap in order to examine the system's state when a particular event occurs, like an `ArrayIndexOutOfBoundsException` in Java. TraceBack also supports a program API for snapping. Indeed, the main products of TraceBack from a user's perspective are the trace snaps. Users control when snaps will occur, and how much data they will contain.

TraceBack provides a variety of snap triggers, including program exceptions (language and low-level exceptions, UNIX signals), alerts from other runtime systems such as a memory fault detector, calls to a supplied “snap” API, and an external “snap” utility to deal with hung or unresponsive processes. Triggers are controlled by entries in a textual policy file that the runtime reads as it starts up in each instrumented process.

In order to provide a consistent version of thread histories during a snap, the TraceBack runtime suspends all threads during a snap, writing the trace data to disk while the threads are suspended. Suspending threads gives the user a globally consistent picture of the threads in their execution histories, and allows for a synchronized dump of memory including stack and heap data.

The snap file includes the raw trace buffers and their trace record contents as well as trace metadata describing details about the process – its name, the details about the host OS, the modules loaded into the process, the reason the trace file was generated, and so on. For instrumented modules, the metadata includes a copy of the module MD5 checksum, and the actual DAG ID ranges used by the module.

Snaps may also include a memory or object dump, so that TraceBack can display the values of variables or objects at the point of the snap.

3.6.1 Coordinating Related Processes

In practice, the user's concept of an “application” may include a group of related processes running on a machine, or across several machines. Sometimes a fault in one of these processes is actually the result of a failure in another of the related processes. Users configure process groups that are all snapped together if any one experiences a fault.

In order to implement group snaps, each machine hosting TraceBack-instrumented processes also runs a separate service process. The TraceBack runtime in each instrumented process communicates with the service process using a local protocol, notifying it of snaps, and potentially getting snap requests from the service process. Group snaps are not perfectly synchronized, but they're useful in practice, particularly for RPC-style interactions, where the calling thread in one process will suspend as the callee thread in another process executes the remote request.

3.6.2 Snap suppression

TraceBack aggressively suppresses snap triggers that appear to be redundant, such as the same exception coming from the same program location. This feature is under runtime policy

control, and is a key factor in producing a usable system. Useless snaps consume runtime resources to produce, disk space to store, and user attention to analyze.

3.7 OS/Runtime Interactions

The runtime typically requires some access to services of the host OS. At a minimum, the runtime must be able to write files or persist trace data in some other fashion. For maximum flexibility, the runtime needs fairly broad access to system services. But giving the runtime access to library and OS services is tricky, because:

- Runtime calls should not modify the visible state of a process. This typically means not sharing the C runtime library, or carefully saving and restoring shared state like the `errno` value.
- The runtime can be invoked in unpredictable contexts, especially at exception points. The thread that enters the runtime may hold locks or other critical resources, and a careless call into the OS might cause deadlock. The runtime itself also requires locking.
- The thread that enters the runtime might invoke operations that cause thread synchronizations that did not exist in the original program.
- The thread that enters the runtime might be operating with restricted privileges (say, from client impersonation).
- The runtime might inadvertently invoke instrumented code or cause an exception, which can lead to an infinite regress as the runtime continues to invoke the same code which takes the same exception.

To deal with these complications in full requires a fair amount of mechanism. To enter the runtime, a thread must save and restore any shared state, register itself as having entered the runtime (so that any exceptions caused by the thread can be suppressed), attempt to amplify its privilege level, and temporarily switch itself into the desperation buffer so that any trace records generated by instrumented code while in the runtime do not corrupt the user trace. Runtime entry is layered, so the runtime only performs the operations it must for a given function.

3.7.1 Gaining Control Initially

The runtime must set itself up before instrumented code runs because the instrumentation probes do not include initialization checks (they would degrade performance). The runtime's first task, therefore, is to load itself into a process and gain control before instrumented code executes. Typically, the runtime also needs control as each new thread enters the process for initialization.

For Windows executables, instrumentation places a special probe at the executable main entry point (for `exe` files) or in `DllMain` (for `dlls`), and an `import` is added that resolves to an entry point in the runtime library. The runtime library is then automatically loaded into the process when the instrumented module is loaded, and immediately gains control when the module is given control.

For Java, instrumentation places a special probe at the start of the static constructor for each class.

