New Techniques for Replay Debugging

Daniel B. Price

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-98-05
May 1998

New Techniques for Replay Debugging

Daniel B. Price
Dept. of Computer Science
Brown University
dp@cs.brown.edu

May 1, 1998

Abstract

Many systems have implemented replay of parallel or sequential processes, but none have provided a robust set of debugger tools that take advantage of this power. This paper develops new ideas about how software instruction counters can be used by programmers to facilitate debugging applications under replay. This paper is the discussion of the author's project for Honors in the degree of Bachelor of Science under the direction of Professor Robert H.B. Netzer.

1 Introduction

Future generations of programmers will rely on deterministic execution replay to facilitate debugging applications. While work has been done towards implementing replay debugging efficiently and reliably, new tools and techniques are needed to provide the programmer with a compelling debugging experience. This paper attempts to address these concerns by discussing additions we have made to the RDB replay debugging environment to export powerful new tools to debuggers.

In traditional debugging, programmers repeatedly re-execute programs while using debugging tools such as breakpoints and watchpoints in order to detect at what point a program's execution deviates from the expected behavior. Often, this occurs several instructions before a crash occurs, but this is not always the case. Incorrect program code may have executed an hour or more before the symptoms of a bug become apparent. Post-mortem "core dumps" are helpful, but can only assist a programmer in developing hypotheses about why a crash occurred.

A replay architecture can address these concerns, and provide the programmer with powerful new ways to shorten the edit-compile-debug cycle. However, it is not immediately clear how to extend tools such as breakpoints and watchpoints to assist a programmer working in a replay environment. Clearly, navigating the replay landscape is a different experience for the programmer; while the traditional model only requires the programmer to think about the current state of the process, replay debugging allows developers to utilize all of the states of the replay process.

We seek to design debugging tools that allow programmers to answer queries such as "What was the state of the process at the instruction before this one?" and to perform actions such as "Mark this instance of execution of this instruction so I can return to it later." Additionally, these tools can perform analysis on the target in order to provide functionality such as "roll back to the last time this variable was modified." Such tools will undoubtedly reduce the time it takes to find complex bugs by allowing programmers to develop and test conjectures about why a program has crashed.

Without a frame of reference, time is not a useful debugging tool, and programmers cannot direct a debugger to "stop replaying at instant x." We observe that a software instruction counter (or *SIC*) holds the key to a new class of debugging tools that allow efficient and meaningful process navigation and powerful control over the replay environment. We demonstrate that generalizing SIC and PC breakpoints yields new tools for replay debugging, and explain how we have implemented these ideas in RDB. The techniques we present here presume only that a replay environment will provide a software instruction counter in some form, regardless of whether or not incremental replay or memory tracing is present.

Finally, we address what work remains to be done to transform our ideas into viable tools for programmers, and discuss how to build visualization tools for execution replay.

2 Software Instruction Counting

An instruction counter indexes the changes of state in a process at the level of individual instructions. Some architectures provide hardware instruction counters, clock cycle counters [8], or even configurable event counters [6], but their use is not yet widespread. In the simplest case, *software instruction counters* emulate hardware instruction counters, and are incremented after every instruction executed.

Software instruction counting has been implemented efficiently for a variety of architectures [1, 3, 4] as an alternative to hardware instruction counters. Because they are implemented in software, SIC implementations usually rely heavily on instrumenting the target executable either after or during compilation. Because instructions are added to the executable, runtime performance is impacted, but it has been shown experimentally that such instrumentation can be performed efficiently on both RISC architectures with less than 10% slowdown, and on CISC architectures with less than 20% overhead.

To provide efficient instruction counting, Mellor-Crummey and LeBlanc propose relaxing the instruction counter model by observing the following: Each instruction in a program is located at a distinct memory address, and thus any instruction is part of a sequence of instructions for which no particular PC value is executed twice. The only time an instruction at a particular address can be executed twice is when a control transfer instruction has altered the program counter. Incrementing the SIC every time a backwards branch or a function call is made guarantees that every PC value executed during one value of the SIC is unique. That is, there exists a unique name for every execution of every instruction in the program using a tuple (SIC, PC).

2.1 Extending the Terminology

This section defines some terms used in the rest of the paper, and provides some convenient new terminology with which to talk about software instruction counting and replay debugging. A SIC *quantum* s is the sequence of PC values executed during a particular value s of an instruction counter. Thus, in a system with a hardware instruction counter, each quantum contains exactly one instruction. However, on systems lacking such support, the model proposed by Mellor-Crummey and LeBlanc is commonly used, so that a quantum may contain many instructions. While adding some new complexity, this guarantees unique naming of each instruction execution instance using the (SIC, PC) tuple. Such a representation is referred to as a *control point*. A final SIC related term is the *era*, which is a sequence of contiguous quanta. While *interval* might be a more appropriate term, this term is already in common use to refer to a unit of execution replay.

Additionally, it is useful to separate the common notion of a "breakpoint" into two parts: the *event specification* and the *event action*. For example, a PC breakpoint in a debugger such as `dbx` has a PC value as its event specification, and "stop the target process" as its event action. The event action is the action the debugger takes when the specification is matched— in this example, the target process stops, and the debugger notifies the user, who can then interact with the debugger and the user program. In this light, the term "breakpoint" is no longer completely appropriate, since arbitrary work can be performed as part of an event action. The term *debugger event* refers to the combination of specification and action that form such a control structure.

