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
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

<table>
<thead>
<tr>
<th>Now</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>‣ Non-Stop multi-threading</td>
<td>‣ Full Pretty-printing</td>
</tr>
<tr>
<td>‣ Partial Pretty-printing</td>
<td>‣ Full Multi-process</td>
</tr>
<tr>
<td>‣ Single space multi-process</td>
<td>‣ Multi-core debugging</td>
</tr>
<tr>
<td>‣ Reverse</td>
<td>‣ Global breakpoints</td>
</tr>
<tr>
<td>‣ Any binary debugging</td>
<td>‣ Tracepoints improvements</td>
</tr>
<tr>
<td>‣ Tracepoints</td>
<td>‣ Fast tracepoints</td>
</tr>
<tr>
<td></td>
<td>‣ Static tracepoints</td>
</tr>
<tr>
<td></td>
<td>‣ Observer-mode</td>
</tr>
<tr>
<td></td>
<td>‣ Intelligent trace visualization</td>
</tr>
<tr>
<td>Features</td>
<td>Now</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Non-Stop multi-threading</td>
</tr>
<tr>
<td></td>
<td>Partial Pretty-printing</td>
</tr>
<tr>
<td></td>
<td>Single space multi-process</td>
</tr>
<tr>
<td></td>
<td>Reverse</td>
</tr>
<tr>
<td></td>
<td>Any binary debugging</td>
</tr>
<tr>
<td></td>
<td>Tracepoints</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Now</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Stop multi-threading</td>
<td><strong>Full Pretty-printing</strong></td>
</tr>
<tr>
<td><strong>Partial Pretty-printing</strong></td>
<td>Full Multi-process</td>
</tr>
<tr>
<td>Single space multi-process</td>
<td>Multi-core debugging</td>
</tr>
<tr>
<td>Reverse</td>
<td>Global breakpoints</td>
</tr>
<tr>
<td>Any binary debugging</td>
<td>Tracepoints improvements</td>
</tr>
<tr>
<td>Tracepoints</td>
<td>- Fast tracepoints</td>
</tr>
<tr>
<td></td>
<td>- Static tracepoints</td>
</tr>
<tr>
<td></td>
<td>- Observer-mode</td>
</tr>
<tr>
<td></td>
<td>- Intelligent trace visualization</td>
</tr>
</tbody>
</table>
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)

- No pretty-printing
- Partial pretty-printing
Pretty-printing (Next)

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

Full pretty-printing with editable values
## Features

<table>
<thead>
<tr>
<th>Now</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Stop multi-threading</td>
<td>Full Pretty-printing</td>
</tr>
<tr>
<td>Partial Pretty-printing</td>
<td>Full Multi-process</td>
</tr>
<tr>
<td><strong>Single space multi-process</strong></td>
<td>Multi-core debugging</td>
</tr>
<tr>
<td>Reverse</td>
<td>Global breakpoints</td>
</tr>
<tr>
<td>Any binary debugging</td>
<td>Tracepoints improvements</td>
</tr>
<tr>
<td>Tracepoints</td>
<td>- Fast tracepoints</td>
</tr>
<tr>
<td></td>
<td>- Static tracepoints</td>
</tr>
<tr>
<td></td>
<td>- Observer-mode</td>
</tr>
<tr>
<td></td>
<td>- Intelligent trace visualization</td>
</tr>
</tbody>
</table>
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

Multiple processes in the same Launch in Non-Stop mode
<table>
<thead>
<tr>
<th>Features</th>
<th>Now</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Stop multi-threading</td>
<td></td>
<td>Full Pretty-printing</td>
</tr>
<tr>
<td>Partial Pretty-printing</td>
<td></td>
<td>Full Multi-process</td>
</tr>
<tr>
<td>Single space multi-process</td>
<td></td>
<td>Multi-core debugging</td>
</tr>
<tr>
<td>Reverse</td>
<td></td>
<td>Global breakpoints</td>
</tr>
<tr>
<td>Any binary debugging</td>
<td></td>
<td>Tracepoints improvements</td>
</tr>
<tr>
<td>Tracepoints</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fast tracepoints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Static tracepoints</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Observer-mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intelligent trace visualization</td>
</tr>
</tbody>
</table>
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

Buttons to control reverse execution

Toggle reverse and display execution buttons
# Features

<table>
<thead>
<tr>
<th>Now</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Stop multi-threading</td>
<td>Full Pretty-printing</td>
</tr>
<tr>
<td>Partial Pretty-printing</td>
<td>Full Multi-process</td>
</tr>
<tr>
<td>Single space multi-process</td>
<td><strong>Multi-core debugging</strong></td>
</tr>
<tr>
<td>Reverse</td>
<td>Global breakpoints</td>
</tr>
<tr>
<td>Any binary debugging</td>
<td>Tracepoints improvements</td>
</tr>
<tr>
<td>Tracepoints</td>
<td>- Fast tracepoints</td>
</tr>
<tr>
<td></td>
<td>- Static tracepoints</td>
</tr>
<tr>
<td></td>
<td>- Observer-mode</td>
</tr>
<tr>
<td></td>
<td>- Intelligent trace visualization</td>
</tr>
</tbody>
</table>
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

Cores are shown for both threads and processes
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

<table>
<thead>
<tr>
<th>Now</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>› Non-Stop multi-threading</td>
<td>› Full Pretty-printing</td>
</tr>
<tr>
<td>› Partial Pretty-printing</td>
<td>› Full Multi-process</td>
</tr>
<tr>
<td>› Single space multi-process</td>
<td>› Multi-core debugging</td>
</tr>
<tr>
<td>› Reverse</td>
<td>› Global breakpoints</td>
</tr>
<tr>
<td>› Any binary debugging</td>
<td>› <strong>Tracepoints improvements</strong></td>
</tr>
<tr>
<td>› <strong>Tracepoints</strong></td>
<td>- Fast tracepoints</td>
</tr>
<tr>
<td></td>
<td>- Static tracepoints</td>
</tr>
<tr>
<td></td>
<td>- Observer-mode</td>
</tr>
<tr>
<td></td>
<td>- Intelligent trace visualization</td>
</tr>
</tbody>
</table>
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
Eclipse Tracepoints Disassembly

› Disassembly view support for Tracepoints
› Tracepoint with condition
Eclipse Tracepoints Properties

- Tracepoints properties
  - Location
  - Enablement
  - Condition
  - Pass count
Eclipse Tracepoints Actions

Properties for

Actions for this tracepoint:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect total</td>
<td>Collect Action</td>
<td>collect total</td>
</tr>
</tbody>
</table>

Available actions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect total</td>
<td>Collect Action</td>
<td>collect total</td>
</tr>
<tr>
<td>collect counter</td>
<td>Collect Action</td>
<td>collect counter,$reg</td>
</tr>
<tr>
<td>Untitled Evaluate</td>
<td>Evaluate Action</td>
<td>eval $count=$count+1</td>
</tr>
</tbody>
</table>

Actions:

- Attach
- New...
- Edit...
- Delete

Restore Defaults  Apply
Cancel    OK
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

Trace variables can be used in tracepoint conditions or actions.
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

![Diagram of Eclipse Trace Data Visualization](image)

- **Tracepoint for this trace is selected**
- **Collected values shown**
- **Stop visualization**
- **Change trace**
- **Line where trace was collected is shown**

© Ericsson | GCC Summit 2010
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](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?