Toward Generating Reducible Replay Logs

Kyu Hyung Lee Yunhui Zheng Nick Sumner Xiangyu Zhang

Department of Computer Science, Purdue University, West Lafayette, IN, 47907, USA
{kyuhlee,zheng16,wnsumner,xyzhang}@cs.purdue.edu

Abstract

Logging and replay is important to reproducing software failures and recovering from failures. Replaying a long execution is time consuming, especially when replay is further integrated with runtime techniques that require expensive instrumentation, such as dependence detection. In this paper, we propose a technique to reduce a replay log while retaining its ability to reproduce a failure. While traditional logging records only system calls and signals, our technique leverages the compiler to selectively collect additional information on the fly. Upon a failure, the log can be reduced by analyzing itself. The collection is highly optimized. The additional runtime overhead of our technique, compared to a plain logging tool, is trivial (2.61% average) and the size of additional log is comparable to the original log. Substantial reduction can be cost-effectively achieved through a search based algorithm. The reduced log is guaranteed to reproduce the failure.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Tracing

General Terms Reliability, Performance

Keywords Software reliability, Log reduction, debugging, replay, instrumentation

1. Introduction

Logging and replay is an important technique for software dependability [6, 11, 20]. It records the interactions between a program and its environment during execution by logging system calls and signals. The execution can be replayed as many times as necessary for various purposes, such as diagnosis of software failures and state recovery from failures. The state of the art logging techniques have low overhead, usually less than 10% [6, 18], support concurrent programs [1, 5, 16], and are even provided as part of the operating system. These features make the techniques highly desirable when intensive on-the-fly human attention is infeasible during execution, such as in the emerging cloud computing scenario.

For long running programs such as server programs and user interactive (UI) programs, replay logs may correspond to executions as long as a few hours or even days. Large logs entail long replay times. The programmer may have to wait for hours before a failure is reproduced. Sending such large log files over network could also be problematic. Checkpoints [4] are often created to mitigate the problem. However, as checkpointing often entails taking snapshots of the entire address space of the application, its frequent use cannot be afforded. We aim to develop a practical technique that reduces an execution by reducing its replay log. Given a reduction criterion, such as a program failure, the technique removes events from the log in a way that the reduced log can still properly drive the execution to the criterion. Such a technique is particularly desirable when replay is integrated with debugging techniques that require expensive instrumentation easily causing an order of magnitude slow down [25]. There are also iterative debugging techniques that require repeated replay [24]. In such a context, even a log for a few minutes of execution is hardly affordable. Furthermore, if a failure occurs in remote execution, the large replay log can be reduced on the remote site before it is sent back to the developer.

Challenges of Replayable Log Reduction.

During replay, events from the log are retrieved in order to drive the execution. *The first challenge is that reduction may induce a different control flow path such that the reduced log cannot properly align with the replayed execution.* An event in the reduced log may no longer be encountered during replay due to the different path. Or, replay may take a new path such that an event is expected but not present in the log.

The second challenge is that reduction may change variable values, leading to inconsistency. For example, it may change the number of bytes that are supposed to be read at a system call. That is, the log indicates x bytes were read but the replayed execution expects $x+c$ bytes. Solving such inconsistency demands substantial effort. Existing work tries to achieve replayable reduction by retaining the events that are in the transitive closure of dependences for the criterion [19, 25]. However, it demands replaying the full execution at least once to detect dependences. Tracking all instruction level dependences often incurs 5-10 times slow down [19]. Since reduction is mainly needed for long executions, the resulting cost could be very high. Furthermore, we observe that for complex programs, dependences between events are so pervasive that the transitive closure may involve all events, so reduction can hardly be achieved.

```
1  read(&x);
2  print("x="+x);
3  read(&y);
4  print("y="+y);
5  read(&z);
6  print("z="+z);
```

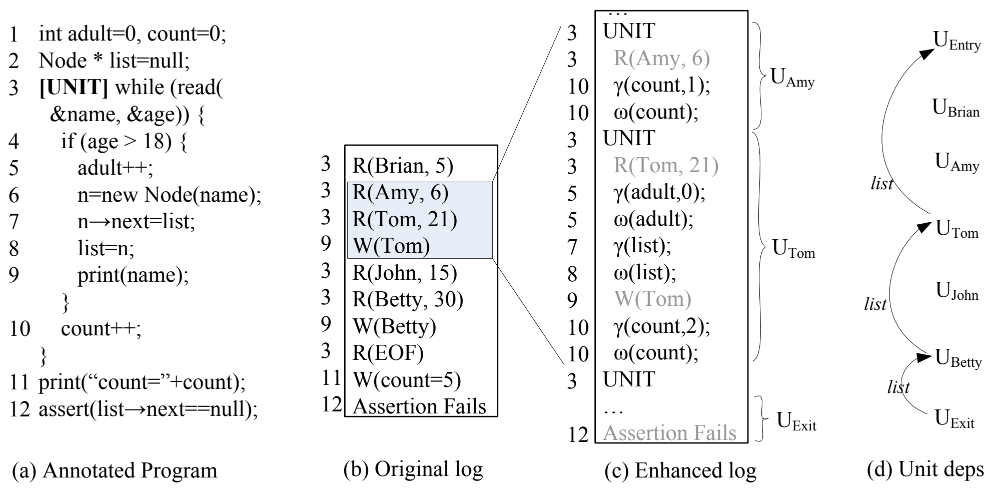
(a) Program

```
1  R(10)
2  W(x=10)
3  R(20)
4  W(y=20)
5  R(30)
6  W(z=30)
```

(b) Sample log

Figure 1. Structural constraints. `R()` stands for an input event. `W()` stands for an output event. The numbers to the left of the log are where events happened.

Dependences are not the only factor needing consideration in reduction. Events independent to the reduction criteria may need to be included due to *structural constraints*. Specifically, *an event is needed if the control flow structure demands its presence even*



```
1  int adult=0, count=0;
2  Node * list=null;
3  [UNIT] while (read(
     &name, &age)) {
4    if (age > 18) {
5      adult++;
6      n=new Node(name);
7      n→next=list;
8      list=n;
9      print(name);
     }
10   count++;
   }
11 print("count="+count);
12 assert(list→next==null);
```

(a) Annotated Program

```
3  R(Brian, 5)
3  R(Amy, 6)
3  R(Tom, 21)
9  W(Tom)
3  R(John, 15)
3  R(Betty, 30)
9  W(Betty)
3  R(EOF)
11 W(count=5)
12 Assertion Fails
```

(b) Original log

```
   …
3  UNIT                 ┐
3    R(Amy, 6)         │
10   γ(count,1);       │ U_Amy
10   ω(count);         ┘
3  UNIT                ┐
3    R(Tom, 21)        │
5    γ(adult,0);       │
5    ω(adult);         │
7    γ(list);          │
8    ω(list);          │ U_Tom
9    W(Tom)            │
10   γ(count,2);       │
10   ω(count);         ┘
3  UNIT
   …                   ┐
12 Assertion Fails     ┘ U_Exit
```

(c) Enhanced log

(d) Unit deps

Figure 2. An example of our design. Events not shaded in (c) are those generated by our technique. γ() and ω() are memory accesses. For readability, we use the name being read in the iteration as the subscript of a unit.

though there is no (transitive) dependence between the criterion and the event. This is the third challenge.

Consider the example in Fig. 1. Ideally, if we want to replay the output of `z` at 6, we only need to replay the read at 5. However, none of the events can be reduced because all of them are expected during replay due to the straight line structure.

Our Design. Our goal is to address the above challenges and develop a practical log reduction technique. The basic idea of this paper is to *selectively record extra information such that reduction can be achieved by analyzing the log*. The approach is *efficient*: it introduces negligible overhead; the additional information needed to be logged is typically not substantial compared to the original log size. It is *effective* in achieving reduction.

In particular, our technique divides an execution into units such that reduction is only carried out at the unit level. In this paper, a unit is an iteration of an event processing loop. For a program driven by external events, its execution is dominated by these iterations. To distinguish the term event from that used in a replay log, we call external events *requests*, and an event loop is hence called a request handling loop. There are many log entries generated inside a unit. They are either reduced entirely or retained for awhile. Such a design allows us easily handle structural constraints. Moreover, there are few dependences crossing unit boundaries because a unit is often supposed to carry out a relatively independent task.

On the other hand, units are not completely independent, so replaying individual units is seldom possible. For example, in the `apache` server program, a unit receives a request from a user, pushes the request to a queue, and then terminates. Another unit pops the request, establishes the connection, and handles the request. It is easy to tell that replaying either unit alone will not succeed. Hence, we need to detect inter-unit dependences. We observe that considering all dependences likely makes the log irreducible as all units are inter-dependent in some way. We propose weaving two kinds of replay. For variables or data structure fields of primitive types, their values are restored at unit boundaries so that we can avoid replaying their dependences. For variables of pointer types or other complex types, we replay their dependences (the units where the variables were defined) because directly restoring their values is either too expensive or problematic. Our compile time analysis instruments the program to collect sufficient runtime information to support such replay. Reduction is done offline by analyzing the enhanced log. Driven by the reduced log, the replay ensures two properties: (1) *the replay control flow must be identical to the corresponding part of the original execution*; (2) *variables must have identical values at identical control flow points*. These properties promise the reproduction of the criterion.

Overview Example. Consider the example in Fig. 2 (a). The program reads personal information and stores the adult information to a linked list. At the end, it asserts the linked list has precisely one element. The user annotates the main request handling loop using keyword "**[UNIT]**", meaning each iteration of the loop is considered a unit. Figure (b) shows the original replay log of an example execution in which the information for two adults is provided. Calls to `read()` and `print()` are logged as input/output events.

The data structures that may cause inter-unit dependences are then identified. They are `adult`, `count`, and `list`. The statements relevant to these variables are instrumented. Part of the enhanced log is presented in Fig. 2 (c). In particular, `UNIT` events delimit execution units. Reads and writes of the relevant data structures are logged. As `count` and `adult` have primitive types, their reads are logged with values. In comparison, the reads of `list` are logged without the pointer values. In Figure (d), the inter-unit dependences regarding variable `list` are presented. Each node denotes a unit. The variable causing a dependence is labeled on the dependence edge. Note that non-pointer dependences are usually not considered. In order to replay the assertion failure, unit U_{exit} needs to be replayed. Transitively, U_{betty}, U_{tom} and U_{entry} are included in the reduced log due to the pointer dependences. Replaying the four events faithfully reconstructs the linked list. When replaying a unit, the values of `count` and `adult` are retrieved from the log instead of being recomputed. Note that `count` causes inter-unit dependences between each pair of consecutive units. Hence, if we consider its dependences, we have to replay all units.

