Dual Execution for On the Fly Fine Grained Execution Comparison

Dohyeong Kim[1] Yonghwi Kwon[1] William N. Sumner[2] Xiangyu Zhang[1] Dongyan Xu[1]

[1]Department of Computer Science, Purdue University, USA
[2]School of Computing Science, Simon Fraser University, Canada
[1]{kim1051,kwon58,xyzhang,dxu}@purdue.edu [2]wsumner@sfu.ca

Abstract

Execution comparison has many applications in debugging, malware analysis, software feature identification, and intrusion detection. Existing comparison techniques have various limitations. Some can only compare at the system event level and require executions to take the same input. Some require storing instruction traces that are very space-consuming and have difficulty dealing with non-determinism. In this paper, we propose a novel dual execution technique that allows on-the-fly comparison at the instruction level. Only differences between the executions are recorded. It allows executions to proceed in a coupled mode such that they share the same input sequence with the same timing, reducing nondeterminism. It also allows them to proceed in a decoupled mode such that the user can interact with each one differently. Decoupled executions can be recoupled to share the same future inputs and facilitate further comparison. We have implemented a prototype and applied it to identifying functional components for reuse, comparative debugging with new GDB primitives, and understanding real world regression failures. Our results show that dual execution is a critical enabling technique for execution comparison.

Categories and Subject Descriptors D.2 [*Software Engineering*]

General Terms Algorithms

Keywords Execution Comparison, Dynamic Analysis

1. Introduction

Execution comparison techniques compare multiple executions from the same program or highly similar programs to identify state differences including control flow differences and variable value differences. Execution comparison has been used to debug sequential, concurrent, and regression failures, by reasoning about the causality between execution differences and input differences [50], thread scheduling differences [47], and syntactic differences among program versions [8, 36, 41], respectively. It has also been used to identify and extract functional features for legacy software reuse [25]. For example, to identify the "send email" functionality from an email client, two executions are compared. In one execution, an email is composed and sent, whereas in the other, the same email is composed but not sent. Execution comparison is also used in malware behavior analysis [24]. Advanced malware often has logic to decide whether or not to activate its payload depending on the environment such as the platform, the running applications on the victim machine, the current date and time, and the presence of debuggers or virtual machines. To understand the activation logic and the payload, malware is executed in different environments and the resulting executions are compared. Software diversification creates executables with different structures, e.g. different stack layouts [38]. At runtime, multiple such versions are executed simultaneously. The executions are compared to detect intrusion because an exploit to one version often crashes others.

Execution comparison can be classified into *online* and *offline* techniques. In online comparison, the executions are compared on-the-fly. The intrusion detection technique from Salamat in 2009 is one such online approach [38]. However, such comparison is at the system event level. In particular, the executions of the multiple diversified versions are driven by the same input event sequence and the output event sequences are then compared. This technique assumes all executions consume the same input, which does not hold for debugging, where executions may have different paths and hence consume different sequence of input events. Most execution comparison techniques are offline [24, 25, 36, 47]. They first collect traces and compare them offline. A prominent challenge is hence the space required to store such traces, which can grow to a few GB within a few milliseconds of execution.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..

http://dx.doi.org/10.1145/2694344.2694394

Since executions are performed independently, many execution differences are caused by nondeterminism such as input events being received at different times. These differences are not helpful in understanding functional differences. Note that many existing logging and replay techniques [15, 20] cannot both preclude nondeterminism and allow for functional differences. Viennot et al. [45] can handle functional differences but it requires explorations and it cannot be used in online analysis. For UI applications, comparing their executions also requires repeated manual effort.

In this paper, we propose a novel *dual execution* engine to overcome the aforementioned limitations. It performs on-the-fly instruction level comparison of two executions and only stores traces for parts of executions that are different. There are several challenges in on-the-fly execution comparison. The executions may diverge because of (1) different inputs, (2) different outcomes from interactions with the environment, and (3) nondeterminism in the program. Our technique allows the executions to differ and consume different inputs. It suppresses the differences caused by interactions with the environment and nondeterminism. In particular, it has three execution modes. Initially, the two executions run in a *coupled* mode, in which they are synchronized and one of them, the *slave*, receives most of its system events from the other, the *master*, instead of from the environment. When the user indicates that different inputs should be provided, the two executions get into the *decoupled* mode, in which they execute independently. After introducing the differences, the user indicates the two executions should be coupled again such that they can share the same future inputs to avoid unnecessary nondeterminism. Since the two executions may be at different points when the coupling signal is received, in the *resynchronization* mode, the engine blocks the faster execution until the other one catches up. After that, the two execute in the *coupled* mode again. Since the two executions then have different states, even though they execute in the *coupled* mode, their paths and system event sequences may differ, so our engine detects and handles all these differences.

Our contributions are summarized as follows.

- We have proposed the novel dual execution technique that allows on-the-fly fine-grained execution comparison. It allows executions to take different inputs.
- We have addressed a set of underlying technical challenges such as how to synchronize two executions at the instruction level; how to share system events such that nondeterminism can be suppressed; how to prevent sharing in some cases such that the slave can still have its independent visible interface; how to resynchronize the two executions after they are decoupled; how to handle signals and threading.
- We have implemented a prototype and applied it to three applications: functional component extraction, on-the-fly comparative debugging with new GDB primitives, and understanding regression failures. The results show that the engine is a critical enabling technique. It also substantially reduces the space and time overhead compared to offline comparison.

```
1  busy_cue() {
2    if (status_message_remaining()) ...
3  }
4  status_message_remaining() {
5    return display_time - time() + min_time > 0;
6  }
```

Figure 1. Nondeterminism in `pine`.

2. Motivation

In this section, we use a real world example to explain why dual execution is preferable in execution comparison and illustrate some of the technical challenges in dual execution.

Suppose we want to identify the functional components that change an email subject and log a sent email in `pine`, an email client. In `pine`, a user can get to the email composition interface through a sequence of menu operations, where he/she can provide the email body, subject, and recipient. He/she can also choose whether a copy of the email will be saved in the `sent_mail` file after it is sent, through the `ctrl-r` hot-key. During the whole process, `pine` periodically pulls incoming emails from the server through a timer control. While an email is being sent, `pine` shows busy messages on the screen.

To identify the desired components, the user provides two executions. In the first, he/she composes an email and sends it, and a copy of the email is saved by default. The second execution is almost identical, except that the user changes the subject and reconfigures `pine` so that a copy is not saved.

We first use an offline technique similar to that of Kim et al. [25], in which the two executions run independently, and we collect instruction level traces for offline comparison. We observe the following problems. *First*, we must repeat the almost identical sequence of user interactions. We must be very careful to not introduce any human error. For example, in the second execution, when we try to type the same email, suppose we mistype a character and use `backspace` to fix it. Although the two emails are identical after the mistake is fixed, the instruction level trace faithfully records the interaction error, which will be identified as a difference during comparison.

Second, even if we manage to avoid introducing human error, there is substantial low level nondeterminism, e.g. from timers, that leads to unnecessary execution differences. Fig. 1 shows a code snippet from `pine` that has nondeterministic behavior. `busy_cue()` is a function that shows a busy message on the screen. Before showing a message it checks whether there already is a message by calling `status_message_remaining()` at line 2, which checks whether the current message is shown on screen for at least a minimum amount of time (line 5). The program behavior is thus dependent on execution timing that

```
1  pico() {
2    while(...) {
3      c = GetKey();
4      if (c == empty || time_to_check())
5        check_new_mail();
6      execute (c);
7    }
8  }
9  GetKey() {
10   if (ReadyForKey(STDIN, timeout))
11     return read(STDIN);
12   else
13     return empty;
14 }
```

Figure 2. Input handling loop in `pine`.

is nondeterministic. There are additional such timing related behaviors in `pine`. We show in Section 5 that execution differences caused by nondeterminism can be as large as half of the overall differences.

