



Memory Debugging Parallel Applications on BlueGene

SciComp
May 21, 2009
Ed Hinkel

Agenda

- **Introduction**
- **Memory (mis) Management**
- **Memory Debug Options & Issues**
- **Memory Debugging Techniques**
- **What's New**



Memory Bugs

Can be very elusive!

- **Memory bugs are often not immediately fatal**
- **Memory bugs can lurk in a code base for long periods of time**
- **Memory bugs can suddenly emerge when**
 - A program is ported to a new architecture
 - Programs are scaled up to a larger size
 - When code is adapted and reused from one program to another.
- **Most memory bugs are not detected by compilers**



Memory bugs are often manifested in unusual ways

- Memory bugs often go undetected until the worst possible time
- Symptoms often surface long after the actual damage is done
- Some only surface after hours or even days of operation
- In many cases, the programs affected are “innocent bystanders”

Memory Issues are on the Rise

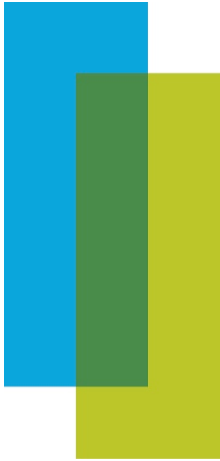
- Core counts are growing at an amazing rate
- But the Memory “per CPU” is trending downward
- Proper memory management is becoming more critical
- More than ever, you need to really know how memory is being used

What is a Typical Memory Bug?

- **A Memory Bug is a mistake in the management of heap memory**
 - **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**

Memory Debugging Options

- **Developers have a range of options (many free) for memory debugging... But:**
 - **Many programs are often singular in function, requiring an array of “solutions”.**
 - **Most often, there is significant “overhead” issues to consider:**
 - **Performance hits can often be huge, with unacceptable slowdowns**
 - **Additional memory usage can make bad things worse**
 - **Special instrumentation requirements can often produce an unwelcome exercise of the Heisenberg uncertainty principle**
 - **Scalability can be a big problem**



Memory Debugging Options

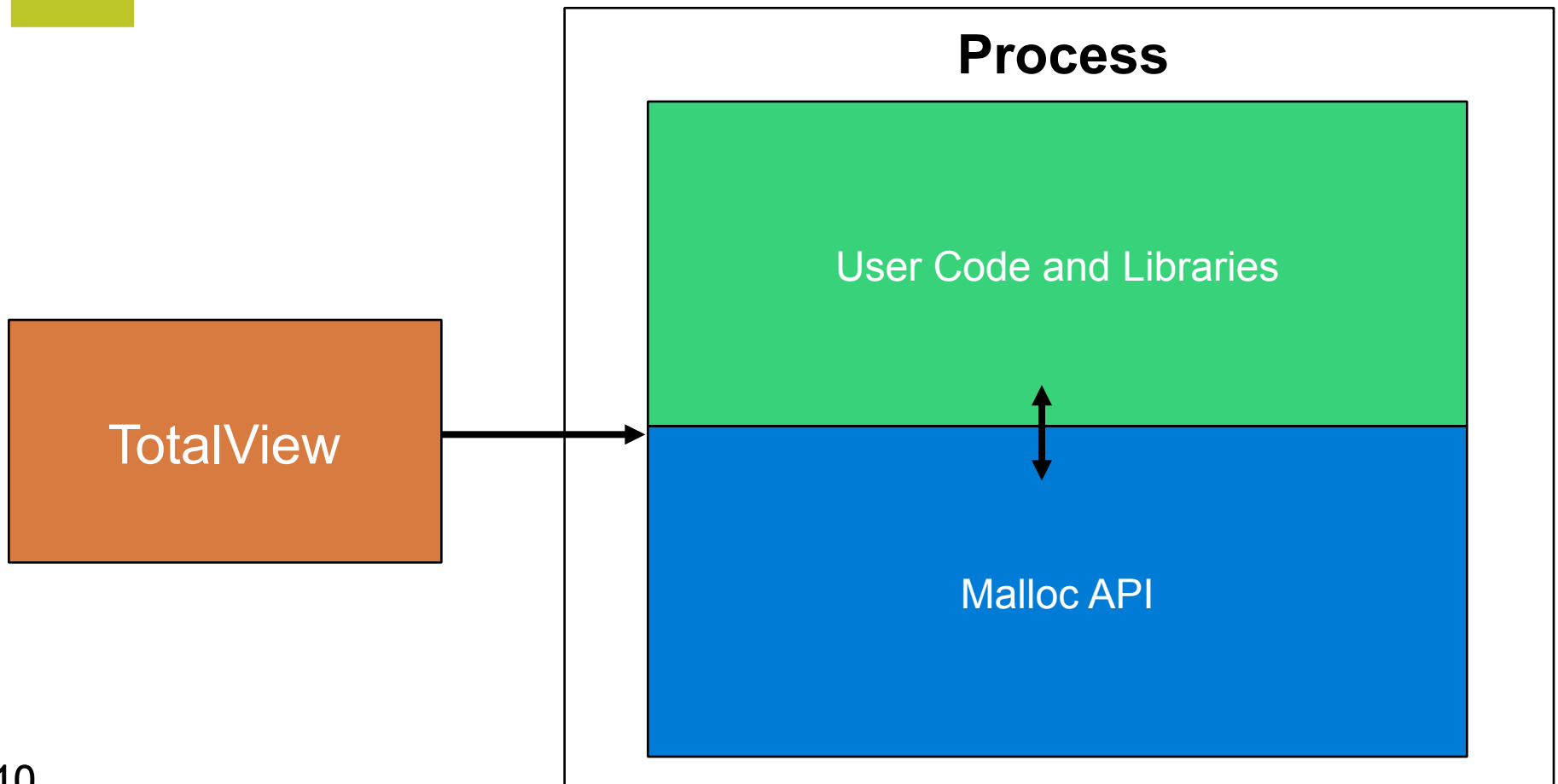
**So,
How Does One
Memory Debug Effectively?**



