Efficient Collection And Storage Of Indexed Program Traces

Robert O'Callahan

April 23, 2007

1. Introduction

Debugging is fundamentally about tracing effects back to causes. However, traditional debuggers for software merely monitor a program's forward execution and are not well suited to the debugging task. There is increasing interest in direct support for "reverse-time debugging" using a combination of techniques such as logging, checkpointing, and replay [BCJ06, Lew03, Boo00, GHS06, KDC05] to give the programmer access to past program states.

Unfortunately there is no technology available today that provides a debugger with the following desirable properties:

- *Complete recording*. The debugger should capture enough information from one run of the program that every detail of the program state during the run can be reconstructed without having to run the program again. This hoists "run the program" out of the loop of the programmer's debugging activity. It also aids debugging of nondeterministic programs; once a bug has manifested once in the debugger, it can be diagnosed.
- *Efficient state reconstruction.* The debugger should be able to reconstruct program state at any time point with a cost proportional to the amount of state reconstructed but independent on the time point chosen. This can be a problem for replay-based techniques, which tend to have a cost proportional to the differences in time points of interest.
- *Efficient reverse dataflow.* A fundamental debugging step is "the value of X at time T is wrong; when and where was it set?" It should be possible to answer such questions efficiently, with a cost independent of the difference in time between T and when X was set.
- *Additional queries*. The debugger should be able to efficiently answer "when was the last execution of program point P (before time T)", and other similar queries.
- *Practical implementation.* An ideal debugger would be usable by as many programmers as possible. Therefore it should run on PC-class hardware, supporting standard operating systems, languages, and runtime systems. It should handle large complex programs, since that is where the hardest debugging problems occur.
- *Reasonable overhead.* A debugger with "complete recording" separates debugging into two phases: recording and diagnosis. When automated testing is available (as most development methodologies recommend) the recording phase can run unattended, and performance during diagnosis is much more important. Nevertheless the space and time requirements for recording must not exhaust the programmer's hardware or patience.

I have created a tool (provisionally called *Chronicle*) which meets these requirements. Chronicle uses Valgrind [NS03] to instrument a Linux process at the binary level and log every memory and register write and all control flow. The log is indexed to provide efficient support for the desired queries, and compressed to reduce storage requirements and especially to avoid the bottleneck of limited disk write bandwidth. During the diagnosis phase, a debugger passes queries to a query engine which answers them from the log data. Figure 1 illustrates the architecture.