Third, we observe that the trace for even one of these executions is over 30GB even though the execution is already very small. It is not surprising given the large number of instructions executed within a second by a modern CPU. Storing and processing such huge trace files is a heavy burden even for modern systems. Note that the two traces are mostly identical.

Next, we use our dual execution engine. Initially, the engine spawns two executions of `pine` at the same time. Then, we only interact with one of the executions, called the *master*. All the interactions between the master and the environment, including user interactions, are relayed to the *slave*. This allows us to avoid repeating the same error-prone user interactions. In addition, the two executions run exactly the same way, without any differences caused by nondeterminism. Since only differences are recorded, we also avoid tracing in this phase.

After composing the email, we press a predefined hot-key. Now we can control the master and the slave separately. We provide different subjects to the two executions and set the save_to_sent_mail option differently. After that, we couple the two executions again by pressing another hot-key. Once again we only interact with the master to confirm sending the email and terminate the program. In this phase, since the engine aligns the two executions at the instruction level and executes them in locksteps, instruction level differences are detected on the fly and recorded. The resulting trace file is only less than 90MB.

We note several technical challenges. *First*, our system relays system events in the master to the slave, but we cannot relay every event. For example, if the slave did not execute the write() system call but rather simulated it using a relayed result during coupled execution, it could not show any interface and thus we could not provide different inputs when decoupled. The engine must identify the events that can be relayed.

Second, when we indicate our intent to recouple the executions after providing different inputs, the engine cannot

```
1  pico();
2  ...
3  call_mailer();
4  ...
5  if (fcc)
6    write_fcc(sentmail);
```

Figure 3. Send-mail function from `pine`.

simply resume relaying events because the two executions may be in different stages due to the different inputs. Fig. 2 shows a code segment in which `pine` receives user inputs. Lines 2-7 use a loop to handle user inputs. At line 3, it reads a key from the user. At lines 4-5, if a certain amount of time has passed, the program checks whether there is a new email. At line 6, the program processes the keystroke. Therefore, depending on the time we spent providing the different inputs, the two executions will be in the different instances of the loop upon recoupling. They may even be in different child functions of the pico() function. The engine needs to resynchronize the two executions at the instruction level.

Third, even though the engine manages to recouple the two executions, they may execute differently due to the input differences. Fig. 3 shows another code snippet from `pine`. At line 1, the program executes pico() to allow the user to compose the email. It also contains the confirmation menu. Recoupling happens inside pico(). At line 3, the program calls call_mailer(), which constructs the email packet and sends it to the SMTP server. After that at line 5, the program checks fcc, which corresponds to the save_to_sent_mail option. If it is set, `pine` makes a copy of the sent mail (line 6). Since we set fcc differently in the two executions, one execution will execute write_fcc(), but the other will not. Note that write_fcc() relies on several events such as file open and file write. Therefore in this case, the engine needs to detect such differences and provide the events appropriately.

3. Design

Our discussion follows the order of the three execution modes: the *coupled* mode in which the master and the slave share system events; the *decoupled* mode in which they execute independently; and the *resynchronization* mode in which the faster execution is blocked until the slower one catches up, when the user wants to recouple the two executions.

3.1 Coupled Execution Mode

The master and slave are in coupled mode when they start and also after different inputs are provided and the executions are resynchronized. Note that after differences are introduced, the two executions may take different paths and have different syscalls, even though they are resynchronized. Thus we focus on detecting and handling such differences that may prevent the executions from sharing events.

We first present the semantics of the master and slave executions and then discuss the monitor that coordinates their behavior.

Table 1. Semantic Rules for Master Execution.

Instruction	Action	Rule
regular instruction $y=^{l}$ instr(op, x_1,...x_n)	execute the instruction; send_trace_entry($\langle INSTR, l, \sigma[y]^1 \rangle$);	M-INSTR

$y=^{l}$ syscall (sid, x_1,...x_n);	send_trace_entry($\langle SYSCALL, l, sid, \sigma[x_1], ..., \sigma[x_n] \rangle$);
	execute the system call;
	send_syscall_outcome($\sigma[y]$);
		M-SYSCALL

1. σ stands for the store that maps a variable to a value.

$TraceEntry$	$t ::=$	$\langle INSTR, l, v \rangle$ $\mid \langle SYSCALL, l, sid, P \rangle$
$Label$	$l ::=$	$\{l_1, l_2, l_3, ...\}$
$Value$	$v ::=$	$\{true, false, 0, 1, 2, ...\}$
$SysCallId$	$sid ::=$	$\{1, 2, ...\}$
$SysCallParameters$	$P ::=$	$\overline{v}$

Figure 4. Trace Syntax

3.1.1 Master Execution in the Coupled Mode.

In coupled mode, the master tracks each instruction execution and sends a trace entry to the monitor. It performs all syscalls faithfully and sends syscall parameters and outcomes to the monitor, who decides if the syscall outcomes need to be relayed to the slave. The semantics is shown in Table 1. We model instructions into two kinds: system calls and others (or *regular instructions*). A regular instruction, with opcode op, takes n operands and produces the result in y. According to rule M-INSTR, the instruction is first executed, then a trace entry is sent to the monitor.

Trace Entry Syntax. As shown in Fig. 4, there are two kinds of trace entries, identified by the types $INSTR$ (denoting regular instructions) and $SYSCALL$, respectively. A regular instruction entry consists of the label (or the program counter) of the instruction and the left hand side value of the instruction. A syscall entry consists of the label, the syscall id, and the parameters. □

Upon a syscall invocation (rule M-SYSCALL), the master first sends the corresponding trace entry containing the syscall id and the parameters to the monitor. It then executes the syscall and sends the outcome to the monitor.

3.1.2 Slave Execution in the Coupled Mode.

As shown in Fig. 2, the slave handles a regular instruction in the same way as the master. For a syscall, the slave sends the trace entry containing the parameters to the monitor, which allows the monitor to decide if the slave should *copy* the syscall outcome from the master or *execute* the syscall. After that, the slave receives the decision from the monitor and acts accordingly. recv_syscall_outcome() is a blocking call, preventing the situation in which the slave execution is faster than the master and manages to perform a syscall before the master reaches the corresponding syscall.

3.1.3 Monitor in the Coupled Mode.

The monitor is the most complex component. It is responsible for synchronizing the two executions, comparing their trace entries on the fly, and instructing the slave how to handle its syscalls.

Algorithm 1 Monitor Algorithm for the Coupled Mode

Input: a pair of executions e_m, e_s

Definition: $\mathcal{I}_{m/s}$ denotes the execution index of the current trace entry from master/slave;

```
MONITOR (e_m, e_s)
 1: while e_m and e_s are not finished do
 2:   t_m = recv_trace_entry(e_m)
 3:   I_m^p = I_m
 4:   I_m = update_index(t_m.l, I_m)
 5:   t_s = recv_trace_entry(e_s)
 6:   I_s^p = I_s
 7:   I_s = update_index(t_s.l, I_s)
 8:   if t_m.type==SYSCALL then
 9:     y_m = recv_syscall_outcome(e_m)
10:     if policy(t_m.sid)==COPY && t_m.P==t_s.P
        then
11:       send_2_slave(y_m)
12:     else
13:       send_2_slave(EXEC)
14:   if t_m.l != t_s.l then
15:     /* Lines 16-21 run parallel with lines 22-27*/
16:     while I_m != dyn_ipdom(I_m^p) do
17:       record(t_m, nil)
18:       t_m = recv_trace_entry(e_m)
19:       I_m = update_index(t_m.l, I_m)
20:       if t_m.type==SYSCALL then
21:         y_m = recv_syscall_outcome(e_m)
22:     while I_s != dyn_ipdom(I_s^p) do
23:       record(nil, t_s)
24:       t_s = recv_trace_entry(e_s)
25:       I_s = update_index(t_m.l, I_s)
26:       if t_m.type==SYSCALL then
27:         send_2_slave(EXEC)
28:   if t_m.v != t_s.v then
29:     record(t_m, t_s)
```

Background: Execution Indexing. Our execution synchronization is built on *execution indexing* (EI) [48]. For completeness of the paper, we briefly introduce the EI technique. It computes a unique identifier for an execution point called an *execution index*, or index. Execution points across multiple runs align when they have the same index. The index of an

Table 2. Semantic Rules for Slave Execution.