3 The RDB Framework

This section presents the RDB framework, into which we have integrated our work, and then focuses on how an RDB replay process communicates with the debugger.

3.1 Overview of RDB

RDB [3] is a framework for incremental replay debugging. In an incremental replay system, tracing and replay are broken down into *intervals*. Trace data is kept in such a way that process state may be restored to the beginning of any interval. This means that it takes much less time to navigate in the process than in traditional replay systems, which must restart process execution from the beginning every time navigation is desired. We have extended the RDB framework in order to add traditional debugger features such as breakpoints, and to experiment with new tools to exploit debugging in a replay system.

There are two major components of the RDB architecture. The first is the RDB engine itself, and the second is an auxiliary library known as the *trace technique*. RDB implements an API for developing such techniques. Our work in this paper is primarily concerned with the RDB engine, but we have used the DFA2-SPL trace technique detailed in [3] as a testbed for our ideas.

RDB involves itself in the user program in three major phases. The first is at compile-time, when ALTR, the Assembly Language TRanslator, is run in order to instrument the user executable according to a series of rules provided by a technique-specific "hooks" file. While the instrumentation is specific to the technique being debugged, it typically implements a software instruction counter, and some kind of memory tracing facilities. This paper is primarily concerned with trace techniques that implement a software instruction counter.

The second major interaction is at trace-time, when RDB's shared libraries provide a harness for the instrumentation, and help to trace system calls and signals. In the DFA2-SPL trace technique considered here, tracing (and symmetrically, replay) is divided into *intervals*, which are typically one to sixty seconds in length. An important implementation detail of this trace technique is that only 16 bits of storage are available for the SIC. Thus, it is necessary to reset the SIC to zero at the start of each interval to prevent overflow. This introduces additional complications because the (SIC, PC) tuple no longer uniquely identifies each instance of execution of an instruction— it only provides unique naming within a particular interval.

Finally, during replay, RDB sets up a different harness which helps to replay the results of system calls, and allows debuggers random access to intervals. It is possible to move through the program's execution, skipping uninteresting intervals, and repeatedly re-executing those of interest. In both trace and replay modes, RDB exports a debugger API. This consists of a debugger library, `librdb_replay`, which cooperates with the target using a dedicated "agent" LWP (lightweight process). The agent can respond to debugger requests and perform complex actions on its behalf.

Trace technique libraries typically need storage for trace data and other miscellaneous state. The DFA2-SPL technique takes a somewhat radical approach by splitting the 32-bit address space in half, and mapping one word in the "lower half" of the address space to each word in the "upper half." However, not all of the addresses in the upper half of the address space are addressable in Solaris. The address range `0xefffffff` to `0xffffffff` is reserved by the operating system. This means that there is a corresponding "mirror" region which is left unused from `0x6fffffff` to `0x7fffffff`; the trace library can use this region as scratch space for storing debugger events, or other miscellaneous data.

3.2 Debugger-Target Interaction

Because RDB is embedded in the user process as a collection of shared objects, it does not naturally have a mechanism for communicating with a debugger process. To facilitate such debugging, RDB provides an *agent LWP* which can service requests made on a debugger's behalf. Figure 1 shows how the agent and debugger communicate via the Solaris procfs mechanism [5]. One important restriction on the agent is that it and the process *target lwp* (the lwp being debugged) run in a mutually exclusive fashion. The agent is used for a variety of reasons, but conceptually it performs a number of major tasks:

1. Update trace or replay runtime options in the target.
2. Help to restore a particular interval for replay.
3. Create, install, uninstall, destroy and manipulate debugger events like SIC and PC breakpoints.
4. Negotiate whether a debugger's request for the target to stop asynchronously is safe, and arrange to stop the process in a safe state (see section 5.4).
5. Examine, copy, and tweak the target LWP's structure as appropriate to hide the instrumentation library from the debugger.
6. Provide memory trace data or other data exported by the trace library.

The agent's interface is easy for individual trace techniques to extend, and this work makes heavy use of this facility. The agent can perform tasks that would otherwise be very costly or very difficult from an external debugger, since reading or writing memory inside the target requires