In Windows and Java the runtime supports late loading into a process; that is, the first instrumented module may appear in the process well after the process has started and created threads. To support delayed loading, the runtime must be able to perform thread discovery, enumerating the threads within the process.

3.7.2 Gaining Control at Exceptions

The runtime must gain control at the point of each exception, preferably before the process has had a chance to do any exception handling (first-chance). This gives the runtime the ability to inspect the process state directly at the point of failure, when the forensic evidence is most complete.

On Windows the runtime intercepts control by rewriting the code that is invoked when an exception is dispatched from the kernel back into the user process. The runtime routine thus has access to the exception context.

In Java there is no portable way for the runtime to gain control when an exception is thrown. Directly modifying the `Throwable` class, perhaps via the `bootclasspath` setting, could work, but this would violate the Java usage license agreement in a commercial product. As a fallback, instrumentation inserts an outermost `try/catch` block into every method, and places a call into the runtime at the start of these new and any pre-existing `catch` blocks. The call passes the exception object to the runtime as an argument. This arrangement allows the runtime to gain control at the time the exception propagates to the first instrumented method and to use the exception object to perform policy checking (and possibly `snap`, if policy dictates). The runtime then returns control to the method. In inserted `catch` blocks the exception object is then immediately rethrown to allow exception propagation to continue as it would have without any instrumentation. The runtime relies on suppression (see section 3.5.2) to avoid snapping repeatedly on the same exception as it propagates down the stack.

3.7.3 Gaining Control at Signals

On Unix the runtime must intercept signals appropriately. `TraceBack` provides a complete re-implementation of the signal API functions in the runtime, interposing on the real `libc` signal API functions. Application code can set handlers without perturbing the handlers installed by the `TraceBack` runtime. The `TraceBack` handlers are set up to handle a number of different scenarios. For fatal signals that are unhandled by the application (`SEGV`, `Control-C`, etc.), the runtime sets up a signal handler that shuts down the runtime, creates a `snap` if requested, and then re-issues the signal appropriately after uninstalling its handler. To re-issue the signal the runtime either returns from the handler to cause the fault to be reissued (for synchronous machine-check signals like `SEGV`) or raises the signal directly from within the handler (for signals like `Control-C`). For handled signals, the `TraceBack` runtime still needs to handle the signal itself, since it must write an *exception* record into the trace buffer (and, if it is specified by the user's policy, performs a `snap`). The exception record allows the reconstruction algorithm to cut the trace at the exact source line where the signal was delivered (see Section 4.2). The user's handler is then invoked, presumably writing more trace records into the buffer. If control returns to the `TraceBack` signal handler (instead of transferring control, for example, via `longjmp`) the runtime writes an "exception end" record to the buffer, allowing `TraceBack` reconstruction to determine the exact source line where control resumed after the exception.

3.7.4 Gaining Control at Termination

To gain control at normal thread and process termination, the runtime hooks the in-process exit points, like `ExitProcess` in Windows. To distinguish normal from abnormal termination `TraceBack` also hooks the so-called last-chance exception handler

in Windows, carefully preserving any previously installed handler.

3.7.5 The Event Thread

The runtime also creates an event thread. This thread remains entirely within the runtime, and has two main purposes: maintaining a heartbeat, and listening for communication from external processes (primarily snap requests). The service process periodically sends a STATUS message to the event thread of every active runtime. If the response times out the service concludes that the event thread (and thus the process) is hung, and can optionally snap or terminate the process.

4 Reconstruction

Trace reconstruction is the process of turning raw trace data into a line-by-line execution trace. Reconstruction requires (1) a trace file, (2) a set of mapfiles from the instrumentation process, one for each instrumented module, (3) (on some platforms) debug information to map from module-relative addresses to source locations, (4) (on some platforms) the instrumented binaries, and (5) source files. This section explains the stages of the reconstruction algorithm, and how the GUI organizes the resultant trace. An illustration of reconstruction is given in **Figure 4**.

4.1 Trace Record Recovery

TraceBack examines the trace file to verify its integrity. Sub-buffer boundaries (if they exist) are removed to produce a contiguous span of trace data. Each buffer is then mined back-to-front (newest record to oldest) to recover the trace records it contains. These record sequences are then split up by thread, if the buffer housed multiple threads.