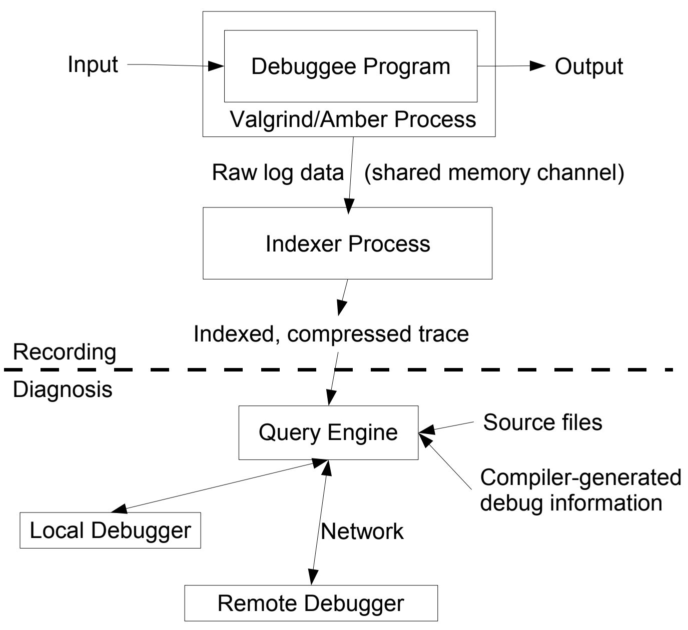


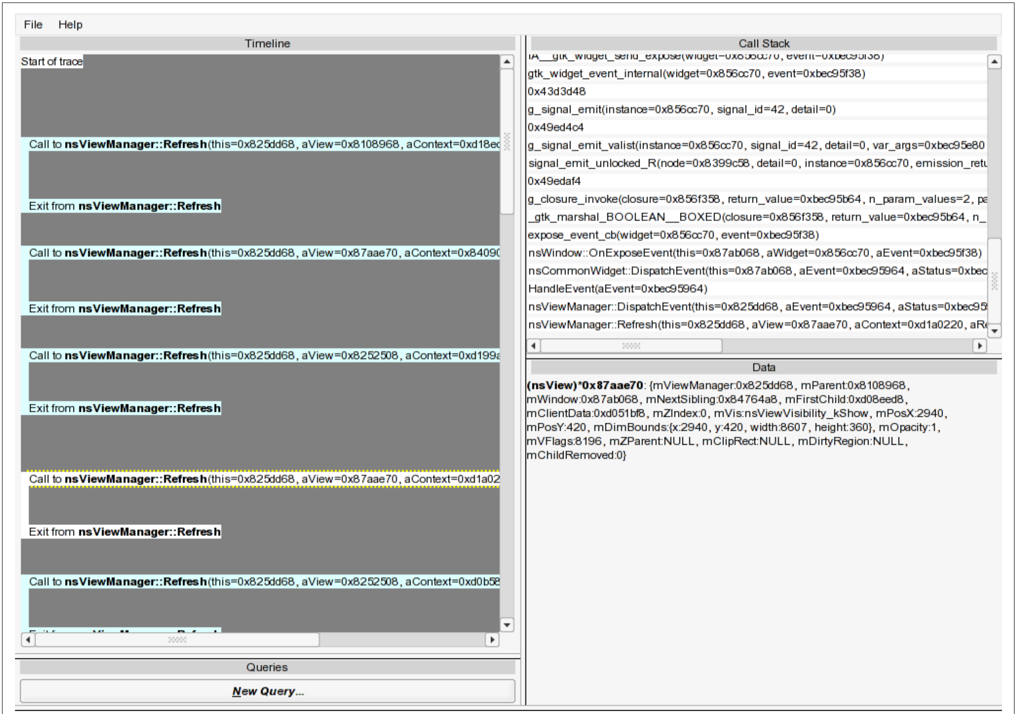
Figure 1: Chronicle Architecture

Chronicle exceeds almost all previous systems in the intensity of its logging for the sake of efficient program state reconstruction and query processing. The only comparable system is Omniscient Debugging, but that system is limited to smallish Java programs, whereas Chronicle was designed from the outset to support debugging of Firefox, which is written in a mixture of C, C++, Javascript and assembly and which executes more than three billion instructions to start up and display a simple Web page (in a debug build). This requires new techniques for gathering, indexing and compressing a torrent of machine-code level log data, taking advantage of multiple CPUs now that multiprocessor systems are mainstream. These techniques are Chronicle's primary contribution to the state of the art. A secondary contribution is a set of experimental results showing that such an aggressive tracing approach is practical on stock hardware, even with programs as large as Firefox, and therefore the time is ripe to adopt this approach to debugging. The rest of this paper describes the Chronicle design and implementation in detail, along with the experimental results. It also shows how a debugger based on complete recording can solve basic tasks, such as call stack reconstruction, more simply and robustly than traditional debuggers.

Fully exploiting the power of Chronicle in a debugger user interface is challenging and beyond the scope of this paper. However, we present some screenshots of a prototype Chronicle-based debugger to illustrate what is possible and to motivate Chronicle's design. Figure 2 shows the main window divided into four panes. The primary pane is the *Timeline*. The Timeline shows events in the program execution history with time increasing from top to bottom. In this example, the programmer has run the program to completion while saving a complete record, then launched the debugger. The programmer then created a query to populate the Timeline with all invocations of the method nsViewManager::Refresh. Each invocation comprises a *Call* event and an *Exit* event; the interval between the events is displayed as a bar on the left. Calls are detected with a query to find all executions of the first instruction of nsViewManager::Refresh; this query returns a set of timestamps, each corresponding to a Call event. In each Call event we display parameter values at the time of the call, using register and memory values reconstructed by Chronicle for the associated timestamp. (In this case, the interpretation of register and memory values is specified by DWARF2 debug information produced by the gcc compiler and consumed by our prototype debugger.)

In Figure 2 the user has selected one particular invocation of nsViewManager::Refresh (shown in white in the Timeline). The debugger has computed the call stack for that timestamp. Each line in the

Call Stack pane displays one stack frame, along with the parameter values for the call that created the stack frame, **evaluated at the time of that call**. (Contrast with traditional debuggers, which attempt to display parameter values for all active stack frames, but only have access to the current contents of stack memory and therefore may display misleading values if the stack locations holding parameters have been modified after subroutine entry.)



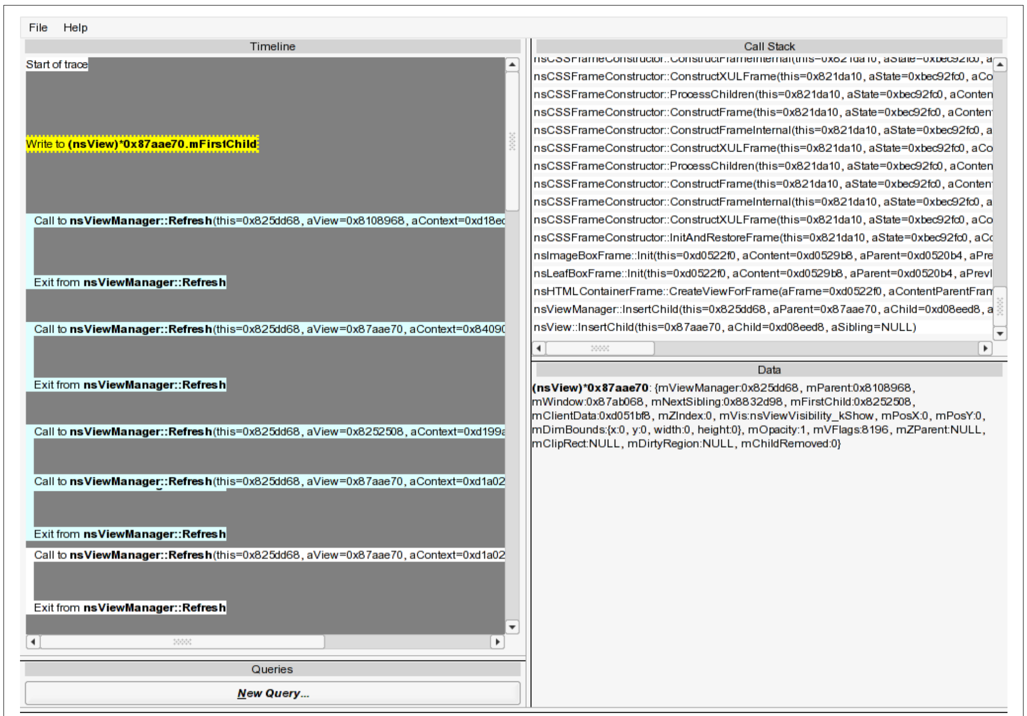
```
Start of trace
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x8108968, aContext=0xd18ec
Exit from nsViewManager::Refresh
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x87aae70, aContext=0x8409(
Exit from nsViewManager::Refresh
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x8252508, aContext=0xd199a
Exit from nsViewManager::Refresh
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x87aae70, aContext=0xd1a02
Exit from nsViewManager::Refresh
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x8252508, aContext=0xd0b58
```

```
IA__gtk_widget_send_expose(widget=0x856cc70, event=0xbec95f38)
gtk_widget_event_internal(widget=0x856cc70, event=0xbec95f38)
0x43d3d48
g_signal_emit(instance=0x856cc70, signal_id=42, detail=0)
0x49ed4c4
g_signal_emit_valist(instance=0x856cc70, signal_id=42, detail=0, var_args=0xbec95e80
signal_emit_unlocked_R(node=0x8399c58, detail=0, instance=0x856cc70, emission_retu
0x49edaf4
g_closure_invoke(closure=0x856f358, return_value=0xbec95b64, n_param_values=2, pa
_gtk_marshal_BOOLEAN__BOXED(closure=0x856f358, return_value=0xbec95b64, n_
expose_event_cb(widget=0x856cc70, event=0xbec95f38)
nsWindow::OnExposeEvent(this=0x87ab068, aWidget=0x856cc70, aEvent=0xbec95f38)
nsCommonWidget::DispatchEvent(this=0x87ab068, aEvent=0xbec95964, aStatus=0xbec
HandleEvent(aEvent=0xbec95964)
nsViewManager::DispatchEvent(this=0x825dd68, aEvent=0xbec95964, aStatus=0xbec95
nsViewManager::Refresh(this=0x825dd68, aView=0x87aae70, aContext=0xd1a0220, aRe
```