The contributions of the paper are highlighted as follows.

- We propose a novel logging technique that generates reducible logs. It selectively logs memory access information with very little runtime overhead. Reduction is achieved by analyzing the log. In particular, we consider unit level reduction so that less runtime information needs to be logged. We consider primitive and pointer variables separately, which allows us to reach a balance in the logging overhead and the achievable reduction.
- We formally introduce the logging semantics, the replay semantics and the reduction algorithm. The logging semantics is designed such that it avoids redundancy. We develop an aggressive reduction algorithm that saves us from reconstructing the whole memory state while still reproducing the criterion (failure).

- We prove that our reduction scheme can faithfully reproduce the criterion.
- We evaluate our technique on a set of real world programs. Results show the runtime overhead is 2.6% on average and the additional space consumption is comparable to traditional techniques. We can reduce executions with tens of thousands of units to just a few units, and still reproduce the bugs.

2. Language

KERNEL-LANGUAGE $\mathcal{L}$

$$\begin{array}{rcl}
P \in \mathcal{L} & ::= & s \\
x \in \mathit{Var} & ::= & \{x_1,\ x_2,\ ...\} \\
c \in \mathit{Const} & ::= & \{\texttt{true},\ \texttt{false}, ...,\ -1,\ 0,\ 1,\ ...\} \\
a \in \mathit{Addr} & ::= & \{0,\ 1,\ 2,\ ...\} \\
d \in \mathit{Dev} & ::= & \{\texttt{stdin}, \texttt{stdout}, \texttt{file}_1, \texttt{file}_2, ...\} \\
e \in \mathit{Expr} & ::= & x^\ell \mid c \mid \&x \mid *x^\ell \mid e_1\ \texttt{binop}\ e_2 \\
s \in \mathit{Stmt} & ::= & x :=^\ell e \mid *x :=^\ell e \mid \texttt{if}\ x^\ell\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \\
 & & \textbf{[UNIT]}\ \texttt{while}\ x^\ell\ \{s\} \mid \texttt{while}\ x^\ell\ \{s\} \mid s_1;s_2 \mid \\
 & & x := \texttt{read}^\ell(d) \mid \texttt{write}^\ell(d,x) \mid x_1 := \texttt{alloc}^\ell(x_2) \\
 & & \mid \texttt{skip} \mid \texttt{assert}^\ell(x) \mid \texttt{exit}
\end{array}$$

Figure 3. Language Syntax

To facilitate discussion, we introduce a kernel language. The syntax of the language is presented in Fig. 3. We explicitly model memory addresses. We allow the *address-of* and *pointer-dereference* operations, pointer manipulation, and dynamic memory allocation through the `alloc()` statement. We explicitly model devices and I/O with `read()` and `write()` statements. Failures are modeled as assertion violations. The `while` loop with the **[UNIT]** annotation denotes that it is a request handling loop. Supporting concurrent programs is discussed in Section 9.

$$\begin{array}{rlrl}
v \in \mathit{Val} & ::= a \mid c & \sigma \in \mathit{Store} & ::= \mathit{Addr} \to \mathit{Val} \\
\alpha \in \mathit{VarAddr} & ::= \mathit{Var} \to \mathit{Addr} & \iota \in \mathit{IOStore} & ::= \mathit{Dev} \to \overline{\mathit{Val}} \\
e_r \in \mathit{RefExpr} & ::= x \mid *x & & \\
\mathit{addrOf}(e_r) & = \begin{cases} \alpha(x) & e_r = x \\ \sigma(\alpha(x)) & e_r = *x \end{cases} & &
\end{array}$$

Figure 4. Definitions

Fig. 4 presents definitions for the semantics. In particular, the variable address mapping α maps a variable to its address. The mapping is static. The device store ι denotes the state of device, which is a mapping from a device to a sequence of values. A reference expression e_r is the shorthand for a variable or a dereference of a pointer variable. Method $\mathit{addrOf}()$ identifies the address of the given reference expression.

The semantics are presented in Fig. 5. We have two sets of rules. The first set evaluates an expression to a value, provided the store and the variable address mapping. The second set evaluates statements. Statement evaluation has the configuration $\langle s, \sigma, \iota \rangle$ with s the statement, σ the store and ι the device store (defined in Fig. 4). Most evaluations are standard. The evaluation of the annotated `while` statement is the same as the regular `while` statement. Evaluation terminates normally at $\langle \texttt{skip}, \sigma, \iota \rangle$ or abnormally at $\langle \texttt{exit}, \sigma, \iota \rangle$.

We model simple stream devices through rules [Read] and [Write]. In particular, one value is read at a time from the head of a stream device; and a value can be written to the tail of the stream. More I/O complexity is omitted to simplify our formal discussion. Our system supports most system calls and signals. Rule [Alloc] describes dynamic allocation behavior. Symbol $\perp$ means undefined. Initially, the addresses of static variables are mapped to value 0 in the store. The remaining addresses are not defined. We assume infinite memory and do not model deallocation. Note that

EXPRESSION RULES

$\boxed{e \xrightarrow{e} v}$ parameterized on α and σ

$$\frac{}{e_r^\ell \xrightarrow{e} \sigma(\mathit{addrOf}(e_r))}\ [\text{Ref}] \qquad \frac{}{\&x \xrightarrow{e} \alpha(x)}\ [\text{Addr-Of}]$$

$$\frac{}{c \xrightarrow{e} c}\ [\text{Const}] \qquad \frac{e_1 \xrightarrow{e} v_1 \quad e_2 \xrightarrow{e} v_2}{e_1\ \mathit{binop}\ e_2 \xrightarrow{e} v_1\ \mathit{binop}\ v_2}\ [\text{Binop}]$$

STATEMENT RULES

$\boxed{\langle s, \sigma, \iota \rangle \xrightarrow{s} \langle s', \sigma', \iota' \rangle}$ parameterized on α

$$\frac{\mathit{addrOf}(e_r) = a \quad e \xrightarrow{e} v \quad \sigma' = \sigma[a \mapsto v]}{\langle e_r :=^\ell e, \sigma, \iota \rangle \xrightarrow{s} \langle \texttt{skip}, \sigma', \iota \rangle}\ [\text{Assign}]$$

$$\frac{x \xrightarrow{e} \texttt{true}}{\langle \texttt{if}\ x^l\ \texttt{then}\ s_1\ \texttt{else}\ s_2,\ \sigma, \iota \rangle \xrightarrow{s} \langle s_1,\ \sigma, \iota \rangle}\ [\text{If-True}]$$

$$\frac{r = \texttt{if}\ x^\ell\ \texttt{then skip else exit}}{\langle \texttt{assert}^\ell(x), \sigma, \iota \rangle \xrightarrow{s} \langle r, \sigma, \iota \rangle}\ [\text{Assert}]$$

$$\frac{}{\langle \texttt{skip};s,\ \sigma, \iota \rangle \xrightarrow{s} \langle s,\ \sigma, \iota \rangle}\ [\text{Skip}] \qquad \frac{}{\langle \texttt{exit};s,\ \sigma, \iota \rangle \xrightarrow{s} \langle \texttt{exit},\ \sigma, \iota \rangle}\ [\text{Exit}]$$

$$\frac{\iota(d) = v \cdot S \quad a = \alpha(x) \quad \sigma' = \sigma[a \mapsto v] \quad \iota' = \iota[d \mapsto S]}{\langle x := \texttt{read}^\ell(d), \sigma, \iota \rangle \xrightarrow{s} \langle \texttt{skip}, \sigma', \iota' \rangle}\ [\text{Read}]$$

$$\frac{\iota(d) = S \quad x \xrightarrow{e} v \quad \iota' = \iota[d \mapsto S \cdot v]}{\langle \texttt{write}^\ell(d,x), \sigma, \iota \rangle \xrightarrow{s} \langle \texttt{skip}, \sigma, \iota' \rangle}\ [\text{Write}]$$

$$\frac{\begin{array}{c} x_2 \xrightarrow{e} \mathit{size} \quad \sigma(a, ..., a + \mathit{size} - 1) = \perp \\ \sigma' = \sigma[a, ..., a + \mathit{size} - 1 \mapsto 0, \alpha(x_1) \mapsto a] \end{array}}{\langle x_1 := \texttt{alloc}^\ell(x_2), \sigma, \iota \rangle \xrightarrow{s} \langle \texttt{skip}, \sigma', \iota \rangle}\ [\text{Alloc}]$$

Figure 5. Operational Semantics

it does not mean our system cannot handle finite memory and deallocation. Such complexity is just not needed for our discussion. The evaluation rule specifies that with an allocation request of a certain size, a consecutive sequence of undefined addresses of the size is allocated. These addresses are then associated with value 0. The base address is associated with variable x_1. Note that multiple valuations of a could satisfy the rule, which models the nondeterminism of dynamic allocation. We do not explicitly model memory errors, which can be modeled by assertion failures if necessary.

3. Logging Semantics

In the logging phase, besides recording the I/O behavior of the execution, our technique divides the execution into units and logs memory reads and writes. The goal of logging memory accesses is twofold. First, it identifies dependences between units. Particularly, pointer dependences allow us to reconstruct relevant memory state. Second, it allows restoring values when we choose not to replay through dependences.

One naïve way logs all accesses. To discover cross-unit dependences, given a read access in unit U_B from address x, the post-processing algorithm traverses backward in the log file to find the latest write to x that precedes the read. If the write is in a different unit U_A, there is a cross-unit dependence between U_A and U_B, denoted as $U_B \xrightarrow{x} U_A$.