Within each thread sequence, each DAG record is checked to determine what module it came from, by extracting the DAG ID and comparing it to the ranges in the trace metadata. When a DAG is resolved to a particular module, the reconstruction algorithm makes a note that the associated mapfile is required to further process the trace. The metadata may describe modules that do not appear in the trace, if all records for that thread have been overwritten, or if this thread executed only uninstrumented code.

4.2 Execution Path to Source Lines

The next processing pass converts the execution records into a trace of source lines. Conceptually, each DAG record is expanded into a sequence of block records. The recovery algorithm does this by expanding the lightweight probe bits to determine the path through each DAG. Then the algorithm uses the DAG to block mapping data found in the mapfile to get the block trace.

The reconstruction algorithm next expands each block into the source lines that the block covers. If the block is followed by an exception record giving an address within the block, the exception address is used to trim back the set of lines. The exception address may fall outside of the block if the block ends in a call, and the exception address is within an uninstrumented callee.

Reconstruction next looks at adjacent lines in the trace to determine if consecutive entries represent repeated executions of a line or if they are simply redundant. Redundancies arise from constructs like an expression with multiple function calls— instrumentation will break this into several blocks, since callee lines may need to be interposed, but if the callee is not instrumented no interposition will take place, and the now-adjacent lines in the caller will be redundant.

4.3 Trace Display

With the completion of the above steps, trace reconstruction is complete – the trace displays a line-by-line history of the thread’s execution. Separate columns of the history can show the module, source file, and so on. However, this format is not always an ideal one for human comprehension. Users often have trouble keeping track of context in a flat list of lines.

To provide neighboring line context, the GUI displays the trace in a lower pane and the source file containing the “current line” of the trace in an upper pane. The two panes are synchronized so that a user can step forward (or backward) through the trace and see the source pane update.

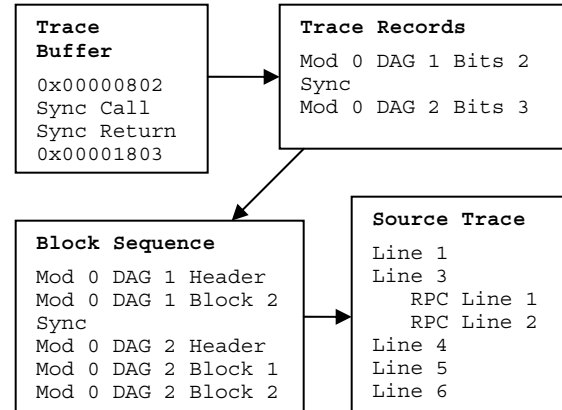


Figure 4: Trace Reconstruction. Execution of the program in Figure 2 produces the trace buffer contents shown at top left. TraceBack parses the trace buffer contents to produce a stream of records. DAG records are then turned into block sequences, and sync points are kept as annotations. Blocks are then expanded into lines, and sync records guide the interleaving of records from other trace buffers to give a complete history

4.3.1 Displaying the call hierarchy

To provide call context, blocks that contain procedure entry or exit points, or a call or a return point are annotated as such in the mapfile. Reconstruction uses these annotations to recreate the stack of activation records. The trace pane can then support a hierarchical display mode where spans of records corresponding to a callee can be collapsed or expanded as desired. The GUI uses this information to provide useful debugger-like stepping operations, like “step out” or “step over” as well as “step back out” and “step back over”.

4.3.2 Multi-threaded trace display

To provide a sense of what other threads were doing when the current thread was executing this line, trace reconstruction produces a “plausible” interleaving of trace records from different threads (recall that timestamp instrumentation provides partial ordering relationships). The GUI also supports a multi-trace display, each one focused on a given thread. Stepping through one thread will highlight the potentially concurrently executing lines of other thread traces.

4.3.3 Fault-directed view selection

The GUI also attempts to display the most relevant data using heuristics driven by the “reason” the trace was produced. If this trace was triggered by an exception on a particular thread, the

GUI sets itself up with a call tree view with all active procedures expanded, and the exception-causing line highlighted. If, on the other hand, the trace was a snap in response to a potential deadlock, the trace will show one line per thread, to aid the user in understanding what is blocking each thread’s execution.

5 Distributed Tracing

Distributed tracing stitches together trace data from separate runtimes into a single master trace. For multi-language systems, these runtimes may coexist within a single process, but in general they could come from separate processes or even separate machines.