(nsView)*0x87aae70: {mViewManager:0x825dd68, mParent:0x8108968, mWindow:0x87ab068, mNextSibling:0x84764a8, mFirstChild:0xd08eed8, mClientData:0xd051bf8, mZIndex:0, mVis:nsViewVisibility_kShow, mPosX:2940, mPosY:420, mDimBounds:{x:2940, y:420, width:8607, height:360}, mOpacity:1, mVFlags:8196, mZParent:NULL, mClipRect:NULL, mDirtyRegion:NULL, mChildRemoved:0}

Figure 2: Debugger Window

The user has double-clicked on the "this" parameter to nsViewManager::Refresh to inspect that object in the *Data* pane. Chronicle has reconstructed the object's field values at the selected timestamp (the event bordered in yellow in the Timeline), and we display these values in the Data pane.



```
Start of trace
Write to (nsView)*0x87aae70.mFirstChild
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x8108968, aContext=0xd18ec
Exit from nsViewManager::Refresh
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x87aae70, aContext=0x8409(
Exit from nsViewManager::Refresh
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x8252508, aContext=0xd199a
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x87aae70, aContext=0xd1a02
Exit from nsViewManager::Refresh
Call to nsViewManager::Refresh(this=0x825dd68, aView=0x87aae70, aContext=0xd1a02
Exit from nsViewManager::Refresh
```

```
nsCSSFrameConstructor::ConstructFrameInternal(this=0x821da10, aState=0xbec92fc0, a
nsCSSFrameConstructor::ConstructXULFrame(this=0x821da10, aState=0xbec92fc0, aCo
nsCSSFrameConstructor::ProcessChildren(this=0x821da10, aState=0xbec92fc0, aConten
nsCSSFrameConstructor::ConstructFrame(this=0x821da10, aState=0xbec92fc0, aConten
nsCSSFrameConstructor::ConstructFrameInternal(this=0x821da10, aState=0xbec92fc0, a
nsCSSFrameConstructor::ConstructXULFrame(this=0x821da10, aState=0xbec92fc0, aCo
nsCSSFrameConstructor::ProcessChildren(this=0x821da10, aState=0xbec92fc0, aConten
nsCSSFrameConstructor::ConstructFrame(this=0x821da10, aState=0xbec92fc0, aConten
nsCSSFrameConstructor::ConstructFrameInternal(this=0x821da10, aState=0xbec92fc0, a
nsCSSFrameConstructor::ConstructXULFrame(this=0x821da10, aState=0xbec92fc0, aCo
nsCSSFrameConstructor::InitAndRestoreFrame(this=0x821da10, aState=0xbec92fc0, aCo
nsImageBoxFrame::Init(this=0xd0522f0, aContent=0xd0529b8, aParent=0xd0520b4, aPre
nsLeafBoxFrame::Init(this=0xd0522f0, aContent=0xd0529b8, aParent=0xd0520b4, aPrevI
nsHTMLContainerFrame::CreateViewForFrame(aFrame=0xd0522f0, aContentParentFram
nsViewManager::InsertChild(this=0x825dd68, aParent=0x87aae70, aChild=0xd08eed8, a
nsView::InsertChild(this=0x87aae70, aChild=0xd08eed8, aSibling=NULL)
```

(nsView)*0x87aae70: {mViewManager:0x825dd68, mParent:0x8108968, mWindow:0x87ab068, mNextSibling:0x8832d98, mFirstChild:0x8252508, mClientData:0xd051bf8, mZIndex:0, mVis:nsViewVisibility_kShow, mPosX:0, mPosY:0, mDimBounds:{x:0, y:0, width:0, height:0}, mOpacity:1, mVFlags:8196, mZParent:NULL, mClipRect:NULL, mDirtyRegion:NULL, mChildRemoved:0}

Figure 3: Found Last Write

Next, suppose the value of field "mFirstChild" looks suspicious. The user control-clicks on the mFirstChild — a command to locate the previous write to that field. The debugger issues a single Chronicle query and receives the timestamp of that last write. This is inserted as a new event in the Timeline, as shown in Figure 3, and selected as the current timestamp. The Call Stack pane updates to show the call stack at the earlier time, and the Data pane updates to show the field values for the selected object at the time. If this write set an incorrect value we will quickly be able to determine why, moving further back in time as necessary.

2. Instrumentation

Valgrind is an open source dynamic binary instrumentation engine, similar to DynamoRio and PIN. It simulates a user-level Linux process at the machine-code level by translating extended basic blocks of instructions on-the-fly to an internal representation, and then recompiling the IR to native instructions for execution. Pluggable "tools" are permitted to modify the IR before recompilation, to introduce instrumentation. This IR rewriting approach can be cumbersome for sparse instrumentation but it is well suited to Chronicle's need for pervasive instrumentation.

Valgrind provides additional capabilities that are useful to Chronicle. It notifies tools of the memory read and write effects of all system calls (introduced to facilitate writing Purify-like memory checkers). Chronicle records the write effects of system calls in its log. Valgrind can also notify tools when memory changes due to subtle effects such as implicit growth of the stack virtual memory area, or the construction of a stack frame for signal delivery. Valgrind runs all program code on a single underlying thread and simulates operating system threads for multithreaded programs. This suits Chronicle, because single-threaded execution is much easier to record. (And as we shall see, Chronicle can exploit multiple CPUs in other ways.) Most importantly of all, Valgrind is robust at handling the intricacies of instruction sets and Linux kernel semantics.

