
Real-time Debugging using GDB Tracepoints and other Eclipse features

GCC Summit 2010

2010-010-26

marc.khouzam@ericsson.com

Summary

- › Introduction
- › Advanced debugging features
 - Non-stop multi-threaded debugging
 - Pretty-printing of complex structures
 - Multi-process debugging
 - Reverse debugging
 - Multi-core debugging
- › GDB Tracepoints

Introduction

- › Many companies deal with embedded systems
- › Linux is widely used in the embedded space
- › Applications are complex and have complex interactions
- › Use of different targets
 - Different OS: Linux, Real-time OS, proprietary OS
 - Different architectures
 - Different environments: design, test, integration, live site
 - Different setups : Simulator, Real hardware, Lab, JTAG

Introduction

- › Need for a debugging tool to address those situations
- › Same tool for design, test, integration, live sites
- › Same tool for simulator, real-target
- › Same tool for different archs, OS
- › Same tool for different types of users
- GDB provides the advanced debugging features
- Eclipse Integration provides the ease-of-use and efficiency

Features

Now

- › Non-Stop multi-threading
- › Partial Pretty-printing
- › Single space multi-process
- › Reverse
- › Any binary debugging
- › Tracepoints

Next

- › Full Pretty-printing
- › Full Multi-process
- › Multi-core debugging
- › Global breakpoints
- › Tracepoints improvements
 - Fast tracepoints
 - Static tracepoints
 - Observer-mode
 - Intelligent trace visualization

Features

Now

- > **Non-Stop multi-threading**
- > Partial Pretty-printing
- > Single space multi-process
- > Reverse
- > Any binary debugging
- > Tracepoints

Next

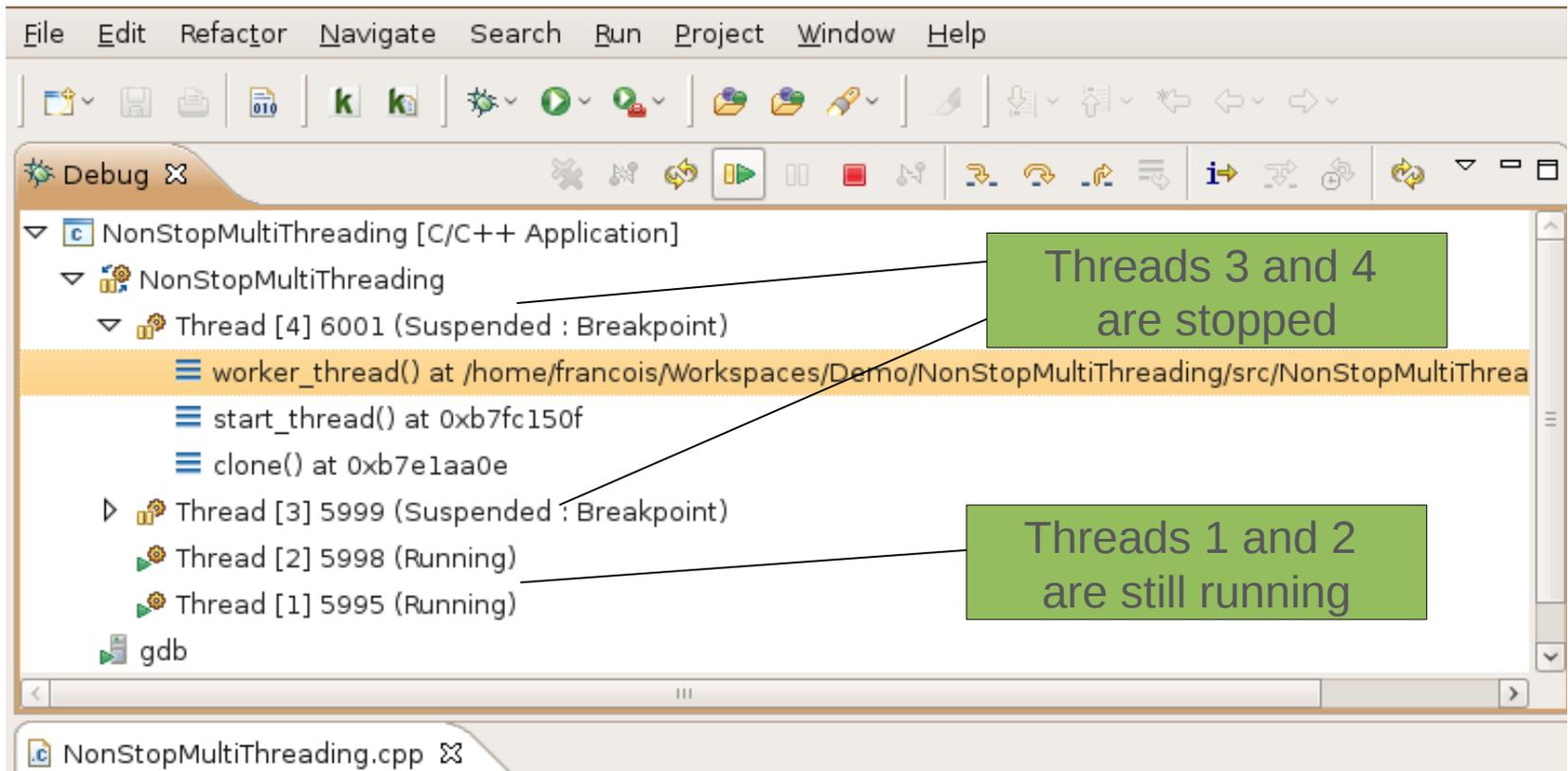
- > Full Pretty-printing
- > Full Multi-process
- > Multi-core debugging
- > Global breakpoints
- > Tracepoints improvements
 - Fast tracepoints
 - Static tracepoints
 - Observer-mode
 - Intelligent trace visualization

Non-Stop multi-threading

- Debugging a process by stopping its execution might cause the program to change its behavior drastically
- Some threads should not be interrupted for proper program execution
 - Heartbeat threads
 - Monitoring threads
 - Server threads
- Non-stop allows to stop and examine a subset of threads, while other threads continue to run freely.

Non-Stop multi-threading

- Allows to individually control threads
 - Step, Resume, Suspend



Features

Now

- › Non-Stop multi-threading
- › **Partial Pretty-printing**
- › Single space multi-process
- › Reverse
- › Any binary debugging
- › Tracepoints

Next

- › **Full Pretty-printing**
- › Full Multi-process
- › Multi-core debugging
- › Global breakpoints
- › Tracepoints improvements
 - Fast tracepoints
 - Static tracepoints
 - Observer-mode
 - Intelligent trace visualization

Pretty-printing

- Content of complex abstract data structures should be presented to the user while keeping the abstraction.
 - Vectors
 - List
 - Maps
 - User-defined structure
- GDB provides Python pretty-printing feature which is STL-ready

Pretty-printing (Now)

The screenshot shows a debugger's Variables window with the following table:

Name	Type	Value
coll	std::vector<std::vector<int, std::alloca...	{...}
std::_Vector_base<std::vector<...	std::_Vector_base<std::vector<int, std:...	{...}
_M_impl	std::_Vector_base<std::vector<int, std:...	{...}
std::allocator<std::vector<...	std::allocator<std::vector<int, std::allo...	{...}
_M_start	std::vector<int, std::allocator<int> > *	0x8055358
_M_finish	std::vector<int, std::allocator<int> > *	0x8055388
_M_end_of_storage	std::vector<int, std::allocator<int> > *	0x8055388
str	std::string	{...}

Below the table, the details for the selected variable 'coll' are shown:

```
Name : coll
Details: {<std::_Vector_base<std::vector<int, std::allocator<int> >, std::allocator<std::vect
Default: {...}
Decimal: {...}
Hex: {...}
Binary: {...}
Octal: {...}
```

A green box with the text "No pretty-printing" has an arrow pointing to the "Details" line of the variable 'coll'.

