NCCS Brown Bag Series
TotalView on Discover:
Part 2

ReplayEngine and MemoryScape

Chongxun (Doris) Pan
doris.pan@nasa.gov
June 14, 2012
Agenda For Part 1 (see previous tutorial)

• Overview
• Basic Navigation and Control
  • Demo 1: Basic Navigation and Control
    ❖ Using an OpenMP Space weather application
• Debugging MPI Applications
• Demo 2: MPI Debugging:
  ❖ Using GEOS5-AGCM
Agenda For Part 2

- Overview
- **Reverse Debugging with ReplayEngine**
  - Demo 1: Replay Engine
    - Using an OpenMP Space weather application
- Memory Debugging with MemoryScape
- Demo 2: MemoryScape
  - Using GEOS AGCM
What is ReplayEngine?

- **Reverse Debugging**
  - Let you step backward by function, line, or instruction
  - Capture and deterministically replay executions
- **Major features**
  - No recompilation
  - Designed to be used for parallel applications
- **Why use ReplayEngine?**
  - Eliminate restart cycle and hard-to-reproduce bugs
ReplayEngine

- **Record Mode** (saving execution history)
  - Capture the order of executions and changes to the data

- **Replay Mode** (when using a ReplayEngine command)
  - Like a “rewind” button on a DVR

- **Implications**
  - To ensure deterministic execution trajectory, it forces single thread execution at a time
  - During Replay mode, some operations (i.e., setting variables or full asynchronous thread control) are not available
  - Record mode could incur significant overhead (both in execution time and memory usage)
Enable ReplayEngine

1. Starting Totalview with ReplayEngine enabled

2. After launching totalview, click the “Enable ReplayEngine” Button. ReplayEngine will be enabled after “Restart”
ReplayEngine Usage Models

- **GoBack** (vs. **Go**): back to the last action point or the start of the recorded history
- **Prev** (vs. **Next**): back to previous statement executed. If the line has a function call, it skips over the call
- **Unstep** (vs. **Step**): back to previous statement executed. If the line has a function call, it moves to the last statement in the call
- **Caller** (vs. **Out**): back to before the current routine was called.
ReplayEngine Usage Models (Cont’d)

- **BackTo** (vs. **Run To**): back to the line you select
- **Live**: to shift from Replay mode to Record mode. It displays the statement that would have executed if you had not moved into Replay mode.
ReplayEngine Preferences

- Setting Maximum History size (default is unlimited)
- setenv TVD_REPLAY_TMPDIR to control the directory of saving the history information (default is / tmp)
Demo 1: ReplayEngine

- Debugging an OpenMP Spaceweather application with ReplayEngine enabled
- Toolbar commands
- Preferences
- TVD_REPLAY_TMPDIR
Agenda For Part 2

• Overview
• Reverse Debugging with ReplayEngine
• Demo 1: Replay Engine
  ❖ Using an OpenMP Space weather application
• Memory Debugging with MemoryScape
• Demo 2: MemoryScape
  ❖ Using GEOS AGCM
• Your program data is organized in four segments:
  1. **Text (code):** machine code instructions
  2. **Data section:** static and global variables
  3. **Stack:** local (automatic) variables
     - Memory set aside for each thread of execution. It remains during program execution.
     - When a function is called, a block is reserved on the top of the stack for local (automatic) variables. It will go out of scope when the function returns and automatically deallocate.
     - Accessing stack in LIFO order – the most recently reserved block is the next block to be freed.
     - Easy to keep track of the stack -- Stack typically maps to the cache, so it is faster than the heap but smaller and expensive.
4. **Heap**: all variables created or initialized *at* runtime are stored
   - Dynamic memory allocations. Heap size can grow as space is needed.
   - Variables created on the heap must be destroyed manually and never fall out of scope.
   - There's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time
   - Slower to allocate on the heap in comparison to variables on the stack
   - Can have fragmentation when there are a lot of allocations and deallocations
   - Responsible for memory leaks and many other memory bugs
What is MemoryScape?

• Runtime Memory Analysis to detect memory bugs
  ❖ A memory bug is a mistake in heap memory usage
    ❖ Failure to check for error conditions
    ❖ Leaking: failure to free memory
    ❖ Dangling references: failure to clear pointers
    ❖ Memory corruption
      ❖ Writing to memory not allocated
      ❖ Over running array bounds

• Designed to be used with parallel applications.
Why are memory bugs hard to find?

