Reverse Debugging with ReplayEngine™

Version 2020
February, 2020

PERFORCE

www.perforce.com

ACKNOWLEDGMENTS

Use of the Documentation and implementation of any of its processes or techniques are the sole responsibility of the client, and Perforce Software, Inc., assumes no responsibility and will not be liable for any errors, omissions, damage, or loss that might result from any use or misuse of the Documentation

PERFORCE SOFTWARE, INC. MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THE DOCUMENTATION. THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. PERFORCE SOFTWARE, INC. HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE DOCUMENTATION, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT SHALL PERFORCE SOFTWARE, INC. BE LIABLE, WHETHER IN CONTRACT, TORT, OR OTHERWISE, FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT, PUNITIVE, OR EXEMPLARY DAMAGES IN CONNECTION WITH THE USE OF THE DOCUMENTATION.

The Documentation is subject to change at any time without notice.

TotalView by Perforce
http://totalview.io

Contents

Understanding ReplayEngine

TotalView ReplayEngine: A New Paradigm in Debugging

The hardest step in locating software bugs centers on working backward from a failure to the error that caused it. Conventional debugging techniques do not make it easy to find the cause of an error as they allow you to control program execution only in the forward direction.

Instead of going back to the beginning to try to recreate the conditions of a problem, ReplayEngine lets you start from the point of failure and work backward in time to find the cause. Recreating the conditions of a crash, sometimes the hardest problem in conventional forward debugging, is no longer necessary. You can now move to locate errors that occurred long before the failure they caused.

ReplayEngine is embedded within TotalView which means you must know how to use TotalView to take advantage of it. TotalView documentation is available both in this help set and at https://docs.roguewave.com/en/totalview/current/.

How ReplayEngine Works

ReplayEngine lets you move backward in your program. To do this, it saves state information as your program executes. This information includes the order in which your program executes as well as changes to its data. When ReplayEngine is saving state information, it is in its *record mode*.

The saved state information is the program's execution *history*. You can save the execution history at any time and then reload the recording when debugging the executable in a subsequent session. See Saving and Loading the Execution History below.

Using a ReplayEngine command shifts ReplayEngine into its *replay mode*. In this mode, you can move to any previously executed statement. When you move to one of these statements, ReplayEngine displays its saved state information. The information you see in replay mode is identical to the information that you saw in record mode.

Most debugging commands work the same in replay mode as they do in record mode. Commands such as diving on a variable or setting a breakpoint work as you would expect them to. The debugging commands that do not work are those that change or alter a recorded state. Typically, these are commands that:

- Change a variable's value.
- Call functions that alter memory.
- Run threads asynchronously.

If your program calls a routine that displays information, the routine will not display this information. For example, suppose your program calls **printf()**. When the **printf()** is executed in record mode, it writes text. However, when the **printf()** is replayed, this text is not rewritten. Similarly, if your program unlinks a file in record mode; the file will not be linked before the unlink statement when you are in replay mode.

When executing in record mode, your program runs more slowly than if you were not using ReplayEngine. Usually, you will not notice the extra execution time. However, when you are in replay mode, the computational overhead required to recreate the program's state may be noticeable. When it needs extra time, ReplayEngine displays a dialog box that allows you to cancel the operation.

You can save a ReplayEngine recording to a Replay recording file and later reload it into TotalView for further analysis. These Replay recording files are then loaded into TotalView so the execution of your program can easily be examined and understood.

System Resources ReplayEngine Uses

ReplayEngine writes internal information in **/tmp.** Normally, very little space is used for this, but there are some situations where it can grow large, and if your system has a small **/tmp** area, ReplayEngine may fill it up. If this occurs, you can:

- Increase the amount of storage allocated to **/tmp.**
- Use the **TMPDIR** environment variable to point to another disk location.
- Define a special TotalView variable, **TVD_REPLAY_TMPDIR**, for ReplayEngine to use as the base directory for writing its temporary information. For example:

  ```
  setenv TVD_REPLAY_TMPDIR /home/user/smith/replayTempDir
  ```

  In some parallel environments (such as Cray), files on the compute nodes may reside on memory devices as opposed to physical storage devices, resulting in less memory being available for your application when temporary files are created. For this reason, it may be more efficient to define **TVD_REPLAY_TMPDIR** as a directory on a physical file system shared by the login node and the compute nodes.