The screenshot shows the details for the variable 'coll' with the following text:

```
Name : coll
Details: std::vector of length 4, capacity 4 = {std::vector of length 3, capacity 3 = {1, 2,
Default: {...}
Decimal: {...}
Hex: {...}
Binary: {...}
Octal: {...}
```

A green box with the text "Partial pretty-printing" has an arrow pointing to the "Details" line of the variable 'coll'.

Pretty-printing (Next)

- Display content in user-friendly fashion
- Allows to modify content directly!

The screenshot shows a debugger's Variables window with the following structure:

Name	Type	Value
coll	std::vector<std::vector<int, std::allocator<int> >	{...}
[0]	std::vector<int, std::allocator<int> >	{...}
(x)= [0]	int	1
(x)= [1]	int	2
(x)= [2]	int	3
[1]	std::vector<int, std::allocator<int> >	{...}
(x)= [0]	int	10
(x)= [1]	int	11
(x)= [2]	int	12

Below the table, the details for the selected variable 'coll' are shown:

```
Name : coll
Details:std::vector of length 4, capacity 4 = {std::vector of length 3, capacity 3 = {1, 2,
Default:{...}
Decimal:{...}
Hex:{...}
Binary:{...}
Octal:{...}
```

A green box with the text "Full pretty-printing with editable values" is positioned over the details section. Two arrows point from this box to the "(x)= [0]" and "(x)= [1]" entries in the variable list, indicating that these values are directly editable.

Features

Now

- › Non-Stop multi-threading
- › Partial Pretty-printing
- › **Single space multi-process**
- › Reverse
- › Any binary debugging
- › Tracepoints

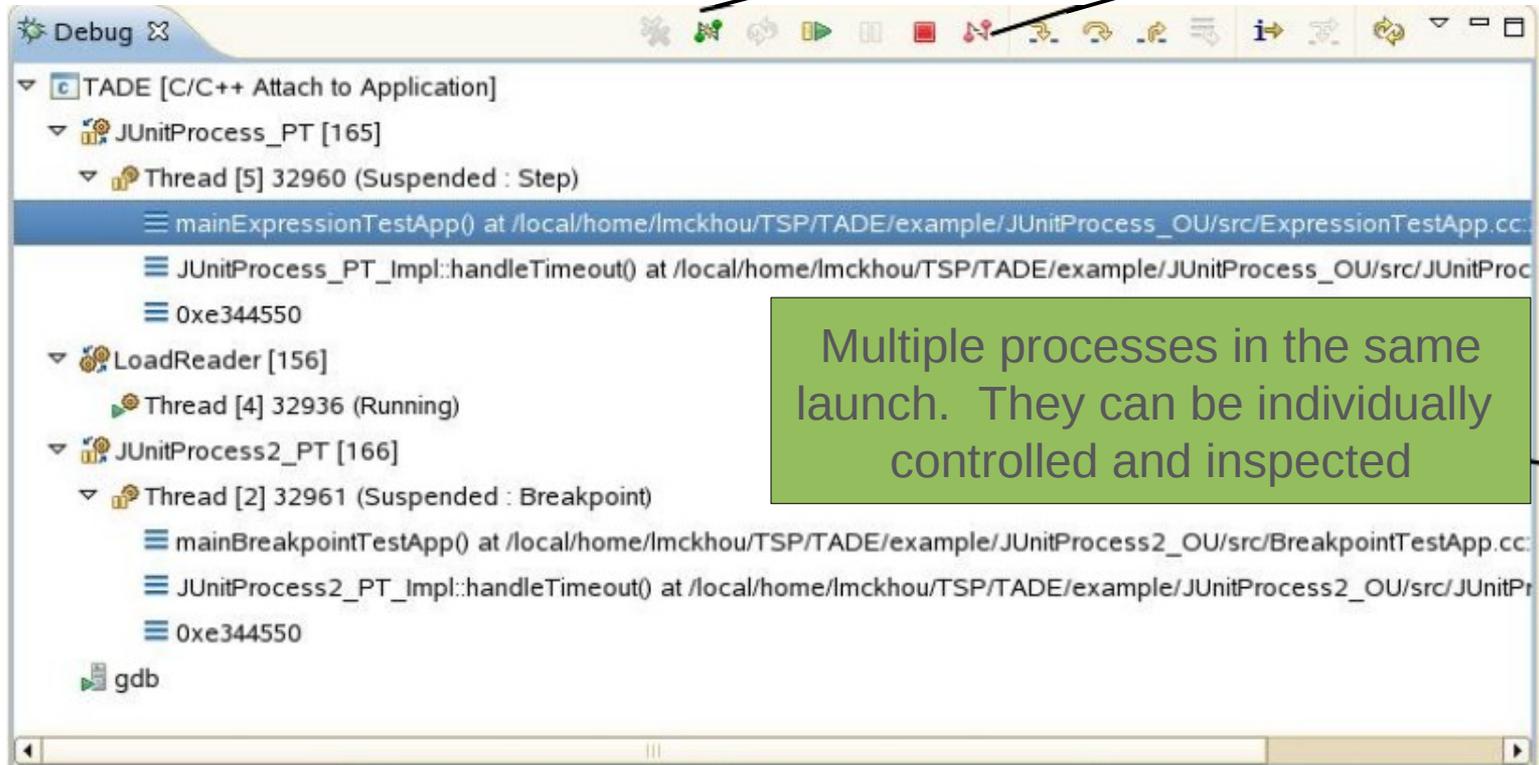
Next

- › Full Pretty-printing
- › **Full Multi-process**
- › Multi-core debugging
- › Global breakpoints
- › Tracepoints improvements
 - Fast tracepoints
 - Static tracepoints
 - Observer-mode
 - Intelligent trace visualization

Multi-process (Now)

- Currently available for targets that have a single memory space for all processes