Instruction	Action	Rule
regular instruction $y =^{l}$ instr(op, x_1,...x_n)	execute the instruction; send_trace_entry($\langle INSTR, l, \sigma[y] \rangle$)	S-INSTR
$y=^{l}$ syscall (sid, x_1,...x_n)	send_trace_entry($\langle SYSCALL, l, sid, \sigma[x_1], ..., \sigma[x_n] \rangle$); t=recv_syscall_outcome(); `if` ($t ==$ EXEC) execute the system call; `else` $\sigma[y] = t$;	S-SYSCALL

$ExecIndex \quad \mathcal{I} ::= \bar{l}$

update_index : $Label \times ExecIndex \rightarrow ExecIndex$

update_index$(l, nil) = l$ update_index$(l, \mathcal{I}_{head} \cdot l_{tail}) = \begin{cases} \mathcal{I}_{head} \cdot l_{tail} \cdot l & \texttt{if } l \text{ control dep. on } l_{tail} \\ \text{update_index}(l, \mathcal{I}_{head}) & \texttt{otherwise} \end{cases}$

dyn_ipdom : $ExecIndex \rightarrow ExecIndex$

dyn_ipdom$(\mathcal{I}_{head} \cdot l_{tail}) = \mathcal{I}_{head} \cdot l$, with l the immediate postdom. of l_{tail}

Figure 5. Execution indexing primitives. The syntax is presented in boxes.

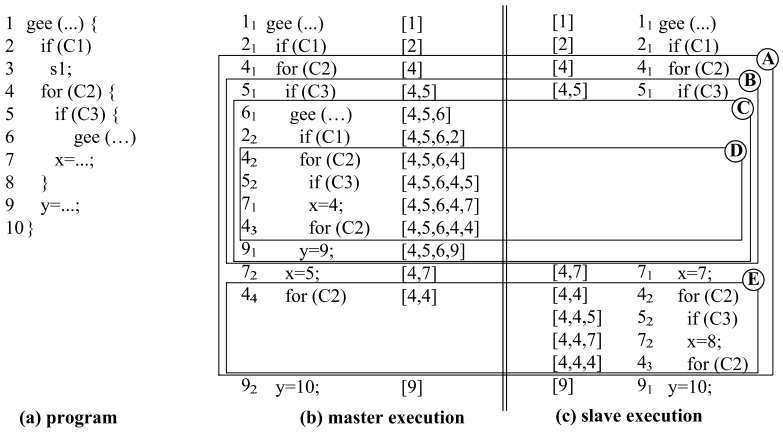

Figure 6. Example for execution indexing and synchronization.

execution point is analogous to the calling context of the point, but instead of describing the nesting of function calls, the index describes the nesting of the control dependence regions of the execution point. Note that multiple instances of an instruction may share the same calling context (and hence calling contexts cannot be used to distinguish the instances), but they must have different control dependence region nestings.

The syntax of execution indexing and the primitives to compute indices are described in Fig. 5. Syntactically, an index is a sequence of labels (or PCs). It is constructed by the update_index() primitive, which takes the label of the current execution point and the index of the previous point, produces the new index. According to the rules, if the previous index is nil, the resulting index is the label. If the previous index is not nil, and if the current label l is control dependent on the tail label l_{tail} of the previous index, indicating l is nesting in the region of l_{tail}, l is appended to the previous index. Otherwise, labels at the tail of the previous index are popped one by one until l finds its control dependence region.

Consider Fig. 6. A code snippet is presented in (a) and two executions are presented in (b) and (c). The two executions differ due to some state differences introduced earlier. The indices of the execution points are presented in the center for easy comparison. Lets look at the master execution (b). Initially, the index of the first instance of statement 1 $\mathcal{I}(1_1) = [1]$ as its previous index is nil; then update_index(2, $\mathcal{I}(1_1)$)= update_index(2, $[1]$)= update_index(2, nil)=$[2]$, because 2 is not control dependent on 1; update_index(5, $\mathcal{I}(4_1)$)= update_index(5, $[4]$)=$[4, 5]$ because 5 is control dependent on 4. Intuitively, it means 5 is nesting in the region of 4, denoted by the box Ⓐ. Similarly, $\mathcal{I}(5_2) = [4, 5, 6, 4, 5]$, denoting 5_2 is nesting inside regions Ⓓ, Ⓒ, Ⓑ, and then Ⓐ. Note that our system only maintains the index for the current execution point, similar to maintaining the call stack.

The indices for the slave (c) are similarly computed. By comparing the indices, only $1_1, 2_1, 4_1, 5_1, 7_2, 4_4, 9_2$ in the

master have correspondence in the slave. We can see the essence of EI is to align the control dependence regions. For example, the alignment of 5_1 in both executions indicates the Ⓑ regions align. But inside the region, different branch outcomes are taken such that there are no further alignments. □

Synchronizing Master and Slave Executions. The monitor component continuously updates the current indices of the master and the slave based on the trace entries received. The indices allow the monitor to achieve lockstep synchronization of the two executions. Initially, the two executions follow the same path such that they are perfectly synchronized, indicated by identical indices. When a predicate is encountered but its branch outcomes are different in the two runs, the paths start to diverge. As such, the monitor decouples the two executions, allowing them to proceed independently to the immediate postdominator of the predicate. Note that since the predicates in the two runs share the same immediate postdominator, they are perfectly synchronized again.

Details are in Algorithm 1, in which a **while** loop continuously processes trace entries until the two executions finish. In the loop, it first receives an entry from the master and updates the current index for the master (lines 2-4). We use $t_m.l$ to represents the label field of the trace entry. Lines 5-7 do the same thing for the slave. In normal cases, the two entries from the two respective runs have the identical labels. But if the previous label is a predicate with different branch outcomes, the pair of entries will have different labels (line 14). In this case, the monitor processes trace entries from the two executions in parallel (lines 16-21 and 22-27), until the *dynamic* immediate postdominator (IPDOM) of the previous predicate is encountered (line 16 and line 22). Note that the trace entries are recorded (line 17 and line 23) because they denote *control flow differences*. In contrast, in lines 28-29, if the values are different, the two entries (with the same label) are also recorded. Note that the monitor must leverage indices to identify the *dynamic* IPDOM (lines 16 and 22). It cannot simply looks for the next occurrence of the *static* IPDOM, because in the presence of recursive functions, the next occurrence of the static IPDOM may not mean the end of the control dependence region. In Fig. 5, dyn_ipdom($\mathcal{I}$) computes the dynamic IPDOM of $\mathcal{I}$ by replacing the tail label, a predicate label, with its static IPDOM label, demanding the prefixing control dependence regions remain the same.

Example. Consider the example in Fig. 6. The first four pairs of trace entries have identical labels. Only the entries for 5_1 are recorded as the branch outcomes are different. The monitor detects that the fifth pair have different labels, i.e. 6_1 from the master and 7_1 from the slave. As such, it computes dyn_ipdom($\mathcal{I}[5_1]$)=dyn_ipdom($[4,5]$)=$[4,7]$. Thus, both executions proceed to index $[4,7]$, that is, 7_2 in (b) and 7_1 in (c). Note that if the static IPDOM 7 were used, due to the recursive call of gee(), we would mistakenly consider 7_1 to be the end of region Ⓑ. □

Handling Syscalls. The monitor also needs to handle syscalls because we want the two executions to share the sequence of external inputs as much as possible to reduce nondeterminism and hence the meaningless differences caused by such nondeterminism. It also reduces error-prone manual efforts. From the earlier discussion, we know both the master and the slave send their syscall parameters to the monitor. In addition, the master also sends its syscall outcome. For a syscall that occurs in both executions, the monitor compares their parameters, e.g. the size of a file read. If they are identical, the monitor relays the syscall outcome from the master to the slave. In the case that they are different due to differences introduced earlier, the monitor instructs the slave to execute the syscall instead of copying from the master.