• Memory problems can be hidden
  - For a given scale or platform or problem, they may not be fatal
  - Failures could occur until modification, reuse of a component, or moving the application to a different cluster with a new OS
  - Libraries could be a source of problem
  - Can also manifest as “random crashes” or “random wrong answers”
MemoryScape Features:

• Stopping execution when heap problems occur
• Viewing the heap graphically
• Leak detection and dangling pointer detection
• Memory Comparisons between processes
• Memory Corruption detection – Bound errors
  - Guard Blocks - allocated additional memory (8 bytes default) before or after a heap block. An event notification occurs when overwriting guard blocks.
  - Red Zone – allocated additional buffer before or after a heap block. An event notification occurs when either writing or reading to the Red Zone. (High memory overhead!)
Memory Debugger Features: (Cont’d)

• Block painting
  ❖ Writing a bit pattern into allocated and deallocated blocks
  ❖ Detecting the use of memory either before it is initialized or after it is deallocated

• Hoarding
  ❖ Holding onto deallocated memory so it cannot be reused immediately
  ❖ Not an often-used feature
  ❖ Mostly used to detect problems related to memory being deallocated by one thread while another thread is using this memory
Start MemoryScape

1. Make sure your program is compiled with –g
2. module load tool/tview-8.9.2-2
3. setenv PATH $PATH:/usr/local/toolworks/memoryscape.3.2.2-2/bin/
4. Launch totalview as usual, i.e.,
   totalview <executable>

Then click “Enable memory debugging” in the “Startup Parameters” window

Or just type:
memscape <executable>
Strategies for Memory Debugging for Parallel Applications

• Run (or step through) the application and see if memory events are detected

• View memory usage across the MPI job
  ❖ Compare memory footprints of the processes

• Gather heap information/Leak reports in all processes of the MPI job
  ❖ Select and examine individually
    ❖ Look at the allocation pattern
    ❖ Look for leaks
  ❖ Compare with the “diff” mechanism
    ❖ Look for major differences
Memory Debugging Options

Low is the default; Medium adds guard blocks; High adds Red Zone detection; Extreme adds above all plus paint memory and hoarding.

Select Medium only when you need to check for corrupted memory!
Select High only when you need to check for memory overruns!
What are memory events?

When one of these errors occurs, MemoryScape places event indicators by the process and at the top of the window.
Analysis of memory debugging data

- Heap status
  - Graphical report
  - Source report
  - Backtrace report
- Leak detection
  - Source report
  - Backtrace report
- Memory usage
  - Chart report
  - High-level table
  - Detailed table
- Memory corruption reports
  - If guard blocks are chosen
- Memory comparison among processes
Filtering reports is usually necessary to suppress the display of too much information.

Right-click routine name or line number in Heap Status Source or Backtrace report or in a Heap Status Graphical report, and choose “Filter out this entry”
### Heap Status Source Report

#### Process 9: GEOGcm debug

<table>
<thead>
<tr>
<th>Process</th>
<th>Bytes</th>
<th>Count</th>
<th>Begin Address</th>
<th>End Address</th>
<th>Backtrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEOGcm debug</td>
<td>4170.94MB</td>
<td>1</td>
<td>128747</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libdap6.so.2</td>
<td>128.00KB</td>
<td>1</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libdap7.so.2</td>
<td>33.55KB</td>
<td>1</td>
<td>296</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Process 10: GEOGcm debug0

<table>
<thead>
<tr>
<th>Process</th>
<th>Bytes</th>
<th>Count</th>
<th>Begin Address</th>
<th>End Address</th>
<th>Backtrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEOGcm debug0</td>
<td>4164.92MB</td>
<td>1</td>
<td>122980</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libdap6.so.2</td>
<td>128.00KB</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libdap7.so.2</td>
<td>33.55KB</td>
<td>1</td>
<td>296</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Backtrace**

```
-ID | Function          | Line # | Source                  
---|-------------------|--------|-------------------------
  1 | malloc             | 106    | malloc                  
  2 | _LMPR_init_thread |        | _LMPR_init_thread       
  3 | _LMPR_init_thread |        | _LMPR_init_thread       
  4 | _ZNESMCI14MKInitEI | 15      | _ZNESMCI14MKInitEI      
  5 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
  6 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
  7 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
  8 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
  9 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
```

**Source**

```
module m_ioutl
```

**Interface**

```
11 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
12 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
13 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
14 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
15 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
16 | _ZNESMCI12Mk10InitizeEI | 15      | _ZNESMCI12Mk10InitizeEI |
```
Leak reports are essentially the same as the Heap status reports, only with fewer controls and less data.
Leak Detection Backtrace Reports