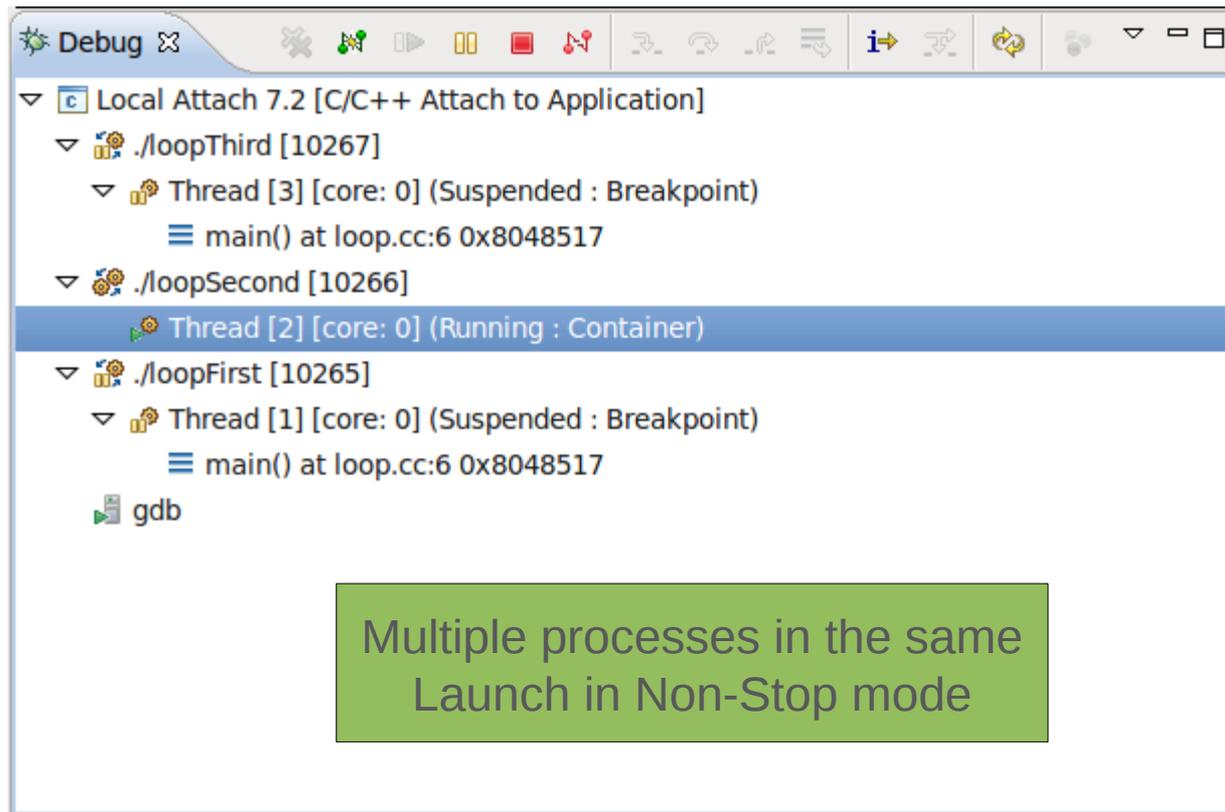
Dynamically connect/disconnect



Multiple processes in the same launch. They can be individually controlled and inspected

Multi-process (Next)

- Current work to bring this to Linux using GDB 7.2 for next release



Features

Now

- › Non-Stop multi-threading
- › Partial Pretty-printing
- › Single space multi-process
- › **Reverse**
- › Any binary debugging
- › Tracepoints

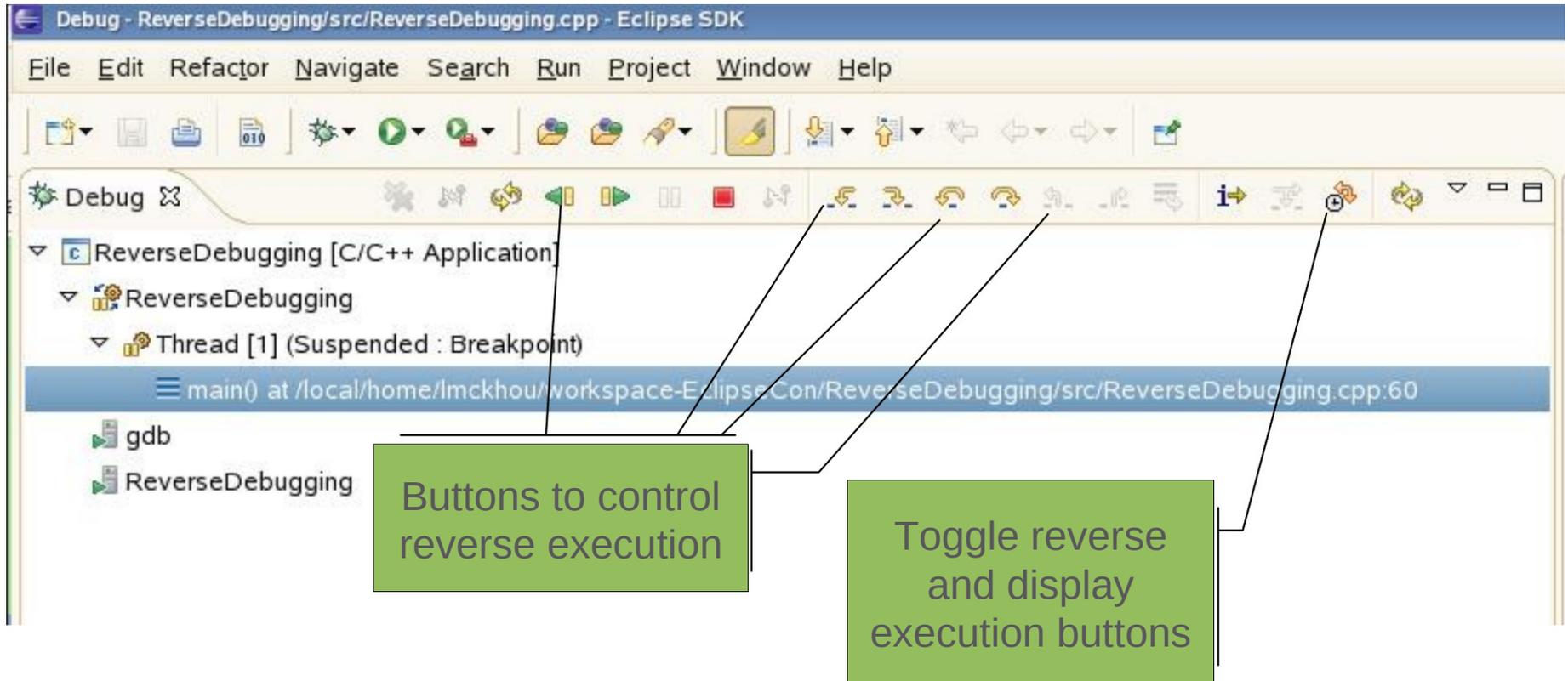
Next

- › Full Pretty-printing
- › Full Multi-process
- › Multi-core debugging
- › Global breakpoints
- › Tracepoints improvements
 - Fast tracepoints
 - Static tracepoints
 - Observer-mode
 - Intelligent trace visualization

Reverse debugging

- Often, when debugging, you realize that you have gone too far and some event of interest has already happened.
- Restarting execution to reach that same failure can be tedious and time consuming
- Why not simply *go backwards*?
- Undo the changes in machine state that have taken place as the program was executing normally i.e., revert registers and memory to previous values
- GDB provides Process Record and Replay for Linux
- Allows to go backwards, modify memory/registers, then resume execution on a new path!