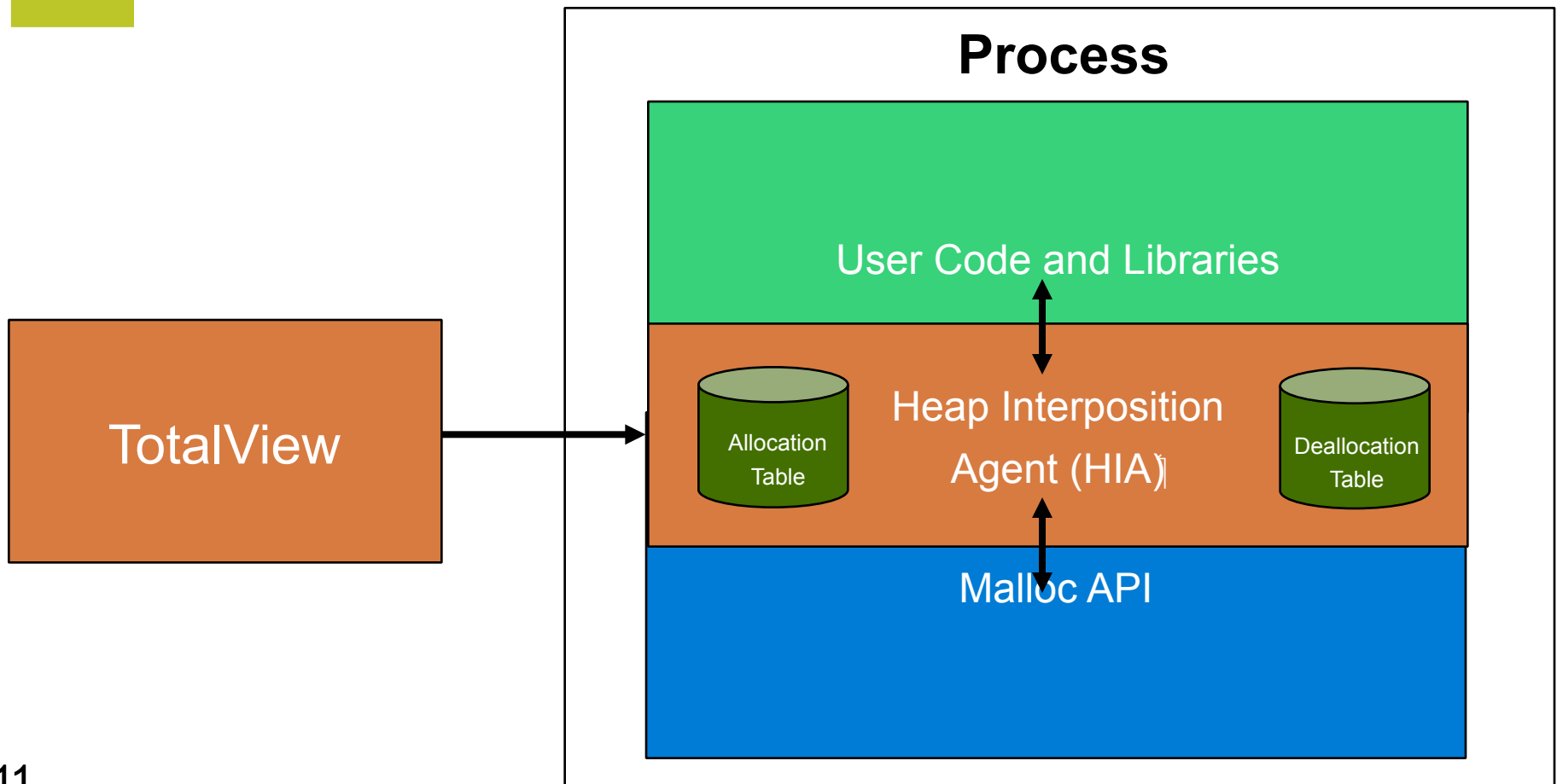
TotalView Memory Debugging Products

- **TotalView Source Code Debugger**
 - Fully integrated Memory Debugging Capabilities
- **MemoryScape Memory Debugger**
 - Standalone Memory Debugging
 - Non-developer environments
 - Quality Assurance
 - Test groups
 - Customers

TotalView's Interposition Agent

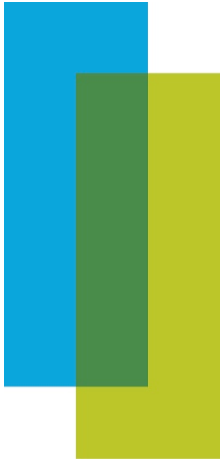


TotalView's Interposition Agent



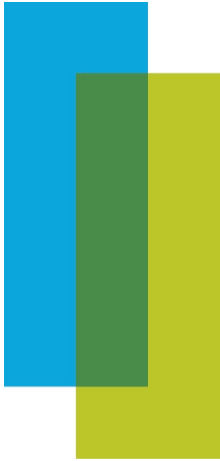
TotalView HIA Technology

- **Advantages of TotalView HIA Technology**
 - **Use it with your existing builds**
 - No Source Code or Binary Instrumentation
 - **Programs run nearly full speed**
 - Low performance overhead
 - **Low memory overhead**
 - Efficient memory usage
 - **Support for a wide range of platforms – including Cell**



Memory Debugger Features

- Automatic allocation problem detection
- Heap Graphical View
- Leak detection
- Block painting
- Dangling pointer detection
- Deallocation/reallocation notification
- Memory Corruption Detection - Guard Blocks
- Memory Hoarding
- Memory Comparisons between processes
- Collaboration features



Enabling Memory Debugging



Setting up a memory debug session...

Fexibility is Key

Enabling Memory Debugging Memory Event Notification

Halt execution on memory event or error

Use the **Advanced** button to control actions for individual events.

[Advanced...](#)

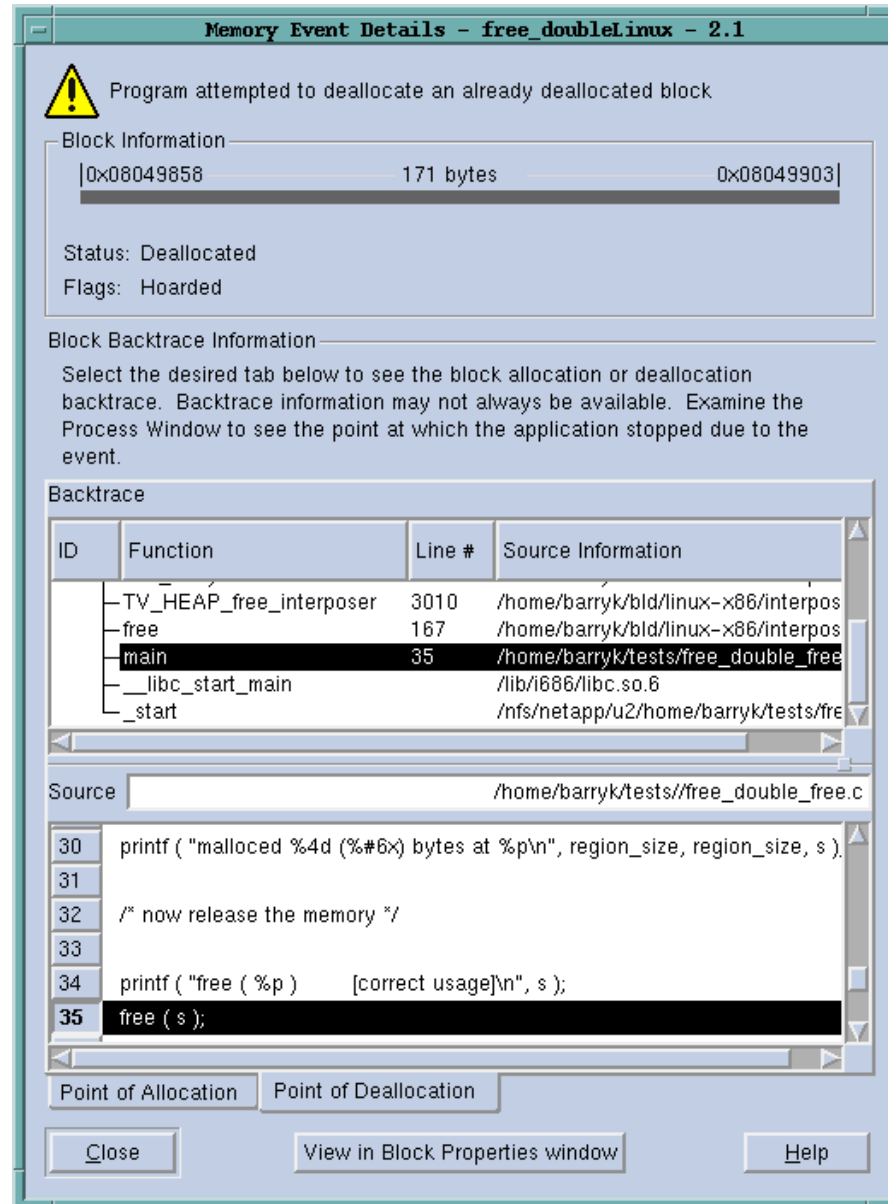
Memory Event Notification

Notify me when these events trigger:


Event	Description
<input checked="" type="checkbox"/> Allocation Failed	An allocation call failed or the address returned is NULL: probably out of memory
<input checked="" type="checkbox"/> Double allocation	Allocator returned a block already in use: heap may be corrupted
<input checked="" type="checkbox"/> Double free	Program attempted to free an already freed block
<input checked="" type="checkbox"/> Free interior pointer	Program attempted to free a block incorrectly, via an address in the middle of the block
<input checked="" type="checkbox"/> Free notification	A block for which notification was requested is being freed
<input checked="" type="checkbox"/> Free unknown block	Program attempted to free an address not in the heap
<input checked="" type="checkbox"/> Guard corruption	The guard areas around a block have been overwritten, suggesting a bounds error
<input checked="" type="checkbox"/> Invalid aligned allocation request	Program supplied an invalid alignment argument to the heap manager
<input checked="" type="checkbox"/> Misaligned allocation	Allocator returned a misaligned block: heap may be corrupted
<input checked="" type="checkbox"/> Realloc notification	A block for which notification was requested is being reallocated
<input checked="" type="checkbox"/> Realloc unknown block	Program attempted to reallocate an address not in the heap
<input checked="" type="checkbox"/> Termination notification	The target is terminating, memory analysis can be performed
<input checked="" type="checkbox"/> Unknown error	Some unknown error has occurred

[Help](#) [OK](#) [Cancel](#)

Memory Event Details Window



Memory Event Details - free_doubleLinux - 2.1

 Program attempted to deallocate an already deallocated block

Block Information

|0x08049858 | 171 bytes | 0x08049903|

Status: Deallocated
Flags: Hoarded

Block Backtrace Information

Select the desired tab below to see the block allocation or deallocation backtrace. Backtrace information may not always be available. Examine the Process Window to see the point at which the application stopped due to the event.

Backtrace

ID	Function	Line #	Source Information
	TV_HEAP_free_interposer	3010	/home/barryk/bld/linux-x86/interpos
	free	167	/home/barryk/bld/linux-x86/interpos
	main	35	/home/barryk/tests/free_double_free
	__libc_start_main		/lib/i686/libc.so.6
	_start		/nfs/netapp/u2/home/barryk/tests/fre

Source: /home/barryk/tests/free_double_free.c

```
30 printf ( "malloced %4d (%#6x) bytes at %p\n", region_size, region_size, s );
31
32 /* now release the memory */
33
34 printf ( "free ( %p ) [correct usage]\n", s );
35 free ( s );
```

Point of Allocation | Point of Deallocation

Memory Event Details - Process 3: filterapp-mpi.1 - 3

Process 3: filterapp-mpi.1 - 3 Time: 06:55:56

Program attempted to free an already freed block

Event Location | Allocation Location | Deallocation Location | Block Details

Backtrace

ID	Function	Line #	Source Information
	free_body	4281	malloc_interposers.c
	TV_HEAP_free_interposer	4488	malloc_interposers.c
	free	184	malloc_wrappers_dlopen.c
	double_free	80	main.cxx
	main	166	main.cxx
	_libc_start_main		libc-2.7.so
	_start		filterapp-mpi

Source: /home/ubuntu/src/main.cxx

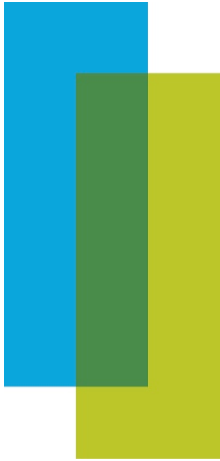
```

73 // Now show that the deallocation stack is available now
74 junk = 0;
75
76 // Now release the memory the second time - illegal
77 #ifdef USEMPI
78 if( rank == 1 )
79 #endif
80 free ( p );
81

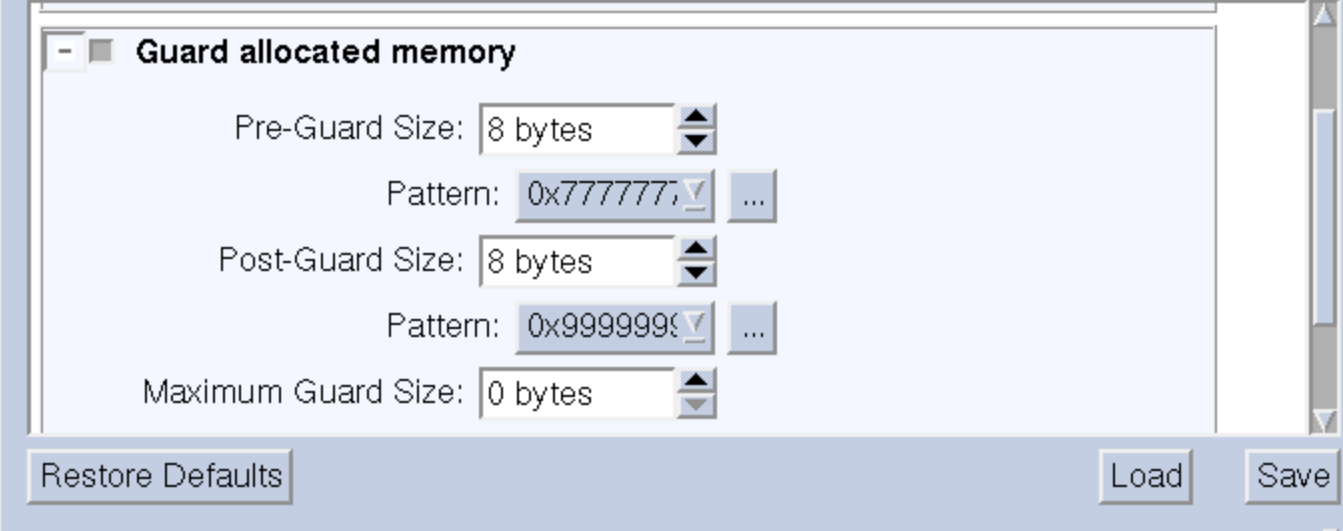
```

Generate

Memory File

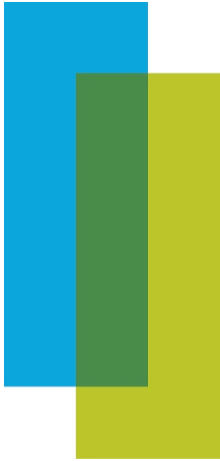


Memory Corruption Detection (Guard Blocks)



Guard blocks provide a buffer around each memory allocation. Using the following controls users can adjust the size of the "pre-guard", the area preceding each memory allocation and the "post-guard", the area following each memory allocation.

The pre-guard and post-guard sizes can be adjusted independently and each can be assigned individual patterns that they are painted with. These patterns are used to determine if the guard areas have been overwritten at all. Keep in mind that the larger the guard areas are made the more memory is required to provide the guard blocks.



Memory Corruption Detection (Guard Blocks)

Memory Event Details - Process 1 (9939): filterapp - 1

Process 1 (9939): filterapp - 1
The guard areas around a block have been overwritten, suggesting a bounds error Time: 07:22:46