However, this simple logging strategy introduces a lot of redundancy. In particular, if an address is written inside a unit, the values of the following reads from the address in the same unit are deter-

$$
\begin{array}{lcl}
\mu_r \in \mathit{ReadMask} & ::= & \mathit{Addr} \rightarrow \mathit{UnitId} \\
\mu_w \in \mathit{WriteMask} & ::= & \mathit{Addr} \rightarrow \mathit{UnitId} \\
\mathcal{L} \in \mathit{Log} & ::= & \overline{U} \qquad\qquad id \in \mathit{UnitId} \ ::= \ Z^+ \\
U \in \mathit{LogUnit} & ::= & \mathtt{UNIT}(id,\ell)\cdot\overline{E} \\
E \in \mathit{LogEntry} & ::= & \mathtt{R}(id,d,\ell,v) \mid \mathtt{W}(id,d,\ell,v) \mid \\
 & & \mathtt{FAIL}(id,\ell) \mid \gamma(id,\ell,a,v) \mid \omega(id,\ell,a) \\
 & & \\
e_{nr} \in \mathit{NoRefExpr} & ::= & c \mid \&x \mid \mathtt{nil} \\
s_o \in \mathit{RegularStmt} & ::= & \mathtt{skip} \mid \mathtt{exit} \mid \mathtt{while}\, x^\ell\ \{s\} \mid s_1;s_2 \\
 & & \mathtt{if}\ x^\ell\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \mid x :=^\ell e \mid \\
 & & *x :=^\ell e \mid x_1 := \mathtt{alloc}^\ell(x_2) \\
 & & \\
\mathit{expr}(s_o) & = & \begin{cases} e & s_o = x :=^\ell e \\ e, x^\ell & s_o = *x :=^\ell e \\ x_2^\ell & s_o = x_1 := \mathtt{alloc}^\ell(x_2) \\ x^\ell & s_o = \mathtt{if}\ x^\ell\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 \\ \mathtt{nil} & \mathit{others} \end{cases} \\
 & & \\
\mathit{def}(s_o) & = & \begin{cases} x & s_o = x :=^\ell e \\ *x & s_o = *x :=^\ell e \\ x_1 & s_o = x_1 := \mathtt{alloc}^\ell(x_2) \\ \mathtt{nil} & \mathit{others} \end{cases} \\
 & & \\
\mathit{need2logwrite}(s_o) & = & \mathtt{let}\ x = \mathit{def}(s_o)\ \mathtt{in}\ x \neq \mathtt{nil} \\
 & & \wedge\, \mu_w(\mathit{addrOf}(x)) \neq \mathit{uid} \\
\mathit{accessed}(a) & = & \mu_w(a) = \mathit{uid} \vee \mu_r(a) = \mathit{uid}
\end{array}
$$

uid: the shorthand for $\sigma(\alpha(\mathtt{unit\_id}))$;
$\mathit{uid}\uparrow$: the shorthand for $\alpha(\mathtt{unit\_id}) \mapsto \sigma(\alpha(\mathtt{unit\_id}))+1$.

Figure 6. Definitions for Logging Semantics

	Execution	Log
1_1	$x = \ldots;$	$\omega(\&x)$
2_1	**[UNIT]** `while` {	
3_1	$\ldots = x;$	$\gamma(\&x)$
4_1	$x = x + 1;$	$\omega(\&x)$
5_1	$\ldots = x;$	
6_1	x=...;	
2_2	**[UNIT]** `while` {	
3_2	$\ldots = x;$	$\gamma(\&x)$
	...	

Figure 7. The accesses at 5_1 (that is, the 1st instance of line 5) and 6_1 are redundant and not logged. Symbols $\gamma()$ and $\omega()$ represent memory read and write logs.

ministic and such reads cannot cause any cross-unit dependence[1]. Hence, logging them is unnecessary. Similarly, the following writes in the same unit do not need to be logged. When a cross-unit dependence involves one of these writes, keeping the first write in the log is sufficient for disclosing the same unit level dependence. Furthermore, if a read from an address has been logged, the following reads in the same unit don't need to be logged because they must have the same value and they only reveal the same cross-unit dependence that the first read reveals.

Consider the example in Fig. 7. It shows a trace of two iterations of an annotated loop. The read in 4_1 is not logged because of the read at 3_1. The read at 5_1 is not logged because of the definition at 4_1. The write at 6_1 is not logged because of the write at 4_1.

The rules are presented in Fig. 8 and the relevant definitions can be found in Fig. 6. We compute a few relations during evaluation. According to the definitions in Fig. 6, relation μ_r maps an address to the id of a unit that most recently reads the address. It is used to avoid logging redundant reads. Similarly, μ_w identifies the most recent unit that writes to an address. Symbol $\mathcal{L}$ denotes the generated log. A log comprises a sequence of log units. A log unit U

[1] This assumes sequential semantics and Section 9 discusses handling threads.

EXPRESSION LOGGING RULES

$\boxed{\langle e,\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r',\mathcal{L}'\rangle}$ parameterized on α, σ, and μ_w.

$$\frac{\mathit{addrOf}(e_r) = a \quad \neg\mathit{accessed}(a) \quad v = \sigma(a)}{\langle e_r^\ell,\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r[a\mapsto \mathit{uid}],\ \mathcal{L}\cdot\gamma(\mathit{uid},\ell,a,v)\rangle} \quad \text{[LE-Log]}$$

$$\frac{\mathit{addrOf}(e_r) = a \quad \mathit{accessed}(a)}{\langle e_r^\ell,\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r,\mathcal{L}\rangle} \quad \text{[LE-NoLog]}$$

$$\frac{}{\langle e_{nr},\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r,\mathcal{L}\rangle} \quad \text{[LE-NoRef]}$$

$$\frac{\langle e_1,\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r',\mathcal{L}'\rangle \quad \langle e_2,\mu_r',\mathcal{L}'\rangle \overset{e}{\Rightarrow} \langle \mu_r'',\mathcal{L}''\rangle}{\langle e_1\ \mathit{binop}\ e_2,\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r'',\mathcal{L}''\rangle} \quad \text{[LE-Binop]}$$

STATEMENT LOGGING RULES

$\boxed{\langle s,\sigma,\iota,\mu_r,\mu_w,\mathcal{L}\rangle \overset{s}{\Rightarrow} \langle s',\sigma',\iota',\mu_r',\mu_w',\mathcal{L}'\rangle}$ parameterized on α

$$\frac{s_T =\ s;\ \mathtt{while}\ x^\ell\ \{s\} \quad \sigma' = \sigma[\mathit{uid}\uparrow] \quad \mathcal{L}' = \mathcal{L}\cdot\mathtt{UNIT}(\mathit{uid},\ell)}{\begin{array}{c}\langle \textbf{[UNIT]}\mathtt{while}\ x^\ell\ \{s\},\ \sigma,\iota,\mu_r,\mu_w,\mathcal{L}\rangle \overset{s}{\Rightarrow} \\ \langle \mathtt{if}\ x^\ell\ \mathtt{then}\ s_T\ \mathtt{else\ skip},\ \sigma',\iota,\mu_r,\mu_w,\mathcal{L}'\rangle\end{array}}$$
[LS-Unit-While]

$$\frac{\begin{array}{c}\langle x = \mathtt{read}^\ell(d),\sigma,\iota\rangle \xrightarrow{s} \langle \mathtt{skip},\sigma',\iota'\rangle \quad a = \alpha(x) \\ \mu_w' = \mu_w[a\mapsto \mathit{uid}] \quad \mathcal{L}' = \mathcal{L}\cdot\mathtt{R}(\mathit{uid},d,\ell,\sigma'(a))\end{array}}{\langle x = \mathtt{read}^\ell(d),\sigma,\iota,\mu_r,\mu_w,\mathcal{L}\rangle \overset{s}{\Rightarrow} \langle \mathtt{skip},\sigma',\iota',\mu_r,\mu_w',\mathcal{L}')\rangle}$$
[LS-Input]

$$\frac{\begin{array}{c}\langle \mathtt{write}^\ell(d,x),\sigma,\iota\rangle \xrightarrow{s} \langle \mathtt{Skip},\sigma,\iota'\rangle \quad x \xrightarrow{e} v \\ \langle x^\ell,\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r',\mathcal{L}'\rangle \quad \mathcal{L}'' = \mathcal{L}'\cdot\mathtt{W}(\mathit{uid},d,\ell,v)\end{array}}{\langle \mathtt{write}^\ell(d,x),\sigma,\iota,\mu_r,\mu_w,\mathcal{L}\rangle \overset{s}{\Rightarrow} \langle \mathtt{skip},\sigma,\iota',\mu_r',\mu_w,\mathcal{L}''\rangle}$$
[LS-Output]

$$\frac{\langle x^\ell,\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r',\mathcal{L}'\rangle \quad x \xrightarrow{e} \mathtt{false} \quad \mathcal{L}'' = \mathcal{L}'\cdot\mathtt{FAIL}(\mathit{uid},\ell)}{\langle \mathtt{assert}(x^\ell),\sigma,\iota,\mu_r,\mu_w,\mathcal{L}\rangle \overset{s}{\Rightarrow} \langle \mathtt{exit},\sigma,\iota,\mu_r',\mu_w,\mathcal{L}''\rangle}$$
[LS-Assert-Fail]

$$\frac{\langle x^\ell,\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r',\mathcal{L}'\rangle \quad x \xrightarrow{e} \mathtt{true}}{\langle \mathtt{assert}(x^\ell),\sigma,\iota,\mu_r,\mu_w,\mathcal{L}\rangle \overset{s}{\Rightarrow} \langle \mathtt{skip},\sigma,\iota,\mu_r',\mu_w,\mathcal{L}'\rangle}$$
[LS-Assert-Pass]

$$\frac{\begin{array}{c}\langle \mathit{expr}(s_o),\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r',\mathcal{L}'\rangle \quad \mathit{need2logwrite}(s_o) \\ \langle s_o^\ell,\sigma,\iota\rangle \xrightarrow{s} \langle r,\sigma',\iota\rangle \quad a = \mathit{addrOf}(\mathit{def}(s_o))\end{array}}{\langle s_o^\ell,\ \sigma,\iota,\mu_r,\mu_w,\mathcal{L}\rangle \overset{s}{\Rightarrow} \langle r,\sigma',\iota,\mu_r',\mu_w[a\mapsto \mathit{uid}],\mathcal{L}'\cdot\omega(\mathit{uid},\ell,a)\rangle}$$
[LS-Log-Wrt]

$$\frac{\begin{array}{c}\langle \mathit{expr}(s_o),\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r',\mathcal{L}'\rangle \quad \neg\mathit{need2logwrite}(s_o) \\ \langle s_o^\ell,\sigma,\iota\rangle \xrightarrow{s} \langle r,\sigma',\iota\rangle\end{array}}{\langle s_o^\ell,\ \sigma,\iota,\mu_r,\mu_w,\mathcal{L}\rangle \overset{s}{\Rightarrow} \langle r,\sigma',\iota,\mu_r',\mu_w,\mathcal{L}'\rangle}$$
[LS-NoLog-Wrt]

Figure 8. Logging Semantics

corresponds to an execution unit, containing a `UNIT()` event followed by a sequence of other events. The `UNIT()` event contains the id of the unit and the program location ℓ of the loop that generates the unit. Besides the `UNIT()` event, we model other 5 kinds of events, described by symbol E. In particular, `R()` and `W()` are the input and output events, in which we log the unit id, the device id, the program location of the event, and the input/output value. The `FAIL()` event records an assertion failure. Events $\gamma()$ and $\omega()$ represent memory reads and writes. The remaining definitions in Fig. 6 are auxiliary to the rules.