5.1 Logical Threads

Our distributed trace model looks for RPC-style interactions across entities. Two physical threads that participate in an RPC call-enter-exit-return sequence are fused into a single logical thread for tracing purposes.

TraceBack relies on a variety of mechanisms to connect up the discrete parts of logical threads. Each runtime creates a unique ID for itself when it is initialized, using a standard generation technique. If the runtime is able to detect when an RPC call is taking place, it allocates a unique logical thread ID and binds it to the calling physical thread. Associated with this thread ID is a sequence number. The runtime then attempts to augment the RPC payload with a triple of (runtime ID, logical thread ID, sequence number). In some cases, like Java calling native via JNI, this information can be passed directly, out of band. In other cases, like COM, a payload extension can be used to attach the data to the marshaled call arguments. The logical thread ID and sequence number are also written (along with a timestamp) into the current physical thread’s trace buffer as a SYNC record.

On the receiving end, the callee runtime can access this extra payload. It first looks at the runtime id, to see if this data comes from a known runtime, and if not the runtime adds it to the runtime partner list for the callee. Next, the receiving thread is bound to the logical thread, then the the runtime increments the sequence number and adds a SYNC record to the thread’s trace buffer. The callee then proceeds normally. When it executes the RPC return, the runtime increments the sequence number again,

Test	Normal	TraceBack	Ratio
ammp	374.9	462.4	1.23
art	330.4	364.3	1.10
bzip2	198.4	340.8	1.72
crafty	101.7	180.5	1.77
eon	122.5	208.2	1.70
equake	105.5	118.3	1.12
gap	98.6	171.9	1.74
gcc	97.7	193.8	1.98
gzip	152.7	300.1	1.97
mcf	237.4	288.2	1.21
mesa	203.9	239.6	1.18
parser	201.3	369.9	1.84
perlbmk	158.0	395.7	2.50
vortex	155.0	330.4	2.13
vpr	224.4	332.0	1.48
Geo Mean			1.59

and puts another SYNC into the callee buffer. Return status can also be captured at this time (e.g. a COM HRESULT). The callee runtime ID, logical thread ID, and updated sequence number are sent back as extra payload, and the caller uses these to update its runtime partner table and places one final SYNC record into the trace buffer of the caller.

The net effect of an RPC call is four SYNC records with the same logical thread ID, successive sequence numbers and (assuming synchronized time sources) increasing timestamps, distributed in two separate trace buffers in two runtimes. The runtimes also have exchanged IDs. If the RPC callee itself makes RPC calls it will likewise pass the logical thread ID along, establishing a causality chain of physical thread trace segments.

5.2 Timestamp Correlation

The sequencing provided by the SYNC records helps trace reconstruction compensate for clock skew among runtimes. Without SYNC records, trace reconstruction can use the real-time timestamps to suggest causality between events occurring in different runtimes, but this only works well for clocks with minimal skew.

6 Experimental Results

The focus of TraceBack is to provide useful data about program crashes in multi-threaded, multi-language, and multi-computer environments. However, the performance of instrumented code must be good enough for users to run it in production. We estimate TraceBack development on x86 consumed 6 engineer-years, and about 20 engineer-years total for the functionality described in this paper on the major platforms.

For CPU-intensive programs like SPEC, TraceBack’s overhead is in the neighborhood of 60%. Table 1 shows measurements made on a 3GHz P4 system, with 2GB of RAM, with SPECint2000 benchmarks compiled with VC7.1, and run on the reference inputs. All programs were instrumented with VERITAS’ Application Saver 1.2.0.36, the incarnation of TraceBack current when these experiments were run. In these binaries, the text section grew by approximately 60%. Ratio is the ratio of TraceBack instrumented performance to Normal performance.

TraceBack’s performance overhead, while high for several programs, compares favorably to previous approaches that report small integer factor slowdowns [18] or more recently, 87% average slowdown [28]. TraceBack collects less information than whole program profiling (because it allows older records to be overwritten), and more complete information than interprocedural path profiling, but it is closer to capturing whole program paths.

Metric	Normal	TraceBack	Ratio
Response (ms)	347.7	364.8	1.049
ops/sec	60.3	57.5	1.049
Kbits/sec	345.3	328.7	1.051

Table 2 SPECweb99 performance for native code (Normal) and its instrumented version (TraceBack).