Event Location | Allocation Location | Deallocation Location | Block Details

Block Information

0x0804c068 64 bytes 0x0804c0a7

Status: Allocated Flags: Operation in Progress

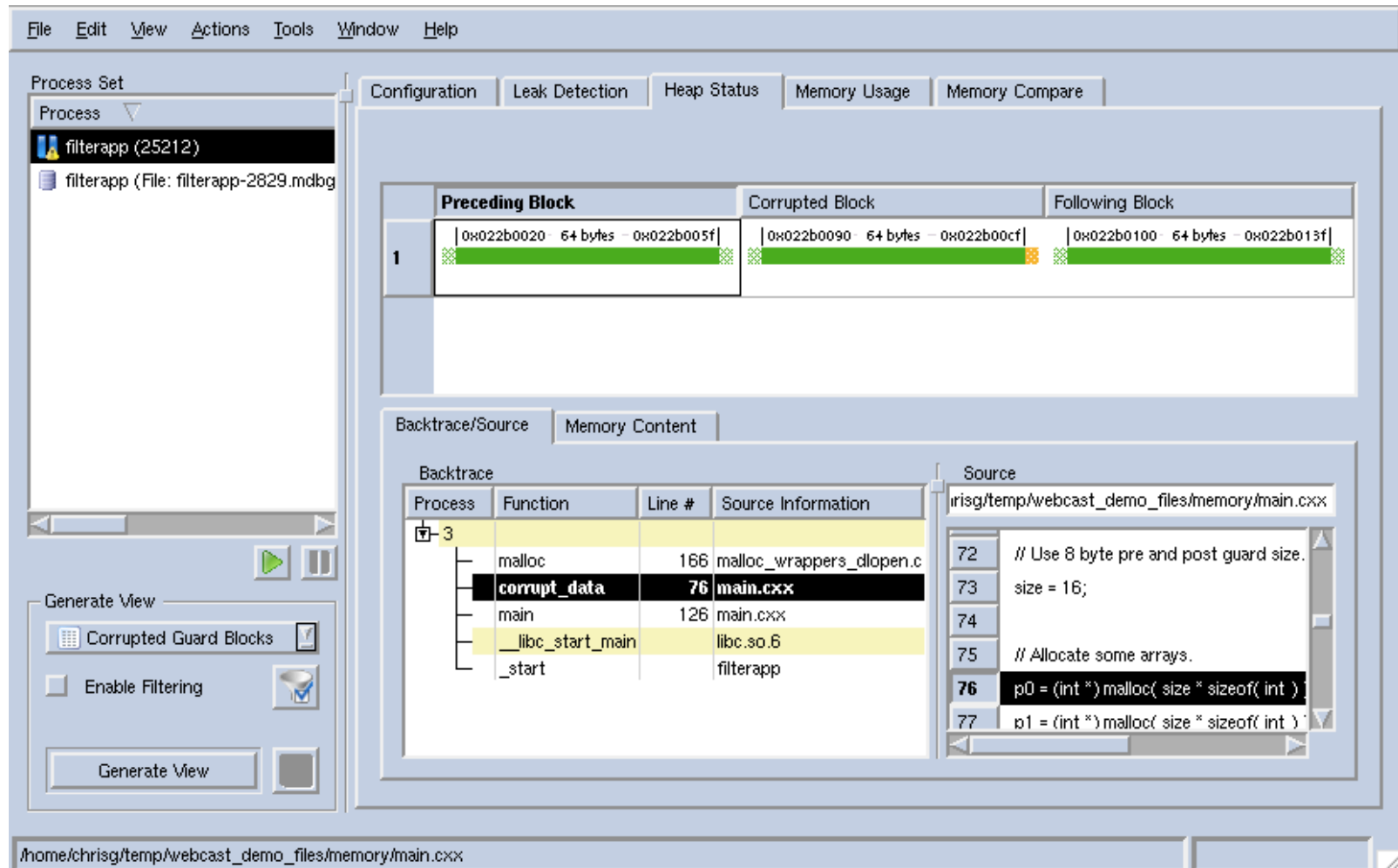
Hexidecimal Count: 80 Bytes 1 2 4 8

0x0804c060	0x77	0x77	0x77	0x77	0x77	0x77	0x77	0x77	0x10	0x00
0x0804c06a	0x00	0x00	0x0f	0x00	0x00	0x00	0x0e	0x00	0x00	0x00
0x0804c074	0x0d	0x00	0x00	0x00	0x0c	0x00	0x00	0x00	0x0b	0x00
0x0804c07e	0x00	0x00	0x0a	0x00	0x00	0x00	0x09	0x00	0x00	0x00
0x0804c088	0x08	0x00	0x00	0x00	0x07	0x00	0x00	0x00	0x06	0x00
0x0804c092	0x00	0x00	0x05	0x00	0x00	0x00	0x04	0x00	0x00	0x00
0x0804c09c	0x03	0x00	0x00	0x00	0x02	0x00	0x00	0x00	0x01	0x00
0x0804c0a6	0x00	0x00	0x00	0x00	0x00	0x00	0x99	0x99	0x99	0x99

Generate
Memory File

Close View in Block Properties window Help

Memory Corruption Report



The screenshot displays the TotalView Memory Corruption Report interface. The main window shows a memory corruption report for a process named 'filterapp (25212)'. The report is organized into several sections:

- Process Set:** Lists the process 'filterapp (25212)' and its file 'filterapp (File: filterapp-2629.mdbg)'.
- Configuration:** Includes tabs for 'Leak Detection', 'Heap Status', 'Memory Usage', and 'Memory Compare'.
- Corrupted Block:** Shows a table with three columns: 'Preceding Block', 'Corrupted Block', and 'Following Block'. The 'Corrupted Block' is highlighted in red, indicating the location of the corruption. The address is 0x022b0090, size is 64 bytes, and the end address is 0x022b00cf.
- Backtrace/Source:** Contains a 'Backtrace' table and a 'Source' code view.

Process	Function	Line #	Source Information
3	malloc	166	malloc_wrappers_dlopen.c
	corrupt_data	76	main.cxx
	main	126	main.cxx
	__libc_start_main		libc.so.6
	_start		filterapp

The 'Source' view shows the following code snippet:

```
irisg/temp/webcast_demo_files/memory/main.cxx
72 // Use 8 byte pre and post guard size.
73 size = 16;
74
75 // Allocate some arrays.
76 p0 = (int *) malloc( size * sizeof( int ) );
77 p1 = (int *) malloc( size * sizeof( int ) );
```

The status bar at the bottom indicates the file path: /home/chrisg/temp/webcast_demo_files/memory/main.cxx.

Enabling Memory Debugging Painting & Hoarding



Paint memory

Paint allocations Pattern: ...

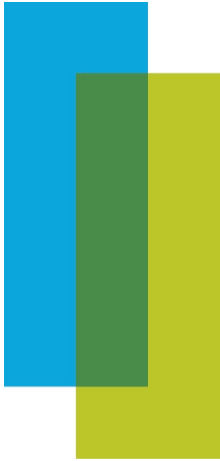
Paint deallocations Pattern: ...