Chronicle's instrumentation outputs a series of trace records to an indexing and compression engine running in another process, described below. We store a sequence of records in a shared memory buffer and the processes synchronize only when switching buffers. There are several types of records:

- *INIT*. This is always the first record.
- *SET_ADDR_MAP*. Emitted when the address space of the process changes, e.g. due to an `mmap` system call. Chronicle emits a series of these after INIT to record the initial address space configuration. The record describes the address space region and its new mapping (if any).
- *BULK_WRITE*. Emitted when a large amount of memory is written atomically, typically due an I/O system call, but also in conjunction with SET_ADDR_MAP when data has been mapped into the address space. We record the address range and data written. When read-only files are mapped, Chronicle skips issuing a BULK_WRITE, assuming that the file contents will be available to the eventual debugger. This avoids having to write the contents of all executables into the log.
- *SYSTEM_WRITE*. Emitted when a system call writes memory as a side effect. The addres range is recorded. One or more BULK_WRITE records follow with the actual data. (We separate out data into multiple BULK_WRITEs because our implementation limits the size of a record to the size of the shared memory channel buffer.)
- *RESET_STATE*. Periodically the instrumentation emits the contents of all registers in one of these records. Certain pseudo-registers are also included (see below). Chronicle also emits one of these records at startup, and whenever registers are changed by a thread context switch.
- *DEFINE_CODE*. After Chronicle has instrumented an extended basic block, it emits one of these records. The record contains a unique ID assigned to the block, and describes what is statically known about the memory and register effects of the block. See below for more details.

- *EXEC.* An instrumented extended basic block emits one of these records every time it is executed. The record includes the block ID, from which the indexer locates the block's static descriptor. Any dynamic information about the block's effects needed by the indexer (e.g., the address of a memory write, if not statically known) is included in the EXEC record. The exact format of each EXEC record is tailored to the block, but can be reconstructed given the block static descriptor.

Valgrind's extended basic blocks (EBBs) are single-entry, multiple-exit instruction sequences connected by fall-through or direct control transfers. Valgrind can select overlapping EBBs, but this is irrelevant to Chronicle. One of the key items of dynamic information in each EXEC record is the number of instructions retired, which can be less than the total number of instructions due to a unexpected conditional branch or an exception.

Valgrind does not allow instrumentation to take control or easily detect when an exception occurs, so it is impossible to set the instructions-retired counter when an exception causes early termination of an extended basic block. (Often an exception will be simply set up an activation record and transfer control to a signal handler registered by the debugee program.) Therefore an instrumented EBB allocates its EXEC record on entry, fills in the dynamic effect information as the associated instructions execute, and updates the retired-instructions counter after each instruction completes. For EBBs that can terminate early, the instrumentation zeroes out the EXEC record after allocation so that upon early termination, any unused log entries are zero; the indexer will eventually compress this data and garbage could hurt compressor performance.

We use a pair of shared memory buffers to communicate between the Valgrind instrumentation and the spawned indexer process. The instrumentation fills one shared memory buffer with records while the indexer processes records from the other buffer. The processes use a pipe to signal each other that a buffer has been filled or emptied. The indexer automatically processes the last incompletely filled buffer when it detects the instrumented process has terminated. (This is useful for debugging Chronicle because it means when an error in the instrumentation (or Valgrind itself) causes a crash, we still get a complete record right up to the point before the crash.)

In Chronicle, a timestamp is simply the count of instructions retired since the start of the program.

3. Register Effects

The Chronicle instrumentation engine extracts from the Valgrind IR a static list of all register modifications performed by each program instruction (including instructions such as system calls that may indirectly change many registers). Each modification is assigned one of four classes:

- *DYNREG_WRITE.* Low bits of some register are set to some value. Neither the register nor the value are statically known, so both are recorded by the instrumentation.
- *REG_WRITE.* Low bits of a statically known register are set to some value that will be recorded dynamically by the instrumentation.
- *REG_SETCONST.* Low bits of a statically known register are set to a statically known value that is expressible as a signed 16-bit value.
- *REG_ADDCONST.* The low bits of a statically known register are set to the low bits of some (possibly other) statically known register plus some statically known value expressible as a signed 8-bit value.

Each of these classes comes in flavours specifying the number of bits affected — 8, 16, 32, 64 or 128.

DYNREG_WRITE is the most general class but requires the most information to be recorded dynamically; we prefer classes that require less information to be recorded dynamically, especially REG_SETCONST and REG_ADDCONST, which require no information to be recorded dynamically. In our implementation DYNREG_WRITE is actually only required by x87 floating point operations — Valgrind treats the x87 floating point stack as a register array indexed by a hypothetical “FPTOP” register.

We could further reduce the amount of data that must be dynamically recorded by extending our set of modification classes, for example by adding REG_MULCONST to multiply a register by a constant or REG_ADD to add two registers. In the limit we could reexecute actual machine instructions (assuming they don't read memory or hidden processor state and have no side effects). This limited set of classes was chosen to be simple, easy to detect from the IR alone, trivial to serialize and deserialize, and still reduce dynamic register logging by a large factor.

The instrumentation engine transmits the static register modification list to the indexer in each DEFINE_CODE record. Then each time an EBB is executed, for each executed register modification, the instrumentation logs any necessary dynamic values (and possibly register numbers) and includes this log as part of the payload to the EXEC record.

With this static and dynamic information we can reconstruct the contents of all registers at any point by replaying register modifications, given the initial states of registers, which we know from RESET_STATE records. RESET_STATE records are emitted periodically to ensure that reconstruction costs are bounded. Along with the normal architected registers, RESET_STATE records also contain a “thread” pseudo-register recording the current kernel thread ID. Valgrind notifies the Chronicle tool whenever a thread switch occurs, and Chronicle immediately outputs a new RESET_STATE record with the new thread ID and register values.

4. Memory Effects

Chronicle records memory writes and instruction executions using a single unified mechanism: memory effect maps. A memory effect is simply an event that has an associated memory range and timestamp. The only difference between recording memory writes and instruction executions is that memory write events have associated data — the values written. Conceptually, all we need to do is send all these events to the indexer.