  In the Cray environment, you must define **TVD_REPLAY_TMPDIR** prior to launching the MPI starter **aprun** or **srun**, or it will not be inherited by the TotalView server.

ReplayEngine also changes the amount of memory your program uses as it keeps history and state information in memory. For information on controlling the history information storage, see Setting Preferences on page 13.

While in replay mode. ReplayEngine creates extra processes, usually around ten, but you may see up to thirty. You should ignore these processes as they are only used buy ReplayEngine.

Replaying Your Program

Before you replay your program's statements, you must stop your program's execution. You can do this by halting your program, or TotalView can stop execution when your program encounters a breakpoint. When execution stops, TotalView ungrays the ReplayEngine buttons you can use on its tool bar (Figure 1)

Figure 1, ReplayEngine Tool Bar Commands.

The ReplayEngine commands are as follows:

- **Record**, a toggle that enables and disables ReplayEngine. See Enabling and Disabling ReplayEngine on page 9 for details.

- **GoBack**, which tells ReplayEngine to display the state that existed at the last action point. If no action point is encountered, ReplayEngine displays the state that existed at the start of its recorded history.
- **Prev**, which tells ReplayEngine to display the state that existed when the previous statement executed. If that line had a function call, **Prev** skips over the call.
- **Unstep**, which tells ReplayEngine to display the state that existed when the previous statement executed. If that line had a function call, ReplayEngine moves to the last statement in that function.
- **Caller**, which tells ReplayEngine to display the state that existed before the current routine was called.
- **BackTo**, which tells ReplayEngine to display the program's state for the line you select. This line must have executed prior to the currently displayed line. If you wish to move forward within replay mode, select a line and select the **Run To** button.
- **Live**, which tells ReplayEngine to shift from replay mode to record mode. It also displays the statement that would have executed if you had not moved into ReplayMode.
- **Save**, which saves the current replay recording session to a file. If the **Save** icon does not appear on the toolbar, then expand the Process window.

NOTE: The ReplayEngine tool bar commands appear only if you are using TotalView on a Linux-x86 (32-bit) or Linux-x86-64 machine. On these platforms, these buttons are permanently grayed out if you do not have a ReplayEngine license.

When you need to move forward within the program's history, you can use the **Step**, **Next**, **Run To**, and **Out** buttons. These commands do the same thing in replay or record modes.

You can also set breakpoints in previously executed statements. After setting a breakpoint, pressing the **Go** button will move you to that statement. You can transform a breakpoint to an eval point if the eval point uses simple expressions such as "`if (x==y+z) $stop`". You cannot, however, create barrier points.

If you reach the line that would have been executed if you hadn't gone into replay mode, you are automatically switched back to record mode and you can then resume program execution. You can also switch back to record mode by pressing the **Live** button.

Threads and Processes

ReplayEngine runs one thread at a time, and it decides which thread will run in a multi-threaded or multi-process program. In record mode, ReplayEngine saves state information for each thread as it executes.

The order in which threads originally execute cannot be changed when you are in replay mode. In replay mode, all actions that occur must be in the same order as previously occurred.

If you need to control the way threads execute, use the TotalView asynchronous threading commands while in record mode. Using these commands you can:

- Single-step a process or lockstep group.
- Hold threads so they do not run.

If you are in replay mode, you cannot hold a thread or a process as they run in the same order as they ran originally.

Attaching to Running Programs

If you attach to a program, ReplayEngine begins recording that program's execution at the time you attached to it. This means that you cannot go back further than when you attached to it.

Saving and Loading the Execution History

TotalView has support to save the current execution history at any given time to a recording file. The saved recording file can then be loaded into TotalView and all the replay options will be available to go back and forth within the time boundaries of the saved recording.

To save a recording, either select the **Save** button on the toolbar or the "**Save Recording File ...**" menu item in the File menu. In addition, you can use the CLI **dhistory** command:

dhistory -save *filename*

The ***filename*** can be either a path or a simple file name, in which case it is saved into the current working directory. If no ***filename*** is specified, the recording is saved in the current working directory as `replay_pid_hostname.recording`.