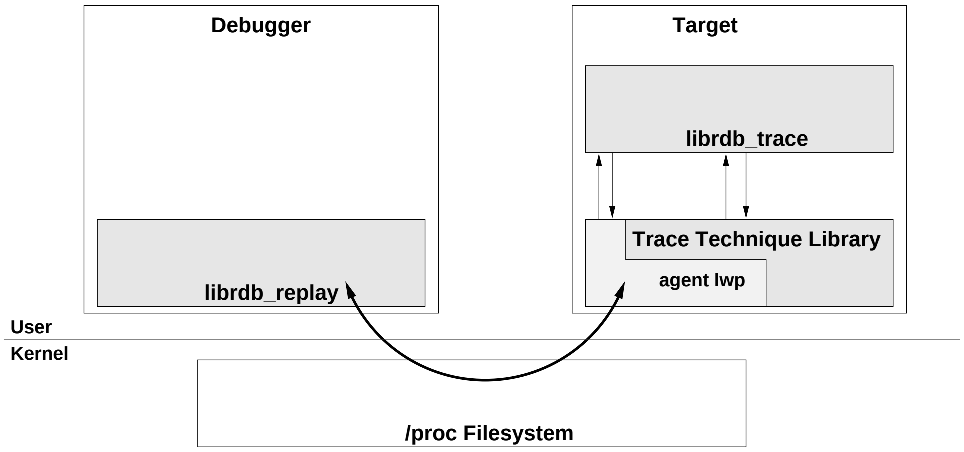

Figure 1: Debugger/Target Interaction

the operating system to context switch. To this end, the debugger library, `librdb_replay`, uses the agent extensively in order to create, install, and remove debugger events, which have storage allocated directly inside the the target process in a region of memory controlled by the trace library.

This approach has a number of advantages. One is that trace techniques like DFA2_SPL may need to use PC and SIC breakpoints in order to accomplish tracing. In this case, relying on the debugger to implement these would create a dependence on the debugger that applications compiled with RDB do not currently have. Placing responsibility for debugger events inside the target is thus the best solution, and it remains possible to replay a program from the command line without debugger intervention.

The last job of the agent is to cooperate with `librdb_replay` in order to help RDB remain invisible inside the target. This is crucial because debuggers typically have the ability to halt the target process at any time during its execution. Under Solaris, this is commonly achieved by communicating with the procfs mechanism; writing a `PCSTOP` message to the target will cause it to halt as soon as the operating system can arrange to do so. At this point, the user may direct the debugger to stack backtrace the target, investigate the state of memory, install breakpoints, etc. This presents a major problem for RDB, which may potentially have code running in the target at the time of such a request to stop. Section 5.4 presents our solution to this problem, which utilizes the agent LWP to cloak RDB's activity inside the target.

4 Using a SIC for Debugging

This section examines how tools that take advantage of a software instruction counter can be used by programmers debugging an application. The SIC provides a powerful facility for implementing debugger features efficiently, and reveals a new "taxonomy of debugger events." [2]

4.1 A Taxonomy of Debugger Events

In its most basic role, an instruction counter can be utilized to specify a simple instruction counter breakpoint. Such a breakpoint is specified only by the quantum q at which its action should be taken, regardless of PC value. Lack of hardware support for instruction counters makes such breakpoints harder to implement than PC breakpoints, but Elnozahy, Shapiro, and others have shown how to instrument the target in order to achieve this efficiently. Pseudocode for the instrumentation needed to accomplish this is shown in figure 2. A SIC breakpoint of this type allows rudimentary control over the target, by allowing the programmer to stop the process at the start of a particular quantum, perhaps just before an error condition or crash occurred.

```
...
  increment the SIC
  if (SIC = q)
    call SIC breakpoint handler
...
```

Figure 2: Implementation of a SIC Breakpoint

While figure 2 shows the implementation of what is commonly called a "SIC breakpoint," halting the target is not the only possibility— arbitrary action can be taken in the SIC breakpoint handler code. In fact, we have presented the implementation of a *quantum event*, which is an event whose action is executed at the start of a particular quantum. We can similarly show that special breakpoint instructions or traps used to implement a "PC breakpoint" can be broadened to implement the class of *PC events*, which take arbitrary action at a particular PC value.

	PC Specification	SIC Specification	Action
PC Event	p	$*$	PC Breakpoint Conditional Breakpoint
Quantum Event	$*$	s	SIC Breakpoint
Control Point Event	p	s	Bookmark Execution Location Redeliver Signal Signal end of interval
Era Event	p	$[s_{begin}, s_{end})$	Era Watchpoint Era Breakpoint
Periodic Event	$*$	$[s \times k, (s+1) \times k, ...)$	Update Visualizations Collect Profile info

Figure 3: A Partial Taxonomy of Debugger Events

Figure 3 proposes a partial taxonomy of control structures that include purely PC based events, purely SIC based events, and some events which merge these notions. In this table, a $*$ means that the indicated specifier is ignored, or always matched, and a variable indicates the value or values for which that specifier is matched. This taxonomy suggests that the traditional SIC and PC *breakpoint* mechanisms can be used to implement a much wider class of debugger events.

Combining the notion of a PC and a quantum event yields a *control point event*. This event receives its activation if and only if the PC *and* SIC of the current target match the control point specified by the event. Control point events are a special case of a wider class of events, called *era events*. These are events that occur over an era (a consecutive range of SIC values), when a particular PC value is matched.

Control point and era events can provide useful mechanisms to developers, who can use control points to uniquely specify a particular instance of execution of a particular instruction. This is helpful because it makes that instance easy to return to later, or to share with a colleague. More fundamentally, it captures perfectly the information needed to replay asynchronous events, such as signals or thread scheduling decisions. [4] Era events may be useful for installing watchpoints, breakpoints, or profiling mechanisms over a particular era instead of over the process lifetime, especially if such mechanisms incur a high performance penalty.

4.2 Building Debugger Events

This section shows that it is possible to build control point, era, and other complex events by composing PC and quantum events. The key idea is to use the event action of a quantum or PC event to install or remove *other* quantum or PC events. Previous systems have used this property in an ad-hoc way, either as a mechanism for controlling replay (i.e. redelivering signals or scheduler decisions), or as a means of rollback-recovery. We seek to generalize, extend and export this property to tools that programmers can use.