We observe that within a single EBB, we frequently see a set of memory effects whose ranges are statically known to partition some larger memory range. For example, the execution of a block of N sequential instructions induces N instruction execution effects whose ranges partition the memory range of the executed block. Byte writes to locations r3, r3-1, and r3-2 are three write effects whose ranges partition the range [r3-2, r3+1). We leverage this observation by grouping such related effects into *bunches*. A bunch is a compound memory effect that has an associated memory range and *base timestamp*. It also has a list of (timestamp-offset, length) pairs describing how the compound effect can be decomposed into a sequence (not necessarily in timestamp order) of basic memory effects that partition the range. The timestamp of each basic memory effect is obtained by adding its timestamp-offset to the base timestamp.

Our key optimization is to form bunches statically, when we instrument an EBB. For instruction execution this is trivial: each run of consecutive instructions is one bunch, and instruction addresses are statically known. For memory writes we form bunches using a simple heuristic as we scan the IR from beginning to end: if we can combine the current write into a bunch with the previous write (or bunch), then we do so. Write addresses are usually not statically known, but in many cases simple dataflow analysis can show that two symbolic address expressions are equal, letting us add a write to the beginning or end of the previous bunch. For example, if the current bunch partitions [r3-1, r3+1), then a write to [r3-2, r3-1) is easily seen to form a bunch partitioning the union [r3-2, r3+1).

In practice we impose some implementation constraints on bunches: we restrict bunches to at most 8 component effects, we restrict the length of each component effect to at most 15 bytes, and we restrict the timestamp-offset of each component effect to at most 15. This lets us encode the static decomposition of a bunch into components in just 8 bytes.

For simplicity even single memory effects are encoded as bunches. (As a special case, these singleton bunches can affect up to 255 bytes ... some x86 SSE2 instructions can write 16 bytes of memory, and some x86 instructions can be longer than 15 bytes!) We statically determine the overall length of the bunch and its decomposition into component effects. We also check whether the base address is statically known. If not, it must be emitted by the instrumentation dynamically when the first instruction in the bunch is executed. However, as a further optimization we detect when the dynamic base address is a known static offset from the dynamic base address of the previous bunch of the same type; if so, we can avoid logging the address of the current bunch. This is useful when we can't form a single bunch because there is a "hole" in the range of memory affected.

All static information about bunches is recorded in the DEFINE_CODE record for the EBB. Residual dynamic information is emitted in the EXEC record each time the EBB is executed. For memory writes, the written value is always emitted dynamically by the instrumentation around the write instruction.

Static bunch detection significantly reduces the number of memory events that must be handled by each stage of Chronicle, resulting in a major gain in efficiency.

5. Indexer

The Chronicle Valgrind tool spawns the indexing and compression engine as a separate process. This allows the indexer to use standard C libraries. (Valgrind tools cannot use standard libraries because they are instrumented.) It also limits the impact of Chronicle on the address space layout seen by the program being debugged. It allows the indexer to detect and recover from unexpected failure in the Valgrind process.

An architecture that put the indexing and compression engine into the Valgrind process and allowed direct calls from Chronicle instrumentation into the indexing and compression engine might be more efficient, but having indexing and compression in a separate thread or process allows that work to be performed by another CPU core. (Putting the indexer in a different thread of the same process would lose the advantages of separate processes for no gain.)

To maximise write bandwidth, the indexer always appends to the end of the database file, never seeking within it or writing to another file.

Memory map changes recorded by SET_ADDR_MAP records are accumulated into a single array along with their timestamps and written out in one chunk at the end of the run.

Beyond that, indexer outputs three main kinds of data to the database: the static information about EBBs as sent by the instrumentation, lists of EBBs executed plus dynamic register data (all compressed), and indexed and compressed memory effects. The volume of static EBB data is negligible compared to the rest of the data, so its storage format is not important as long as lookup by EBB ID is efficient. The next sections describe the storage of the dynamic data.

After formatting, dynamic log data is compressed using a compression engine, currently zlib [Zlib06]. The formatting breaks data into chunks that are compressed and written independently and asynchronously. In practice there are many chunks and compression is currently the performance bottleneck, so Chronicle uses all available CPU cores to compress multiple chunks simultaneously.

6. Indexed Execution History

The execution history is divided into *epochs*; each RESET_STATE event starts a new epoch. An array containing a list of all epochs is appended to the database; each array entry contains the timestamp of the epoch start, the thread ID of the epoch, a bitmask indicating which registers were written during the epoch, and the file offset of the compressed epoch details. The bitmask and thread ID allow reasonably efficient searches to determine which epochs belong to a given thread or which epoch contains a write to given register(s) by a given thread.

At the start of the details we record the contents of all registers at the start of the epoch, and the number of EBBs executed (or partially executed). Then follows an array with the number of instructions executed by each EBB (only one byte required per EBB execution), followed by another array with the ID of each executed EBB. (The arrays are separated to avoid wasted space due to alignment constraints, and also to make determining which EBB execution contains a given instruction timestamp as fast as possible.) After those arrays is the raw register log data for all EBB executions, as described in Section 3. We store out complete register log data even for for EBBs that exited early; unused log entries will have been zeroed out by the instrumentation as described above.

The key to achieving a high compression ratio is to ensure that repetitious code in the instrumented program produces repetitious input to the compressor. It is common for an instruction at address X that writes a register R to write the same value each time it is executed [LWS96]. However it is also common for the values written to R to form an arithmetic progression (e.g. as a counter increments or as a pointer strides through memory), and these values are not compressed well by a simple compressor such as `zlib`. Therefore we exploit our domain knowledge to improve the compression by preprocessing the data. When copying register log data for an EBB execution to the details buffer for compression, if this is not the first execution of the EBB in this epoch, we actually store the difference between the log data for this execution and the log data for the previous execution of the same EBB. To simplify the code and increase speed, we ignore the structure of the log data and simply perform word-by-word differencing of the memory blocks. With this optimization, repeated executions of instruction X that write the same values to register R produce zeroes as input to the compressor, and when the values form an arithmetic progression with stride C, the input to the compressor is very likely to be a string of Cs (interspersed with other values from other instructions).