Hoard deallocated memory

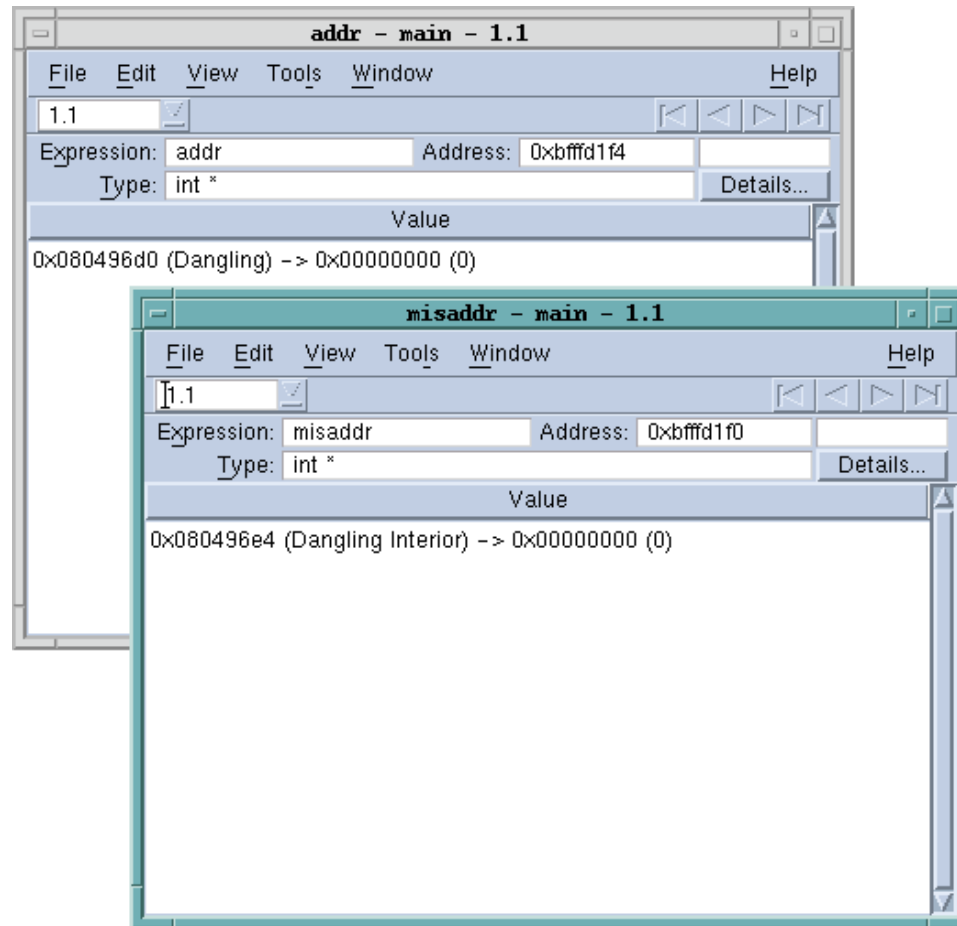
Maximum KB to hoard: ▲▼

Maximum blocks to hoard: ▲▼

Current Size: KB Blocks

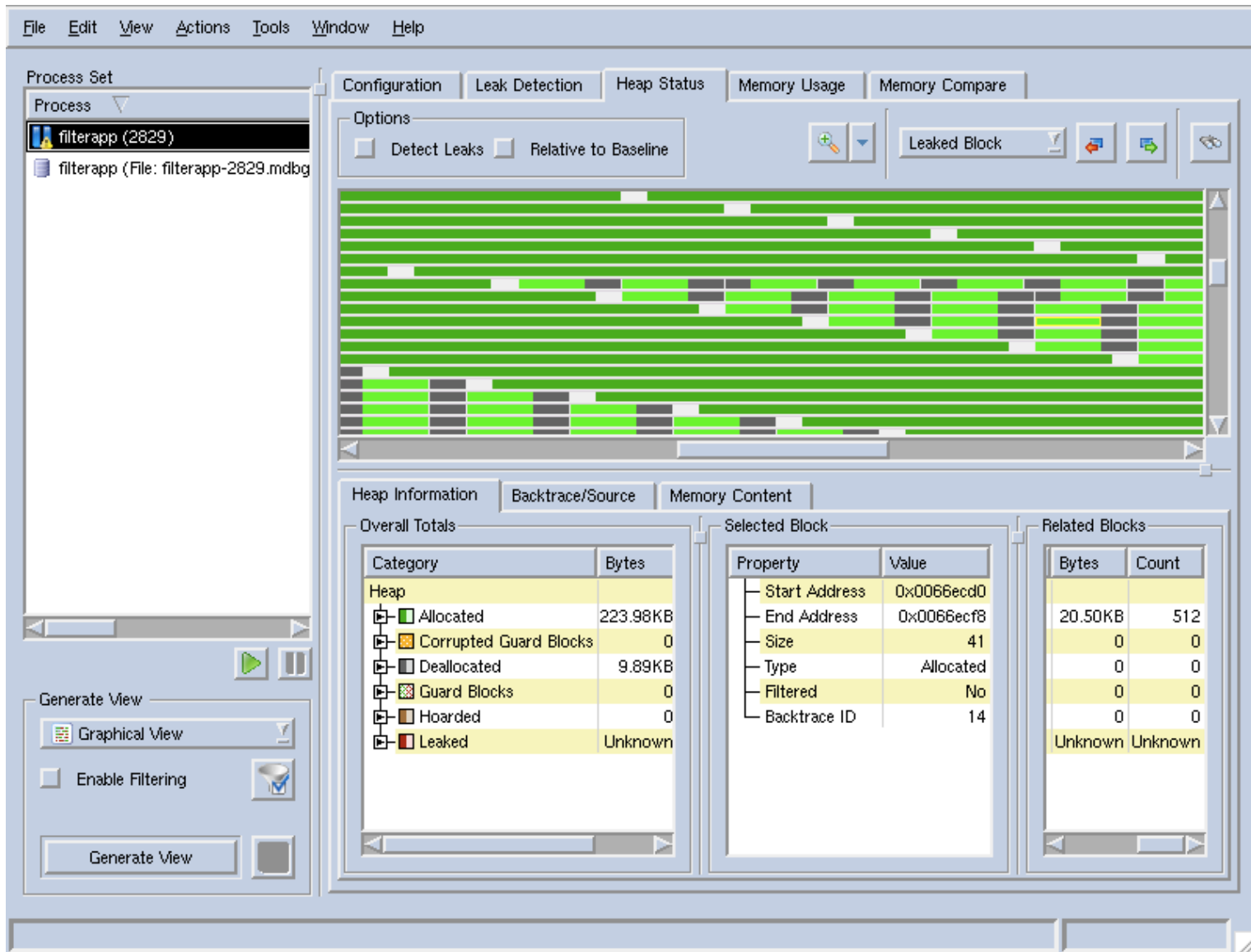


Dangling Pointer Detection



The image shows two debugger windows from TotalView. The top window, titled 'addr - main - 1.1', displays the expression 'addr' at memory address '0xbfffd1f4' with type 'int *'. The value shown is '0x080496d0 (Dangling) -> 0x00000000 (0)'. The bottom window, titled 'misaddr - main - 1.1', displays the expression 'misaddr' at memory address '0xbfffd1f0' with type 'int *'. The value shown is '0x080496e4 (Dangling Interior) -> 0x00000000 (0)'. Both windows have a menu bar with 'File', 'Edit', 'View', 'Tools', 'Window', and 'Help'.

Heap Graphical View



The screenshot displays the TotalView Heap Graphical View interface. The main window is titled "Heap Graphical View" and contains several panels and tabs.

Process Set: A list of processes is shown, with "filterapp (2829)" selected.

Configuration: Includes tabs for "Configuration", "Leak Detection", "Heap Status", "Memory Usage", and "Memory Compare". The "Options" section has checkboxes for "Detect Leaks" and "Relative to Baseline".

Graphical View: A large area displaying a graphical representation of memory blocks, with green bars indicating allocated memory and grey bars indicating deallocated or leaked memory.

Heap Information: A table showing overall totals for various heap categories.

Category	Bytes
Heap	
Allocated	223.98KB
Corrupted Guard Blocks	0
Deallocated	9.89KB
Guard Blocks	0
Hoarded	0
Leaked	Unknown

Selected Block: A table showing properties for a selected block.