The expression rules are of the format $\langle e,\mu_r,\mathcal{L}\rangle \overset{e}{\Rightarrow} \langle \mu_r',\mathcal{L}'\rangle$, provided α, σ, and μ_w. Rules [LE-Log] and [LE-NoLog] evaluate reference expression e_r, which is either x or $*x$ (Fig. 4). A reference expression involves a memory read. It tests whether the address being referred to has been accessed in the same unit (through method *accessed*() defined in Fig. 6). If so, we don't need to log the read. Otherwise, we attach a γ event to the log and update the most recent read unit of the address. Note that we introduce a program variable *unit_id* to denote the dynamic unit id. Symbol *uid* is shorthand for its value. Rule [LE-NoRef] handles expressions not involving memory reads.

Statement evaluation has the configuration of $\langle s, \sigma, \iota, \mu_r, \mu_w, \mathcal{L} \rangle$. Rule [LS-Unit-While] specifies that *uid* is incremented and a $\texttt{UNIT}()$ event is logged when an annotated while statement is evaluated. Note that the $\texttt{UNIT}()$ event leads the unit corresponding to the iteration if the loop predicate evaluates true, otherwise, it leads the unit corresponding to the execution from the end of the loop to the beginning of the next annotated loop. Rules [LS-Input] and [LS-Output] attach $\texttt{R}()$ and $\texttt{W}()$ entries, respectively, to the log. In [LS-Input], the most recent write unit to the left hand side variable x is updated. In [LS-Output], the read of x may be logged by the expression evaluation. When an assertion fails (rule [LS-Assertion-Fail]), a $\texttt{FAIL}()$ event is logged. It is then the last event in the log as the `exit` statement leads to the termination of the evaluation.

Rules [LS-Log-Wrt] and [LS-NoLog-Wrt] describe evaluation of the other statements, denoted as s_o, whose definitions are in Fig. 6. Two auxiliary functions $expr(s_o)$ and $def(s_o)$ are used. The former function returns the expressions that need to be evaluated for a s_o statement. Such expressions lead to memory reads. For instance, $expr(*x :=^{\ell} e) = \; e, x^{\ell}$, with e, x being the concatenation of the two expressions. When the concatenated expression is evaluated under rule [LE-Binop], e is evaluated first and then x, ensuring the reads are logged in the proper order. Function $def(s_o)$ returns the target of a memory write if s_o involves one, or `nil` otherwise. In rules [LS-Log-Wrt] and [LS-NoLog-Wrt], the expressions s_o are first evaluated. If the write target has not been written previously in the same unit, a $\omega()$ event is logged (rule [LS-Log-Wrt]). Evaluation starts with $\mathcal{L} = \texttt{UNIT}(0,0)$ such that the execution from the beginning to the start of the first annotated loop belongs to U_0.

4. Reduction Algorithm

$pointerdep(U,X) = \; \exists \omega(id, \ell, a) \in U$ and $\gamma(id', \ell', a, v) \in X, s.t.\ ispointer(\ell) \wedge \nexists \omega(id'', \ell'', a) \in S$ with $\omega(id, \ell, a) \cdot S \cdot \gamma(id', \ell', a, v) \subset \mathcal{L}$

$isloopexit(U) \; = \; U = \texttt{UNIT}(uid, \ell) \cdot S \; \wedge \; U \cdot U' \subset \mathcal{L} \; \wedge \; U' = \texttt{UNIT}(uid + 1, \ell') \cdot S' \; \wedge \; \ell \neq \ell'$

$X \in ReducedLog ::= \; \overline{U}$

LOG REDUCTION $\boxed{\langle \mathcal{L}, X \rangle \rightsquigarrow \langle \mathcal{L}', X' \rangle}$

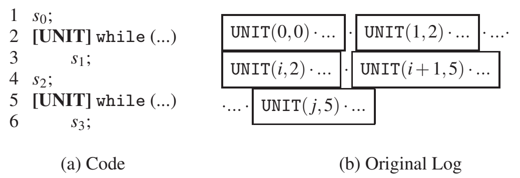
$$\frac{\texttt{FAIL}(id, \ell) \in U_c}{\langle \mathcal{L}_h \cdot U_c, \texttt{nil} \rangle \rightsquigarrow \langle \mathcal{L}_h, U_c \rangle} \text{[X-Crit]} \qquad \frac{pointerdep(U,X)}{\langle \mathcal{L}_h \cdot U, X \rangle \rightsquigarrow \langle \mathcal{L}_h, U \cdot X \rangle} \text{[X-PointerDep]}$$

$$\frac{isloopexit(U)}{\langle \mathcal{L}_h \cdot U, X \rangle \rightsquigarrow \langle \mathcal{L}_h, U \cdot X \rangle} \text{[X-LoopExit]} \qquad \frac{}{\langle U, X \rangle \rightsquigarrow \langle \texttt{nil}, U \cdot X \rangle} \text{[X-FirstUnit]}$$

$$\frac{\neg pointerdep(U,X) \quad \neg isloopexit(U)}{\langle \mathcal{L}_h \cdot U, X \rangle \rightsquigarrow \langle \mathcal{L}_h, X \rangle} \text{[X-Reduce]}$$

Figure 9. Log Reduction.

Given a replay criterion, e.g. a failure, our reduction algorithm identifies a small subset of units that are replayable and produce the criterion. The approach includes units that are relevant to the criterion through cross-unit pointer dependences or other complex dependences such as those through buffers and complex fields in data structures. We call this *replay by dependence*. For uses of variable having primitive types and defined in other units, we retrieve the values from the log, avoiding replay of their dependence precedence. We call this *replay by value*. For example, if a pointer x is defined in unit U_B and then used in U_A, we need to replay U_B if we replay U_A. Assume x has a primitive field f, and $x \rightarrow f$ is defined in U_C and used in U_A. We don't need to replay U_C when replaying U_A as we acquire its value directly from the log.



```
1  s_0;
2  [UNIT] while (...)
3          s_1;
4  s_2;
5  [UNIT] while (...)
6          s_3;
```

$\boxed{\texttt{UNIT}(0,0) \cdot \ldots} \cdot \boxed{\texttt{UNIT}(1,2) \cdot \ldots} \cdot \ldots \cdot$
$\boxed{\texttt{UNIT}(i,2) \cdot \ldots} \cdot \boxed{\texttt{UNIT}(i+1,5) \cdot \ldots}$
$\cdot \ldots \cdot \boxed{\texttt{UNIT}(j,5) \cdot \ldots}$

(a) Code (b) Original Log

Figure 10. Ex. for reduction with structural constraints. Variables i and j are ids and the 2nd numbers in $\texttt{UNIT}()$ are line numbers.

The reason for this design is that restoring pointer values from the log is problematic due to dynamic allocation. Furthermore, for dependences on buffers and complex fields, restoring values implies logging their values, which is much more expensive than logging primitive variables/fields.

Note that our technique nonetheless allows the developer to replay primitive variables/fields by dependences if needed. All the necessary information is available in the log. One can imagine cases in which the root cause resides in a unit different than the criterion and affects the criterion through primitive dependences, the developer can choose to replay the dependence of a primitive value if she/he decides the value is suspicious and wants to inspect the computation leading to the value.

Fig. 9 presents the log reduction algorithm. The evaluation starts with the original log $\mathcal{L}$ and an empty reduced log $X = \texttt{nil}$. The units in $\mathcal{L}$ are evaluated in a backward fashion. The evaluation terminates with all units processed, and X is the reduced log. Rule [X-Crit] adds the unit containing the criterion event to X. In this paper, we assume the criterion is a failure. Rule [X-PointerDep] mandates that the last unit in $\mathcal{L}$ is added to X if some unit in X has a pointer dependence on the unit. Method $pointerdep(U,X)$ tests if there is a memory write in U and a read in X with the same address, and there is not other writes to the address in-between. If so, the unit is added to X.

Rule [X-LoopExit] tests whether a unit under consideration denotes a loop exit, meaning the unit encompasses the execution between the exit of an annotated loop and the beginning of the next annotated loop. If so, we need to add the unit to X to respect the structural constraint. If we did not, and the unit were not replayed, the execution wouldn't be able to get out of the annotated loop. Method $isloopexit(U)$ decides whether U is a loop exit by testing whether the label of the leading $\texttt{UNIT}()$ event of the unit is different from that of the next unit. A unit is reduced if there are no pointer dependences between the unit and the previously computed X and it is not a loop exit (rule [X-Reduce]). Finally, the first unit, which corresponds to the execution from the beginning to the first annotated loop, is always included (rule [X-FirstUnit]). This is for the consideration of structural constraints.

Consider the example in Fig. 10. There are two annotated loops. Figure (b) shows a sample log. Each box represents a unit. Assume the replay criterion is U_j, the shorthand for the unit starting with $\texttt{UNIT}(j,5)$, and it has no cross-unit pointer dependences with any other units. According to the rules, units U_0 and U_i are included in the reduced log. U_0 drives the execution to the loop at 2 and U_i forces the execution get out of loop 2 and reach loop 5.