Backtrace Report contains similar info as the source report. It is useful in helping you coordinate info in different screens and tabs as it does not change from report to report.
Memory Usage Chart Report

Differ slightly from the Heap Status reports:
- Memory usage data is obtained from the OS perspective, including overhead of the MemoryScape itself;
- Heap status is obtained from monitoring program requests and releases of memory
A detailed program/library report table – breakdown for the program and libraries for each process:

You can also see just a high level process report table – breakdown for each process.
Memory Comparison Report

Data Source
- Allocations
- Leaks
- Duplications
- Hard
- Red Zones

Process Comparisons
- Session 1: GEOSSGCMDebugger.x.x
- Session 2: GEOSSGCMDebugger.x.x

Process
- GEOSSGCMDebugger.x.x
- MAPL_CFIQ
- MAPL_Generic
- MAPL_Comms
- m_lszdl
- for
- got
- vm
- line 10
- ZNمضغ
- ZNمضغ
- for

Memory Comparison Report

<table>
<thead>
<tr>
<th>Process</th>
<th>Bytes Session 2</th>
<th>Bytes Session 1</th>
<th>Bytes Difference</th>
<th>Count Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEOSSGCMDebugger</td>
<td>413.49MB</td>
<td>427.46MB</td>
<td>-13.96MB</td>
<td></td>
</tr>
<tr>
<td>MAPL_CFIQ</td>
<td>413.49MB</td>
<td>427.46MB</td>
<td>-13.96MB</td>
<td></td>
</tr>
<tr>
<td>MAPL_Generic</td>
<td>670.00KB</td>
<td>330.00KB</td>
<td>330.00KB</td>
<td></td>
</tr>
<tr>
<td>MAPL_Comms</td>
<td>24.36KB</td>
<td>24.09KB</td>
<td>272</td>
<td></td>
</tr>
<tr>
<td>m_lszdl</td>
<td>0</td>
<td>338.00KB</td>
<td>-338.00KB</td>
<td></td>
</tr>
<tr>
<td>for</td>
<td>412.01MB</td>
<td>426.77MB</td>
<td>-13.96MB</td>
<td></td>
</tr>
<tr>
<td>got</td>
<td>5.01MB</td>
<td>744</td>
<td>5.01MB</td>
<td></td>
</tr>
<tr>
<td>vm</td>
<td>5.01MB</td>
<td>764</td>
<td>5.01MB</td>
<td></td>
</tr>
<tr>
<td>line 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZNمضغ</td>
<td>173.56KB</td>
<td>383.09KB</td>
<td>-209.53KB</td>
<td></td>
</tr>
<tr>
<td>ZNمضغ</td>
<td>294.75KB</td>
<td>819.43KB</td>
<td>-524.68KB</td>
<td></td>
</tr>
<tr>
<td>for allocate</td>
<td>407.34MB</td>
<td>426.79MB</td>
<td>-18.46MB</td>
<td></td>
</tr>
</tbody>
</table>

NASA Center for Climate Simulation
Memory Corruption Reports are generated only if you select “Medium” or higher debugging option.

- The top section graphically displaying each corruption.
- The bottom section contains a backtrace and the allocation source line.

**Orange**: corrupted guard block  
**Light green**: uncorrupted guard block  
**Dark green**: allocated data block
Agenda For Part 2

• Overview
• Reverse Debugging with ReplayEngine
• Demo 1: Replay Engine
  - Using an OpenMP Space weather application
• Memory Debugging with MemoryScape
• Demo 2: MemoryScape
  - Using GEOS AGCM
Demo 2

- `xsub -I -V -l select=1:ncpus=12:mpiprocs=12,walltime=1:00:00 -W group_list=<your_group>`

Once the compute node is available:

- `cd "$GEOS_EXPDIR"`
- `source $GEOSBIN/g5_modules`
- re-compile GCM with BOPT=g
- `mpdboot -n 1 -r ssh -f $PBS_NODEFILE`
- `module load tool/tview-8.9.2.2`
- `cp and link necessary files ..(all in gcm_run.j)`
- `totalview ./GEOSgcm.debug.x`
More info and references…

• MemoryScape_User_Guide.pdf
• ReplayEngine_Getting_Started_Guide.pdf
Under /usr/local/toolworks/totalview.8.9.2-2/doc/pdf

• “Memory Debugging Parallel Programs” tutorial offered by the Totalview in SC009
http://www.crc.nd.edu/~rich/SC09/docs/tut120/tut120.pdf