Computing the value of any given register(s) at a given timestamp now requires locating the epoch containing the timestamp in the epoch array (binary search can be used), reading the compressed epoch data block from disk and decompressing it, initializing register values to the initial values and then replaying the register operations of the EBBs executed before the given timestamp. The instrumentation engine bounds the number of EBB executions between RESET_STATE events, so the cost of reconstructing registers is generally a single disk seek and some bounded amount of CPU time.

7. Indexed Memory Effects

Memory effects are stored per memory “page”. Chronicle's page size need not correspond to the system page size; currently we use 64K as the page size. For each memory page, we divide history into *page epochs*. We start a new page epoch when the current timestamp is at least 2^{32} instruction executions after the start of the current epoch, or when the number of effects in the current epoch exceeds some limit (currently 60000) chosen to make the compressed data for each page epoch be on the order of 100KB.

The database contains a page directory, organized as a hash table mapping page numbers (the page address divided by the page size) to page records. Each page record contains the file offsets of an uncompressed array of page epoch records and an uncompressed array of bitmap records.

Each bitmap record contains the offset and length of compressed data for N (currently 8) bitmaps, each showing which memory locations were affected during a particular page epoch. (For example, the instruction execution effects bitmap for a page epoch indicates which instructions were executed in the page during the epoch.) The bitmaps for each page epoch are compressed using run-length encoding before being aggregated and sent to the general compressor. (In practice the run-length encoding is extremely effective, which is why we aggregate the results before doing a relatively expensive compress-and-write.) The query engine searches these bitmaps to determine which page epochs affected particular memory locations, especially to determine which page epoch was the last epoch to affect a particular location before some designated timestamp. It can actually perform this analysis directly using the run-length encoded form of the bitmap.

A page epoch record contains the timestamps of the first and last effect in the epoch. It also contains the file offset and length of a compressed list of all effects in the epoch. Each effect in the list records the timestamp of the effect relative to the start of the epoch (stored in 32 bits; this can't overflow because we have restricted the epoch duration), the 16-bit address within the page of the affected memory, and the 16-bit length (with zero interpreted as 2^{16} bytes). The effects are actually bunches, so we also store the 8-byte bunch descriptor to indicate how the bunched effect breaks down into individual effects.

Again we apply the principle that repetitious program behaviour should produce repetitious input to the compressor. Loops that repeatedly write into a page with constant address stride are common, and timestamps of each write also often form an arithmetic progression (because the number of instructions executed per iteration is often constant). Therefore, for each effect we actually store the difference in timestamp and address between the effect and the previous effect.

It is often desirable to traverse the effects in reverse-time order, so we include in the page epoch record the timestamp and absolute address offset of the last effect. We can then reconstruct effects in reverse order by subtracting the differences instead of adding them.

For effect types that have associated data values (memory writes), the values for each effect are appended to the effect list before compression.

When the indexer receives an EXEC record from the instrumentation, it extracts the effects from the static EBB descriptor, merges them with the dynamic data to compute the final address of each effect, and appends each effect to the current page epoch for the affected memory page and effect type. An effect may span multiple pages (x86 permits unaligned accesses), so a slow path breaks up such effects into multiple effects (a nontrivial task when bunches must be decomposed). When an EBB exits early, we take another slow path that determines which effects actually happened and applies them (again, nontrivial when some of the effects in a bunch were not executed). Fortunately these slow paths are rarely required. Bunched effects almost always pass through the indexer without requiring decomposition, which is why they are a big performance win.

Memory writes not performed by program code, including user-space writes by the kernel and memory "writes" performed by address space changes, are received as BULK_WRITE records by the indexer, which turns them into one memory write effect to each affected page.

The indexer can consume a lot of memory as it stores partially-completed page epoch data for each active page. This overhead is roughly a constant times the debugee's memory usage; the constant can be tuned to trade off indexer memory usage against page epoch size (larger page epochs require more interim storage but compress better and reduce CPU overhead).

8. Query Engine

The query engine that reads Chronicle databases is simple. It accepts JSON-formatted queries over a socket and sends replies over the same socket. Queries are stateless and the engine never writes to the database, so it is easy to run queries in parallel. Almost the entire design of the engine is determined by the database structure described above. In fact, the hardest part of the engine was designing a query API to expose compiler-generated debug information to the debugger, and implementing that API for DWARF2. (Working with DWARF2 directly from a high-level language is undesirable since parsing debug information is often a performance problem in debugging sessions, and low-level memory manipulation is helpful. Also, if Chronicle is to be used for remote debugging, it will be helpful to avoid sending large program binaries over the connection.)

We have already described how to reconstruct register state for any time T. The memory write effect data lets us reconstruct memory contents for any time T. First the memory area required is divided into pages. For each page, we use the summary bitmaps to determine which page epochs up to time T wrote to the memory locations of interest. (For the page epoch containing T, the bitmap tells us which locations *may* have been written before T; for earlier page epochs, the bitmap tells us that locations were *definitely* written before T.) Then we fetch the effect list for each relevant page epoch and traverse it backwards to find the last write to each location of interest. (This step can be parallelized.) We also have to check the address space map events in case the last change to the contents of the memory was a memory map change. Note that this approach to memory reconstruction also reveals when the last write to each location occurred.

This approach may require a large number of page epochs to be considered, and a large number of effects to be scanned in each page epoch. We could avoid all this by saving the compressed contents of the page at the start of each page epoch, at the cost of a considerable increase in bandwidth. So far it appears this is not necessary; the debugger generally only reconstructs small amounts of data from any given page, or when it requires large amounts of data, the writes to store the data had temporal locality. More experience is required.

9. History-Based Stack Reconstruction

One of the most difficult issues in a traditional debugger is reconstructing the current call stack by inspecting the contents of stack memory. Compilers like to optimize the layout of stack frames, and often it is profitable to avoid constructing the standard linked list of frame pointers. Therefore for debugging optimized code, formats such as DWARF2 include very complex (indeed, Turing-complete) descriptions of how stack memory should be interpreted. Of course, for many bodies of optimized code, debug information is not available. Even if it is, in practice, this is an error-prone area of debugger implementation and some popular debuggers often display broken call stacks. Worse, some kinds of errors can corrupt stack memory or make it unintelligible by wiping key registers such as the program counter, stack pointer or frame pointer. Certain optimizations such as reusing stack frames for tail calls make true call stack reconstruction impossible.