In Algorithm 1, when the monitor receives a pair of syscall trace entries (line 8), it first retrieves the syscall outcome from the master (line 9). Then it consults a *policy table* that defines the actions for different kinds of syscalls. While the details of the policy table are discussed in Section 4, all we need to know now is that there are two possible actions for each syscall ID. COPY indicates the slave should copy the result from the master, and EXEC indicates the slave should execute the syscall. At line 10, the algorithm checks whether the action is COPY and if the parameters are identical. If so, it relays the outcome from the master (line 11). Otherwise, it instructs the slave to execute (line 13). When the two executions take different branches, the monitor retrieves and discards the syscall outcome from the master (lines 20-21), and instructs the slave to always execute its syscall (lines 26-27).

```
1  /*different file names are provided*/
2  filename = read();
3  ...
4  while (1) {
5    menu = selectMenu();
6    if (menu == MENU_QUIT)
7      quit();
8    else if (menu == MENU_WRITE_MESSAGE) {
9      message = read();
10     if (exists(filename))
11       rename(filename, filename + ".bak");
12     fd = open(filename);
13     write (fd, message);
14   }
15 }
```

Figure 7. Example for syscall handling.

Example. Consider an example in Fig. 7. When the user selects the menu item `write message`, the program reads a message and writes it to a file with a name provided earlier. If the file already exists, the program first makes a backup. Suppose the user wants to compare two executions with two different file names `A` and `B`, and there is already a file with name `B`. Fig. 8 shows the master and the slave executions. Note that since the file name is different at 2_1, the following syscalls at 10_1, 11_1, 12_1, and 13_1 in the slave are executed instead of copied, which correctly exercises the intended different semantics. In contrast, the syscalls at 5_1 and 9_1 are copied. □

Master		Slave		
2_1	filename = read();	2_1	filename = read();	E
4_1	**while** (1)	4_1	**while** (1)	
5_1	menu = selectMenu()	5_1	menu = selectMenu()	C
6_1	**if** (menu == ...)	6_1	**if** (menu == ...)	
8_1	**else if** (menu == ...)	8_1	**else if** (menu == ...)	
9_1	message = read();	9_1	message = read();	C
10_1	**if** (exists (filename))	10_1	**if** (exists (filename))	E
-		11_1	rename(filename, ...);	E
12_1	fd = open(filename);	12_1	fd = open(filename);	E
13_1	write (fd, message);	13_1	write (fd, message);	E

Figure 8. Executions of the example in Fig. 7. The last column shows if the slave executes (E) the syscall or copies (C) result. The boxed entries are affected by the differences from 2_1.

3.2 Decoupled Execution Mode

When the user wants to decouple the two executions and provide different inputs, he/she presses `ctrl-c` in the master execution, which sends an interrupt to the master. Note that the user does not interact with the slave execution (in most cases). The master checks for the interrupt before executing a system call. If it was received, the master informs the monitor. When the slave is about to execute the corresponding system call, the monitor notifies the slave to start executing its syscalls. In other words, the two start to interact with the environment directly. Both continue to send their trace entries to the monitor to update their indices.

3.3 Resynchronizing Master and Slave Executions

After providing the different inputs, the user presses another hot-key in both the master and the slave to indicate that he/she wants to resynchronize the two executions such that they can share the same input events again to reduce nondeterminism and manual interactions. However, when the interrupts are received, the two executions may be at different locations. The monitor needs to determine which execution is faster and block it until the slower one catches up. After the two resynchronize, they resume in the coupled mode.

We leverage execution indices to determine which execution is ahead of the other. Intuitively, an index represents the nesting of control dependence regions. Two indices sharing some common prefix means that the current execution points in the two runs are nesting in a common set of control dependence regions, called *aligned regions*. By comparing the relative positions of the two points inside the aligned regions, we can decide which execution is ahead. Consider Fig. 6. Point 7_1 in (c) is ahead of 5_1 in (b) because $\mathcal{I}_m(5_1) = [4, 5]$ and $\mathcal{I}_s(7_1) = [4, 7]$; they share a prefix $[4]$ and 7 is ahead of 5 inside the region of 4. Similarly, 7_2 in (c) is ahead of 5_2 in (b). In particular, their indices share the prefix $[4]$; 7_2 in (c) is in the region denoted by index $[4, 4]$, i.e. Ⓔ representing the second iteration of the loop, whereas 5_2 in (b) is in the region denoted by $[4, 5]$, i.e. Ⓑ representing the **if** statement at line 5 in the first iteration of the loop). Ⓔ is ahead of Ⓑ.

```
1  while (1) {
2    if (select(STDIN, ...) == SUCCESS)
3      /* process the user input */
4      s1;
5      ...
6  }
```

Figure 10. Event handling loop example.

The ahead_of() primitive in Fig. 9 defines how to decide index order. If the two indices share the same head label, it recursively checks the order of their tails. If the head labels l_1 and l_2 are different, and there is a path from l_2 to l_1, the first index is ahead. In Fig. 6,

$$\begin{aligned}\text{ahead\_of}(\mathcal{I}_s(7_2), \mathcal{I}_m(5_2)) &= \text{ahead\_of}([4, 4, 7], [4, 5, 6, 4, 5]) \\ &= \text{ahead\_of}([4, 7], [5, 6, 4, 5]) = \mathit{true}.\end{aligned}$$

There are cases where neither execution is ahead of the other if they are in the different branches of a predicate. In this case, the monitor will resynchronize the two executions at the dynamic IPDOM of the predicate.

Dealing With Event Handling Loops. When the user sends a resynchronization signal, the two executions are likely inside some event handling loop, which receives and handles external events. During the independent executions in decoupled mode, the two runs may have received different events, e.g. different numbers of key strokes, causing them to execute a different number of iterations. Index based resynchronization recognizes that the execution that iterated more is ahead[1]. As such, it tries to execute the other that iterated less, but in fact the other execution cannot make progress as it has received all the needed external inputs and expects no more. When the user signals resynchronization, he/she knows that the two executions have received the different inputs, and they should start to receive the same inputs again from then on. Hence, the iteration number differences of the event loop are not important. We introduce a normalization step to remove the excessive entries (in the indices) corresponding to unnecessary iteration differences. The primitive is defined in Fig. 9. It takes three indices, with the first two requiring normalization and the third an auxiliary parameter whose initial value is nil, and it produces two normalized indices. The auxiliary parameter represents the current common prefix during computation. According to the rules, it initially compares the two head labels. If different, the differences are not caused by the event loop, so it returns the two input indices as normalized (rule N-INIT-NEQ). Rules N-INIT-EQ and N-EQ-HEADER detect equivalent heads and move it to the tail of the common prefix. In rule N-RM-FIRST, if the two heads are different but the head of the first index is identical to the tail of the prefix, indicating repetitive entries, i.e. loop iterations, the head is removed. Rule N-RM-SECOND is symmetric. Rule N-END states the termination condition, in which the difference is not caused by repetition. The final normalized indices are the common prefix concatenated with the two current indices.