Table 2 measures performance for SPECweb99 on a 2.4 GHz Intel Pentium 4, with 256MB of RAM running Windows XP, Apache version 2.0.5, compiled with Visual Studio 6. The client

Table 1 SPECint2000 performance for native code (Normal) and its instrumented version (TraceBack).

is a 700MHz Pentium 3 system with 128MB of RAM running Windows XP. The two systems are connected with 100Mbps ethernet cards and a hub. The results are for the full SPECweb99 test with 21 connections, which was the maximum sustainable load for both the instrumented and uninstrumented server (server CPU was the performance bottleneck). All apache executables and dlls are instrumented.

The table shows that for both latency (average response time in milliseconds) and throughput metrics (operations per second, and kilobits transferred per second), instrumentation slows down the web server by 5%. This is similar to other web-based workloads we tested. We ran the Microsoft .NET PetShop version 3.2 on a Dell 600SC running Windows 2003 (2.4GHz P4, 512MB RAM), and used the Windows load generator from 3 machines to saturate the CPU. The baseline was 1,649 req/sec; with TraceBack it dropped to 1,633 req/sec, or a 1% throughput reduction.

Table 3 shows the performance of SPECJbb, a server-side Java benchmark, for normal and instrumented code. Instrumentation reduces throughput for this benchmark from 16%–25%.

System	Normal	TraceBack	Ratio
Win 1W	4,189	3,600	1.164
Win 5W	3,978	3,294	1.207
Lin 1W	4,128	3,376	1.223
Lin 5W	3,418	2,780	1.229
Sun 1W	3,238	2,612	1.240
Sun 5W	7,928	6,347	1.249

Table 3 Performance of SPECJbb. 1W means one warehouse, while 5W means 5 warehouses. The Win system ran Windows NT on an Intel P3 550MHz, 2GB RAM. The Lin system ran Linux RedHat 7 on an Intel P3 600MHz, 384MB RAM. The Sun system ran Solaris 9, on a 4-way UltraSPARC 450MHz, 1 GB of RAM. The **Normal** and **TraceBack** columns have throughput measures (number of completed transactions) for the uninstrumented and instrumented version of each benchmark.

These measurements are commensurate with the performance of TraceBack in the field. For example, TraceBack was deployed at Phase Forward (a data capture and management company) in an environment in which clients used web browsers to interact with a pharmaceutical trials application running on hundreds of centralized servers. TraceBack overhead was measured below 5%. These “real” applications have more system calls, more disk accesses, and they execute more code than standard benchmarks. All of these factors reduce the impact of instrumentation on performance. In SPECint, the high overhead from gzip is due to a very tight loop which contains a DAG header probe. The routine `longest_match` contains a DAG header, 2 lightweight probes and a register spill/restore which account for 30% of the total execution slowdown. Most commercial applications spread their execution history over a larger number of basic blocks.

6.1 Fault Diagnosis

The most important use of TraceBack is also the hardest to quantify, namely how it helps software developers understand and fix production bugs. Because a quantitative study would require access to bug fix information that companies do not generally make public, we offer several example uses of TraceBack for fault diagnoses.

Phase Forward used TraceBack to diagnose an intermittent hang in their production C++ application. The cross-machine traces demonstrated conclusively that the problem was in a third party database connection dynamically loaded library (dll). Phase Forward used this evidence to get a fix from the database company.

A Fidelity C++ application was not stable in production, and would only stay up for three to four hours at a time. TraceBack revealed that numerous calls to `memcpy` were overwriting allocated buffers and corrupting neighboring data structures. With the problem narrowed down, the developers were able to fix the worst corruptions giving them days of uptime.

A customer who declined to be identified had a C++ application that crashed after being instrumented. The trace and dump file revealed that the program was passing uninitialized data to a routine, and it gave them the file name and line number of the bug.

At Oracle, TraceBack revealed problems with a Java/C++ application. Application performance was slow because it was taking a large number of Java exceptions. TraceBack revealed that a call to `sleep` had been wrapped in a try/catch block. The argument to `sleep` was coming directly from a random number generator, which could return a negative number. When `sleep` was called with a negative argument, it threw an exception.