Complete recording enables a different approach to call stack reconstruction, by inspecting the history of program execution and never examining the contents of stack memory. The basic idea is that the current call stack at time T consists of all procedure calls that have not yet returned; from the execution history we can find all procedure calls up to time T and determine which ones have returned.

We need to define what exactly comprises a “procedure call” and “return”. Our implementation defines a procedure call as any control transfer to an instruction which is labelled as a procedure by debug information or symbol tables, or any control transfer via an “call” instruction unless the target is statically known to be the next instruction. (A sequence “call LABEL; LABEL: pop reg” is often used to obtain the program counter.) A “call” instruction is defined as any instruction that stores the address of the next instruction into memory anywhere. These definitions are heuristics, but very effective. Note that indirect control transfers require a dynamic check to determine whether the target is a procedure.

Defining a “return” is more difficult. In our implementation, we say that a procedure activation whose first instruction is at T1 has returned before time T2 if, for some T, $T1 < T \leq T2$ and the value of the stack pointer at time T is greater than the value of the stack pointer at time T1 (assuming that the stack grows downwards, as it does on x86). That is, a procedure activation returns when the stack pointer first exceeds the value it had on procedure entry. This could happen because of a return instruction popping off the return address, or due to any other kind of stack unwinding such as C++ exceptions or C's longjmp.

We need an algorithm Entry(T) to determine for a given timestamp T, the timestamp of the first instruction of the activation current at T. This is the most recent procedure entry that has not already returned, and is undefined if there is no such procedure entry. Then we can apply Entry iteratively to reconstruct the call stack for a given time T: first find the call to the current procedure, say let T1 = Entry(T). Then the instruction at T1-1 was the call instruction, so find the entry to its procedure, letting T2 = Entry(T1-1). Repeat setting Ti = Entry(Ti-1-1) until Entry(Ti-1-1) is undefined.

Let SP(T) denote the value of the stack pointer at time T. To determine Entry(T), note that SP(Entry(T)) must be greater than or equal to SP(T). Thus we look from time T backwards for procedure calls where the after-call stack pointer is greater than or equal to SP(T). To do this efficiently, we reuse our memory effect machinery and introduce a kind of synthetic memory effect: "enter-SP". The Chronicle instrumentation detects procedure calls and records an enter-SP effect after each call, affecting one word at the after-call stack pointer address (where the return address is stored). Then we can efficiently scan the enter-SP page histories and summary bitmaps to find the last Te < T such that Te immediately follows a procedure call and SP(Te) ≥ SP(T). (For efficiency and to handle multiple thread stacks, we also bound SP(Te) above, to the top of the thread's stack, which can be computed from recorded address space maps.)

This Te may actually be the start of a procedure that terminated before time T. Therefore we compute SP' = max { SP(T') | Te < T' ≤ T }; if SP' ≤ SP(Te), then Te did not return before time T, and therefore Entry(T) = Te. Otherwise we need to look further backwards, so we set Te' to the last Te' < Te such that Te' immediately follows a procedure call and SP(Te') ≥ SP(Te). Then we set Te to Te' and repeat. (There is no point in considering SP(Te) > SP(Te') ≥ SP(T), because the Te' procedure call must have returned before Te and thus T.)

Computing SP' could be expensive. To bound that cost, we introduce another Chronicle extension to store in each register epoch an upper bound on the value taken by the stack pointer during the epoch. The instrumentation tries to compute for each EBB a static upper bound on the difference between the entering stack pointer value and the maximum stack pointer value taken during the EBB. The instrumentation at the head of each EBB adds this bound to the current stack pointer value and updates the epoch upper bound. When no reasonable static upper bound exists (e.g., because the stack pointer is read from memory during the EBB), instrumentation is injected to dynamically update the epoch upper bound based on the new stack pointer value. With this stack pointer upper bound for each epoch, computing SP' requires a detailed scan of at most two register epochs.

This approach to stack reconstruction is extremely robust. It can reconstruct the call stack for time T even when stack memory and all registers except the stack pointer were zeroed out before that time. Even if the stack pointer is invalid (e.g., outside the thread stack area), the debugger can search backwards for a time when the stack pointer was valid. It is also robust to compiler transformations; it requires no knowledge of stack frame layout, no knowledge of which instructions belong to which procedures (i.e., it is robust to arbitrary code placement optimizations), and it can reconstruct the actual call chain in the presence of tail call optimization.

10.Results

The design goal was to squeeze the entire database into less than one byte per instruction executed, for realistic applications. This goal has been achieved. With a Firefox debug build running on a 32-bit x86 laptop (single-core 1.5 GHz Pentium M, 1GB memory), a test run that starts up the browser, displays a Web page and shuts down executes more than 4.8 billion instructions and the resulting database size is 3870 MB. The number of bytes per instruction is about 0.838. The entire run takes about 20 minutes (about 300 times slower than a regular execution). This is slow, but it is easy to automate these runs so no human attention is required. It is a relatively short run, but very many interesting bugs can be reproduced in runs of this length. Recording is CPU-bound; the disk is mostly idle. Profiling suggests that almost all time is spent in the zlib compression code.

11.Related Work

Omniscient Debugging [Lew03] is the closest work to Chronicle in spirit, sharing the vision of debugging from trace data without reexecution, and using indexes to facilitate operations such as "find last write to this location". However the Omniscient Debugger is limited to Java programs and has not addressed the challenges of indexing, compressing and storing large volumes of trace data.

Nirvana [BCJ06] is the closest work to Chronicle in implementation. Nirvana is a Valgrind-like dynamic code rewriting framework, which has been used to construct a tool for complete program recording to support debugging with reverse execution via reexecution and replay. Nirvana records the results of memory read instructions to allow deterministic replay of multithreaded applications on multiprocessors. Register state is periodically checkpointed as in Chronicle, so that register values can be reconstructed with bounded overhead. Unlike Chronicle, memory writes are not separately indexed, so it appears that many operations (such as finding the last write to a location) will require possibly costly reexecution. Indeed it is unspecified how, and how effectively, the Nirvana debugger supports reconstruction of general memory contents. In the language of the Introduction, “efficient state reconstruction” and “efficient reverse dataflow” are not directly supported. On the other hand, Nirvana supports lower overhead and smaller traces than Chronicle in the recording phase (5-15x time overhead is reported), but debugging requires an additional, slower reexecution phase (for which a further factor of 3-4x slowdown is suggested), so in total the overhead is still high. Nirvana's trace data per instruction is 0.1-0.5 bits per instruction executed, which is a lot less than 6.5 bits per instruction with Chronicle.