Example. Fig. 10 shows a very simple event loop. Assume it iterates two times and stops at line 5 in the master when

[1] In indexing, an iteration nests within the region of the previous iteration such that the one iterating more has a longer index.

$$\boxed{\mathsf{ahead\_of} : ExecIndex \times ExecIndex \rightarrow Boolean}$$

$$\mathsf{ahead\_of}(l_1 \cdot \mathcal{I}_1, l_2 \cdot \mathcal{I}_2) = \begin{cases} false & \texttt{if } l_1 \neq l_2 \ \wedge \text{ there is not a path from } l_2 \text{ to } l_1; \\ true & \texttt{if } l_1 \neq l_2 \ \wedge \text{ there is a path from } l_2 \text{ to } l_1; \\ \mathsf{ahead\_of}(\mathcal{I}_1, \mathcal{I}_2) & \texttt{otherwise i.e., } l_1 \equiv l_2 \end{cases}$$

$$\boxed{\mathsf{normalize} : ExecIndex \times ExecIndex \times ExecIndex \rightarrow ExecIndex \times ExecIndex}$$

$$\mathsf{normalize}(l_1 \cdot \mathcal{I}_1,\ l_2 \cdot \mathcal{I}_2,\ nil) = \begin{cases} \langle l_1 \cdot \mathcal{I}_1,\ l_2 \cdot \mathcal{I}_2 \rangle & \texttt{if } l_1 \neq l_2 \quad \text{[N-INIT-NEQ]} \\ \mathsf{normalize}(\mathcal{I}_1,\ \mathcal{I}_2,\ l_1) & \texttt{if } l_1 \equiv l_2 \quad \text{[N-INIT-EQ]} \end{cases}$$

$$\mathsf{normalize}(l_1 \cdot \mathcal{I}_1,\ l_2 \cdot \mathcal{I}_2,\ \mathcal{I}_p \cdot l_p) = \begin{cases} \langle \mathcal{I}_p \cdot l_p \cdot l_1 \cdot \mathcal{I}_1,\ \mathcal{I}_p \cdot l_p \cdot l_2 \cdot \mathcal{I}_2 \rangle & \texttt{if } l_1 \neq l_2 \ \wedge\ l_2 \neq l_p \ \wedge\ l_1 \neq l_p \quad \text{[N-END]} \\ \mathsf{normalize}(l_1 \cdot \mathcal{I}_1,\ \mathcal{I}_2,\ \mathcal{I}_p \cdot l_p) & \texttt{if } l_1 \neq l_2 \ \wedge\ l_2 \equiv l_p \quad \text{[N-RM-SECOND]} \\ \mathsf{normalize}(\mathcal{I}_1,\ l_2 \cdot \mathcal{I}_2,\ \mathcal{I}_p \cdot l_p) & \texttt{if } l_1 \neq l_2 \ \wedge\ l_1 \equiv l_p \quad \text{[N-RM-FIRST]} \\ \mathsf{normalize}(\mathcal{I}_1,\ \mathcal{I}_2,\ \mathcal{I}_p \cdot l_p \cdot l_1) & \texttt{if } l_1 \equiv l_2 \quad \text{[N-EQ-HEADER]} \end{cases}$$

Figure 9. Resynchronization Primitives.

the user signals resynchronization, and it iterates four times and stops at 3 in the slave, yielding the indices $\mathcal{I}_m = [1, 1, 2, 5]$ and $\mathcal{I}_s = [1, 1, 1, 2, 4]$, respectively. Recall that loop iterations produce consecutive entries in the indices as an iteration directly nests in the region of the previous iteration. We have the following:

$$\begin{aligned} & \mathsf{normalize}(\mathcal{I}_m, \mathcal{I}_s, nil) \\ = \ & \mathsf{normalize}([1,2,5],\ [1,1,2,4],\ [1]) && \text{[N-INIT-EQ]} \\ = \ & \mathsf{normalize}([2,5],\ [1,2,4],\ [1,1]) && \text{[N-EQ-HEADER]} \\ = \ & \mathsf{normalize}([2,5],\ [2,4],\ [1,1]) && \text{[N-RM-SECOND]} \\ = \ & \mathsf{normalize}([5],\ [4],\ [1,1,2]) && \text{[N-EQ-HEADER]} \\ = \ & [1,1,2,5],\ [1.1,2,4]) && \text{[N-NED]} \end{aligned}$$

The rules applied are presented on the right. Note that at the second step, the heads of the indices are different, but the second head is the same as the tail of the prefix, indicating repetition. The head is thus removed. Observe that after normalization, the two executions are within the same control dependence region, and the master is ahead.

Overall Resynchronization Procedure. The procedure is informally described as follows. Assume $\mathcal{I}_m$ and $\mathcal{I}_s$ are the indices when a resynchronization signal is received. The monitor first normalizes $\mathcal{I}_m$ and $\mathcal{I}_s$. If one execution is ahead of the other, it is blocked until the other one catches up. If neither is ahead, meaning they are in the two branches of some predicate, the monitor allows both to execute independently until the dynamic IPDOM of the predicate.

4. Handling Practical Challenges

Syscall Policy for Slave Execution in Coupled Mode. In previous discussion, whether the slave should execute a syscall mainly depended on whether the syscall corresponded to one in the master and whether they had different parameters. However, the decision also depends on the type of the syscall. This is reflected by the policy table look up on line 10 of Algorithm 1. For example, we would like to execute user interface (UI) related syscalls such that the slave has its own UI.

We categorize syscalls into 7 groups. Table 3 presents the groups and their policies. There are two possible policies: *execute* and *copy*. The former means that the slave executes the syscall whereas the latter means that the slave copies the result from the master and does not execute the syscall.

Table 3. Category of syscalls and the default policy.'E' and 'C' stand for execute and copy, respectively.

Category	Description	Policy
Memory	Allocate/free memories (mmap(), munmap(), ...)	E
Process	Create/kill/block processes (fork(), clone(), ...)	E[1]
Open	Open a file descriptor (open(), create(), ...)	E
Input	Read data from a file/environment (read(), stat(), ...)	C
Output	Write data to a file (write(), ...)	C,E[2]
GUI	Special case of network syscalls with X-Windows server	E,C[1,3]
Utility	Nondeterministic library functions (time(), ...)	C

1. there is ID translation during execution.
2. outputs to stdout and UI are executed, others copied.
3. requests and replies are executed and events are copied.

Process management syscalls follow the execute policy. In addition, the slave and the master both manage a mapping between process IDs. Syscalls creating new processes such as fork() and clone() return different process IDs in the two executions. To prevent propagating these different values to other parts of the executions, we would like the corresponding processes in the two runs to have the same IDs. In particular, when clone() is called to create a thread, both runs assign the same logical thread ID to the newly created thread and map the real thread ID returned by the syscall to the logical ID. Our wrapper around clone() then returns the logical ID. Later, when the logical ID is used in other syscalls such as exit(), our wrapper maps it to the original real ID.

Open syscalls also use an execute policy because the descriptors they return may be used by other syscalls such as mmap or when the user introduces differences between the executions. It is possible that the master succeeds in opening a file but the slave fails to open the same file, for instance, when the master creates a file with the O_EXCL option first. In this case, the slave opens a new temporary file. Since I/O syscalls in the slave mostly copy their outcomes from the master, opening a different file will not affect the slave execution.

Most *input/output syscalls* follow the copy policy. One exception is that we *execute* output syscalls emitting to standard output or user interfaces to allow the slave to have its interface. In addition, both the master and the slave record

Table 4. Applications in feature identification.

Program	Feature Description
alpine-1	Send an e-mail
alpine-2	Create a directory in remote imap server
xv	Convert two different bitmap images to jpeg images
smbc	Create a directory in remote samba server
ncmpc	Add a song to playlist

the buffer content of an output syscall and meta information such as the definition point of the buffer, for further analysis, e.g. slicing.

GUI applications in Linux communicate with the X server through sockets. There are three types of messages: *requests*, *replies*, and *events*. Requests are sent from a client to the server to request a service such as creating a window and querying properties. Upon a request, the server sends the requested information with a reply. Events are sent by the server without the corresponding requests, denoting user interactions. Requests/replies are handled in a way similar to process management syscalls, which use the execute policy and ID translation, to provide the proper user interface. The master and the slave must map their window IDs to the same logical ID when the windows correspond to each other. Events have the copy policy, meaning the slave copies events from the master and ignores its own events.

The slave also copies signals from the master in coupled mode, ignoring its own signals except segfaults. We support threads by implementing a deterministic scheduler [31] that allows one thread to execute at a time. This suppresses nondeterminism caused by different thread schedules.