Finally, the TraceBack GUI itself (written in C++) is instrumented with TraceBack. While at eBay, one author (Ayers) was looking at traces in the GUI when it became unresponsive. Ayers took a snap, and sent the trace, in real time, to another author (Metcalf) who was back at corporate headquarters. Metcalf quickly determined that there was an $O(n^2)$ algorithm in the GUI which was making it unresponsive. Ayers told the engineers at eBay on the spot what the GUI bug was and how it would be fixed.

6.2 Cross-language and Distributed Traces

This section shows the output of TraceBack for a cross-language and a cross-machine trace that depict typical failure scenarios.

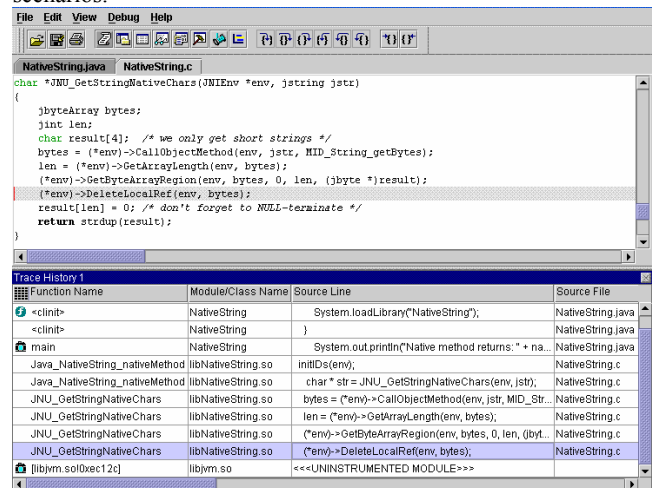


Figure 5 Cross-language trace, Java to C on Solaris.

Figure 5 shows a trace from a Java program that uses the Java native interface (JNI) to call C code. The Java code passes the C code a string, but unfortunately, the C code has only allocated 4 characters for the string (`result`). The comment betrays the

programmer’s bad thinking, “we only get short strings.” The Java code passes a long string, and the result is a stack corruption and a wild return which would prevent an accurate stack backtrace in a standard debugger. The TraceBack trace does show the flow of control from the Java program to the native C code. The figure depicts which source file each line is from (NativeString.java or NativeString.c). The figure also shows the debugger-like controls at the top with which the user navigates through the trace as with a standard debugger.

Figure 6 shows a trace across two machines in a C++ program that uses Microsoft’s distributed component object model (DCOM) to communicate. This example is a modification to the Labrador COM example from MSDN. The code makes a DCOM call to set the name of the pet, and then it retrieves the name. However, the programmer made `m_szPetName` a `const WCHAR*` instead of a `WCHAR[32]`, so the `wscpy()` into the string buffer fails with an access violation in the C runtime library code (`msvcr70d.dll`). The server process catches the exception and sends it back to the client where it is converted into an `RPC_E_SERVERFAULT` exception in the client (the kernel explicitly raises this exception via a call to `RaiseException`, so the client sees the source of the exception as `kernel32.dll`). The client code does not properly check the returned error code, so it continues on to call the `GetPetName` method, which succeeds, though the name the server returns is incorrect.

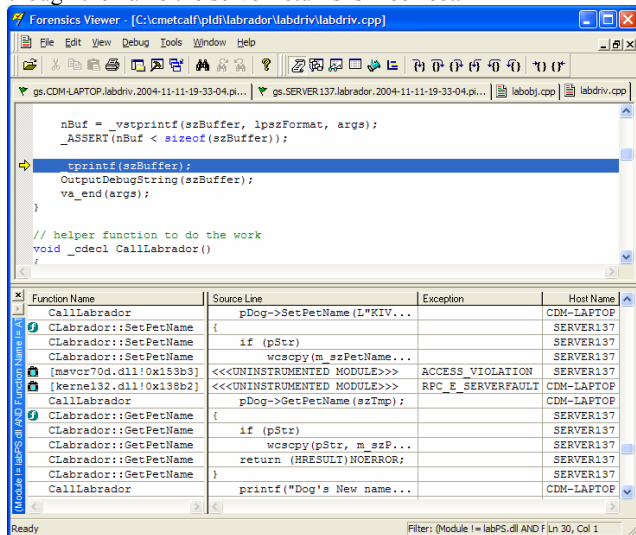


Figure 6 Cross-machine trace, C++ on Windows using DCOM.

7 Related Work

There has been a great deal of interest in systems for defect-finding and reliability such as TraceBack in both research and commercial spheres.