TTVM [KDC05] is a virtual-machine level trace-and-replay framework with associated gdb extensions to help debug the replayed VM's kernel. By recording only the I/O and and asynchronous events affecting the VM, slowdown is reduced to less than a factor of two. The main reason TTVM's recording is much more efficient than Nirvana's is probably that TTVM does not support replay of parallel execution of the guest VM on multiple processor cores, and therefore does not need to log memory reads in any way. There are proposals for hardware extensions to support deterministic replay of parallel execution much more efficiently [XBH03].

12.Conclusions And Future Work

With careful design it is not hard to meet the goals set out in the introduction. Complete recording (with indexing) can be carried out on commodity desktop hardware with reasonable efficiency. The challenge now is to exploit the full power of the model in an actual debugger. Adding “time travel” features to a traditional debugger is relatively easy but focusing on a single point in time and applying traditional debugger algorithms and user interfaces will not reap the full benefits of the data available. History-based call stack reconstruction is just one example of this.

This approach does have some limitations. Valgrind serializes thread execution so certain kinds of concurrency problems (e.g., memory coherence issues) are not captureable by Chronicle. However, the Valgrind scheduler could be made very aggressive to reveal many kinds of concurrency bugs. Furthermore Chronicle only needs to reproduce a bug once, and then it becomes much easier to debug than using traditional methods.

Chronicle does not capture the state of the environment. For example, the programmer cannot observe the contents of a program's window during the debugging session. A natural extension would be to support recording of environment state (e.g., window snapshots) into the database.

Another natural extension would be to support dynamic attachment. A program could run normally for a while before we attach Valgrind/Chronicle and start running it with instrumentation. (Other Valgrind-like frameworks (e.g., PIN [LCM05] and Nirvana [BCJ06]) support this.) This would let Chronicle debug long-running programs as long as the root causes of bugs occur after attachment.

Currently, when a process forks we disable recording of the child process. It would be interesting to record multiple processes to separate databases, but with enough synchronization for a debugger to reconstruct multi-process interactions.

Another possible extension is to record memory and/or register reads as well as writes. This would let the programmer see where values are used, and allow reconstruction of a complete dataflow graph for the program. This would require a large increase in bandwidth, and it is not yet clear how useful it would be in a debugger. (Chronicle already implements this as an option.)

Much work could be done to improve the efficiency of the entire process, for example by optimizing the compressor or the instrumentation. This is probably less important than building a better debugger using the existing infrastructure. Another option would be to use a low-overhead recorder (e.g., TTVM [KDC05]) to record at nearly normal speed, and then replay that execution offline under Chronicle to build a rich trace database. This is probably the best overall approach.

Currently Chronicle is focused on debugging C and C++ programs, but the infrastructure is very general and any machine-code program can be recorded. Any language could be supported as long as we have a mapping from process state to language-level state. Even interpreted languages can be debugged, since we can reconstruct the state of the interpreter. Multi-language debugging could also work well, if we can design a practical user interface for it.

As we increase the power of the debugger, we will probably need to extend Chronicle in new directions. One obvious direction is to record allocation and deallocation as memory effects, so the debugger can know the allocation state of all memory at all times, and efficiently determine when a particular memory location was allocated or freed. C++ construction and destruction could be recorded the same way -- then we would know the type of many memory locations. This would assist regular debugging and make it even easier to find certain kinds of memory errors.

13.References

[BCJ06] *Framework for Instruction-level Tracing and Analysis of Program Executions*. Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinic, Darek Mihocka, Joe Chau. Proceedings of the Second International Conference on Virtual Execution Environments (VEE 2006). http://research.microsoft.com/manuvir/papers/instruction_level_tracing_VEE06.pdf

[Boo00] *Efficient Algorithms For Bidirectional Debugging*. Bob Boothe. Proceedings Of The ACM SIGPLAN 2000 Conference On Programming Language Design And Implementation (PLDI 2000).

[GHS06] *Time Machine Debugger For Embedded Systems*. Green Hills Software. http://www.ghs.com/products/timemachine.html

[KDC05] *Debugging Operating Systems With Time-Traveling Virtual Machines*. Samuel T. King, George W. Dunlap, and Peter M. Chen. Proceedings of USENIX 2005. http://www.eecs.umich.edu/virtual/papers/king05_1.pdf

[LCM05] *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation.* Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace ,Vijay Janapa Reddi, and Kim Hazelwood. Proceedings Of The ACM SIGPLAN 2005 Conference On Programming Language Design And Implementation (PLDI 2005).

[Lew03] *Debugging Backwards In Time*. Bil Lewis. Proceedings of the Fifth International Workshop On Automated And Algorithmic Debugging (AADEBUG 2003). http://www.lambdacs.com/debugger/AADEBUG_Mar_03.pdfSee also http://www.lambdacs.com/debugger/debugger.html.

[LWS96] *Value Locality And Load Value Prediction.* Mikko H. Lipasti, Christopher B. Wilkerson and John Paul Shen. ASPLOS VII. October 1996. pp. 138-147.

[NS03] *Valgrind: A Program Supervision Framework*. Nicholas Nethercote and Julian Seward. Electronic Notes in Theoretical Computer Science 89 (2003), no. 2. See also http://valgrind.org.

[XBH03] *A "Flight Data Recorder" For Enabling Full-System Multiprocessor Deterministic Replay*. Min Xu, Rastislav Bodik, and Mark D. Hill. International Symposium on Computer Architecture (ISCA '03) (2003), 122–35. http://www.cs.wisc.edu/multifacet/papers/isca03_flight_data_recorder.pdf.

[Zlib06] *Zlib*. http://zlib.net.