Reverse debugging



Features

Now

- › Non-Stop multi-threading
- › Partial Pretty-printing
- › Single space multi-process
- › Reverse
- › Any binary debugging
- › Tracepoints

Next

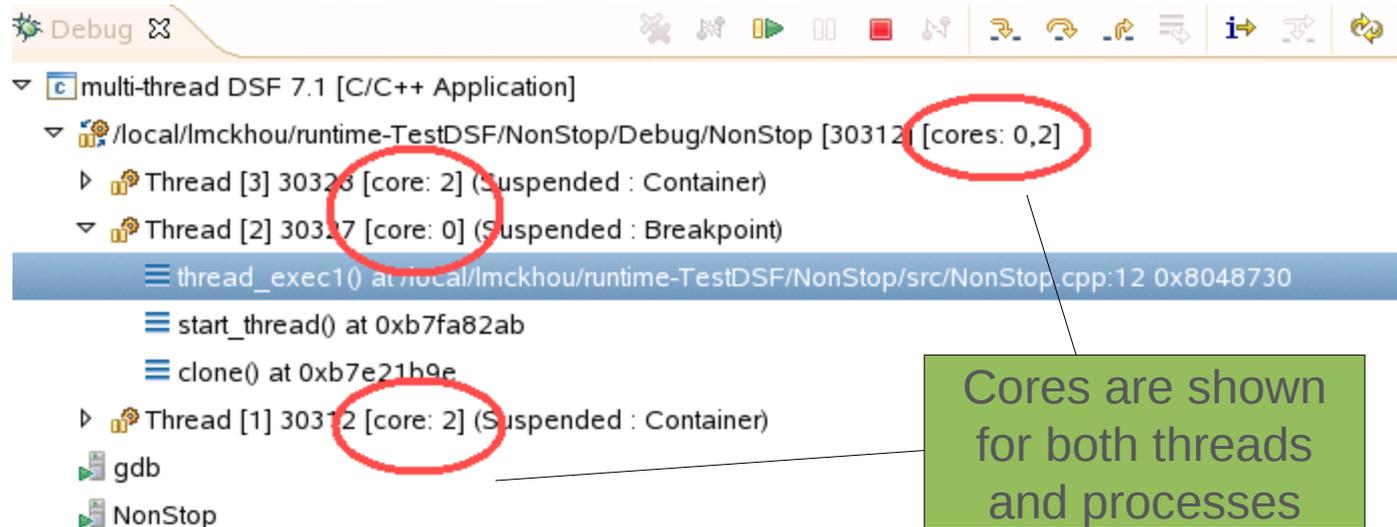
- › Full Pretty-printing
- › Full Multi-process
- › **Multi-core debugging**
- › Global breakpoints
- › Tracepoints improvements
 - Fast tracepoints
 - Static tracepoints
 - Observer-mode
 - Intelligent trace visualization

Multi-core debugging

- As systems get more complex, so does the software running on them
- Debugging tools must provide more information to describe these complex systems
- Multi-core systems are the default now
- Troubleshooting requires having knowledge of what is running where

Multi-core debugging

- First step in upcoming broader multi-core debugging support
- Indicates core information to the user



Others

- Any binary debugging (Now)
 - Allows to debug any binary without having to build it in Eclipse
 - Almost immediate debugging of GDB or GCC!
- Automatic remote launching (Next)
 - Will automatically start gdbserver on your target
- Global breakpoints (Next)
 - Allows to stop processes that don't have the debugger attached to it
 - Essential for short-lived processes
 - Essential for startup-sequence debugging on a real target

Features

Now

- › Non-Stop multi-threading
- › Partial Pretty-printing
- › Single space multi-process
- › Reverse
- › Any binary debugging
- › **Tracepoints**

Next

- › Full Pretty-printing
- › Full Multi-process
- › Multi-core debugging
- › Global breakpoints
- › **Tracepoints improvements**
 - Fast tracepoints
 - Static tracepoints
 - Observer-mode
 - Intelligent trace visualization

Dynamic Tracing

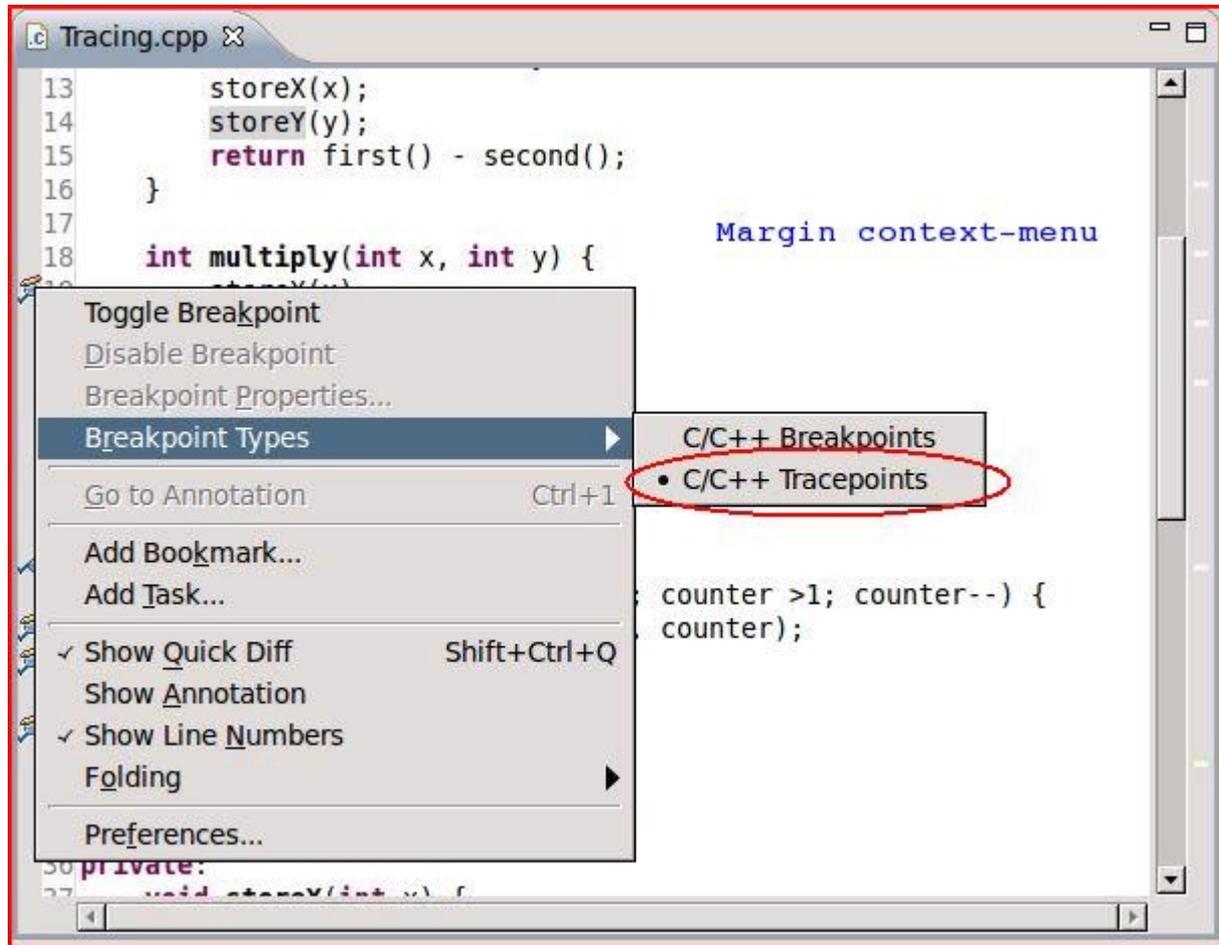
- › Using a debugger drastically changes execution
- › In some cases, a debugger is too intrusive :
 - Debugging a race condition
 - Investigating user-interface issues
 - Live sites
 - Real-time systems
- › Low-overhead tracing is the answer: LTTng, UST
- › What if existing static traces don't give info needed?
- › What about systems that are not instrumented?
- Eclipse's integration of GDB's Dynamic Tracepoints

Eclipse Tracepoints

- › Creation of tracepoints is done as for breakpoints
- › Enable/Disable
- › Dynamic condition
- › Specification of data to be gathered using symbolic expressions and memory addresses (actions)
- › Pass count
- › Trace-state variables can be used in conditions and actions
- › Tracepoints are only in effect if tracing is enabled

Eclipse Tracepoints Selection

- › Tracepoints treated as breakpoints



Eclipse Tracepoints Display

- › Tracepoints
- › Tracepoints with actions

The screenshot displays the Eclipse IDE interface. On the left, the 'Tracing.cpp' editor shows C++ code with several tracepoints (bug icons) placed at lines 19, 29, 30, and 32. On the right, the 'Breakpoints' window lists these tracepoints with their file paths and line numbers. A red circle highlights the tracepoint at line 7 in the Breakpoints window, and another red circle highlights the tracepoints at lines 19, 29, 30, and 32 in the code editor.