EXPRESSION REPLAY RULES

$\boxed{\langle e,\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma',\mu_r',X'\rangle}$ parameterized on α, σ, and μ_w.

$$\frac{addrOf(e_r)=a \quad \neg accessed(a) \quad \neg ispointer(\ell) \qquad \boxed{X=\gamma(id,\ell,a',v)\cdot X'} \quad \sigma'=\sigma[a\mapsto v]}{\langle e_r^{\ell},\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma',\ \mu_r[a\mapsto uid],\ X'\rangle} \text{[RE-ByValue]}$$

$$\frac{addrOf(e_r)=a \quad \neg accessed(a) \quad ispointer(\ell) \quad \boxed{X=\gamma(id,\ell,a',v)\cdot X'}}{\langle e_r^{\ell},\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma,\mu_r[a\mapsto uid],X'\rangle} \text{[RE-ByDep]}$$

$$\frac{addrOf(e_r)=a \quad accessed(a)}{\langle e_r^{\ell},\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma,\mu_r,X\rangle} \text{[RE-NoLog]}$$

$$\frac{}{\langle e_{nr},\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma,\mu_r,X\rangle} \text{[RE-NoRef]}$$

$$\frac{\langle e_1,\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma',\mu_r',X'\rangle \quad \langle e_2,\sigma',\mu_r',X'\rangle \twoheadrightarrow^e \langle \sigma'',\mu_r'',X''\rangle}{\langle e_1\ binop\ e_2,\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma'',\mu_r'',X''\rangle} \text{[RE-Binop]}$$

STATEMENT REPLAY RULES

$\boxed{\langle s,\sigma,\mu_r,\mu_w,X\rangle \twoheadrightarrow^s \langle s',\sigma',\mu_r',\mu_w',X'\rangle}$ parameterized on α

$$\frac{\boxed{X=\mathtt{UNIT}(id,\ell)\cdot X'} \quad \sigma'=\sigma[uid\uparrow] \quad s_T=s;\ \mathtt{while}\ x^{\ell}\ \{s\}}{\langle \mathbf{[UNIT]}\mathtt{while}\ x^{\ell}\ \{s\},\ \sigma,\mu_r,\mu_w,X\rangle \twoheadrightarrow^s \langle \mathtt{if}\ x^{\ell}\ \mathtt{then}\ s_T\ \mathtt{else\ skip},\ \sigma',\mu_r,\mu_w,X'\rangle} \text{[RS-Unit-While]}$$

$$\frac{\boxed{X=\mathtt{R}(id,d,\ell,v)\cdot X'} \quad a=\alpha(x) \quad \sigma'=\sigma[a\mapsto v]}{\langle x=\mathtt{read}^{\ell}(d),\sigma,\mu_r,\mu_w,X\rangle \twoheadrightarrow^s \langle \mathtt{skip},\sigma',\mu_r,\mu_w[a\mapsto uid],X'\rangle} \text{[RS-Input]}$$

$$\frac{\langle x^{\ell},\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma',\mu_r',X'\rangle \quad x \xrightarrow{e}_{\sigma'} v \quad \boxed{X'=\mathtt{W}(id,d,\ell,v)\cdot X''}}{\langle \mathtt{write}^{\ell}(d,x),\sigma,\mu_r,\mu_w,X\rangle \twoheadrightarrow^s \langle \mathtt{skip},\sigma',\mu_r',\mu_w,X''\rangle} \text{[RS-Output]}$$

$$\frac{\langle x^{\ell},\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma',\mu_r',X'\rangle \quad x \xrightarrow{e}_{\sigma'} \mathtt{false} \quad \boxed{X'=\mathtt{FAIL}(id,\ell)}}{\langle \mathtt{assert}^{\ell}(x),\sigma,\mu_r,\mu_w,X\rangle \twoheadrightarrow^s \langle \mathtt{exit},\sigma',\mu_r',\mu_w,\mathtt{nil}\rangle} \text{[RS-Assert-F]}$$

$$\frac{\langle x^{\ell},\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma',\mu_r',X'\rangle \quad x \xrightarrow{e}_{\sigma'} \mathtt{true}}{\langle \mathtt{assert}^{\ell}(x),\sigma,\mu_r,\mu_w,X\rangle \twoheadrightarrow^s \langle \mathtt{skip},\sigma',\mu_r',\mu_w,X'\rangle} \text{[RS-Assert-T]}$$

$$\frac{\begin{array}{c}\langle expr(s_o),\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma',\mu_r',X'\rangle \quad need2logwrite(s_o)\\ \langle s_o^{\ell},\sigma',\mathtt{nil}\rangle \xrightarrow{s} \langle r,\sigma'',\mathtt{nil}\rangle \quad a=addrOf(def(s_o))\\ \mu_w'=\mu_w[a\mapsto uid] \quad \boxed{X'=\omega(id,\ell,a')\cdot X''}\end{array}}{\langle s_o^{\ell},\ \sigma,\mu_r,\mu_w,X\rangle \twoheadrightarrow^s \langle r,\sigma'',\mu_r',\mu_w',X''\rangle} \text{[RS-Log-Wrt]}$$

$$\frac{\begin{array}{c}\langle expr(s_o),\sigma,\mu_r,X\rangle \twoheadrightarrow^e \langle \sigma',\mu_r',X'\rangle\\ \neg need2logwrite(s_o) \quad \langle s_o^{\ell},\sigma',\mathtt{nil}\rangle \xrightarrow{s} \langle r,\sigma'',\mathtt{nil}\rangle\end{array}}{\langle s_o^{\ell},\ \sigma,\mu_r,\mu_w,X\rangle \twoheadrightarrow^s \langle r,\sigma'',\mu_r',\mu_w,X'\rangle} \text{[RS-NoLog-Wrt]}$$

Figure 11. Replay Semantics.

Note that dividing an execution into units simplifies the structural constraints. We do not support nested annotated loops. In such cases, the outer annotations are ignored.

5. Replay

Replay is driven by the reduced log. It can be considered a dual process of logging. The rules are presented in Fig. 11. Lets first consider the expression rules. If a memory read is not on a pointer value, we replay by restoring the value from the log (rule [RE-ByValue]). The store is updated with the logged value. Note, the rule dictates that the logged label and the current label match, ensuring the read happens in the same program point as before. During replay, the masks μ_r and μ_w are maintained in the same way as in the logging phase so that the replay algorithm can avoid restoring values that were found to be redundant during logging. This is important to the synchronization of the replay and the reduced log. Hence, in rule [RE-ByValue], μ_r is updated to reflect that the address has been read in the current unit. If the memory read is on a pointer value (rule [RE-ByDep]), we don't restore the value. Instead, the value is presently available due to earlier replay operations. Hence, the algorithm simply removes the current log entry. The other expression rules are similar to the logging rules.

The statement rules are similar to those in the logging phase. We summarize the key features of the replay rules as follows.

- In order for replay to progress, the reduced log has to perfectly align with the evaluation, indicated by the label of the current evaluation must be identical to that in the current log entry. This is highlighted in the boxed premises.
- The rules do not require that the current unit id *uid* matches the logged id (e.g. rule [RS-Unit-While]), because the current id could be different due to reduction.
- Replay does not concern devices. The I/O rules only interact with the log file.
- Many rules involve evaluating expressions to values. Such evaluation is conducted after the proper values are restored from the log. For instance, in rule [RS-Output], evaluation $x \xrightarrow{e}_{\sigma'} v$ is parameterized on the updated store.
- In rule [RS-Output], the output value v is constrained. It serves as a validation rule. Intuitively, we demand that the observable outputs match those recorded in the reduced log.

6. Replayability

We say a reduced log is *replayable* if it successfully drives the replay to the criterion. It is equivalent to the *progress* property for the rules in Fig. 11. Intuitively, the replay control flow has to perfectly align with the reduced log such that proper events can be retrieved. The replay outputs have to match the logged values. Otherwise, the replay evaluation will get stuck.

$$\begin{array}{lll} T \in Trace ::= \bar{t} & t \in TraceEntry ::= \mathtt{trace}(s^{\ell},v,p) & \\ p \in Pointer ::= \mathtt{pnt}(b,z,f) & & \\ b \in BaseAddr ::= Addr & z \in Size ::= Z^{+} & f \in Offset ::= Z^{+} \end{array}$$

We leverage traces to formally discuss replayability. As presented above, a trace is a sequence of entries. A trace entry is a triple that records the evaluated statement s, the right hand side value v and the canonical pointer value if it is a pointer. Canonical pointer values allow us to reason about pointer equivalence between the original run and the replay. It is a triple tracking the base address, the size of the allocated region, and the offset in the region. A pointer obtains its canonical base address and region size at the address-of operation or the `alloc()` statement, with initial offset 0. The region size for an address-of operation is 1, indicating it is pointing to a variable. Offsets are updated by pointer manipulations. The semantics for tracing is omitted.

Definition 1. (REPLAYABILITY)
Let X be a reduction of $\mathcal{L}$ for a given criterion. Let the subtrace corresponding to X in the original run be T, the replay trace be $\tilde{T}$. We say X is a replayable reduction if $T \doteq_T \tilde{T}$.

Note that T corresponds to the executions of all the units that are in the reduced log. The trace equivalence operator $\doteq_T$ is defined in Fig. 12. It demands the two traces have the same number of entries and the corresponding entries are equivalent. If the corresponding

$$\frac{\begin{array}{c}\ell_1=\ell_2 \quad ispointer(\ell_1) \quad p_1=\texttt{pnt}(b_1,z_1,f_1) \quad p_2=\texttt{pnt}(b_2,z_2,f_2) \\ \exists x,\ b_1=b_2=\alpha(x) \quad f_1=f_2=0 \quad z_1=z_2=1\end{array}}{\texttt{trace}(s_1^{\ell_1},v_1,p_1) \doteq_t \widetilde{\texttt{trace}}(s_2^{\ell_2},v_2,p_2)} \quad \text{[TEQ-P1]}$$

$$\frac{\begin{array}{c}\ell_1=\ell_2 \quad ispointer(\ell_1) \quad p_1=\texttt{pnt}(b_1,z_1,f_1) \\ p_2=\texttt{pnt}(b_2,z_2,f_2) \quad z_1=z_2 \quad f_1=f_2 \\ \boxed{\begin{array}{c}\exists\, i,\ t_i=\texttt{trace}(x_1^{\ell}:=\texttt{alloc}(x_2),\ b_1,\ \texttt{pnt}(b_1,z_1,0)\ ) \text{ and} \\ \tilde{t}_i=\widetilde{\texttt{trace}}(x_1^{\ell}:=\texttt{alloc}(x_2),\ b_2,\ \texttt{pnt}(b_2,z_2,0)\ )\end{array}}\end{array}}{\texttt{trace}(s_1^{\ell_1},v_1,p_1) \doteq_t \widetilde{\texttt{trace}}(s_2^{\ell_2},v_2,p_2)} \quad \text{[TEQ-P2]}$$

$$\frac{\ell_1=\ell_2 \quad \neg ispointer(\ell) \quad v_1=v_2}{\texttt{trace}(s_1^{\ell_1},v_1,p_1) \doteq_t \widetilde{\texttt{trace}}(s_2^{\ell_2},v_2,p_2)} \quad \text{[TEQ-Val]}$$

$$\frac{|T|=|\tilde{T}| \quad \forall 0\le i<|T|,\ t_i \doteq_t \tilde{t}_i}{T \doteq_T \tilde{T}} \quad \text{[TEQ]}$$

Figure 12. Trace Equivalence.

entries involve non-pointer values, their equivalence is determined by the equivalence of the statements and the values (rule [TEQ-Val]). If the entries involve pointers, the pointer values must be identical if they point to some variable (rule [TEQ-P1])[2]. If they point to dynamically allocated regions (rule [TEQ-P2]), the regions must be allocated at corresponding execution points and have the same size and the same offset (t_i and $\tilde{t}_i$ are the ith entries in the two traces). Note that the base addresses of t_i and $\tilde{t}_i$ might be different as less memory is allocated in the reduced replay.