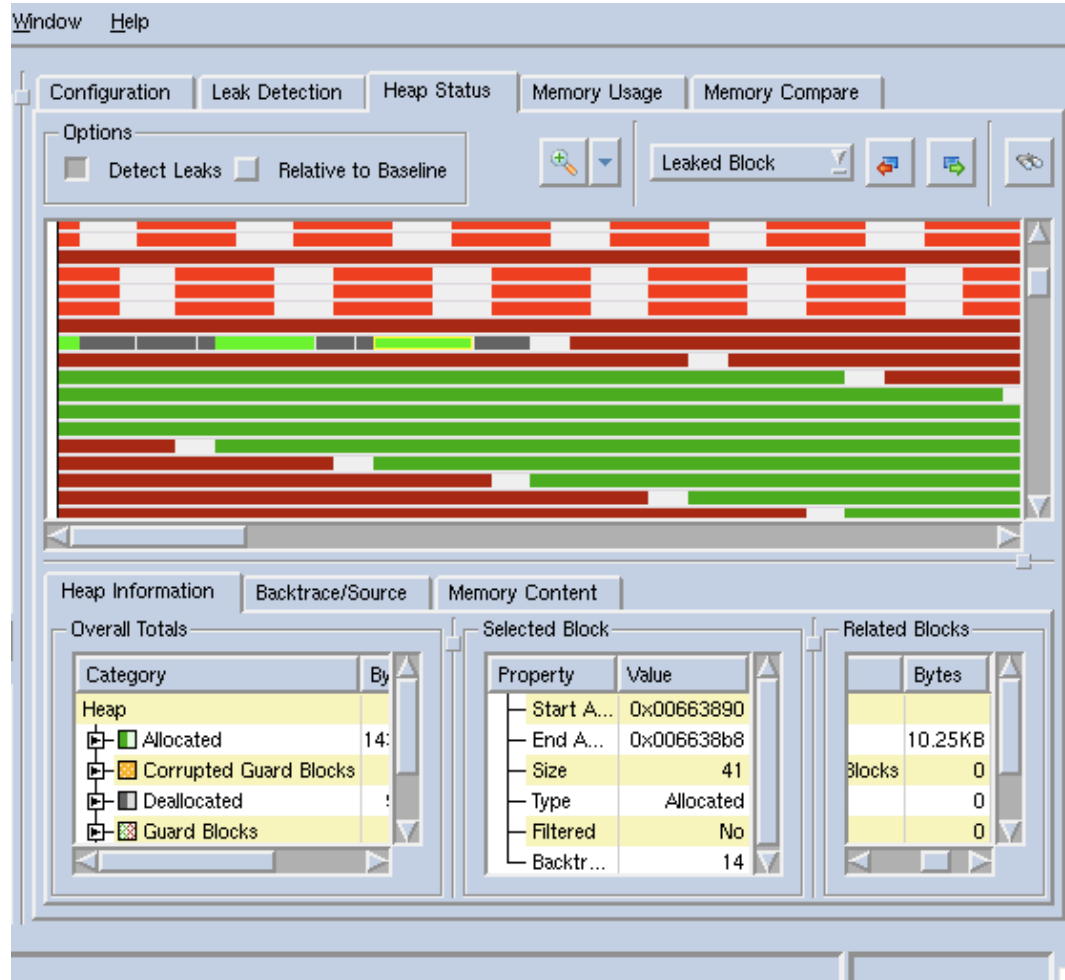
Property	Value
Start Address	0x0066ecd0
End Address	0x0066ecf8
Size	41
Type	Allocated
Filtered	No
Backtrace ID	14

Related Blocks: A table showing related blocks.

Bytes	Count
20.50KB	512
0	0
0	0
0	0
0	0
Unknown	Unknown

Generate View: A section with a "Generate View" button and a "Graphical View" dropdown menu.

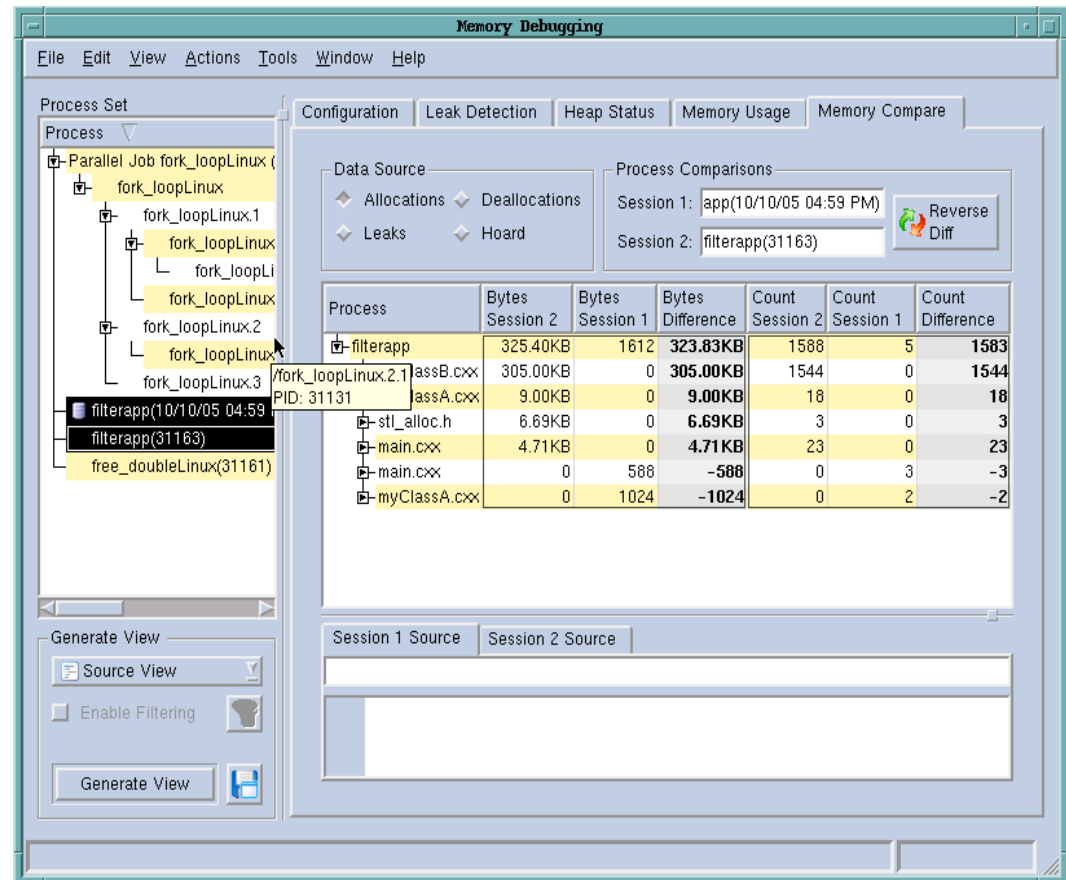
Leak Detection



- Based on Conservative Garbage Collection
- Can be performed at any point in runtime
- Helps localize leaks in time