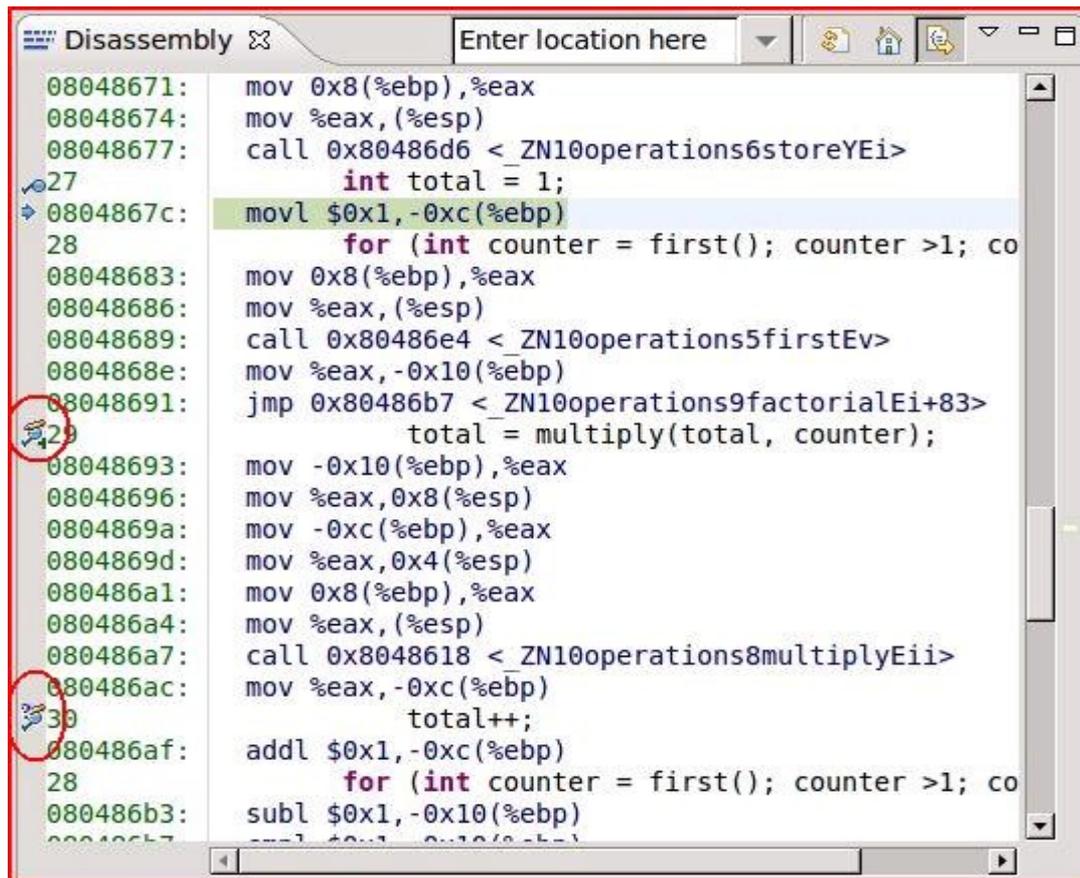
```
13     storeX(x);
14     storeY(y);
15     return first() - second();
16 }
17
18 int multiply(int x, int y) {
19     storeX(x);
20     storeY(y);
21     return first() * second();
22 }
23
24 int factorial(int y) {
25     storeY(y);
26
27     int total = 1;
28     for (int counter = first(); counter > 1; counter--) {
29         total = multiply(total, counter);
30         total++;
31     }
32
33     return total;
34 }
35
```

Breakpoints window:

- /home/lmckhou/runtime/Tracing/src/Tracing.cpp [line: 27]
- /home/lmckhou/runtime/Tracing/src/Tracing.cpp [line: 64]
- /home/lmckhou/runtime/Tracing/src/Tracing.cpp [line: 7]
- /home/lmckhou/runtime/Tracing/src/Tracing.cpp [line: 19]
- /home/lmckhou/runtime/Tracing/src/Tracing.cpp [line: 29]
- /home/lmckhou/runtime/Tracing/src/Tracing.cpp [line: 30]
- /home/lmckhou/runtime/Tracing/src/Tracing.cpp [line: 32]

