Supporting Parallel Component Debugging in Embedded Systems Using GDB Python Interfaces.

Kevin Pouget, Miguel Santana, Vania Marangozova-Martin and Jean-François Mehaut
Context

Embedded System Development

- High-resolution multimedia app. ⇒ high performance expectations.
  - H.265 HEVC
  - augmented reality,
  - ...

- Sharp time-to-market constraints

⇒ Important demand for
  - powerful parallel architectures
    - MultiProcessor on Chip (MPSoC)
  - convenient programming methodologies
    - Component-Based Software Engineering
  - efficient verification and validation tools
    - Our problematic
# Context

## MultiProcessor on Chip (MPSoC)

- **Parallel architecture**
  - more difficult to program
- **Maybe heterogeneous**
  - hardware accelerators,
  - GPU-like accelerators (OS-less)
- **Embedded system**
  - constrained environment,
  - on-board debugging complicated
    - performance debugging only
  - limited-scale functional debugging on simulators
Context

Component-Based Software Engineering

- Focus on design of independent building blocks
- Applications built with interconnected components
- Allows the adaptation of the application architecture according to runtime constraints
- Runnable components able to exploit MPSoC parallelism
Agenda

1. Component Debugging Challenges
2. Component-Aware Interactive Debugging
3. Feature Details
4. Python Implementation
5. Conclusion
Agenda

1. Component Debugging Challenges
2. Component-Aware Interactive Debugging
3. Feature Details
4. Python Implementation
5. Conclusion
Component Debugging Challenges

Component-based applications are dynamic

- various set of components deployed during the execution
- components are dynamically inter-connected
Component Debugging Challenges

Component-based applications are **dynamic**

- various set of components deployed during the execution
- components are dynamically inter-connected
Component Debugging Challenges

Component-based applications are **dynamic**

- various set of components deployed during the execution
- components are dynamically inter-connected
Component Debugging Challenges

Components interact with one another

- their execution is driven by interface communications
- complex framework-dependent steps between an interface call and its execution
Component Debugging Challenges

Components interact with one another

- their execution is driven by interface communications
- complex framework-dependent steps between an interface call and its execution
Component Debugging Challenges

Components interact with one another

- their execution is driven by interface communications
- complex framework-dependent steps between an interface call and its execution
Component Debugging Challenges

Components interact with one another

- their execution is driven by interface communications
- complex framework-dependent steps between an interface call and its execution
Component Debugging Challenges

Components interact with one another

- their execution is driven by interface communications
- complex framework-dependent steps between an interface call and its execution
Component Debugging Challenges

Components interact with one another

- their execution is driven by interface communications
- complex framework-dependent steps between an interface call and its execution
Component Debugging Challenges

Components interact with one another

- their execution is driven by interface communications
- complex framework-dependent steps between an interface call and its execution
Component Debugging Challenges

Components interact with one another

- their execution is driven by interface communications
- complex framework-dependent steps between an interface call and its execution
Component Debugging Challenges

Information *flows* over the components

- A corrupted data may be carried over various component before triggering a visible error
Component Debugging Challenges

Information flows over the components

- a corrupted data may be carried over various component before triggering a visible error
Component Debugging Challenges

Information **flows** over the components

- a corrupted data may be carried over various component before triggering a visible error
Component Debugging Challenges

Information flows over the components

- a corrupted data may be carried over various component before triggering a visible error
Component Debugging Challenges

Information *flows* over the components

- A corrupted data may be carried over various component before triggering a visible error.
Component Debugging Challenges

Information flows over the components

- a corrupted data may be carried over various component before triggering a visible error
Agenda

1. Component Debugging Challenges
2. Component-Aware Interactive Debugging
3. Feature Details
4. Python Implementation
5. Conclusion
Component-Aware Interactive Debugging

Objective: Bring the debugger closer to the component model
Component-Aware Interactive Debugging

**Objective:** Bring the debugger closer to the component model

- Show application architecture evolutions
  - component deployment
  - interface binding
- ...
Component-Aware Interactive Debugging

Objective: Bring the debugger closer to the component model

- Show application architecture evolutions
  - component deployment
  - interface binding
  - ...

- Follow the execution flow(s) over the component graph
  - runnable component execution,
  - execution triggered by an interface call
  - ...

Component-Aware Interactive Debugging

**Objective:** Bring the debugger closer to the component model

- Show application architecture evolutions
  - component deployment
  - interface binding
  - ...

- Follow the execution flow(s) over the component graph
  - runnable component execution,
  - execution triggered by an interface call
  - ...