Theorem 1. *If the program only allows comparisons and subtractions between two pointers, the reduction scheme in Fig. 9 is replayable in the absence of overflow.*

Overflow is defined as follows.

$$\frac{ispointer(\ell) \quad p=\texttt{pnt}(b,z,f) \quad f\ge z}{overflow(\texttt{trace}(s^{\ell},v,p))}$$

For example, in the following program snippets "`1. b=&x; 2. p=b+1;`" and "`1. b=alloc(10); 2. p=b+12;`", overflows occur in statement 2s.

Proof Sketch of the Theorem 1 To prove replayability, we want to prove trace equivalence. According to rule [X-FirstUnit], we always replay the first unit. As a result, the theorem holds for the trace entries corresponding to the first unit.

Next, we prove by induction. We assume the theorem holds for the first k entries, and prove that it holds for the $(k+1)$th entry, that is, $t_{k+1} \doteq_t \tilde{t}_{k+1}$.

First, the statement under evaluation for the $(k+1)$th entry must be the same as the result of the equivalence of the previous entries. In particular, any predicate guarding the statement must have been evaluated previously in equivalent entries. Also, it must have been evaluated to the same result according to the assumption, leading to the execution of the same statement in the $k+1$ entry. Hence, we only need to prove all the uses (references) in the entry have equivalent values. Without loss of generality, let's consider a reference of variable x.

In case (1), x has a non-pointer value. Assume in the original run, the value is defined by a previous evaluation, denoted by trace entry t_j. If t_j and t_{k+1} are within the same unit, an equivalent evaluation must have occurred in $\tilde{t}_j$. In the absence of overflow, there are no other assignments to the same address during replay. Hence, x must have an equivalent value in $\tilde{t}_{k+1}$ during replay. If t_j and t_{k+1} are in different units, the value of x is retrieved from the log according to the replay semantics, and x is equivalent across runs.

Note that execution reduction causes `alloc()` to return different values during replay because many allocations may be reduced away. An overflow pointer may cause different variables to be overwritten. For instance, assume dynamically allocated buffers A and B are next to each other in the original run, A and C are adjacent in the replay. Assume an overflow pointer based on A overwrites B, the equivalent pointer overwrites C during replay. In other words, in the presence of overflows, writes though pointers with equivalent values at equivalent points may write to different variables (regions).

In case (2), x is a pointer. Assume it is defined at t_j. When t_j and t_{k+1} are in the same unit, the equivalence can be derived as in case (1). If they are in different units, denoted as U_m and U_n respectively, U_m is replayed according to our reduction rules. In the absence of overflow, it can be inferred that x is not redefined in-between $\tilde{t}_j$ and $\tilde{t}_{k+1}$. Otherwise, the same redefinition must have occurred in the original run. The value equivalence at the $(k+1)$th entries can be derived from the equivalence at the jth entries.

Similarly, we can prove the equivalence when the reference is through a pointer, i.e. $*x$.

Finally, it can be inferred that the same expression must yield an equivalent value if all references in it have equivalent values, assuming only the supported operations. □

According to the theorem, replayability is not guaranteed in our model in the presence of overflow. In real-world scenarios, however, if the overflow is a stack overflow, due to the determinism of variable layouts, we can always replay. For heap overflows, although we may not encounter the original failure, we very likely encounter a memory failure in a different place due to the non-determinism of heap allocations. We can attach the replay to a memory error detection tool, such as valgrind's *memcheck*, to identify heap overflows.

Note that traces and canonical pointer values just facilitate our definitions and proofs. The technique does not need to trace executions or compute canonical pointer values.

7. Aggressive Reduction

Although we divide executions into relatively independent units, non-trivial pointer dependences may limit the possible reduction. The reduction scheme in Fig. 9 is sometimes too rigid. Namely, the scheme requires that pointers used in equivalent evaluation points are defined at equivalent points with equivalent canonical values. However, as long as the pointer is pointing to the same variable or to the same offset of a region with the same size, the places they are defined are irrelevant because the value stored at the pointed-to address can be either restored from the log (if defined external to the unit) or recomputed (if defined internally). Moreover, we need to ensure that pointers to the same variable or to the same region in the original run retain such aliasing relations during replay. Otherwise, undesirable overwriting and different results in pointer comparison may occur so that the replay fails to make progress.

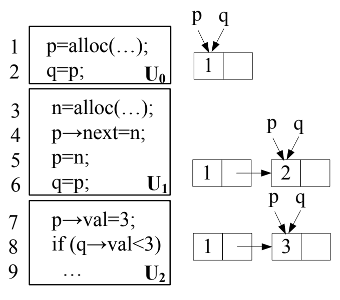

```
p=alloc(…);
q=p;                U0
n=alloc(…);
p→next=n;
p=n;
q=p;                U1
p→val=3;
if (q→val<3)
  …                 U2
```

Figure 13. According to the reduction rules in Fig. 9, the log is not reducible. However, U_0 and U_2 are a replayable reduction assuming the criterion is in U_2.

[2] We assume the variable-address mapping α is static.

Consider Fig. 13. Three units are presented on the left. The execution maintains a linked list, the state of which is presented on the right at the end of each unit. Pointers p and q always point to the tail of the list. Assume the criterion is in U_2. Because $U_2 \xrightarrow{p,q} U_1 \xrightarrow{p} U_0$, we cannot achieve reduction by rules in Fig. 9. However, U_0 and U_2 compose a replayable reduction because the region pointed to by p at the end of U_0 is compatible with that at the end of U_1, and the aliasing relation of p and q is the same at the end of the two units. In other words, the state of the list at the end of U_0 is sufficient to drive the execution in U_2. Note that if after U_0 p and q were pointing to different regions, the predicate at line 8 would take a different branch, making the reduction not replayable.

In the following, we formally define compatibility.

Definition 2. [COMPATIBILITY]
Let X be a reduction and T be the sub-trace corresponding to X in the original run. X is compatible if and only if for each $U \in X$, the following conditions are satisfied.

1. *for each read to an address a logged in U (hence its definition must be external to the unit) for a pointer value $\texttt{pnt}(b,z,f)$, there must be a closest preceding write to a in T with the canonical value $\texttt{pnt}(b',z,f)$.*
2. *for each pair of reads from addresses a_1 and a_2 in U with pointer values $\texttt{pnt}(b,z,f_i)$ and $\texttt{pnt}(b,z,f_j)$, respectively, the closest preceding writes to a_1 and a_2 in T must have the values $\texttt{pnt}(b',z,f_i)$ and $\texttt{pnt}(b',z,f_j)$.*

In condition 1, the read would receive its value from the closest preceding write in T instead of the original write during replay. This dictates the two writes have the same size and offset. Condition 2 dictates the region level aliasing be retained. Note that the pointer offsets f_i and f_j may be different. Consider the code snippet within a unit "`1. x=p+2; 2. *x=9; 3. ...=*q;`". Assume p and q are defined externally to the unit, pointing to the same region and with offsets 0 and 2, respectively, so they do not alias. Variable x is defined internally. After line 1, x and q point to the same address and hence the dereference at line 3 has the value 9. To ensure faithful replay, we must ensure p and q point to the same region.

Theorem 2. *A compatible reduction is replayable.*

The proof is omitted for brevity.

Search for Compatible Reduction. Given a criterion, there may be multiple compatible reductions. We use a BFS algorithm to find a compatible reduction. In particular, given a criterion, the algorithm first includes only the units dictated by structural constraints (the first unit and all loop exit units) and tries to replay with such a log. If replay fails (gets stuck), the algorithm further includes units that are 1 pointer dependence edge away, then 2 dependence edges away, and so on, until a compatible reduction is found.

8. Optimizing Instrumentation

To reduce logging overhead, we limit instrumentation to the data structures and variables that may cause inter-unit dependences. We further remove unnecessary instrumentation through static analysis. The analysis is similar in spirit to *definite assignment* analysis that decides if a local variable must have been initialized. The analysis computes the set of data structure fields and variables that must have been either read or written before a program point within a unit. Such information can be used to discover redundancy. For instance, instrumenting an object field read is not needed if the field must have been read or written before. Details are elided.

9. Handling Concurrent Programs

Our technique supports concurrent programs. The system currently supports only a single core due to limitations of the underlying *jockey* infrastructure [18]. More discussion resides in Section 10. We leave supporting multiple cores to future work. Presently, we discuss the extensions to support concurrent programs.

Maintaining Unit Id. The rules in Fig. 8 assume sequential semantics. Since units do not interleave, we can use a unit id variable across the entire program, which is not sufficient for concurrent programs. We consider two possible threading models. In the first one, an annotated loop does not spawn any thread but rather resides in a thread. We use a thread local unit id for the loop. In the other model, threads may be spawned inside an annotated loop. We need to properly attribute thread execution with the unit id of the loop. We use a thread local unit id, which is inherited from the parent when a thread is spawned.

Remote Definitions. With sequential semantics, we identify redundant reads and writes by comparing the current unit id and that stored in μ_r and μ_w. In the presence of thread interleaving, the value of a memory read may be defined remotely by a unit in a different thread. In such a case, the read should be logged even if it has a local definition in the same unit. To handle such cases, we use a pair comprising a thread id and a thread local unit id to denote a unit globally in the masks μ_r and μ_w. Upon a shared variable write, both masks are updated. If the remotely defined value is read for the first time, the difference between the global id in the masks and the current id would entail logging the read. This allows us replay by value. Replay by dependence is supported by recovering the total order of memory reads and writes for the same variable, which is easily achievable by inspecting the log. Note that the logging rules in Fig. 8 don't need to change except the above extensions.