Load TotalView ReplayEngine saved recording files into TotalView as follows:

- At startup, using the same syntax as when opening a core file:

  totalview *executable recording-file*

  TotalView recognizes the recording file for what it is and acts appropriately.

- After TotalView is running, using the **dattach** command with option **-c**:

  dattach *executable* **-c** *recording-file*

- On the Root or Process window, by selecting "**Debug Core or Replay Recording File ...**" or on the Starting a Debug Session window, by selecting "**A core file or replay recording file**".

  Performing any of the above displays the dialog for selecting the record file and application used during the recording session when the recording session was saved.

  Again, TotalView recognizes it is dealing with a recording file.

Using ReplayEngine

There is very little difference between running TotalView and running ReplayEngine. With ReplayEngine enabled on a running program, use the special buttons **GoBack, Prev**, **UnStep**, **Caller**, or **BackTo** to go back in your program's history to the statement you wish to examine.

Enabling and Disabling ReplayEngine

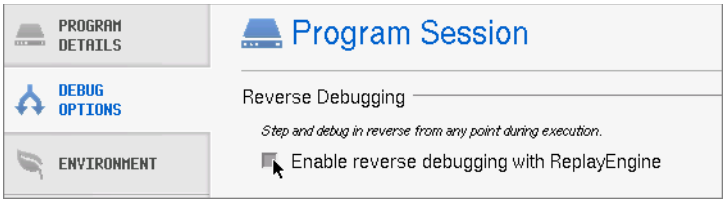
You can prepare a program for replay when you first load it into TotalView. Once the program is loaded, there are a number of ways to enable replay.

Enabling ReplayEngine at Program Load

To enable ReplayEngine when loading a program into TotalView, select the checkbox **Enable reverse debugging with Replay Engine** in either

- the **File > Debug New Program > Debug Options** dialog
- the **File > Debug New Parallel Program > Debug Options** dialog
- the **File > Attach to a Running Program > Debug Options** dialog

Figure 2, Enabling Replay Engine



This can also be accomplished from the command line through the **-replay** option.

CLI: dload -replay *program-path*
dattach -replay *program-path*

For a new program, ReplayEngine begins recording instructions as soon as you begin executing the program. For a running process you have attached to, ReplayEngine starts recording the next time you restart the process.

Enabling and Disabling ReplayEngine for a Loaded Program

Once a program is loaded into TotalView, there are many ways to enable and disable replay.

Enabling Replay

Replay behavior differs depending on whether program execution has begun or not.

The program is not yet executing

If the program is loaded but has not started executing, enable ReplayEngine in any of the following ways:

- Click the **Record** toolbar button (shown selected)

- Select the **Debug > Enable ReplayEngine** menu item
- Execute the CLI command **dhistory -enable**

  CLI: dhistory -enable

- Select **Process > Startup Parameters** from the Process Window, select the **Debug Options** tab, and then the **Enable reverse debugging with ReplayEngine** option. Click **Apply**.

ReplayEngine begins recording when the process starts executing. If you restart the process, ReplayEngine begins recording from the beginning of process execution.

To stop recording, exit the program and explicitly disable ReplayEngine. You cannot turn replay off while a process is executing.

For a parallel job, replay is enabled for *all* processes in the entire job, including at start and restart, unless you explicitly disable it.

The program is executing but halted

If a process is executing and stopped, you can immediately enable replay with any of the first three methods described above, but replay is enabled *only* while the program executes that single time. At process exit and restart, ReplayEngine will no longer be enabled unless you explicitly reenable it.

For a parallel job, only that process in the Process window will have replay enabled; other processes in the job will not.

Enabling ReplayEngine during program execution also means that you cannot step backward beyond the point at which ReplayEngine was enabled.

The fourth method discussed above, **Process > Startup Parameters**, requires a restart before it takes effect, meaning that Replay Engine will not be enabled on the currently executing process until it is restarted. For a parallel job, all processes will have replay engine enabled at restart.

Disabling Replay

Once a process with replay enabled is executing, the Record button and **Debug > Enable ReplayEngine** menu item are inactive, so you cannot simply deselect them to end replay.