Memory Comparisons

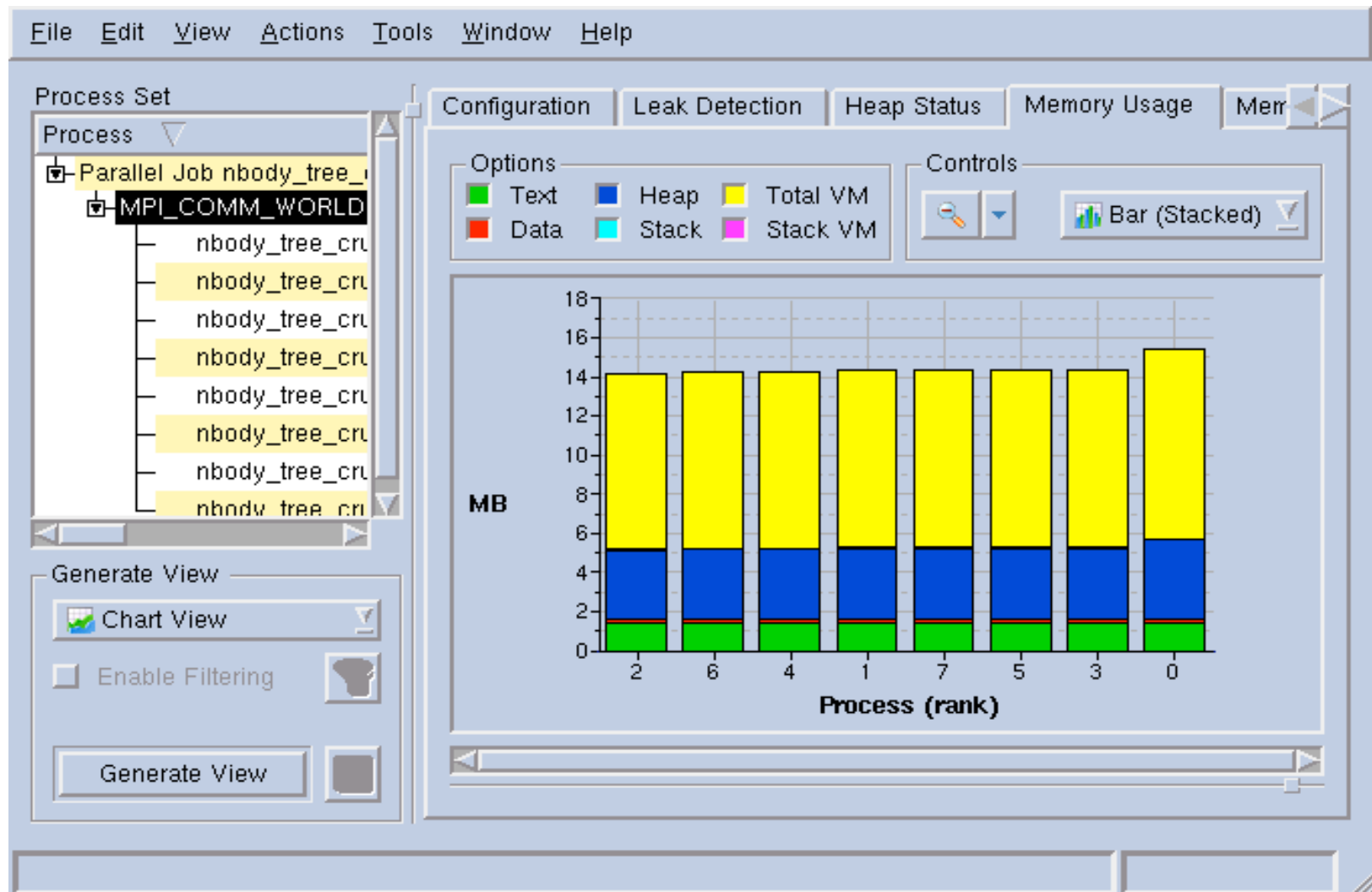
- **“Diff” live processes**
 - Compare processes across cluster
- **Compare with baseline**
 - See changes between point A and point B
- **Compare with saved session**
 - Provides memory usage change from last run



The screenshot shows the 'Memory Debugging' application window. The 'Process Set' tree on the left lists several processes, including 'filterapp(10/10/05 04:59)' and 'filterapp(31163)'. The 'Memory Compare' tab is active, showing a comparison between 'Session 1: app(10/10/05 04:59 PM)' and 'Session 2: filterapp(31163)'. A table displays the following data:

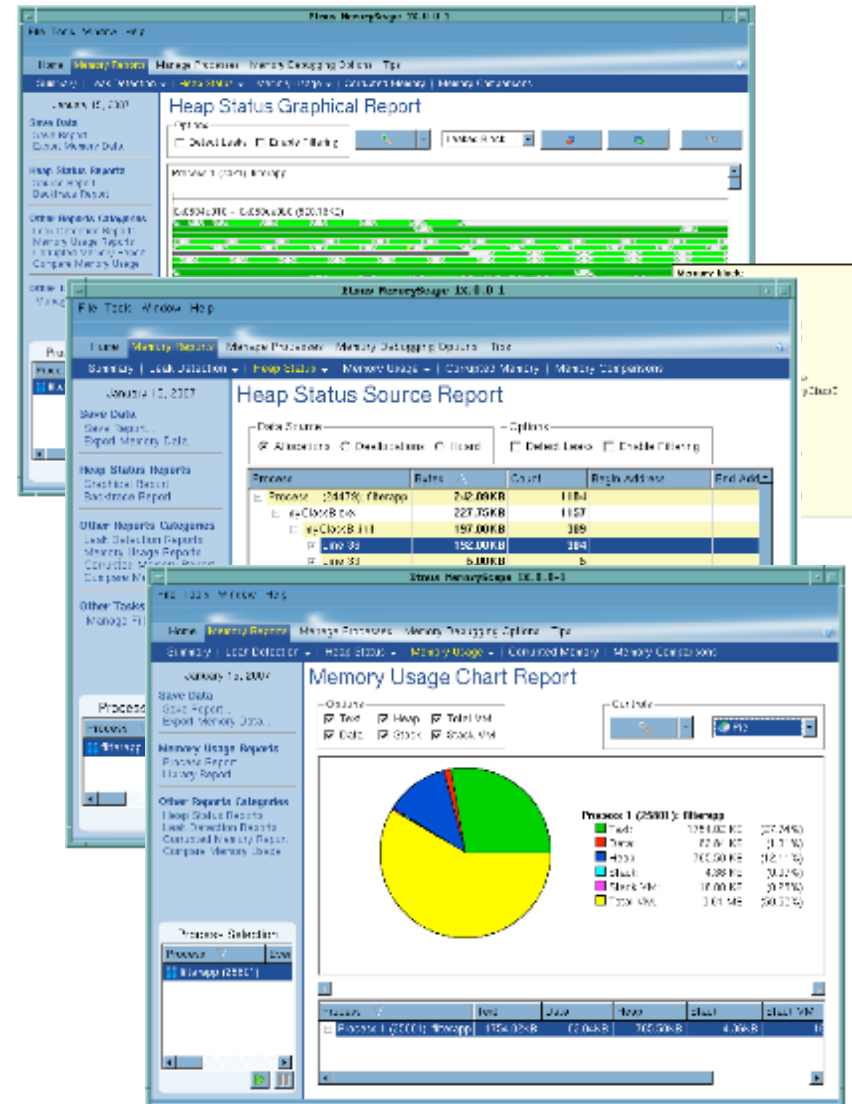
Process	Bytes Session 2	Bytes Session 1	Bytes Difference	Count Session 2	Count Session 1	Count Difference
filterapp	325.40KB	1612	323.83KB	1588	5	1583
fork_loopLinux.2\assB.cxx	305.00KB	0	305.00KB	1544	0	1544
fork_loopLinux.2\assA.cxx	9.00KB	0	9.00KB	18	0	18
stl_alloc.h	6.69KB	0	6.69KB	3	0	3
main.cxx	4.71KB	0	4.71KB	23	0	23
main.cxx	0	588	-588	0	3	-3
myClassA.cxx	0	1024	-1024	0	2	-2

Memory Usage Statistics



Memory Reports

- **Multiple Reports**
 - Memory Statistics
 - Interactive Graphical Display
 - Source Code Display
 - Backtrace Display
 - HTML - interactive format
- **Allow the user to**
 - Monitor Program Memory Usage
 - Discover Allocation Layout
 - Look for Inefficient Allocation
 - Find Memory Leaks



The screenshots illustrate the tool's capabilities in monitoring memory usage. The top window displays a 'Heap Status Graphical Report' showing memory allocation over time. The middle window shows a 'Heap Status Source Report' with a table of memory allocations:

Process	File	Line	Size	Count	Begin Address	End Address
Process 1 (25476): libexpat			242,000B	1164		
Process 1 (25476): libexpat			227,750B	1157		
Process 1 (25476): libexpat			157,000B	389		
Process 1 (25476): libexpat			152,000B	384		
Process 1 (25476): libexpat			6,000B	6		

The bottom window shows a 'Memory Usage Chart Report' with a pie chart and a table of memory usage by process:

Process	File	Line	Size	Count	Begin Address	End Address
Process 1 (25476): libexpat			102,000B	27,24%		
Process 1 (25476): libexpat			77,000B	19,1%		
Process 1 (25476): libexpat			703,500B	17,6%		
Process 1 (25476): libexpat			4,388B	1,1%		
Process 1 (25476): libexpat			17,000B	4,3%		
Process 1 (25476): libexpat			1,614B	0,4%		

