Getting Started with Replay Engine

version 1.7

TotalView Technologies

Contents

Understanding ReplayEngine

TotalView Technologies 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. TotalView documentation is available , on our web site at http://www.totalviewtech.com.

ReplayEngine Overview

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 and 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*.

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 slower than it would run 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.

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 Replay-Engine to use as the base directory for writing its temporary information. For example:

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

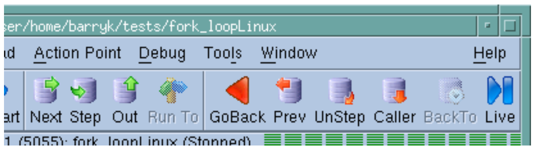
ReplayEngine also changes the amount of memory your program uses as it keeps history and state information in memory. For information on controlling the history information storage, see "*Setting Preferences*" on page 6.

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 by 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. (See Figure 1.)

Figure 1: ReplayEngine Tool Bar Commands



The ReplayEngine commands are as follows:

- **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.

The ReplayEngine tool bar commands only appear if you are using TotalView on a Linux-x86 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.

Using ReplayEngine

There is very little difference between running TotalView and running ReplayEngine. The first step is enabling ReplayEngine. Do this by selecting **Enable Replay Engine** in the **File > New Program** dialog box or in the **Process > Startup Parameters** dialog box. Figure 2 shows the **New Program** dialog box.

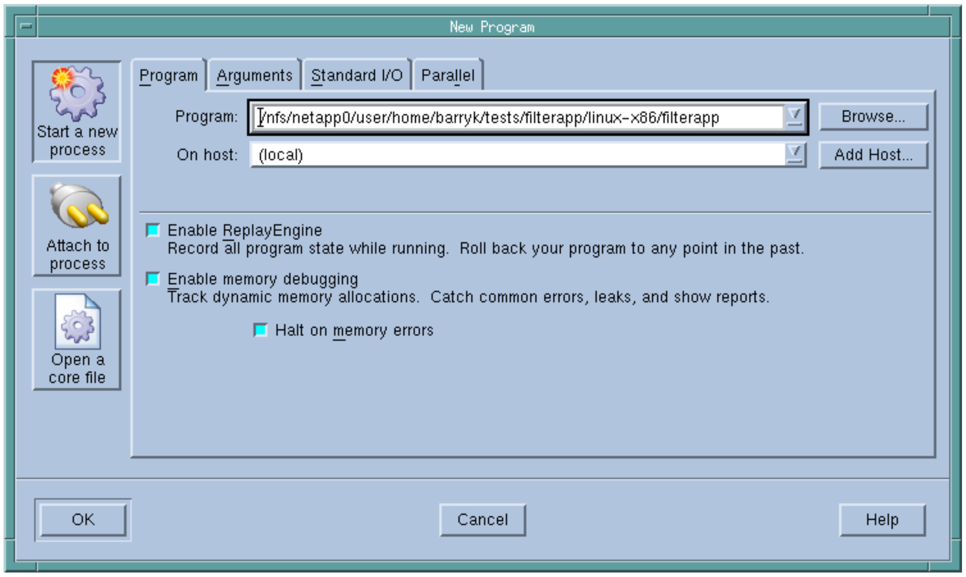


Figure 2: Enabling using the File > New Command Dialog Box

Program: /nfs/netapp0/user/home/barryk/tests/filterapp/linux-x86/filterapp

On host: (local)

Enable ReplayEngine
Record all program state while running. Roll back your program to any point in the past.

Enable memory debugging
Track dynamic memory allocations. Catch common errors, leaks, and show reports.

Halt on memory errors

If you are using the **New Program** dialog box, ReplayEngine begins recording instructions when you begin execution. If you are using the **Startup Parameters** dialog box, ReplayEngine is enabled when you restart your program.

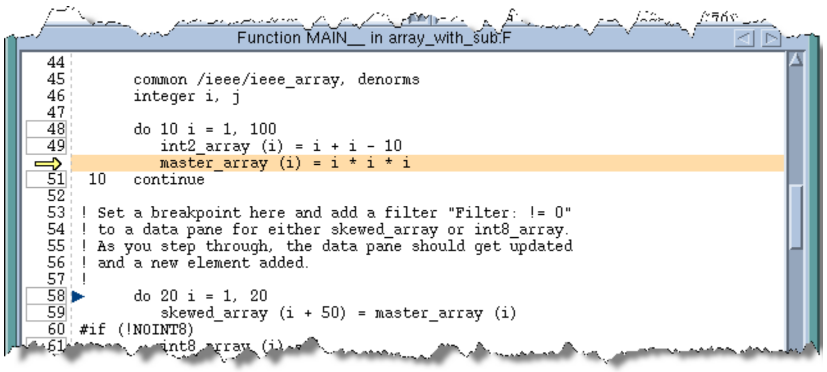
You can also enable ReplayEngine by using the TotalView –**replay** command-line option.