To disable replay, you must:

1. Kill the executing process.
2. Disable ReplayEngine by clicking the **Record** button or deselecting the **Debug > Enable ReplayEngine** menu item (both are toggles).
3. Restart the process.

Another way to disable replay is to:

- Select **Process > Startup Parameters** from the Process Window, then the **Debug Options** tab, and deselect the **Enable ReplayEngine** debugging option.
- From a CLI prompt focused on the process, enter **dhistory -disable**.

```
CLI: dhistory -disable
```

If you then restart the process, ReplayEngine will be disabled for the executing process.

ReplayEngine and CUDA

You can enable ReplayEngine while debugging CUDA code; that is, ReplayEngine record mode works on the Linux x86-64 host. However, ReplayEngine does not support replay operations when focused on a CUDA thread since it does not support the GPU architecture itself. If you attempt this, you will receive a Not Supported error.

ReplayEngine and Expression Evaluation

The TotalView features that evaluate expressions, including Eval points, the Expression Window, the Variable Window, and the Expression List Window, do still function with ReplayEngine enabled, but there are limitations. Please see "Expression Evaluation with ReplayEngine" in the *TotalView for HPC User Guide*.

Examining Program State and History

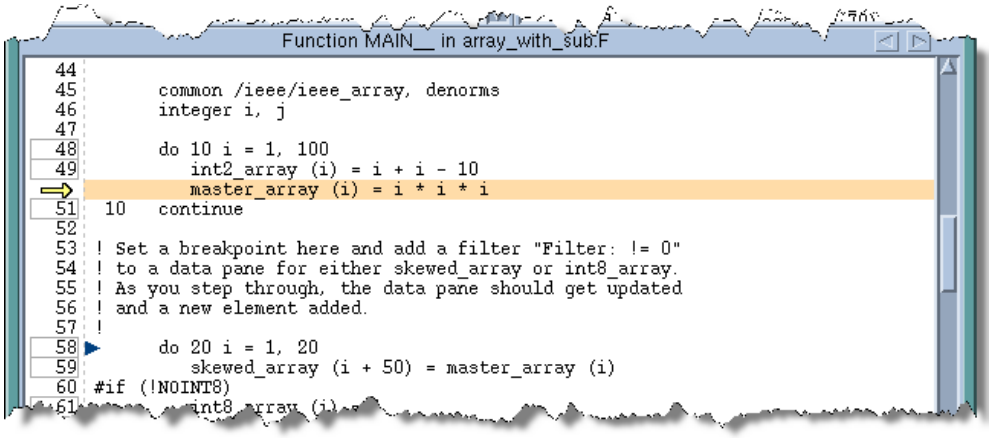
After enabling ReplayEngine, you can begin controlling your program's execution using the same execution commands you use when ReplayEngine is not enabled. For example, you might set a breakpoint and press the **Go** button or select a line and press the **Run To** button.

When you wish to view the program's state, halt your program, then use the **GoBack, Prev**, **UnStep**, **Caller**, or **BackTo** buttons to go to the statement you wish to examine. These four buttons are similar to the **Next**, **Step**, **Out**, and **Run To** tool bar buttons, differing only in that the Replay buttons go backwards in the program's history. The **Debug** pull-down menu contains the menu bar equivalents to these commands.

While you are in replay mode, notice that the **Next**, **Step**, **Out**, and **Run To** tool bar buttons are still displayed to move forward in the history.

When you're in replay mode, TotalView changes the highlight line from yellow to orange within the Source Pane**.**