  `Create Quantum event` $qe(s_{begin})$

wait for activation of $qe(s_{begin})$
  `On activation of` $qe(s_{begin})$ {
    `Destroy` $qe(s_{begin})$
    `Create SIC event` $qe(s_{end})$
    `Create PC event` $pc(p)$
  }

wait for activation of $qe(s_{end})$ *or* $pc(p)$
  `On activation of` $qe(s_{end})$ {
    `Destroy` $qe(s_{end})$
    `Destroy` $pc(p)$
  }

  `On activation of` $pc(p)$ {
    `perform era event action`
  }

Figure 4: Implementing an era event using PC and quantum events

Creating era (and thus control point) events is somewhat more challenging. Although any quantum event (within the bounds of the SIC values used) and any PC event (that maps to a valid text address) is valid, not all control point or era events will necessarily ever occur, since it is possible to specify a (SIC, PC) pair that never occurs in the execution of the program. Figure 4 shows how to implement an era event for the event specification (s_{begin}, s_{end}, pc). It is easy to see from the definition of an era event that a control event can be implemented as a special case of the era event, by setting $s_{end} = s_{begin} + 1$.

Debugger events such as those described here become more flexible and powerful as the amount of available trace data increases. For example, tracing information about which function each quantum corresponds to would yield the ability to service very complex debugger requests such as "stop on modification of *var* in *func*." Additionally, we can see at this point how the debugger events presented can be used to implement the types of analysis tools that developers really need. Coupled with memory tracing information of the type DFA2_SPL implements, quantum events can help to implement debugger features such as rollback to the last modification of a particular address.

Simple SIC and PC breakpoints are the building blocks for a much wider class of *debugger events* exists. We have also shown that these are not of simply abstract interest— replay debugging systems should export such features to developers. The taxonomy of debugger events is a good starting place for such an effort. The following section shows how such events can be implemented in a replay debugging system and exported to a debugger.

5 Implementation in the RDB system

The ideas for different kinds of debugger events originally came from our desire to add PC and SIC breakpoints to the RDB framework. In this context we wanted control structures that supported traditional debugging, SIC based replay debugging, and structures that could control RDB's end-of-interval mechanism in a simple way. Because RDB is a framework for implementing trace-and-replay techniques, a chief design goal was to build an open interface for designing and implementing new kinds of debugger events. We wanted a mechanism sufficiently generic to build the five event types presented above, as well as *trace interval breakpoints* and *replay interval breakpoints* which are special cases of quantum events which control the end-of-interval detection during trace and replay.

As stated before, we view debugger events as composed of specification and action parts. RDB arranges that pending quantum and PC events cause the target to stop. Once the process stops on an event of interest, RDB needs to figure out which event (or events) have specifications which match the current state of the user program. For example, a user program could have two events installed: a PC event with PC=0x1234 and a quantum event with SIC=0x5678. If the target arrives at a control point (PC=0x1234, SIC=0x5678), both the PC event and the quantum event should have their event actions performed. To accomplish this in RDB, we create a pool of active events, each of which is modeled by an object. When the target process comes to a stop, a matching algorithm is run, which calls an "activate" method on each debugger event whose specification is matched. The following sections detail this implementation, and discuss related concerns.

5.1 RDB's Software-Event Mechanism

Previous sections of this paper referred to PC and quantum events; our mechanism for creating and managing various kinds of *debugger events* is called the *Software-Event* API.

RDB implements Software-events (*SEs*) as "objects" in the typical object-oriented sense. The Software-event structure, `rdb_se_t`, shown in figure 5, is loosely based on various UNIX System V interfaces, such as the vnode and device interfaces. [7] In this object-oriented approach, a structure containing event specific function pointers is associated with each Software-event, forming

```
typedef struct rdb_se {
  rdb_bp_t se_bp;              /* breakpoint storage */
  ulong_t se_sic;              /* sic value          */
  ulong_t se_sic_pair;         /* pair value (gets swapped w/se_sic) */

  ushort_t se_installed;       /* nonzero of se is currently installed */
  ushort_t se_state;           /* state of se when active */
  ushort_t se_match_mask;      /* which event specifiers are valid */
  const rdb_se_ops_t *se_ops; /* operations vector */

  ulong_t se_userstate;        /* user data fields       */
  void* se_userdata;           /* (ignored by se engine) */

  rdb_se_list_t *se_list;      /* list that se is currently on    */
  struct rdb_se *se_next;      /* for list of sicevts or freelist */
  struct rdb_se *se_prev;      /* for list of sicevts or freelist */
} rdb_se_t;
```

Figure 5: rdb_se_t structure

an *operations (ops) vector*. By implementing sets of these simple operations, it is easy to define new types of Software-event, and extend the trace technique's functionality. The ops vector contains six operations: `op_destroy`, `op_install`, `op_uninstall`, `op_stepover`, `op_activate`, and `op_postactivate`. By providing these six functions, along with a `create` function, developers can implement SEs for various trace techniques.