Script Mode – MemScript - Tvscript

- **Automation Support**

- Scripting lets users run tests and check programs for memory leaks without having to be in front of the program
- Simple command line program
 - Doesn't start up the GUI
 - Can be run from within a script or test harness
- The user defines
 - What configuration options are active
 - What thing to look for
 - Actions to be taken for each type of event that may occur

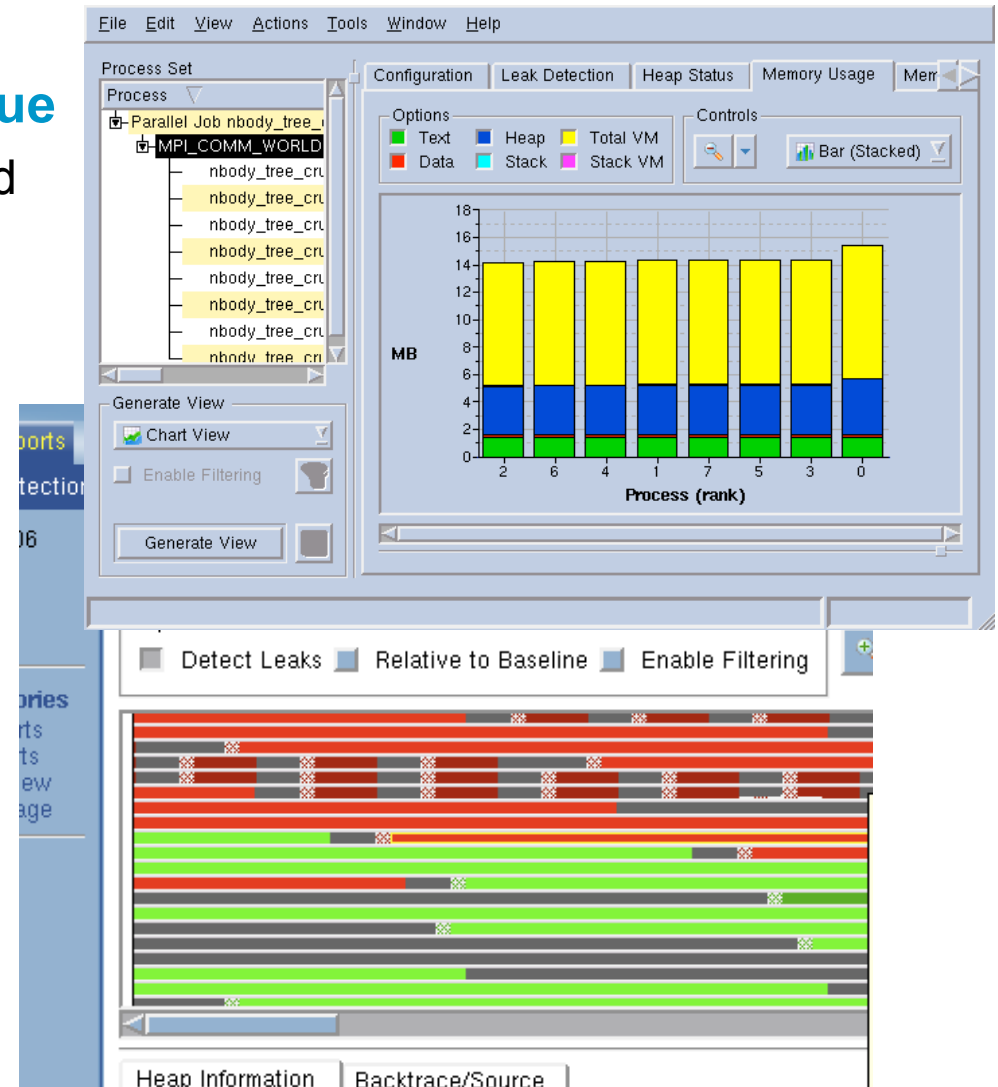
Parallel Memory Debugging

- **Memory is a growing issue**

- Node resources are limited
- Predicting and managing memory usage across parallel applications is complex

- **Analysis may include**

- Comparing usage across
 - Processes of job
 - Time
 - Datasets
- Exploring layout of allocations
- Leak detection
- Buffer overflow detection



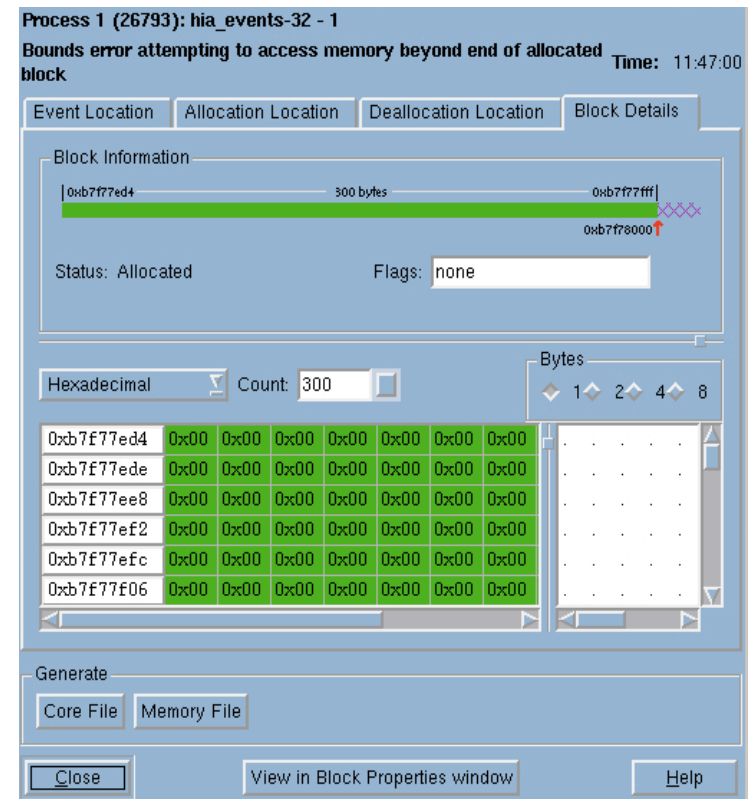
TotalView provides complimentary set of memory ‘tools’

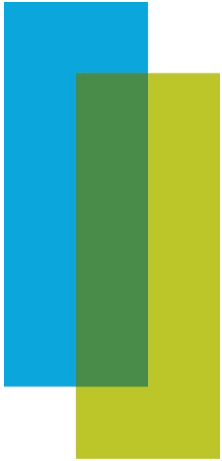
- **Guard Blocks**
 - Low runtime overhead, small size, over- and under-runs
 - Identify heap allocation bounds errors after the fact
- **HIA Events**
 - Low overhead, only catch certain types of errors
- **Memory Statistics**
 - No overhead, very high level, pick out outliers and patterns
- **Heap Graphical Display**
 - Detailed view, understand re-allocation and fragmentation behavior
- **Leak Detection**
 - Analysis of state
- **(de)allocation Hoard**
 - Helps identify dangling pointers
- **RedZones (TV 8.7, MS 2.5)**
 - Low runtime overhead, large size, over- or under-runs
 - Flags heap allocation bounds errors as they happen

Coming in TotalView 8.7 and MemoryScape 3.0

- Redzones -

- **Allocates a “protected page”**
 - adjacent to selected heap allocations
 - Before or after
- **A write into this space triggers immediate events**
 - Event occurs as the write is happening
- **Pages have a fixed size**
 - If there are many heap allocations this can potential have a large memory usage overhead
- **Ways to manage Redzones memory overhead**
 - Turn redzones on and off manually
 - Specify (by size) what allocations you want to have redzones on





Thanks!

QUESTIONS?

totalviewtech.com