5. Evaluation

We implement a prototype on Pin [26]. It can work on stripped binaries as it can generate dynamic control flow graphs (for indexing). In our experiments, we first quantitatively evaluate the space and time savings by our technique in comparison with offline techniques. We then apply the prototype to three comparative analysis tasks.

5.1 Examined Comparative Tasks

Feature identification. As seen in Sec. 2, feature identification involves locating the portion of a program responsible for a feature [2, 11, 17–19, 34, 35, 46], and this can be done by comparing executions of a program over different inputs [25]. Table 4 presents the full set of examined feature identification tasks, which have appeared in published work [25].

Comparative Debugging. To help developers, we integrate our technique into `gdb` and provide a few new `gdb` primitives that allow developers to modify variables and compare the execution with changes to an execution without them. Details on the changes to `gdb` are presented in a later case study.

Understanding regression. We also apply our technique to executions of an old, working version of a program and a new, failing version to understand regressions.

All the examined bugs are real world bugs from [7, 41]. Table 5 describes them with GNU Savannah bug IDs if any.

Table 5. Subjects in comparative debugging and regression understanding. The regression bugs are tagged with *.

Program	Bug #	Description
grep-1	12128	Numeric parameters are incorrectly interpreted
grep-2	16567	-i option does not work with a regular expression
grep-3	27919	
grep-4	27919	
grep-5	21199	-w option causes incorrect search results
make-1	25493	Incorrectly handle dependencies in rules
make-2	18622	User-defined rules conflict with default rules
tar-1	508199	Cannot restore files from backup
tar-2	598636	Cannot handle broken symbolic links
tar-3	637085	Cannot handle longer filenames
grep-6*		Search incorrect if -i, -n options are used together
grep-7*	36567	-i does not work with multi-byte characters
grep-8*		
find-1*	34976	Fail to save working directory
find-2*	29949	-execdir does not change working directory
make-3*	31155	Incorrect order in parsing patterns
make-4*	39310	Commandline options are applied multiple times
rm*		-I, –interactive=once does not work same
seq*		[seq 1 3 1] treated as [seq 1 3]
cp*		–no-preserve=mode exits 1
cut*		Incorrect error message
expr*		Incorrect computation with negative value

5.2 Disk Usage and Performance

In this experiment, we examine the impact of dual execution on the running time and disk usage, when compared with offline techniques. We first collect traces of the two executions for all the subjects (in the three tasks). We then perform offline comparison. The results are used as the base line. We then use our prototype to generate difference traces on the fly and compare the results. We used an Intel Core i7 machine running Arch Linux 3.15.5 with 16GB RAM.

Table 6 presents the trace size comparison. The Traces column shows the size of full traces. The Diffs (W/O Dual Exec.) column shows the size after offline trace comparison. The Diffs (W/ Dual Exec.) column shows the size of difference traces generated online. The % Full column shows the size of the difference traces as a percentage of the full traces. Note that we do not use any trace compression technique. However, our technique is orthogonal to those techniques and users can combine the techniques to achieve smaller traces. Also many trace compression techniques targets specific trace abstractions but our technique can be used on a fine-grained trace including data and control dependencies.

Observe that using dual execution always produces much smaller (difference) traces than the *full traces* (5.41% on average). This is because the full traces include every instruction of both executions, while dual execution only records differences. For tasks comparing executions, there is often substantial similarity between the executions. There is also a lot of similarity across executions from initialization and configuration behavior that is irrelevant to the analysis tasks.

Dual execution also consistently produces smaller differences than the differences by offline comparison. This occurs because many applications have nondeterministic behaviors

Table 6. Trace size comparison.

Program	W/O Dual Exec.		W/ Dual Exec.	
	Traces	Diffs	Diffs	% Full
seq	62MB	15MB	15MB	24.19
make-3	1.8GB	409MB	358MB	19.42
make-4	4.6GB	887MB	886MB	18.81
grep-7	4.0GB	978MB	653MB	15.94
find-1	4.2GB	562MB	541MB	12.58
find-2	4.0GB	462MB	447MB	10.91
grep-8	4.0GB	432MB	424MB	10.35
grep-5	12.5GB	2.7GB	1.0GB	8.00
cut	79MB	6.0MB	6.0MB	7.59
grep-4	13GB	438MB	428MB	3.22
rm	3.8GB	117MB	117MB	3.00
xv	14GB	559MB	380MB	2.65
cp	68MB	1.4MB	1.3MB	1.91
tar-1	377MB	17MB	6.8MB	1.80
ncmpc	6GB	120MB	93MB	1.51
tar-2	457MB	6MB	4.0MB	0.88
smbc	55GB	627MB	432MB	0.77
make-1	15.7GB	1.5GB	105.2MB	0.65
expr	58MB	306KB	305KB	0.51
make-2	7.3GB	104.9MB	32.2MB	0.43
alpine-2	57GB	320MB	205MB	0.35
grep-3	14GB	41.2MB	40.7MB	0.28
grep-6	4.0GB	9.1MB	7.7MB	0.19
tar-3	11.4GB	35MB	6.4MB	0.055
grep-1	10.6GB	8.6MB	832KB	0.0074
alpine-1	60GB	344MB	170MB	0.0003
grep-2	14.8GB	3.4KB	1KB	0.000006
			Geometric Mean	5.41

Table 7. Execution time comparison.

Program	Native	W/O Dual Execution			W/ Dual Execution		
		Tracing	Comp	$Total_1$	$Total_2$	Tot_2/Tot_1	Tot_2/Nat
expr	.002s	3s	0.5s	4s	6s	1.50	3000
seq	.002s	4s	0.5s	5s	7s	1.40	3500
cut	.002s	4s	0.7s	5s	7s	1.40	3500
grep-6	.01s	10s	29s	39s	49s	1.26	4900
cp	.01s	4s	0.5s	5s	6s	1.20	600
make-3	.01s	19s	21s	40s	36s	0.90	3600
grep-7	.01s	11s	42s	53s	42s	0.79	4200
find-2	.03s	11s	39s	50s	38s	0.76	1267
find-1	.02s	12s	46s	58s	44s	0.76	2200
ncmpc	14s	1m 12s	1m 3s	2m 15s	1m 42s	0.76	*
tar-1	.03s	18s	1s	19s	12s	0.63	400
tar-2	.01s	20s	2s	22s	12s	0.55	1200
grep-5	.05s	1m 16s	1m 45s	3m 1s	1m 32s	0.51	1840
make-4	.01s	42s	38s	80s	40s	0.50	4000
grep-8	.01s	12s	45s	57s	28s	0.49	2800
rm	2s	11s	32s	43s	19s	0.44	*
tar-3	.02s	1m 0s	1m 58s	2m 58s	1m 15s	0.42	3750
xv	16s	2m 20s	3m 42s	6m 2s	2m 31s	0.42	*
grep-3	.03s	1m 10s	1m 9s	2m 19s	45s	0.32	1500
grep-2	.02s	1m 16s	1m 11s	2m 27s	47s	0.32	2350
grep-4	.02s	1m 6s	59s	2m 5s	38s	0.30	1900
make-2	.03s	52s	39s	1m 31s	27s	0.30	900
smbc	40s	5m 32s	7m 2s	12m 34s	3m 10s	0.25	*
grep-1	.02s	1m 0s	1m 15s	2m 15s	26s	0.19	1300
alpine-2	16s	5m 20s	6m 31s	9m 51s	1m 47s	0.18	*
alpine-1	12s	6m 42s	8m 59s	15m 41s	2m 14s	0.14	*
make-1	.01s	1m 28s	1m 59s	3m 27s	20s	0.10	2000
				Geometric Mean		0.49	2022

that the user cannot control, which introduce additional differences. For example, `alpine` reads and uses the system time quite often, causing a lot of non-deterministic differences. Dual execution eliminates such differences by sharing system events as much as possible.