Recent work [5][15][12][21] on path profiling [4] has decreased execution time overheads while maintaining most path information. TraceBack collects a full program trace but retains only the most recent portion, so is more similar to interprocedural extensions to path profiling [28]. TraceBack breaks its path records at calls (see Section 2.4), reducing the effectiveness of many path profile optimizations. Path profiling’s goal is to efficiently aggregate path information (i.e., into a path frequency table), while TraceBack and whole program path profiling

maintain the temporal relationship between taken paths (i.e., an ordered path trace).

TraceBack does not use the Ball-Larus path profiling algorithm because it must provide the exact location of exception points even in the face of abrupt thread termination. TraceBack minimizes state kept in registers because it can be difficult to get register state information (including the PC) from a hung process, or one that terminated abruptly.

Recent work [19] has demonstrated statistical techniques for aggregating multiple correct and incorrect runs of an application in order to find program bugs. TraceBack does not use sampling, opting to reproduce control flow perfectly, and confine data sampling to snaps. Both systems require a programmer to examine the data returned by the system.

Virtutech recently announced support for Hindsight [29], an extension to their SimICS machine simulation environment that supports reverse execution in the debugger. Their support indicates the utility of reverse execution, though Hindsight is not appropriate for production software because machine simulators are not fast enough to run in production.

There is recent work on static analysis to find defects without needing to run the program. Examples include: Aiken’s work on alias analysis, applied to static data race detection [1], Dor’s CSSV to detect buffer overflows [11], Heine and Lam’s work on detecting memory leaks [14], and CCured for memory safety [10]. TraceBack uses static analysis to determine the control-flow of binaries to enable instrumentation, not to find defects directly.

Recent work in problem diagnosis has used machine learning techniques to find faults in distributed systems [6]. These systems attempt to isolate program components that are involved in failures, and so operate at a coarser granularity than TraceBack, which provides detailed control flow information within a binary.

TraceBack’s reconstruction of control flow across machines is unique among binary instrumentation systems, but the techniques for reconstructing control flow using RPC semantics across machines is similar to other work [2].

Enabling software to trace its own execution is an example of a systems capability that implements autonomic computing [16], where computers shoulder a larger burden of bug finding and bug fixing.

Previous systems used binary analysis and re-writing largely for performance-related purposes: ATOM [27] optimizes binaries and instruments them for performance analysis. Larus and Schnarr used EEL [17] to build profiling and tracing tools for SPARC/Solaris executables, primarily for performance and architectural research. Etch [20] has been used to build instrumentation and optimization tools for Windows/x86 binaries. Cifuentes and Gough [8] used binary analysis to reverse engineer legacy binaries for program understanding. VEST [25] and FX!32 [7] translate from VAX and x86 (IA32) code, respectively, to Alpha code. TraceBack uses binary instrumentation not for performance analysis or profiling, but rather to trace execution for debugging. Microsoft’s Phoenix project [20], which supports binary instrumentation of Windows executables, should make creation of TraceBack-like tools simpler and more reliable.

Systems have been developed that analyze and transform programs “on the fly,” as they run: Dynamo [3] rewrites instructions for faster execution. Shade [9] is an instruction set simulator and trace generation tool for performance analysis. Valgrind [23] uses dynamic binary translation to instrument application code. Embra [30] is a processor and memory system

simulator. TraceBack uses static binary instrumentation to avoid run-time cost.

Purify is a well-known product for detecting memory leaks based on binary modification. However, it can degrade performance 9 to 29 times according to a report published by Rational Software, the company that makes Purify [20]. Similarly, reverse execution debuggers, e.g., [11], are 4 to 7 times slower. Hence, Purify, gdb and other debuggers are ill-suited for production use.

Recent commercial systems have also arisen in similar areas, including: Identify Software's AppSight, and OC Systems' RootCause. Empirically, these systems appear to use heavier-weight instrumentation techniques, forcing users to selectively insert instrumentation to avoid excessive performance overhead. In contrast, TraceBack's low overhead enables it to run in production and to instrument entire executables without hints.

8 CONCLUSION

As performance of computer systems increase, users increasingly look to other features to distinguish systems. TraceBack trades some performance for ease of fault diagnosis. It is a very robust system, supporting multi-threading, multiple languages and distributed tracing. The overheads of TraceBack instrumentation vary greatly with the workload, from 60% average overhead for SPEC integer applications to 5% latency and throughput overhead for the Apache web server. There is ample anecdotal evidence that the TraceBack system has helped find production execution bugs, even when the bugs are irreproducible in a test environment.