Software-events have a lifecycle which begins when a particular event is instantiated. Once created, an SE may be *installed*, at which point it can exercise control over the target. Eventually, a Software-event may be uninstalled by the trace technique (or at the request of the programmer employing the event), and finally destroyed. Most of an SE's interesting behavior occurs while it is installed.

An installed Software-event has a series of states which determines its behavior. An SE may be:

- *unmatched*, in which case it is currently active, and awaiting a match;
- *matched*, indicating that the event specification is matched by the current state of the process;
- *spent*, which means that the event is no longer active, typically because it is a one-time event whose quantum is past.

The sequence of states of installed SEs is well defined, and can be modeled with a simple state diagram, shown in figure 6.

Once the set of matched SEs has been processed by the trace technique library, each event's `post_activate` callback is performed. This allows the event to specify where it belongs— it can either become *spent*, in which case it is transferred to the spent SE queue, or request a 'requeue,' in which case its state is returned to *unmatched* and it is returned to the unmatched event queue. This is useful because some events have specifications which are met at most once in the course of an execution interval, while others may occur many times. For example, a traditional PC breakpoint

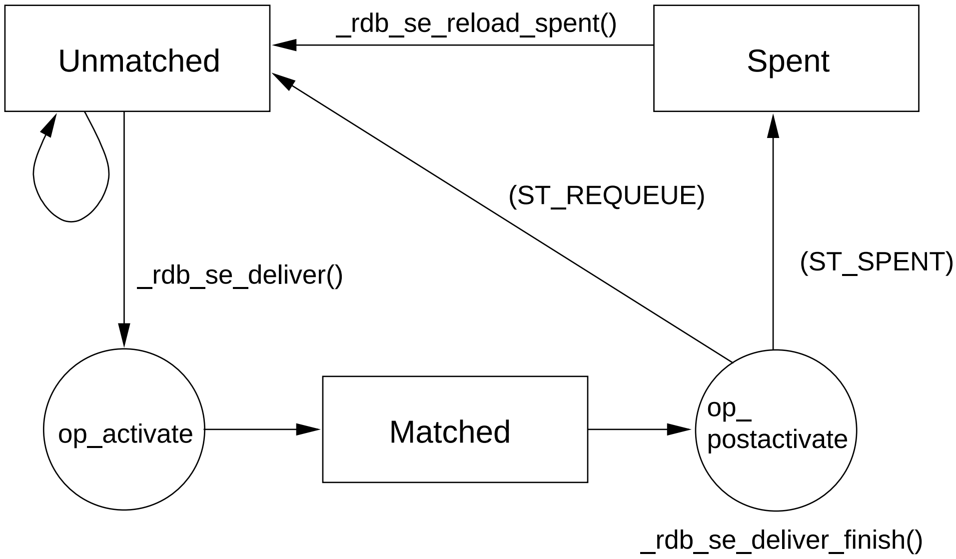

Figure 6: States of an installed Software-event

may be matched tens or thousands of times during an execution interval, while a SIC breakpoint will be activated only once. This mechanism of requeuing or marking "spent" allows these policy decisions to be made by the SE itself, simplifying the matching engine.

Because RDB supports incremental replay, the programmer may at any point choose to restore to the beginning of the current (or any other) execution interval. At this point, SEs which were previously marked *spent* are transferred back to the *inactive* state, where they are again candidates for matching and activation. This is accomplished with a call to `_rdb_se_reload_spent()`.

5.2 Matching and Activating Software-Events

The Software-event *engine* works behind the scenes to maintain queues and buckets of SEs. This section describes the data structures used to organize and manage SEs which have been installed in the user process. SEs whose specifications are partially or fully met by the current state of the process are said to be *matched*, and are moved to a special list that is presented to the trace technique which is the client of the interface.

In order to accomplish this matching, the trace technique invokes the `_rdb_se_deliver()` algorithm when the process stops on an event of interest. Matched events have been *activated* because the algorithm calls the `op_activate` method on each matched event. Again, this makes it easy to install events with widely ranging behaviors.

As shown in section 4.2, complex debugger events like era events may be built by composing quantum and PC events. The pseudocode presented there creates and destroys multiple different events in order to accomplish the functionality of an era event. RDB's Software-event mechanism takes a slightly different approach. The `rdb_se_t` structure has fields for storing PC and SIC information, and the `se_match_mask` field is used to notify the SE engine that particular fields are valid. When the target stops, the engine looks through its list of SEs and makes a determination about how those events match the current state of the process. This means that each SE is limited to having one PC event and one SIC event installed at any given time.

An immediate observation is that having one PC event and one SIC event available for each Software-event means that there are four basic cases:

1. Only the PC specifier is valid. The object is place in a bucket of PC events.
2. Only the SIC specifier is valid. The event is placed in a queue of SIC based events.
3. Both the PC and SIC specifiers are valid. The event is placed in a queue of SIC and PC based events.
4. Neither are valid.