Eclipse Tracepoints Disassembly

- › Disassembly view support for Tracepoints
- › Tracepoint with condition

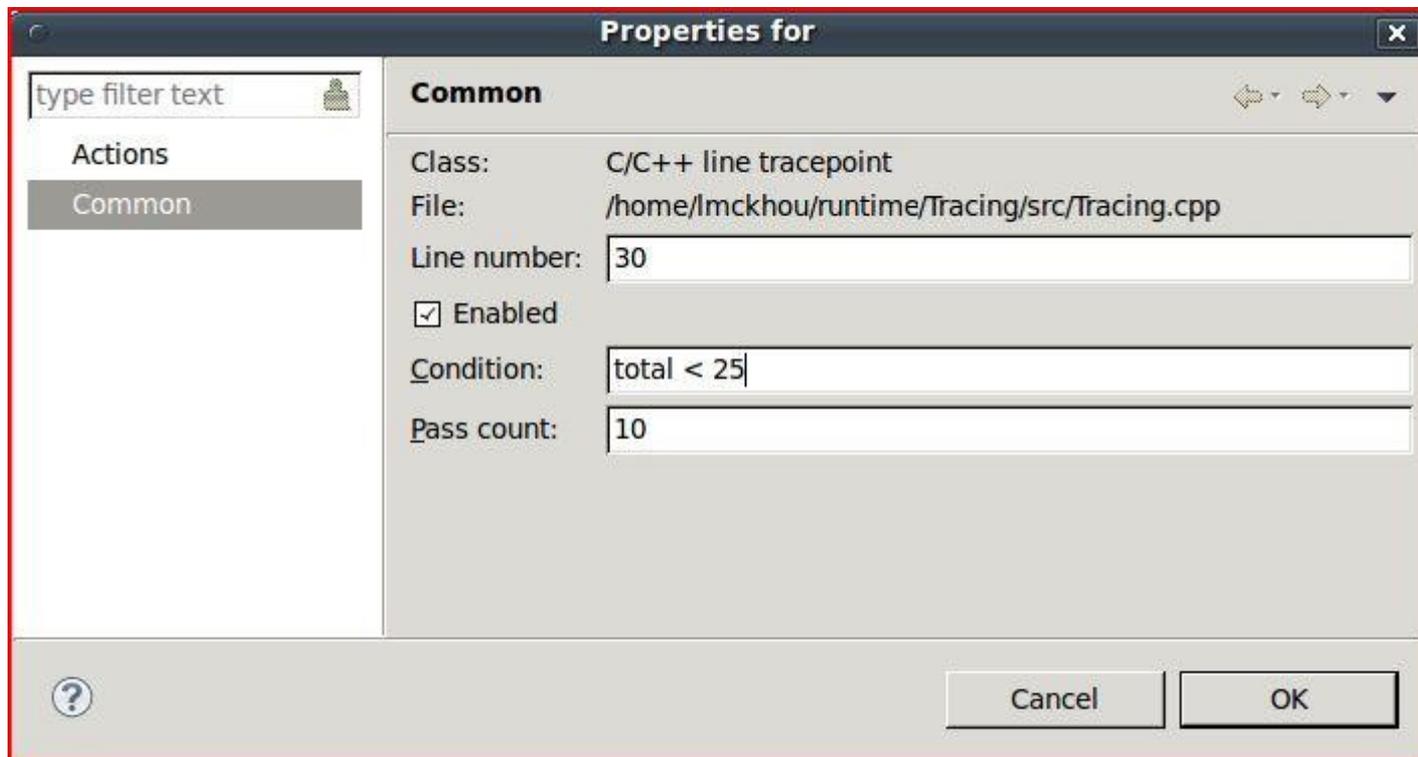


The screenshot shows the Eclipse IDE's Disassembly view. The window title is "Disassembly" and it has a search bar "Enter location here". The assembly code is displayed in a list format with addresses on the left and instructions on the right. A red box highlights the entire window. Two specific instructions are circled in red: the instruction at address 08048691, which is a jump instruction, and the instruction at address 080486ac, which is a move instruction. The code includes comments in C++ style, such as "int total = 1;" and "for (int counter = first(); counter > 1; co".

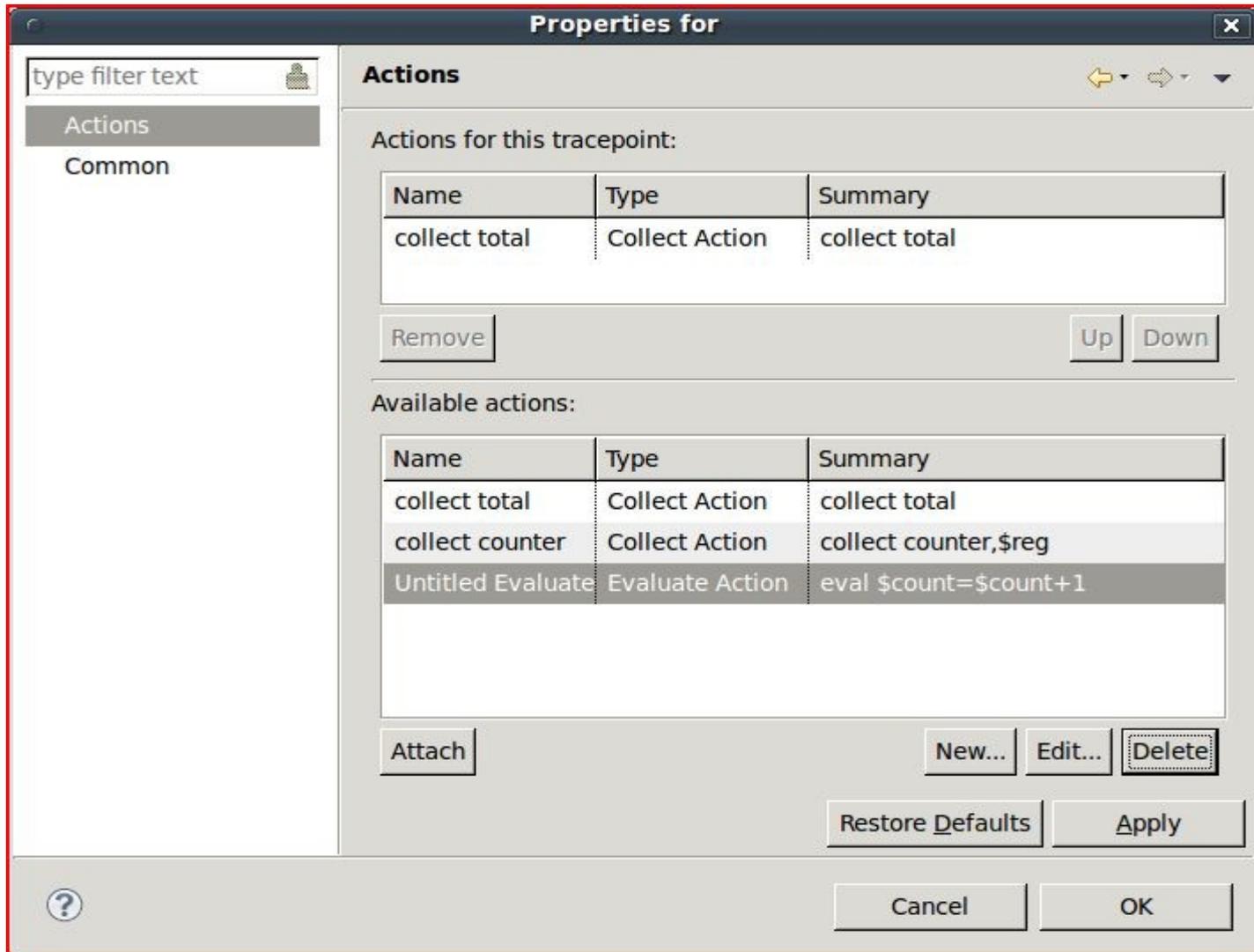
```
08048671: mov 0x8(%ebp),%eax
08048674: mov %eax,(%esp)
08048677: call 0x80486d6 <_ZN10operations6storeYEi>
27      int total = 1;
0804867c: movl $0x1,-0xc(%ebp)
28      for (int counter = first(); counter > 1; co
08048683: mov 0x8(%ebp),%eax
08048686: mov %eax,(%esp)
08048689: call 0x80486e4 <_ZN10operations5firstEv>
0804868e: mov %eax,-0x10(%ebp)
08048691: jmp 0x80486b7 <_ZN10operations9factorialEi+83>
29      total = multiply(total, counter);
08048693: mov -0x10(%ebp),%eax
08048696: mov %eax,0x8(%esp)
0804869a: mov -0xc(%ebp),%eax
0804869d: mov %eax,0x4(%esp)
080486a1: mov 0x8(%ebp),%eax
080486a4: mov %eax,(%esp)
080486a7: call 0x8048618 <_ZN10operations8multiplyEii>
080486ac: mov %eax,-0xc(%ebp)
30      total++;
080486af: addl $0x1,-0xc(%ebp)
28      for (int counter = first(); counter > 1; co
080486b3: subl $0x1,-0x10(%ebp)
```