Table 7 presents the time comparison. The Native column shows the original native execution time, i.e. sum of the master and the slave. Tracing and Comp present the time taken for whole execution tracing and offline comparison. $Total_1$ is their sum. $Total_2$ presents the time for dual execution. The last two columns shows the comparison. Running times of interactive programs are not comparable against Native, denoted by '*'. Entries in the table are sorted by Tot_2/Tot_1. A lower number means that dual execution improved the running time.

Observe that the time spent on dual execution is almost always lower than the time for full execution tracing and offline comparison, 0.49% on average. The benefits are more obvious for larger programs and longer runs. These improvements are mostly due to the smaller traces that are actually recorded when using dual execution. The slowdown of our system is 2022× relative to native on average. It is more suitable for inhouse testing and debugging.

5.3 Case Studies

5.3.1 Feature Identification

In this case study, we use our prototype to identify 5 functional features from 4 real world applications. We use our prototype to trace and compare executions and use an offline technique similar to [25] to identify functions from the trace. For each case, we compare two executions that exercise the feature with different inputs by using our prototype, and we identify the relevant differences, which are supposed to correspond to the features. For example, in alpine-1 case, we compare two executions sending two different mails. The differences between the executions should be the feature responsible for sending different emails if there is no nondeterminism that is not relevant to the input differences. Our prototype suppresses those nondeterminism. Once we get the changes, we consider the highest function(s) on the call graph that can cover all the differences to be the functional component for the feature. The differences generated by our tool allow us to precisely locate the correct functions. For example, all the differences in `xv` can be covered by two functions with the names of `LoadBMP()` and `SaveJPEG()`, which clearly indicate they are the intended functions. For `alpine`, we have located `call_mailer()` for email sending and `add_new_folder()` for directory creation. For `smbc`, `RcreateDir()`. For `ncmpc`, `mpd_async_send_command_v()`.

All these applications contain event handling loops that behave differently between two executions because different user input timings or packet arrival timings perturb the loop behavior, causing undesirable execution differences.

For the `xv` case, from the results in Table 6, 559MB-380MB=179MB trace differences are due to such nondeterminism, which largely originates from an event han-

```
1  _XReply() {
2      while (1) {
3          reply = poll_for_reply();
4          if (reply->sequence == expected) break;
5          ...
6  }
7  poll_for_reply() {
8      while (poll() == -1) ...;
9  }
```

Figure 11. Simplified event handling loop in libX11

dling loop for reply messages from the X server, as shown in Fig. 11. In the de-coupled mode, the different sequences of X messages make the loop at line 2 iterate different times. As a result, full traces cannot be properly aligned, leading to lots of undesirable differences during offline comparison. In contrast, the normalization step in the re-synchronization mode in our technique suppresses such differences.

5.3.2 Comparative Debugging Using GDB.

Table 8. New gdb commands supported by dual execution.

Command	Interface
dslice	dslice [instruction address] [instance]
dset	dset [variable] [value1] [value2]
dprint	dprint [variable]

We integrated our prototype with gdb and provided new debugging commands as shown in Table 8. dslice generates a dual slice [47] of the given instruction and instance. The slice contains execution differences causally related to the difference at the given slicing criterion. dset sets a variable in two executions to different values; dprint shows two values of a variable side by side. Basic commands such as setting a breakpoint and continuing the execution are applied to both executions by default. We then used the enhanced gdb to debug the 10 non-regression failures. Each required fewer than 15 manual steps to capture the causality of the failure. We also tried to use the vanilla gdb to achieve the same results for three cases, grep-5, tar-2, and tar-3, but failed due to the prohibitively tedious manual interactions involved.

Next, we present our experience with the grep-5 bug. The "-w" option in grep selects lines containing whole word matches. The buggy program prints out only a substring of a matched line. We launched the buggy program in our enhanced gdb. Two processes were automatically created and run in coupled mode. We set a breakpoint at prtext(), which printed the incorrect output. Few places in the immediate source code were affected by the "-w" option; the match_word variable was one of them. It had value 1 in the buggy run. We wanted to perturb its value and observe the effect, and more important, the causality. We used dset to set it to 0 and 1 in the two respective runs. At the breakpoint, we observed output differences: the execution with match_word set to 0 produced the whole line. We then used dslice to slice from this output difference. Fig. 12 presents the resulting dual slice. The results from the correct run are on the left side and

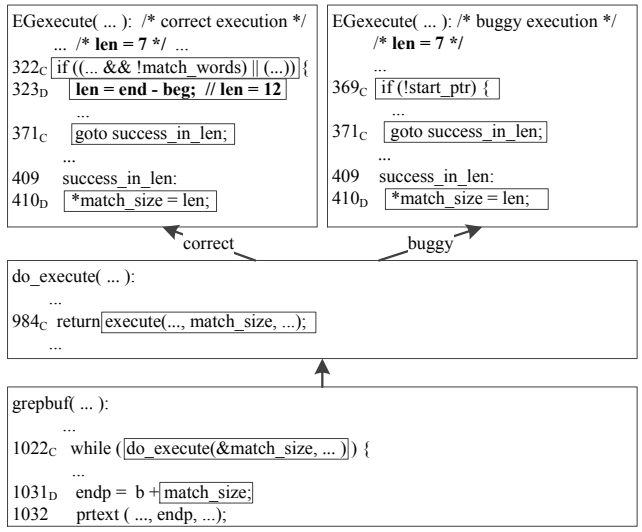

Figure 12. Slicing results for grep

those from the buggy run are on the right. do_execution() and grepbuf() are common to both executions, though they have different dependencies in EGexecution(). The subscript C and D on line numbers represent control and data dependencies respectively. The bug manifests when prtext() prints different results at line 1032 in grepbuf() because of the differences in the variable endp. The slice result shows that the two executions have different match_size values, 12 versus 7. It also shows that the control dependencies include do_execute(), execute(), and EGexecution(). Observe that in the correct run, len is defined at line 324, which eventually allows printing the whole line, whereas there is no such definition in the buggy run. Therefore, the root cause is that such a definition is missing when the option is set, which is confirmed by the bug report.

Next, we show our experience in doing the same comparative debugging with the vanilla gdb. We started two executions of the buggy program and attached them to two separate gdb instances. Then we set the match_word variable to 0 and 1 again. But the challenges lie in monitoring the propagation of the value differences. We first tried to single-step the two executions. But the definition point of match_word and the output point are separated by a substantial amount of computation in EGexecute(). Even worse, there were control flow differences due to our perturbations such that we had to somehow manually align the two executions. The process quickly became unmanageable. Another attempted option was to identify related variables and set break-points and watch-points. However, we could not solely use watch-points as many related variables were stack variables. However, using break points was also problematic. In particular, when we set a break point at the access to match_size in do_execute(), we found that the access occurs more than 200 times and only the 150th instance shows a difference across the two runs. The process is prohibitively tedious and error-prone. In con-

Table 9. Dual slicing regressions.

Program	# of Instr. in differences	# of Instr. in slice	Time
grep-6	44K	35K	0.1s
grep-7	15K	7.7K	16s
grep-8	213K	76K	6s
find-1	42K	2.0K	6s
find-2	172K	120K	5s
make-3	888K	127K	4s
make-4	753K	108K	3s
rm	3.4K	214	1.5s
seq	902	516	0.2s
cp	2179	332	0.05s
cut	952	541	0.05s
expr	787	62	0.03s

trast, our new commands and the underlying dual execution engine make interactive comparative-debugging feasible.

We also point out that the enhanced gdb is more flexible than a stand-alone slicing tool as it allows interactively perturbing program state at any point.