The fourth case can safely be ignored, since an SE that has no valid specifiers can never become activated. Thus, the Software-event engine can handle each of the three important cases separately. This is useful because events with SIC specifiers (cases 2 and 3) *must* occur in sequence according to their event specifiers. These SEs are stored in two separate queues, ordered by SIC value.

```
_rdb_se_deliver(sic, pc) {
  /* (1): match items in the sic queue */
  while(sic_matches(head(sic queue), sic) {
    se ← remove_head(sic queue)
    if (pc_matches(se, pc) = MATCH)
      se→op_activate(se, pc, sic, MATCH_ALL)
    else /* pc_matches(se, pc) = NOMATCH or DONTCARE */
      se→op_activate(se, pc, sic, MATCH_SIC)
    list_append(se, match_list);
  }
  /* (2): match items in the pc bucket */
  foreach (se ∈ pc bucket) {
    if (pc_matches(se, pc) = MATCH) {
      if (sic_matches(se, sic) = MATCH)
        se→op_activate(se, pc, sic, MATCH_ALL)
      else /* NOMATCH or DONTCARE */
                 se→op_activate(se, pc, sic, MATCH_PC)
      list_append(se, match_list);
    }
  }
  ...  /* repeat (2), matching items in the sicpc queue */
}
```

Figure 7: The `_rdb_se_deliver()` matching algorithm

Figure 7 shows pseudocode for the `_rdb_se_deliver()` algorithm, which is in charge of the process of matching and activating Software-events. When the process stops, and `_rdb_se_deliver()` is called, the first element in the SIC queue can be examined to see if its SIC value matches the current SIC value. The PC bucket and SIC-PC queue must be completely scanned, however. As shown in figure 7, matched SEs have their `op_activate` functions called at the time they are considered to be matched, and an indicator of which criteria were successfully matched is passed to the SE subtype in the form of `MATCH_ALL`, `MATCH_PC` or `MATCH_SIC` constants.

5.3 Software-Event Implementation

To illustrate how the `_rdb_se_deliver()` algorithm interacts with SE subtypes, we present code for implementing a quantum event using our Software-event mechanism. Figure 8 shows the operations needed to implement a SIC breakpoint. Note that the SIC breakpoint does not directly cause a break to the debugger, since multiple Software-events might be matched when the target stops. Instead, the SE sets its `se_userstate` to indicate to the trace technique that it is requesting a breakpoint to occur. Once all of the matched SEs get exported to the trace technique library, it can iterate across them and decide what to do.

```
rdb_se_t* sic_se_create(rdb_heap_t *heap, dfa2se_sic_args_t *args) {
  rdb_se_t *se;
  se = _rdb_se_create(heap, &g_sic_se_ops);
  /* set up the sic specifier */
  se->se_sic = args->sesic_sic;
  return se;
}

int sic_se_op_activate(rdb_se_t *se, ucontext_t *ctx, ulong_t sic, ushort_t match)
  /* request that the debugger breakpoint */
  se->se_userstate = DFA2_SEST_BRKPT;
  return 0;
}

int sic_se_op_postactivate(rdb_se_t *se)
  /* clear debugger breakpoint request */
  se->se_userstate = DFA2_SEST_NONE;
  /* Quantum se's only get activated once per interval.
     We can thus mark this se 'spent' */
  return SE_ST_SPENT;
}

int sic_se_op_install(rdb_se_t *se, caddr_t pc, ulong_t sic)
  /* mark that we have a valid SIC specifier */
  se->se_match_mask = SE_MATCH_SIC;
  /* we may have already past the specifier for this se */
  if (sic < se->se_sic)
    return 0;
  else
    /* return -1, which will mark this sicevt as 'already spent' */
    return -1;
}

int sic_se_op_uninstall(rdb_se_t *se, caddr_t pc, ulong_t sic)
  se->se_match_mask = SE_MATCH_NONE;
  return 0;
}
```

Figure 8: Complete implementation of SIC breakpoint

The code also illustrates a further advantage of allowing Software-events to make policy decisions about their lifecycle. The `op_install` function demonstrates that successfully installing events may be conditional on factors like the current value of the SIC, and that placing responsibility for such decisions on the SE dramatically simplifies the `_rdb_se_deliver()` algorithm.

5.4 Handling Asynchronous Debugger Directives

One difficult technical problem that implementing the Software-event mechanism presented was handling process stops caused by the debugger in a clean manner. As stated in section 3.2, such stops may be directed by the debugger using the Solaris procfs mechanism, and the debugger will expect that the stopped process may be interacted with in typical ways, such as stack backtracing, installing debugger events, etc. This presents a problem if the user program is stopped by the debugger while it main LWP is busy carrying out some RDB-related activity, such as updating event related data structures, or performing interval-handling code. The problem is apparent from figure 9, which illustrates that the Software-event engine has data structures which may be accessed by fault handlers from user code and by the agent, causing potential race conditions.