Figure 3, Source Pane While ReplayEngine is in Replay Mode



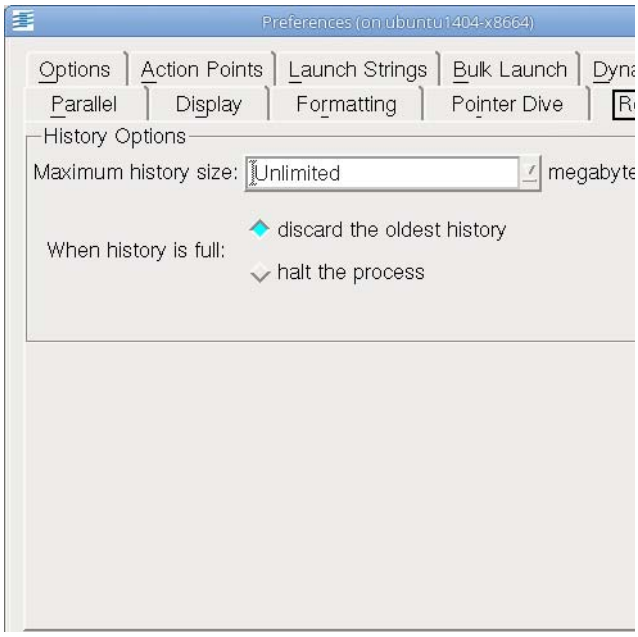
The Process window always shows the last line executed within record mode using the ▶ symbol and the yellow highlight line is on the same line as this symbol. When you are in replay mode, this symbol is where ReplayEngine shifts from replay mode to record mode.

The scoping commands at the far left side of the tool bar have no effect in replay mode as the ReplayEngine only supports process width.

Setting Preferences

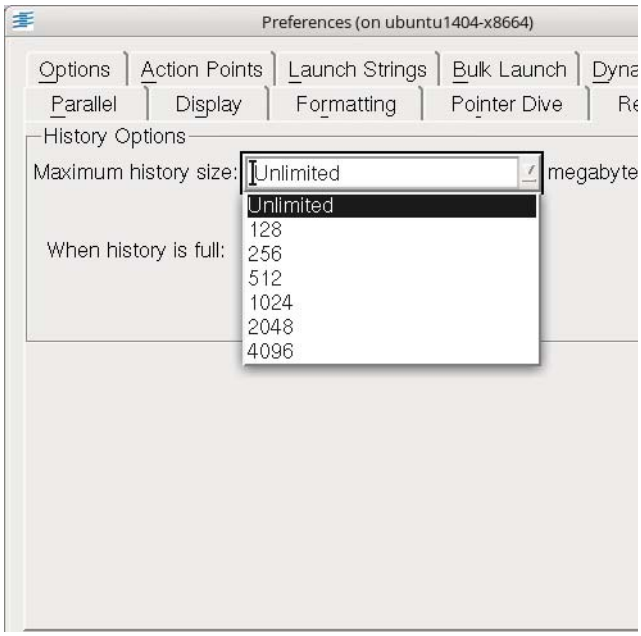
Use the **ReplayEngine** tab in the **Preferences** dialog box to define how ReplayEngine handles recorded history.

Figure 4, Preferences Dialog Box> ReplayEngine Page



The **Maximum history size** option sets the size in megabytes for ReplayEngine's history buffer. The default value, Unlimited, means ReplayEngine will use as much memory as is available to save recorded history. You can enter a new value into the text field or select from a drop-down list.

Figure 5, Preferences Dialog Box> ReplayEngine Page Drop-Down



You can also set these options using the CLI as follows:

CLI: dset TV::replay_history_size <value>

For example:

```
dset TV::replay_history_size 1024M
```
sets the maximum history size to 1024 megabytes.

```
dset TV::replay_history_size 1000000
```
sets the maximum history size to 1000000 bytes.

The second option on the **ReplayEngine** preference page defines the tool's behavior when the history buffer is full. By default, the oldest history will be discarded so that recording can continue. You can change that so that the recording process will simply stop when the buffer is full.

You can also control this behavior using the CLI as follows:

CLI: dset TV::replay_history_mode <1,2>

For example:

```
dset TV::replay_history_mode 1
```
sets the mode to discard the oldest history and continue recording.

```
dset TV::replay_history_mode 2
```
sets the mode to stop the process when the buffer is full.

CLI Support

- The **dload** and **dattach** CLI commands have the **-replay** option for enabling and disabling ReplayEngine. For example:

```
dload -replay myProgram
```

- The **dgo, dnext**, **dnexti**, **dout**, **dstep**, **dstepi**, and **duntil** commands let you step or run backwards by using the **-back** option. For example:

```
dnext -back
duntil -back 22
```

- The **dhistory** command has the following options:

  -info Dumps useful information about ReplayEngine.