Acknowledgements

We would like to thank everyone who helped program TraceBack into existence, especially Mike Decker, Xin Liu, Ian Schechter, Elliot Waingold, Tom Soranno, and Sanjeev Banerjia. Thanks to Kathryn McKinley, Michael Bond, and the anonymous reviewers for commenting on earlier drafts of this paper.

References

- [1] Aiken, A., Foster, J., Kodumal, J. and Terauchi, T. Checking and Inferring Local NonAliasing. In *PLDI*, 2003.
- [2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of SOSP*, 2003.
- [3] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A Transparent Runtime Optimization System. In *PLDI*, 2000.
- [4] Ball, T., and Larus, J.R. Efficient path profiling. In *Proceedings of MICRO-29*, (Paris, 1996).
- [5] Bond, M. D. and McKinley, K. S. Practical Path Profiling for Dynamic Optimizers. In *CGO*, 2005.
- [6] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic, Internet Services, in *Proc. Int. Conf. on Dependable Systems and Networks (IPDS Track)*, 2002.
- [7] Chernoff, A., and Hookway, R. DIGITAL FX!32: Running 32-Bit x86 Applications on Alpha NT. In *Proceedings of the USENIX Windows NT Workshop* (Seattle, WA, Aug. 1997).
- [8] Cifuentes, C. and Gough, K.J. Decompilation of Binary Programs. In *Software - Practice & Experience. Vol 25 (7)*, July 1995.
- [9] Cmelik, B., and Keppel, D. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *SIGMETRICS*, 1994.
- [10] Condit, J., Harren, M., McPeak, S., Necula, G. and Weimer, W. CCured in the Real World. In *PLDI*, 2003.
- [11] Cook, J. C. Reverse execution of Java Bytecode. In *The Computer Journal. Vol 45 (6)*, 2002.
- [12] Duesterwald, E. and Bala, V. Software Profiling for Hot Path Prediction: Less is More. In *ASPLOS 2000*.
- [13] Dor, N., Rodeh, M. and Sagiv, M. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In *PLDI*, 2003.
- [14] Heine, D., and Lam, M. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In *PLDI*, 2003.
- [15] Joshi, R., Bond, M. D., Zilles, C. Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems. In *CGO 2004*.
- [16] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–52, 2003.
- [17] Larus, J. and Schnarr, E. EEL: Machine-Independent Executable Editing. In *PLDI*. 1995.
- [18] Larus, J. Whole Program Paths. In *PLDI*. 1999.
- [19] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.
- [20] Microsoft Corporation. Phoenix compiler infrastructure. <http://research.microsoft.com/phoenix> April 2005.
- [21] Nandy, S., Xiaofeng, G., and Ferrante, J. FFP: Time-sensitive, Flow-specific Profiling at Runtime. In *Workshop on Languages and Compilers for Parallel Computing*, 2003.
- [22] National Software Testing Laboratories. NSTL Final Report for Rational Software: Performance test of Rational Software's software product Purify. October 1997. <http://www.rational.com/media/whitepapers/pnt-nstl.pdf>
- [23] Nethercote, N. Dynamic Binary Analysis and Instrumentation. *Ph.D. dissertation*, University of Cambridge, 2004.
- [24] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., and Chen, J. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop*, 1997.
- [25] Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P. and Robinson, S.G., Binary Translation. In *Communications of the ACM, Vol 36 (2)*, Feb. 1993.
- [26] Szymanski, T. G., Assembling code for machines with Span-dependent Instructions. In *Communications of the ACM. Vol 21 (4)*, April 1978.
- [27] Srivastava, A. and Eustace, A. ATOM: A system for building customized program analysis tools. In *PLDI*, 1994.
- [28] Tallam, S., Zhang, X., Gupta, R. Extending Path Profiling across Loop Backedges and Procedure Boundaries. In *CGO*, 2004.
- [29] Virtutech Corporation. Hindsight. <http://www.virtutech.com> March, 2005.
- [30] Witchel, E. and Rosenblum, M., Embra: Fast and Flexible Machine Simulation. In *SIGMETRICS*, 1996.