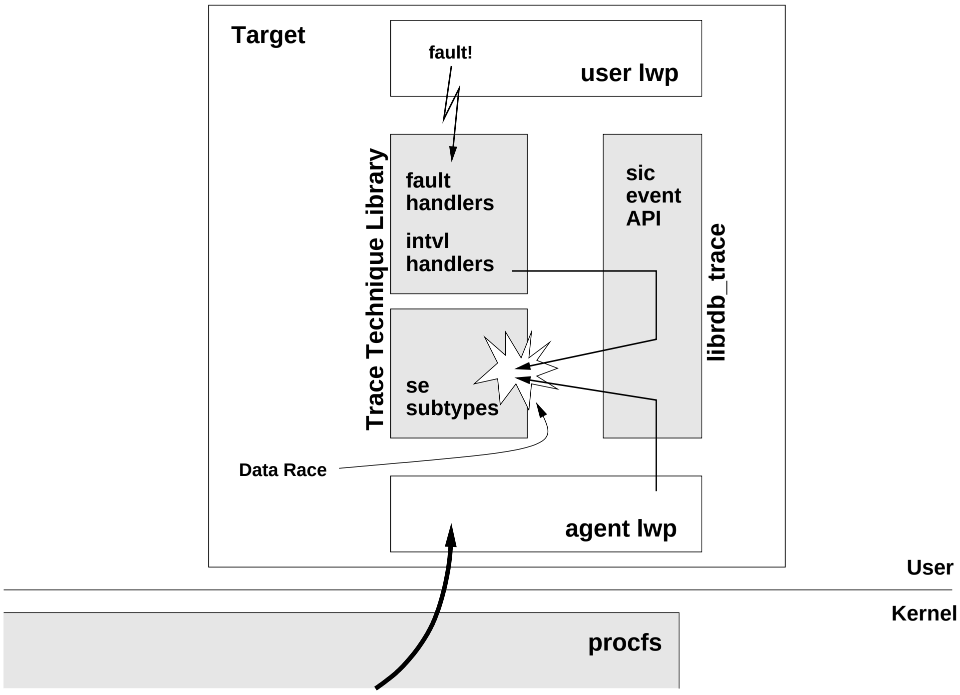

Figure 9: Potential Data Race between the User and Agent LWPs

Our solution is to define a simple protocol between the debugger and the agent LWP which allows the agent to make a determination about whether it is safe for the debugger to operate on the target. If it is not safe to continue, the agent is responsible for arranging for the target to stop in the near future at a safe stopping point, and the debugger must restart the process and wait until it comes to a halt. The process context may also need to be modified in order to mask RDB's presence.

Once the process stops, the agent is activated to determine if the current state is safe. The agent LWP opens the user program's primary LWP using procfs, and examines its context for clues about where the target stopped. If the agent finds a case in which stopping is unsafe, it sets a global flag indicating that a safe stop is needed. The next time the user program transitions back into normal user code, the flag is checked, and the process is halted if necessary. This in itself is problematic, since it can lead to race conditions (see items 4 and 5 below). The agent checks the following cases:

1. The process stopped because some RDB code voluntarily caused a debugger breakpoint. This may be due to an matched PC or SIC breakpoint, a trace event of interest, or a previously scheduled "safe stop" arranged when the process was stopped at an unsafe moment.
2. The process stopped in RDB or trace technique specific code. This is detected by comparing the PC value to the address range which the RDB and trace technique libraries occupy in memory. If so, the `safe_stop` flag is set.
3. The process stopped in a user signal handler. This is problematic since RDB catches all signals and then dispatches them to user signal handlers if installed. This means that a stack backtrace to determine whether the process stopped in user code invoked *from* RDB code is necessary. If so, the `safe_stop` flag is set.
4. The process is in the process of returning to user code at the end of a fault or signal handler, before or after having checked the global `safe_stop` flag.
5. RDB interposes special *system call stubs* that either trace or replay the effects of system calls. On exit from these stubs, the `safe_stop` flag is checked, but similar to case 4, a data race exists between checking the flag and returning from the stub function.
6. The process stopped in normal user code. In this case, the agent can return that the stop is safe, and the debugger can proceed normally.

```
_rdb_safe_setctx(ucontext_t *ctx):
  save input parameters on stack

_rdb_safe_setctx_restart:
  load input parameters from stack

  if(safe_stop == 0)
    goto no_stop

  breakpoint for debugger
  safe_stop = 0

no_stop:
  trap(ST_SYSCALL, SUBSYS_setctx, ctx)
  return

_rdb_safe_setctx_lastpc:
  nop
```

Figure 10: rdb_safe_setctx()

When a *safe stop* is necessary, the agent returns a result of `RESTART` to the debugger, which requests that it restart the user process until a safe stop occurs. Cases 4 and 5 are particularly problematic, since we require atomic "check the flag and return" and "check the flag and set context" operations. To work around this, RDB utilizes the functions `_rdb_safe_setctx()` and `_rdb_safe_sysret()`, respectively. Pseudo-assembly code for `_rdb_safe_setctx()` is shown in figure 10. This function is designed to be *restartable*, which means that if the agent detects that the `safe_stop` flag is about to be set, and the transition from RDB code to user code is occurring (i.e. the PC value falls in the address range [`_rdb_safe_setctx_restart`, `_rdb_safe_setctx_lastpc`]), then `_rdb_safe_setctx()` will need to be restarted from the label `_rdb_safe_setctx_restart` in order to ensure that no data race occurs.