$\mathbf{T}_1$	$\mathbf{T}_2$	$\mu_r(x)$	$\mu_w(x)$
1. **[UNIT]** `while` {			
2. $\boxed{\text{...=x}}$;		$\langle T_1,0\rangle$	
	10. **[UNIT]** `while` {	$\langle T_1,0\rangle$	
	11. $\boxed{\text{x=...}}$;	$\langle T_2,0\rangle$	$\langle T_2,0\rangle$
3. $\boxed{\text{...=x}}$;		$\langle T_1,0\rangle$	$\langle T_2,0\rangle$
4. ...=x;		$\langle T_1,0\rangle$	$\langle T_2,0\rangle$
	12. ...=x;	$\langle T_1,0\rangle$	$\langle T_2,0\rangle$

Figure 14. The accesses to x are interleaved in two threads. The $\mu_r(x)$ and $\mu_w(x)$ columns show the state of the masks **after** each statement execution.The boxed executions are logged.

Consider the example in Fig. 14. Two threads interleave. Statement 2 is logged according to rule [LE-Log]. Statement 11 is logged according to rule [LS-Log-Wrt]. Note, it updates both $\mu_r(x)$ and $\mu_w(x)$. Statement 3 is logged according to rule [LE-Log] even though there is a unit-local write earlier in the same thread. The remotely defined value is hence logged. Statements 4 is not logged because $\mu_r(x)$ is equivalent to the current id (by [LE-NoLog]). Statement 12 is not logged because $\mu_w(x)$ is equivalent to the current id (by [LE-NoLog]). Intuitively, the remote reads in-between 11 and 12 do not modify the value of x and there is no need to log.

10. Evaluation

Our system makes use of LLVM, *jockey*, and *pin*. We use LLVM to analyze and instrument programs. The logging runtime works through *jockey*, an application level logging and replay tool [18]. It works directly on binaries by rewriting the instruction sequences that make syscalls to intercept the calls. This design makes it very efficient; HP used it for debugging distributed systems. Previously, *jockey* logged only syscalls and signals. We extend it to log memory accesses, directed by instrumentation inserted by the LLVM pass. *Jockey* currently only supports one core. The *jockey* replay component is insufficient to support our technique as it cannot intercept memory accesses. We implement our replay component in *pin*, a dynamic instrumentation tool. Using *pin* provides flexibility for integrating reduction with future dynamic analysis. Reduction is implemented in C++. The overall implementation is 10k LOC.

Applications	LOC	threads	Bug description
Apache-2.0.48	157K	16	#1: Atomicity violation (21287)
			#2: Unprotected buffer
			#3: Cache size problem (21285)
BerkeleyDB-4.7.25	172K	5	#1:Failure in leader election
			#2:Panic caused by out-dated msgs
Squid-2.3.4	62K	1	Buffer overflow (4148)
MC-4.5.55	106K	1	Buffer overflow (8658)
W3M-0.5.2	51K	1	Out of memory (492290)
VIM-7.0	230K	1	Hangs (100%CPU usage)
DC-1.3	9.5K	1	Segmentation fault (135029)
YAFC-1.1.1	40.6K	1	Segmentation fault

Table 1. Application and bug description

Applications	annotated loops	data structure	func.	instrumen-. tation	instrmt. after opt.
Apache	2	32	227	2,333	1,775
BerkeleyDB	3	47	1,371	24,338	15,319
Squid	1	21	401	2,877	2,232
MC	1	18	372	1,967	1,460
W3M	1	12	408	5,506	3,609
VIM	1	25	1,319	12,040	8,375
DC	1	2	44	641	521
YAFC	1	8	287	1,785	1,337

Table 2. Static instrumentation.

All experiments were conducted on an Intel L2400 1.66GHz CPU with 3GB of RAM running Linux-2.6.11. We evaluate our technique on a number of real world programs and real bugs from them as the reduction criteria. Table 1 presents the programs and the bugs. `BerkeleyDB` is a distributed database. `Squid` is a proxy server. `MC` is a user interactive file explorer. `W3M` is a text-based web browser. `Vim` is an advanced text editor and `Dc` is a calculator. `Yafc` is a FTP client.

The instrumentation results are presented in Table 2. It presents the number of annotated loops, the data structures and functions that may cause inter-unit dependences, and the instrumentations before and after the static optimization (Section 8). Observe that only a few loops need to be annotated. These loops further call other functions to carry out computation. Some of the functions may cause interesting dependences. Although our technique only instruments access points of certain data structures in certain functions, the number of such points is large due to the program size. The static optimization effectively reduces many of them (1/3 for the `BerkeleyDB` and `VIM`). It is worth mentioning that the large number of static instrumentation points do not induce high runtime overhead because we also avoid redundant logging at runtime. Figure 15 shows an annotation example from the `Apache web server`. `Apache` is a multi-threaded application and two loops were annotated. It is easy to figure out these loops because they are in the main body of each thread.

App	time (sec)	w/o opt	stat	dyn	stat+ dyn	rednt. access	jockey
Apache	210.0	2.8%	1.77%	0.97%	0.17%	87.01%	3.46%
DB	48.3	9.55%	8.47%	5.92%	4.20%	95.61%	2.94%
Squid	79.04	30.6%	12.1%	2.23%	1.1%	88.62%	2.53%
MC	58.1	29.7%	27.5%	3.35%	2.26%	93.13%	3.25%
W3M	N/A	N/A	N/A	N/A	N/A	87.13%	N/A
VIM	55.26	42.15%	34.0%	5.3%	3.93%	92.05%	0.94%
DC	35.63	85.32%	43.93%	13.62%	4.8%	96.8%	7.11%
YAFC	63.03	23.75%	17.25%	4.02%	1.82%	97.17%	3.48%

Table 3. Logging overhead

Table 3 presents the runtime overhead. In this experiment and the next space overhead experiment, we use test inputs provided with the programs if available or randomly generated inputs otherwise. We will also show results on real world workloads later. The 2nd column shows the native execution time. Columns 3-6 present the overhead of our technique over the plain jockey logging with different configurations: without any optimizations (i.e. log all accesses of the relevant data structures), static optimization only, i.e. removing instrumentation through static analysis (Section 8), dynamic optimization only, i.e. avoiding redundant logging during execution via μ_r and μ_w, and both optimizations. The 7th column presents the accesses identified as redundant by the dynamic optimization. The last column presents the *jockey* overhead (without our technique) for reference. The overhead of our technique is very small after both static and dynamic optimizations, but dynamic optimization is the more effective of the two. Its effectiveness is supported by the number of accesses that are found to be redundant. If an access is redundant, we only pay the cost of a few (usually 2) extra memory accesses and one comparison. Otherwise, we have to update μ_r and/or μ_w, store the access information into a buffer, which will be flushed to the log file later. Note that `W3M` is a user-interactive application without a batch mode, thus we could not measure its run time.

App	jockey log		our log w/o opt.		our log with opt.	
	entry	size(MB)	entry	size(MB)	entry	size(MB)
Apache	29k	8.79	58,826k	379.82	7,644k	47.24
DB	539k	6.3	11,447k	85.86	502k	3.43
Squid	734k	22.18	44,838k	265.46	5,101k	30.35
MC	255k	12.9	10,499k	67.94	715k	4.34
W3M	14K	4.42	6,483K	42.34	834K	5.09
VIM	151k	4.14	49,970k	123.48	3,974k	23.27
DC	1,035K	12.45	24,970K	160.55	846K	5.04
YAFC	1,324K	49.11	10,428K	63.95	295K	1.78

Table 4. Recording space overhead

Space overhead is presented in Table 4. We present both the number of log entries and the log size. For comparison purposes, we separate a log file into the *jockey* log and our log. Observe that our log size is comparable to the *jockey* log size for most cases except `apache` and `VIM`, which have a lot of memory accesses with effects crossing unit boundaries. Our log has more entries, but each entry has a smaller size as it only records a memory access.

Table 5 presents reduction effectiveness. In the experiment, we weave inputs used in the previous experiment with failure inducing inputs. For UI programs, the failures are induced by a sequence of user actions. We interleave the sequence with the (much longer) original inputs. For example in `MC`, the bug is triggered by following a directory path to a specific file and opening the file. We interleave the failing sequence with the original normal actions such as browsing different directories and opening other files. For server programs, failures are triggered by setting specific configurations at the beginning and providing specific requests at the end. Note that playing the last few events in the *jockey* log does not work due to dependences. More discussion can be found in the Vim case study.

Columns 2-4 present logs before reduction including the number of units (not event entries), *jockey* and our log sizes. Columns 5-7 present the reduction result if all dependences are considered. Columns 8-10 present the result if only pointer dependences are considered. Columns 11-15 present the result of compatible reduction. That is, we conduct breadth first search along pointer dependences, looking for compatible reduction. Column 14 shows the depth of the BFS search. Observe that pointer dependence reduction sometimes can achieve good reduction. The last column shows the log reduction time including the replay time needed during re-

```
697  static void *listener_thread(apr_thread_t *thd, void * dummy) {
..
729  [UNIT] while(1) {
730            if (requests_this_child <= 0) {

..
896        } // while end
..
910  } // listener_thread function end

[server/mpm/worker/worker.c]
```

(a) Listener thread annotation

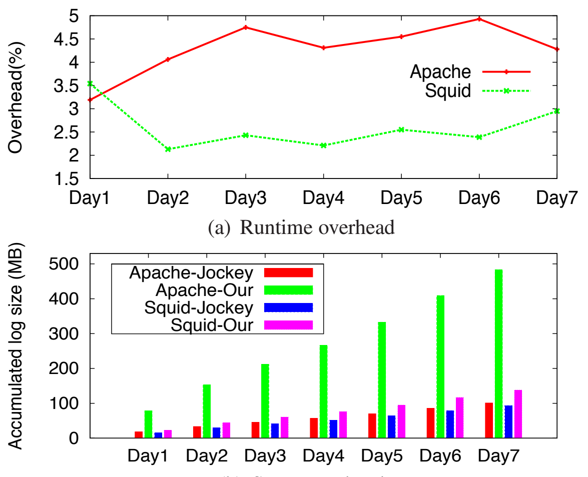
```
923  static void *worker_thread(apr_thread_t *thd, void * dummy) {
..
946  [UNIT] while(!workers_may_exit) {
947            if(!is_idle) {

..
1003       } // while end
..
1015 } // worker_thread function end

[server/mpm/worker/worker.c]
```

(b) Worker thread annotation

Figure 15. Loop annotation example (Apache)