Eclipse Tracepoints Properties

- › Tracepoints properties
 - Location
 - Enablement
 - Condition
 - Pass count

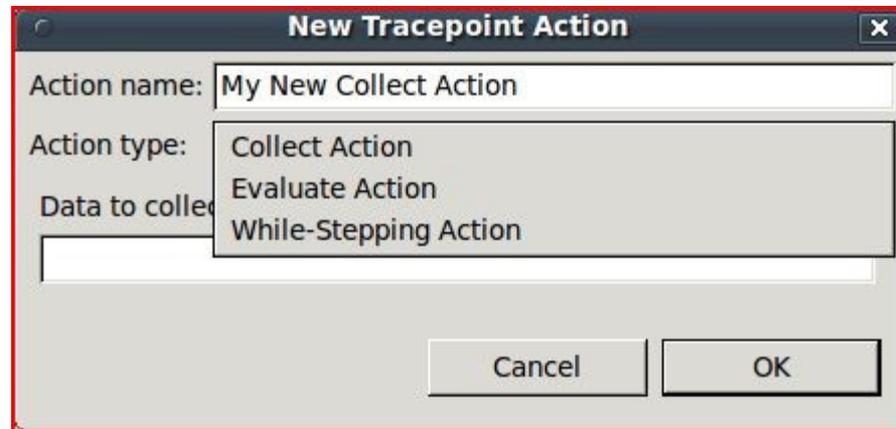


Eclipse Tracepoints Actions

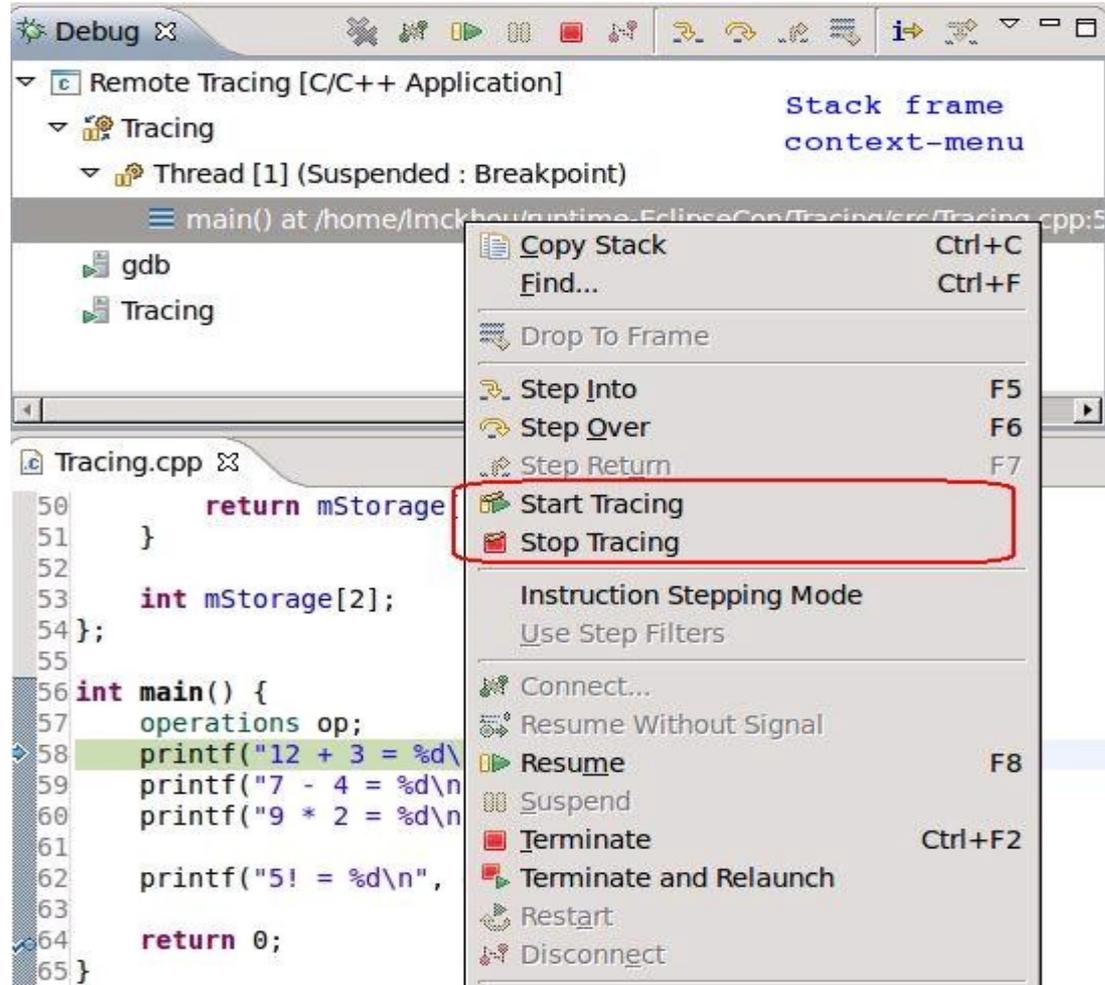


Eclipse Tracepoints Actions

- › Tracepoints action types
 - Collect
 - Evaluate
 - While-Stepping
 - › Collect
 - › Evaluate