  -enable If the program has not been started, ReplayEngine is enabled when it is started. If the program is already running, ReplayEngine is enabled immediately. Recording begins at the point that ReplayEngine was enabled and moving back beyond that point is not possible.

  -disable Disables Replay Engine for the next restart for the process.

  -create_bookmark [*comment*]
  Creates a Replay bookmark at the current execution location so you can return to it later. You can specify an optional comment to this command and it will be stored with the bookmark for display when you use the **show_bookmarks** command. A bookmark is created with a unique numeric ID, which is the return value.

  -goto_bookmark *ID*
  Goes to the bookmark with the specified ID. This returns the focus process to the execution location where the bookmark was first created.

  -go_live Resets the process back to record mode.

  -show_bookmarks
  Displays all Replay bookmarks. This command shows the bookmark ID along with information about what line number, PC and function the bookmark is on. If you added a comment to help you remember the significance of the bookmark, it displays this as well.

  -delete_bookmark *ID*
  Deletes the bookmark with the given ID.

  -clear_bookmarks
  Deletes all Replay bookmarks.

-get_time Deprecated — Use the bookmark options.
Displays the current time. The output of this command shows an integer value followed by an address. The first integer value is a virtual timestamp. This virtual timestamp does not refer to the exact point in time; it has a granularity that is typically a few lines of code. The address value is a PC value that corresponds to a precise point within that block of code.

-go_time *time* Deprecated — Use the bookmark options.
Moves the process to an execution point represented by the ***time*** argument. The ***time*** argument is a virtual timestamp as reported by **dhistory -get_time**. You cannot use this command to move to a specific instruction but you can use it to get to within a small block of code (usually within a few lines of your intended point in execution history). This command is typically used either for roughly bookmarking a point in a code or for searching execution history. It may need to be combined with stepping and **duntil** commands to return to an exact position.

These CLI commands are explained in detail in the *TotalView for HPC Reference Guide*.

Known Limitations and Issues

Limitations

- **Obscure instructions:** Use of AMD 3DNow! and other extended AMD instructions is not supported (though Intel SSE, SSE2, SSE3 and SSE4 instructions are supported). Instructions that modify CS, DS, ES or SS registers are also not supported.
- **AsyncIO:** ReplayEngine does not support asynchronous IO operations. **io_cancel**, **io_destroy**, **io_getevents**, **ioperm**, **iopl**, **io_setup**, and **io_submit** system calls are all unsupported.
- **Exec:** ReplayEngine does not support the **execve** syscall, as used by libc's **execl()**, **execlp()**, **execle()**, **execv()**, **execvp()**, and **execve()** functions. If the target program attempts to issue this system call, forward execution will not be possible beyond this point (though reverse execution is still possible).
- **Fork**: ReplayEngine does not follow **fork()** or **vfork()** system calls.
- **Obscure system calls:** Certain rarely used system calls are not supported. If the target program attempts to issue an unsupported system call, forward execution will not be possible beyond this point (though reverse execution is still possible). The following system calls are either esoteric or obsolete, and only maintained in the kernel for backward compatibility with binaries written for early 2.x series kernels: **ssetmask, modify_ldt, pivot_root, vm86**, and **unshare**.
- **Use of setrlimit():** If the target program uses **setrlimit** to reduce the amount of memory, processes, or other resources consumed, ReplayEngine may not be able to operate properly due to lack of resources.
- **Use of x86 inter-segment (aka 'far') jumps/calls:** ReplayEngine does not support the use of far jumps/calls in the target program. Any such attempt will result in forward execution not being able to continue from the point at which the far jump/call instruction is issued.
- **Non-executable memory:** ReplayEngine ignores the executable status of memory when running code, so code that would usually fail because it is in non-executable memory will run successfully.
- **Disk usage:** Depending on the target program, ReplayEngine can create large temporary files within **/tmp**. See System Resources ReplayEngine Uses on page 4 for information on how to use alternative temporary directories.