5.3.3 Understanding Regressions

In this case study, we applied an existing dual slicing [47] tool on the difference traces generated by the dual execution engine to understand regressions. As mentioned earlier, dual slicing computes the execution differences related to the difference at the slicing criterion. In the experiment, we use a text differencing tool to generate syntactic mappings between statements in the two program versions and propagate such mappings down to the instruction level to facilitate our engine. The slicing criteria are output differences. In the case of crashing bugs, they are the pointer dereferences. The results are shown in Table 9. The second column shows the total instructions in the difference traces, which are already a very small portion of the full traces (Table 6). The third column shows the slice size. Observe that the slices are much smaller than the differences for most cases. We also confirm that all of them include the root causes. We suspect the slice sizes can be further reduced if we use a better syntactic mapping algorithm. But that is beyond our scope.

6. Related Work

Execution comparison. Execution comparison is used in debugging [1, 3, 37, 50], concurrency failure understanding [47], vulnerability detection [24], and binary reuse [25]. Comparative causality [41] produces bug explanations by replacing program states on the fly. Sieve [36] compares traces from program versions to identify the root causes of regressions. Hoffman et al. proposed a semantic-aware trace analysis [22] for understanding executions, particularly identifying the cause of regressions. In comparison to these works, the dual execution engine avoids generating full traces. Instead, it performs online comparison and only records differences.

Execution Replication and Replay. Execution replication has been widely studied [4–6, 10, 13, 42]. The premise is similar to n-version programming [12], which runs different implementations of the same service specification in parallel. Then, voting is used to produce a common result tolerating occasional faults. Vandiver et al. [43] proposed a technique that handles Byzantine faults in database transaction processing using replicated systems. Chun et al. [14] run diversified replicas on multi-core processors to handle Byzantine faults. There are also many security applications [9, 16, 27, 38] of n-variant execution. McDermott et al. [28] proposed a defense technique based on logical replication. They re-execute commands on each replicated system and detect differences among the replicas. TightLip [49] runs a replicated process in parallel to an original process and analyzes the replica to prevent information leakage. There is also a large body of works in execution replay [23, 30, 32, 33, 39, 44] that aim to faithfully reproduce an execution. Compared to these works, our technique allows differences in executions and handles the complex consequences of these differences.

Viennot et al. [45] proposed a technique that replays events from one version of a program with another version of the program. It hence also allows sharing syscalls across different executions. However, it requires exploration steps to find the best replay. In contrast, our technique exploits fine-grained traces and can align executions on-the-fly.

Execution Alignment. Alignment techniques identify corresponding points [21, 21, 29, 48] and memory locations [40] across different executions. Xin et al. proposed Execution Indexing (EI) [48] to precisely locate corresponding points across executions. Our technique is built on EI. We have overcome many new challenges such as lockstep synchronization, resynchronization, and syscall dispatching for our purpose.

7. Conclusion

We develop a dual execution engine that allows two executions to proceed simultaneously in coupled mode, in which they share system inputs, or in decoupled mode, in which different inputs can be provided. On the fly comparison identifies instruction level differences between executions for later analysis. Our experiments on three comparative analyses demonstrate the engine is highly effective.

Acknowledgments

This research has been supported in part by DARPA under contract 12011593 and under cooperative agreement HR0011-12-2-0006, and by NSF under awards 0845870, 0917007, 1320326 and 1409668. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of DARPA and NSF.

References

[1] David Abramson, Ian Foster, John Michalakes, and Rok Sosič. Relative debugging: A new methodology for debugging scientific applications. *Commun. ACM*, 39(11):69–77, November 1996.

[2] G. Antoniol and Y.-G. Gueheneuc. Feature identification: a novel approach and a case study. In *Proceedings of the IEEE*

International Conference on Software Maintenance (ICSM), 2005.

[3] Ansuman Banerjee, Abhik Roychoudhury, Johannes A. Harlie, and Zhenkai Liang. Golden implementation driven software debugging. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 177–186, New York, NY, USA, 2010. ACM.

[4] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[5] Kenneth P. Birman. Replication and fault-tolerance in the isis system. *SIGOPS Oper. Syst. Rev.*, 19(5):79–86, December 1985.

[6] Dave Black, C. Low, and Santosh K. Shrivastava. The voltan application programming environment for fail-silent processes. *Distributed Systems Engineering*, 5(2):66–77, 1998.

[7] Marcel Böhme and Abhik Roychoudhury. Corebench: studying complexity of regression errors. In *ISSTA*, pages 105–115, 2014.

[8] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 15:1–15:16, Berkeley, CA, USA, 2007. USENIX Association.

[9] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replicæ for defeating memory error exploits. *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 0:434–441, 2007.

[10] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. Base: Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.*, 21(3):236–269, August 2003.

[11] Kunrong Chen and Vaclav Rajlich. Ripples: Tool for change in legacy software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2001.

[12] Liming Chen and A Avizienis. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pages 113–, Jun 1995.

[13] M. Chereque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active replication in delta-4. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 28–37, July 1992.

[14] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 287–292, Berkeley, CA, USA, 2008. USENIX Association.

[15] Asaf Cidon, Kanthi Nagaraj, Sachin Katti, and Pramod Viswanath. Flashback: Decoupled lightweight wireless control. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 223–234, New York, NY, USA, 2012. ACM.

[16] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[17] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 2006.

[18] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 2003.

[19] A.D. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2005.

[20] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[21] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately choosing execution runs for software fault localization. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 80–95, Berlin, Heidelberg, 2006. Springer-Verlag.

[22] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 453–464, New York, NY, USA, 2009. ACM.

[23] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 265–276, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 347–362, Washington, DC, USA, 2011. IEEE Computer Society.

[25] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. Reuse-oriented reverse engineering of functional components from x86 binaries. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1128–1139, New York, NY, USA, 2014. ACM.

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming*

Language Design and Implementation, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[27] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 115–124, New York, NY, USA, 2008. ACM.

[28] J. McDermott, R. Gelinas, and S. Ornstein. Doc, wyatt, and virgil: prototyping storage jamming defenses. In *Computer Security Applications Conference, 1997. Proceedings., 13th Annual*, pages 265–273, Dec 1997.

[29] Subrata Mitra, Ignacio Laguna, Dong H. Ahn, Saurabh Bagchi, Martin Schulz, and Todd Gamblin. Accurate application progress analysis for large-scale parallel debugging. In *PLDI*, page 23, 2014.

[30] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 73–84, New York, NY, USA, 2009. ACM.

[31] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, November 2007.

[32] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. *SIGPLAN Not.*, 41(11):229–240, October 2006.

[33] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM.

[34] Maksym Petrenko and VáClav Rajlich. Concept location using program dependencies and information retrieval (depir). *Information and Software Technology*, 2013.

[35] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 2007.

[36] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Sieve: A tool for automatically detecting variations across program versions. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 0:241–252, 2006.

[37] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[38] Babak Salamat. *Multi-variant Execution: Run-time Defense Against Malicious Code Injection Attacks*. PhD thesis, Irvine, CA, USA, 2009. AAI3359500.

[39] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM.

[40] William N. Sumner and Xiangyu Zhang. Memory indexing: Canonicalizing addresses across executions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 217–226, New York, NY, USA, 2010. ACM.

[41] William N. Sumner and Xiangyu Zhang. Comparative causality: Explaining the differences between executions. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 272–281, Piscataway, NJ, USA, 2013. IEEE Press.

[42] A Tulley and S.K. Shrivastava. Preventing state divergence in replicated distributed programs. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*, pages 104–113, Oct 1990.

[43] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Samuel Madden. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *ACM SOSP*, Stevenson, WA, October 2007.

[44] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst.*, 30(1):3:1–3:24, February 2012.

[45] Nicolas Viennot, Siddharth Nair, and Jason Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 127–138, New York, NY, USA, 2013. ACM.

[46] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. Improving feature location practice with multi-faceted interactive exploration. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, 2013.

[47] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 253–264, New York, NY, USA, 2010. ACM.

[48] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 238–248, New York, NY, USA, 2008. ACM.

[49] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 12–12, Berkeley, CA, USA, 2007. USENIX Association.

[50] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 253–267, London, UK, UK, 1999. Springer-Verlag.