Bugs	log b/f reduction			reduced log w. all deps			reduced log w/o search			reduced log with search				
	units	jockey (MB)	ours (MB)	units	jockey (MB)	ours (MB)	units	jockey (MB)	ours MB	units	jockey (MB)	ours (MB)	BFS level	Reduction time(sec)
Apache#1	14,989	8.55	47.1	14,989	8,55	47.1	14,975	8.53	47.1	16	0.14	0.51	1	5.9
Apache#2	14,991	8.54	47.4	14,991	8.54	47.4	14,989	8.53	47.4	40	0.29	1.34	1	6.4
Apache#3	15,005	8.54	47.3	15,005	8.54	47.3	15,004	8.54	47.3	20	0.17	0.69	1	6.1
DB#1	2,033k	144.9	25.7	832	5.85	0.122	10	0.07	0.0015	2	0.06	0.0007	1	5.6
DB#2	459k	37.6	8.36	54	3.52	0.047	8	0.44	0.005	4	0.42	0.0018	1	5.2
Squid	8,000	22.2	30.4	8,000	22.2	30.4	8,000	22.2	30.4	91	0.34	0.35	6	24.8
MC	5,348	13.4	4.35	5,348	13.4	4.35	14	1.25	1.12	5	0.69	0.93	1	5.4
W3M	961	1.67	3.98	961	1.67	3.98	961	1.67	3.98	2	0.03	0.021	1	4.9
VIM	2,544	4.14	23.27	2,544	4.14	23.27	2,369	4.02	22.66	12	0.035	0.279	1	5.8
DC	6,580	12.45	5.04	6,580	12.45	5.04	6,580	12.45	5.04	14	0.94	0.65	1	1.4
YAFC	196	49.11	63.95	196	49.11	1.78	196	49.11	1.78	4	0.11	0.001	1	3.5

Table 5. Log reduction results



(a) Runtime overhead

(b) Space overhead

Figure 16. Runtime and space overhead with real-world workload.

Unit 1: first unit, need to include
Unit 2 : turn on syntax highlights
Unit 3 : open file1
Unit 44 : open file2 **(bug triggering file)**
Unit 508 : *unnecessary unit*
Unit 1066 : open file3
Unit 1187-1188 : *unnecessary units*
Unit 2489 : open file4
Unit 2541 : go back to file2 (from memory buffer)
Unit 2542 : *unnecessary unit*
Unit 2544 : goto the bug triggering line in file 2. **Failure!**

Figure 17. Reduced log of Vim

duction. The replay time of the reduced log is similar to the reduction time and elided. The dominant factor of replay time is the infrastructure overhead of Pin. The search algorithm (compatible reduction) can always achieve good results. The reduced logs are very small, leading to very short replay times. Furthermore, we want to point out that by paying the one time reduction cost, the programmer has a much smaller run to analyze for arbitrary number of times. Hence, the benefit may go beyond the savings compared to the original run. Moreover, the level of search tends to be small. We suspect that many memory states are compatible, allowing us to find one quickly. Note that small search depths do not mean our technique only acquires units towards the end of execution. Recall that our algorithm (Section 7) starts with a skeleton of units based on structural constraints. More units (and their skeletons) are added through search. Even the initial skeleton may contain units distributed along the whole execution. Furthermore, traversing one dependence edge can reach distant units. For instance, a unit close to the end may likely depend on a unit near the beginning.

Practicality Study With Real Workload. In order to evaluate the practicality of our technique, we study its efficiency and effectiveness on the two server programs `apache` and `squid` with real-world workloads. Specifically, we acquired the high level web request log for our institution's web-site for one week. We wrote a script to regenerate the workloads for 1-7 days and fed them to the two server programs. At the end of each workload, we also supplied the failure inducing requests to trigger the failure. The average runtime overhead and aggregated space overhead are presented in Figure 16. Observe that the runtime overhead is more or less consistent. The space overhead is reasonable for a few days' execution. The replayable, reduced logs (not shown) are consistently small. These results show the practicality of our technique.

Case Study. We present the `VIM` case in detail. Fig. 17 shows the reduced version of a large log. The failure occurs as follows. Unit 1 (U1) is the first unit, containing execution from the beginning to before the first request loop iteration. At U2, the user enables syntax highlighting, a precondition of the bug. Next, four files are opened with file editing actions in-between the openings (note that the unit ids of open units are not contiguous). `VIM` allocates a memory buffer for each file, so after U2489, there are four buffers. Here, the user switches back to file2, which is loaded from the buffer. At U2544, the user scrolls down to a specific line. When the syntax highlighter tries to parse the line, the bug is triggered (`VIM` hangs).

The reduced log includes 12 units. The skeleton contains the failing unit U2544 and the first unit U1. The skeleton is not replayable. With one step pointer dependence edge traversal, the 12 units are included. In particular, the dependences between file open units are through the memory buffer data structure `xfilemark`. These units construct a compatible memory state for the execution of U2544, and hence leads to successful failure replay. In particular, the failure demands the four files be loaded otherwise the memory buffer state would be different, leading to control flow difference and hence mis-alignment between the control flow and the replay log. Furthermore, it demands U2 to activate syntax highlighting. We also manually inspect the reduced log and find that 4 units are unnecessary. In other words, the ideal minimal reduced log contains 8 units. The 4 units are included because they are also reachable in one dependence edge and they do not change compatibility.

From their ids, observe that the units are sparsely distributed along the entire execution. A naive idea of replaying the last a few units would not work. We further inspect all the reduced logs and find that the second unit of the reduced log (the first unit is always included in the reduced log) is indeed in the early stage of the original executions except the `W3M` bug and `BerkeleyDB` bug #1.

11. Related Work

Execution fast forwarding (EFF) [19, 25] also has the goal of reducing logs. EFF does not instrument the program during the logging phase. Instead, it analyzes dependences between event entries offline, either through static analysis or by replaying the whole execution once to detect dependences, which is too expensive (usually 10X slowdown). EFF does not guarantee replayability. In this paper, we show that with small cost, we can acquire a reducible log on the fly. Reduction can be achieved by analyzing the log, without static analysis or replaying the full execution. We also observe that EFF is too conservative in considering all dependences, often leading to very little reduction. In contrast, we weave value and dependence based replay to a novel and highly effective scheme.

Language based replay [21] allows users to replay a program component (module). It uses profiling to find a replay cut to minimize logging efforts. Whereas the technique can be considered a reduction on the program dimension, our reduction is on the temporal dimension, and we believe the two dimensions are orthogonal. Moreover, to use their technique, the user has to know about where a bug might be to do selectively logging. In comparison, our logging is general. We also handle failures that cross components. SCARPE [10] is a similar component based technique for Java. Subgroup replay [23] groups processes and records only intergroup messages.

Checkpointing [4] is a standard approach to avoid replaying a whole execution. Incremental checkpointing [3, 15] avoids overhead from capturing memory snapshots. Language based checkpointing [22] allows users to checkpoint at arbitrary program points with contexts. Our technique is complementary to checkpointing, e.g., our technique can further reduce an execution between checkpoints. Furthermore, we allow fine-grained reduction at the unit (loop iteration) level.

There are software based replay systems that record individual memory accesses and their happens-before relations [5]. Such systems induce substantial runtime overhead. There has been substantial work on software based record and replay for parallel and distributed systems [2, 6, 11, 17, 18]. These systems only perform coarse-grained logging at the level of system calls or control flow and hence are not sufficient for reduction. Hardware based logging and replay [7, 8, 12, 14] can faithfully replay executions. While such techniques are effective, they demand special hardware.

In recent years, significant progress has been made in testing and debugging concurrent programs [1, 9, 13, 16]. These techniques search for a failure inducing schedule given certain inputs. In the future, we plan to leverage these techniques to generate reducible logs on multiple cores.

Delta debugging [24] reduces a failure inducing input by repeatedly running the program on subsets of the input. The process could be expensive for long execution.

12. Conclusion

We propose a compiler based technique that generates a reducible replay log. The technique divides an execution into units, mainly iterations of event processing loops. We instrument programs instrument to collect minimal additional information like memory accesses into the replay log. Given a criterion, reduction can be achieved through analyzing just the log. Our technique is automated, only requiring annotating the event processing loops. It is highly efficient: the average runtime overhead is 2.6% and the additional space consumption is comparable to logs containing only syscalls and signals. Our reduction and replay scheme is also novel. It seamlessly weaves value based replay and dependence based replay to achieve both great reduction and faithful reconstruction of memory state. Our results show that we can reduce executions with up to 2,033K units to less than 91 units.

13. Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This research is supported in part by the National Science Foundation (NSF) under grants 0917007, 0845870 and 0834529. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] G. Altekar and I. Stoica. Odr: output-deterministic replay for multicore debugging. In *SOSP'09*.

[2] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. Traceback: first fault diagnosis by reconstruction of distributed control flow. In *PLDI'05*.

[3] G. Bronevetsky, D. Marques, K. Pingali, and R. Rugina. Compiler-enhanced incremental checkpointing. In *LCPC'07*.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[5] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE'08*.

[6] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI'08*.

[7] D. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two hardware-based approaches for deterministic multiprocessor replay. *Communications of the ACM*, 52(6):93–100, 2009.

[8] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ISCA'08*.

[9] P. Joshi, C. S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI'09*.

[10] S. Joshi and A. Orso. Scarpe: A technique and tool for selective capture and replay of program executions. In *ICSM'07*.

[11] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATEC'05*.

[12] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *ASPLOS'09*.

[13] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI'07*.

[14] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS'06*.

[15] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *PLDI'94*.

[16] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. Lee, S. Lu, and Y. Zhou. Pres: Probabilistic replay with execution sketching on multiprocessors. In *SOSP'09*.

[17] M. Ronsse, K. D. Bosschere, M. Christiaens, J. C. d. Kergommeaux, and D. Kranzlmüller. Record/replay for nondeterministic program executions. *Communications of the ACM*, 46(9):62–67, 2003.

[18] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05*.

[19] S. Tallam, C. Tian, X. Zhang, and R. Gupta. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *ISSTA'07*.

[20] L. D. Wittie. Debugging distributed c programs by real time reply. In *PADD'88*.

[21] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang. Language-based replay via data flow cut. In *FSE'10*.

[22] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *FSE'07*.

[23] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker. Mpiwiz: subgroup reproducible replay of mpi applications. In *PPOPP'09*.

[24] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE'02*.

[25] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *FSE'06*.