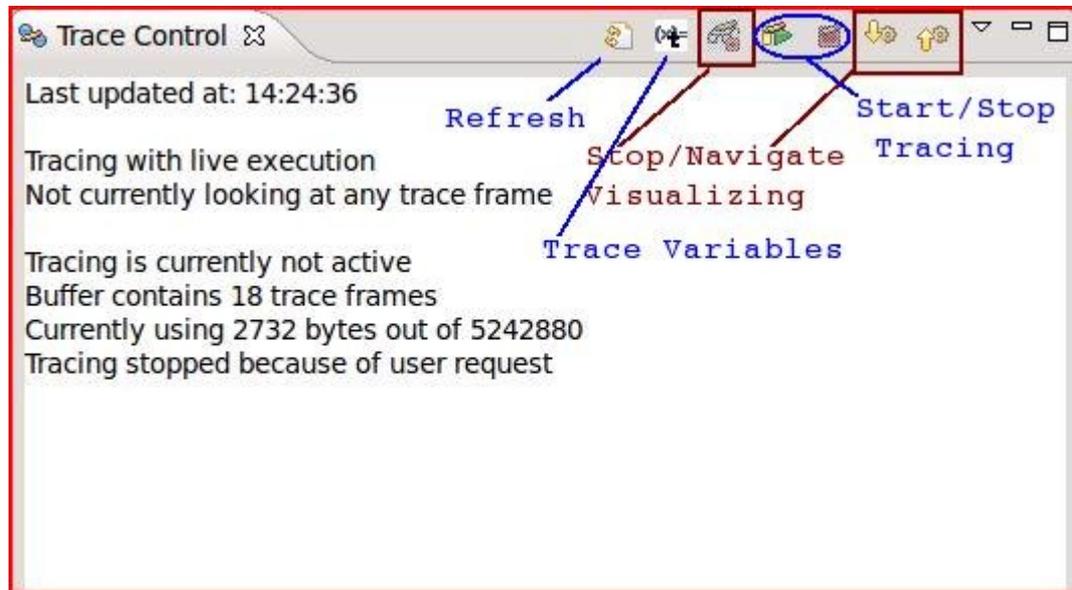


Eclipse Tracepoints Control

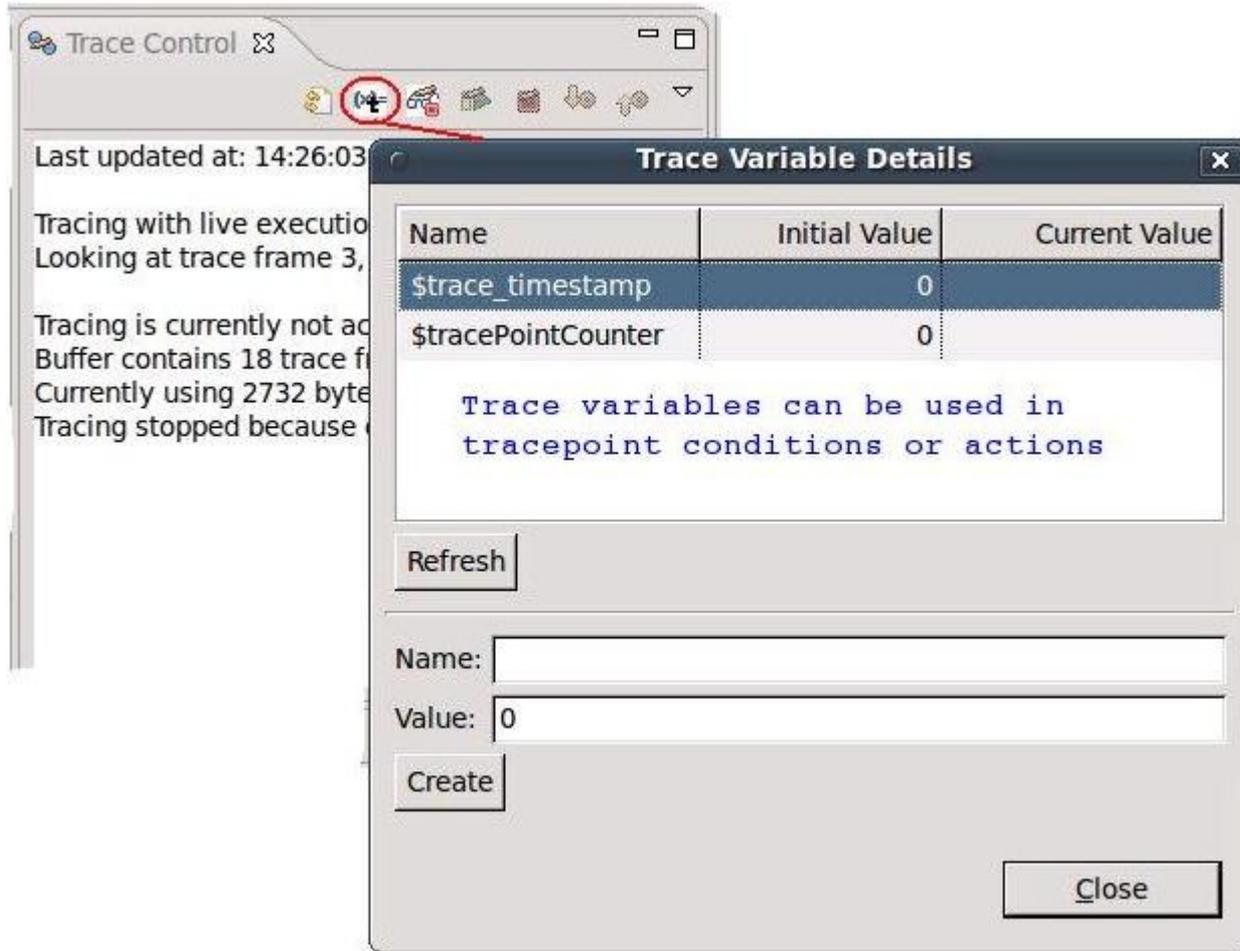


Eclipse Tracepoints Control

- › Trace Control View
 - Refreshing info
 - Trace Variables
 - Start/Stop Tracing
 - Navigate during Visualization
 - Stop Visualization

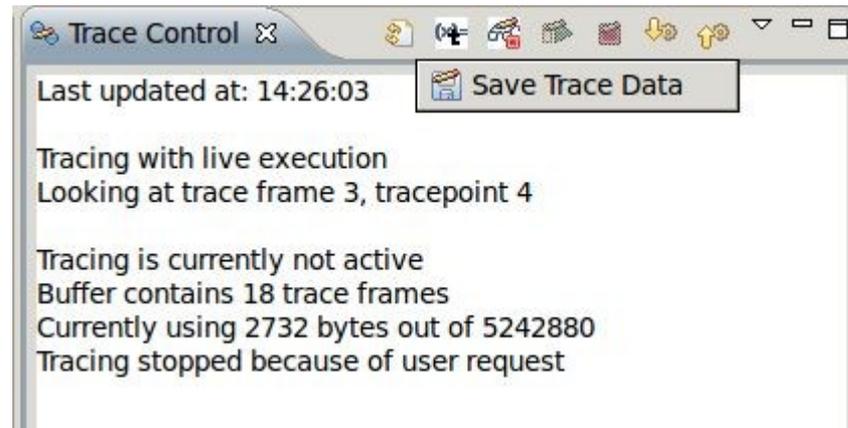


Eclipse Tracepoints Variables



Eclipse Trace Data

- › Resulting trace data
 - can be stored to file
 - can be visualized in Eclipse immediately or in the future



Eclipse Trace Data Visualization

- › Navigation through data records using GDB
- › Each data record is a snapshot of debug information
- › Records can be examined using standard debugger views
 - As if debugger was attached at a specific point in time
 - Only collected information can be shown
 - Highlighting of the tracepoint of interest
- › All collected data of a record can also be dumped as plain text

- › Trace data can be saved to file
- › Saved trace data can be examined offline

Eclipse Trace Data Visualization

The screenshot displays the Eclipse IDE interface during a debug session. The main window shows the source code of `Tracing.cpp` with a tracepoint set at line 29. The `Variables` window shows the state of the program, with the `total` variable highlighted in yellow and annotated as "Collected values shown". The `Breakpoints` window lists several tracepoints, with the one at line 29 selected and annotated as "Tracepoint for this trace is selected". The `Trace Control` window shows the current trace configuration, including the last update time (15:05:51) and the number of trace frames (20). The `Trace Control` window also has two annotations: "stop visualization" pointing to the stop button and "change trace" pointing to the change button. The `Trace Control` window also displays the text "Looking at trace frame 5, tracepoint 4".

Name	Type	Value
▶ this	operations * const	Collected values shown
↳ counter	int	
↳ total	int	10

```
20     storeY(y);
21     return first() * second();
22 }
23
24 int factorial(int y) {
25     storeY(y);
26
27     int total = 1;
28     for (int counter = first(); counter >1; counter++)
29         total = multiply(total, counter);
30     total++;
31 }
32
33 return total;
34 }
35
```

Trace Control window text:

Last updated at: 15:05:51
Tracing with live execution
Looking at trace frame 5, tracepoint 4
Tracing is currently not active
Buffer contains 20 trace frames
Currently using 2744 bytes out of 5242880
Tracing stopped because of user request

Eclipse Static Tracepoints

- › Next phase of development
- › Using GDB and UST
- › Handled like Dynamic Tracepoint, except for creation

Eclipse Static Tracepoints

- › Creation of tracepoint done by designer before compilation
- › As for Dynamic tracepoints:
 - Enable/Disable tracepoints dynamically
 - Dynamic condition
 - Can additionally have dynamic tracing specified (actions)
 - Pass count
 - Trace-state variables
 - ...

Planned Tracepoint Features

- › Support for Fast Tracepoints
 - Explicit or implicit support?
- › Support for Static Tracepoints
- › Support for Observer mode
- › Support for Global Actions (affecting all tracepoints)

Planned Tracepoint Features

- › Disabling tracepoints during Tracing
- › Tracepoints Enhanced Visualization:
 - Currently the user must have an idea of what has been collected
 - Goal is to directly and only show what has been collected
- › Fast Tracepoints on 3-byte instruction
 - Currently fast tracepoints are 5-byte jumps insert in the code
 - New 3-byte jump to a nearby location to the 5-byte jump

Getting it to work for you in five easy steps

1. Downloading Eclipse Linux Package:

- <http://eclipse.org/downloads>
- Choose: “Eclipse IDE for C/C++ Linux Developers”

2. Extract it: `tar xf <packageFile>`

3. Run it: `cd <packageDir> ; ./eclipse`

4. Create a (dummy) C/C++ project: “Hello World” is fine

5. Start debugging... GDB... GCC... etc...

Questions?



ERICSSON