- **Self-modifying code:** ReplayEngine mostly works with self-modifying code, but in some situations the effects of writing into the currently executing "basic block" may be delayed (that is, writing instructions just ahead of the current program counter such that the processor executes the newly written code by virtue of "running in to" rather than "jumping to" it).
- **Shared memory accesses straddling valid and invalid pages:** Accessing shared memory where the instruction's operand straddles a page boundary such that the first part of the operand is in accessible shared memory, but the second part is in mapped shared memory which is not backed by a valid shared object (e.g. because the file which is mapped has been truncated) should receive signal **SIGBUS**. Under ReplayEngine, a target program making such an access will not receive **SIGBUS** but will read zeros for the part of the operand that straddles into unbacked memory. Note that normal attempted access to shared memory not backed by a shared object will generate a **SIGBUS** as normal; the issue applies only when a single instruction's access that lies half in valid memory and half in invalid memory that should generate a **SIGBUS**.
- **Breakpoints:** All breakpoints used with ReplayEngine work like hardware breakpoints. In particular, if the code where the breakpoint resides is not modified, writing to that code will not remove the breakpoint, and setting a breakpoint that is not at the first byte of an instruction will have no effect.
- **System call output buffers:** Any system calls that write to memory must be passed a buffer entirely within writable memory. For example, if **read()** is passed an 8k buffer of which only the first 4k is in user-writable memory, if that **read()** would normally return 4k or fewer characters then natively it may succeed, but on ReplayEngine it will fail with **EFAULT**. If a system call that writes to memory is passed a buffer which is not in writable memory at all, but fails for some other reason before the kernel tries to write to the buffer, then natively it may fail with some error other than **EFAULT**, but on ReplayEngine it may fail with **EFAULT**. If two buffers which overlap are passed to a system call which writes to both of them or reads from one and writes to the other, the behavior in ReplayEngine may differ from the native behavior (although behavior in such cases is liable to vary between kernel versions, too.)
- **Adjust Flag:** According to the Intel manuals, the state of the Adjust Flag (AF) after some instructions is "undefined." On some processor models, different executions of the same code can produce different states of AF. If the behavior of a program depends on the state of AF when it is supposed to be undefined, the program may not run correctly with ReplayEngine.
- **SIGCHLD while attaching:** If a **SIGCHLD** arrives for a process while ReplayEngine is in the middle of attaching to the process, the **SIGCHLD** may be silently lost. Once the process has been attached to, **SIGCHLD** is handled normally.
- **Loading a previous recording session**: The successful reloading and debugging of a previously saved replay recording session requires that both the environment that saved the session and the environment replaying the recording session be exactly the same.

Performance Issues

High TLB rates with certain multi-threaded target programs

When reverse debugging an application in which many threads make frequent system calls on a multi-processor platform, binding the application process to a single processor can improve performance. This is because such applications put stress on ReplayEngine's heap management, which in turn stresses the processor's TLB (translation lookaside buffer). If the application is bound to a single processor, it is less likely to suffer TLB misses caused by process migration. Since user threads are automatically serialized during reverse debugging, there is no loss of concurrency due to binding.

If the application is to be launched under TotalView, one way to accomplish binding is to preface the TotalView command with a **taskset(1)** command specifying a single processor. For example:

```
taskset --cpu-list 3 totalview -replay myapp
```

To accomplish binding when TotalView is to be attached to a running application, find the PID (process identifier) of the application process, and use taskset to bind that process to a single processor before attaching to it with TotalView. For example:

```
taskset --pid --cpu-list 3 <PID of myapp>
```

We have noticed the need for such binding when debugging MySQL applications with ReplayEngine.

Avoiding self-contention in OpenMP target programs

Because threads are serialized during reverse debugging, OpenMP implementations that use non-yielding spins for synchronization can experience self-contention, resulting in poor performance. ReplayEngine has internal knowledge of several OpenMP implementations and tries to avoid this situation. Since this aspect of OpenMP is somewhat loosely standardized, however, ReplayEngine may not always be able to avoid self-contention.

For the Portland Group compilers in particular, ReplayEngine uses environment variables to avoid self-contention. It inserts the settings **OMP_WAIT_POLICY=ACTIVE** and **MP_SPIN=0** into the environment. The effects are, respectively, to cause idle threads to wait using a semaphore check loop, and to cause the semaphore check loop to call **sched_yield** in every iteration. If the user has pre-set either of these environment variables, ReplayEngine will not alter the settings.

Index