5.5 Debugger Interface

This section presents the interface that the replay library exports to the debugger for creating and installing SIC events into the user process. This interface is designed to be as generic as possible. In order to instantiate a new instance of a particular Software-event in the target, the user fills out a structure of arguments that are specific to that Software-event, and passes this to the rdb debugger library, `librdb_replay`. At this point a debugger-side proxy is created which the user can use to manipulate the real SE inside the target. The API is very small, as shown below:

int rdb_softevt_create(rdb_softevt_t **se*, rdb_proc_t **pr*, uint_t *se_type*, void **args*, size_t *n*);
int rdb_softevt_destroy(rdb_softevt_t **se*);
int rdb_softevt_install(rdb_softevt_t **se*);
int rdb_softevt_uninstall(rdb_softevt_t **se*);
int rdb_softevt_stepover(rdb_softevt_t **se*);

5.6 Implementing Execution Backstep

So far, this API has been used to implement a quantum based backstep. In this mechanism, stepping backwards rolls the state of the process back to the previous quantum. Using the API made it straightforward to code, as figure 11 shows. This illustrates that the project goals have been achieved: We are able to rapidly and easily build powerful debugger tools such as execution backstep via cooperation between target-side and debugger-side code.

6 Future Work

We have been developing a graphical debugger as a testbed for RDB's debugging tools. The work presented here has been integrated into a simple command-line debugger for testing, but has not yet been merged into this application; doing so serves as a reference for what work should be completed in the future.

6.1 Graphical Tools

We envision a complete suite of end-user tools for creating and managing SIC and PC breakpoints, watchpoints, and other debugger events like those presented in this paper. Additionally, we believe that an execution time line that provides both visualization and process navigation capabilities for the developer is essential. RDB's Software-event mechanism can be used to keep the visualization

```
quantum_backstep() {
  dfa2se_sic_args_t args;
  rdb_softevt_t se;

  args.sic = rdb_sic_get() - 1;
  rdb_softevt_create(&se, DFA2_SE_SIC, &args, sizeof(args);

  foreach (s ∈ installed_ses)
    rdb_softevt_uninstall(s);

  rdb_softevt_install(se);
  rdb_proc_restore(proc, cur_intvl);

  rdb_proc_run(proc);

  /* when the process stops, it will be backstepped */
  rdb_proc_wait(proc);

  foreach (s ∈ uninstalled_ses)
    rdb_softevt_install(s);
  rdb_softevt_uninstall(se);
  rdb_softevt_destroy(se);
}
```

Figure 11: Implementation of Quantum Backstep

synchronized with the replay process by using "periodic" debugger events that trigger updates to the time line view.

When a traditional debugger is used to install a PC breakpoint, it is very clear to the developer what meaning that PC value has. It corresponds to precisely one line of code in the program source. By contrast, we believe that developers will probably not be able to understand the *meaning* of a particular SIC value without powerful visual feedback that helps them to contextualize the SIC's value with other events occurring in the process. To this end, it will be necessary to allow developers to overlay a variety of trace data onto the execution time line, and use those as reference points. We envision indicating invocations of particular system calls, signals delivered, or more generally, displaying those replay events which match a particular query. For example, a developer might find it useful to view the execution time line with special markers for all calls to the `write` system call which return a result less than zero (indicating an error), and then choose to set quantum or control point breakpoints prior to one or all of these. To achieve this end, we need to develop a queryable database of trace information that developers can draw from in order to analyze the results of replay.

6.2 Signal Replay

A signal is a software interrupt delivered by the operating system to a user process indicating that some sort of exceptional condition has occurred. Signals present a problem because they occur asynchronously, and may interrupt any user code that is not explicitly masking signals. In order to replay a signal, its occurrence must be traced, and the PC and SIC value at which it occurred recorded. During replay, the signal must be redelivered at exactly the same point in the execution to ensure that the program executes in the same manner.

Using the RDB Software-event mechanism should make implementing signal replay in fairly straightforward. This can be achieved my creating a new SE subtype which, when activated, communicates with RDB's signal API in order to construct and deliver the replayed signal. Doing so will greatly extend the class of user programs which RDB can successfully replay.

7 Conclusions

We have presented a set of new ideas about how to capitalize on the power that a software instruction counter provides to a replay environment. We have shown how to construct sophisticated and useful debugger tools, and presented a framework for doing so in the RDB replay environment. Finally, we have explored ways in which the tools presented here can be used to provide sophisticated visualizations for navigating and analyzing replay data; we hope to add such functionality to RDB in the future.

8 Acknowledgements

None of this work would have been possible without the following people: Robert H.B. Netzer, who supported and occasionally funded this effort, and encouraged me to keep the big picture in focus (and to reduce the bug count); Michael Shapiro, who implemented the RDB framework, and tirelessly advised, inspired, answered questions, and helped to specify the notion of a debugger event; Benjamin Boer, who provided valuable feedback on an early draft of this paper and Michael Warres, who helped to debug and refine RDB.

References

[1] J.M. Mellor-Crummey and T.J. LeBlanc. A software instruction counter. In *Proceedings of the Third ASPLOS*, April 1989.

[2] M.W. Shapiro. Personal communication, October 1997.

[3] M.W. Shapiro. Rdb: A system for incremental replay debugging. Masters thesis, May 1997.

[4] J.H. Slye and E.N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 250–259, June 1996.

[5] Sun Microsystems, Inc., Mountain View, CA. */proc, The Process File System*, 1995.

[6] Tera Computer Company. *Hardware Characteristics of the Tera MTA*, April 1998.

[7] U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice-Hall, New Jersey, 1996.

[8] D.L. Weaver and T. Germond. *The SPARC Architecture Manual: Version 9*. Prentice-Hall, New Jersey, 1994.