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. This is because pressing these buttons moves you forward in the history.

When you're in replay mode, TotalView changes the highlight line from yellow to orange within the Source Pane. (See Figure 3.).



```
      common /ieee/ieee_array, denorms
      integer i, j

      do 10 i = 1, 100
         int2_array (i) = i + i - 10
         master_array (i) = i * i * i
 10   continue

! Set a breakpoint here and add a filter "Filter: != 0"
! to a data pane for either skewed_array or int8_array.
! As you step through, the data pane should get updated
! and a new element added.
!
      do 20 i = 1, 20
         skewed_array (i + 50) = master_array (i)
#if (!NOINT8)
         int8_array (i)
```

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.

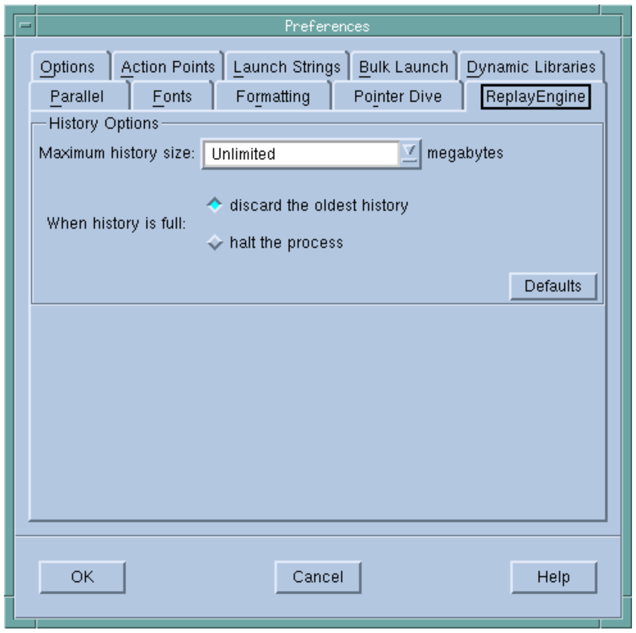
The scoping commands at the far left side of the tool bar have no effect in replay mode as the ReplayEngine only supports process width.

Setting Preferences

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

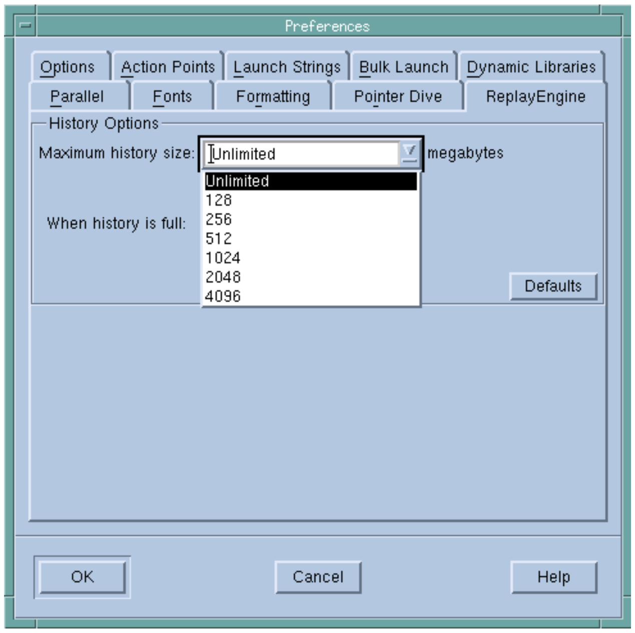
The **Maximum history size** option sets the size in megabytes for Replay-Engine's history buffer. The default value, Unlimited, means ReplayEngine will use as much memory as is available to save recorded history. You can enter a new value into the text field or select from a drop-down list.(See Figure 5.)

Figure 4: Preferences Dialog Box > ReplayEngine Page



History Options
Maximum history size: Unlimited megabytes
When history is full: ◆ discard the oldest history ◇ halt the process

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



History Options
Maximum history size: Unlimited megabytes
When history is full:
Unlimited
128
256
512
1024
2048
4096

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.
-get_time	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***	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 com-

	bined with stepping and **duntil** commands to return to an exact position.
-go_live	Resets the process back to record mode.
-enable	Enables ReplayEngine for the next restart for the process.
-disable	Disables Replay Engine for the next restart for the process.

These CLI commands are explained in detail in the *TotalView 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).
- **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 2 for information on how to use alternative temporary directories.

- **Statically-linked target programs:** ReplayEngine cannot start a statically-linked target program. However, it can attach to an existing statically-linked target program process.
- **Self-modifying code:** Self-modifying code is supported as long as the target program executes at least one branch instruction between the writing of the code and its execution.
- **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 zeroes 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 is only with 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.

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