- Track inter-component data exchanges
  - message route history,
  - message- or interface-based breakpoints
  - ...
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework

Slide 12/29 — kevin.pouget@st.com — Supporting Parallel Component Debugging. — GNU Tools Cauldron 2012, July 9th-11th
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework
Component-Aware Interactive Debugging

Implementation

⇒ Detect and interpret key events in the component framework

Stopped on <Component 1 execution>

Breakpoint hit on <component execution>

Breakpoint hit at @ 0xdeb42

Interface binding  Execution context  Interface  Component
Agenda

1. Component Debugging Challenges
2. Component-Aware Interactive Debugging
3. Feature Details
4. Python Implementation
5. Conclusion
Feature Details
Proof-of-concept environment

Platform 2012
ST MPSoC research platform
- Heterogeneous
- 4x16 CPU OS-less comp. fabric
Feature Details

Proof-of-concept environment

Native Programming Model
- P2012 component framework
- Provides communication components and interface

Platform 2012
ST MPSoC research platform
- Heterogeneous
- 4x16 CPU OS-less comp. fabric
Feature Details

Proof-of-concept environment

**The Gnu Debugger**
- Adapted to low level debugging
- Large user community

**Native Programming Model**
- P2012 component framework
- Provides communication components and interface

**Platform 2012**
ST MPSoC research platform
- Heterogeneous
- 4x16 CPU OS-less comp. fabric
Feature Details

Proof-of-concept environment

The Gnu Debugger
- Adapted to low level debugging
- Large user community

Native Programming Model
- P2012 component framework
- Provides communication components and interface

Platform 2012
ST MPSoC research platform
- Heterogeneous
- 4x16 CPU OS-less comp. fabric
Feature Details

Case study: Debugging a Pyramidal Feature Tracker

- part of an augmented reality application
- analyzes video frames to track interesting features motion
Case study: Debugging a Pyramidal Feature Tracker

List components and their interfaces

```
(gdb) info component +connections
#1 Host[31272]
  DMAPush/0x... <DMA> srcPullBuffer Component... #2
  DMAPull/0x... <DMA> dstPushBuffer Component... #2
* #2 Component[SmoothAndSampleProcessor.so]
  srcPullBuffer <DMA> DMAPush/0x... Host[31272]
  dstPullBuffer <DMA> DMAPull/0x... Host[31272]
```
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

Message 1:
Component A # Message created
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

Message 1:
Component A # Message created
Component A::Interface A.1 # Message sent
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

Message 1:
Component A # Message created
Component A::Interface A.1 # Message sent
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

Message 1:
Component A # Message created
Component A::Interface A.1 # Message sent
Component B::Interface B.1 # Message received
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

Message 1:
Component A # Message created
Component A::Interface A.1 # Message sent
Component B::Interface B.1 # Message received

Message 2:
Component B # Message created
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

Message 1:
Component A # Message created
Component A::Interface A.1 # Message sent
Component B::Interface B.1 # Message received

Message 2:
Component B # Message created
Component B::Interface B.2 # Message sent
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

Message 1:
Component A # Message created
Component A::Interface A.1 # Message sent
Component B::Interface B.1 # Message received

Message 2:
Component B # Message created
Component B::Interface B.2 # Message sent
Component C::Interface C.1 # Message received
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

Message 1:
Component A # Message created
Component A::Interface A.1 # Message sent
Component B::Interface B.1 # Message received

Message 2:
Component B # Message created
Component B::Interface B.2 # Message sent
Component C::Interface C.1 # Message received
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

- messages can be logically aggregated with user-defined routing tables:

Message 1:
Component A # Message created
Component A::Interface A.1 # Message sent
Component B::Interface B.1 # Message received

Message 2:
Component B # Message created
Component B::Interface B.2 # Message sent
Component C::Interface C.1 # Message received
Case study: Debugging a Pyramidal Feature Tracker

Information about messages

- messages can be logically aggregated with user-defined routing tables:

**Message 1:**
Component A # Message created
Component A::Interface A.1 # Message sent
Component B::Interface B.1 # Message received
Component B::Interface B.2 # Message sent
Component C::Interface C.1 # Message received
Case study: Debugging a Pyramidal Feature Tracker

Information about interface activity

(host) info components +counts
#2 CommComponent[SmoothAndSampleProcessor.so]
  srcPullBuffer #35 msgs
dstTmpPushBuffer #36 msgs
srcTmpPullBuffer #35 msgs
dstPushBuffer #34 msgs
Case study: Debugging a Pyramidal Feature Tracker

Information about interface activity

(gdb) info components +counts
#2 CommComponent[SmoothAndSampleProcessor.so]
  srcPullBuffer #35 msgs
dstTmpPushBuffer #36 msgs
srcTmpPullBuffer #35 msgs
dstPushBuffer #34 msgs

- allowed us to find a bug in the application
  (msg sent to the wrong interface)
Case study: Debugging a Pyramidal Feature Tracker

Information about interface activity

Excerpt from a 300 lines-of-code file

```c
/* Compute last lines if necessary */
if (tmp_size > 0) {
    ...
    /* Transmit the last lines computed */
    CALL(srcTmpPullBuffer, release)(...);
    CALL(dstTmpPushBuffer, push)(...);
}
```
Agenda

1. Component Debugging Challenges
2. Component-Aware Interactive Debugging
3. Feature Details
4. Python Implementation
5. Conclusion
Python Implementation

Detect and Interpret Key Events in the Component Framework
Python Implementation

Detect and Interpret Key Events in the Component Framework

Detect

- Internal breakpoints
  - no apparent execution stop
  - no screen notification

→ Python notification for framework events
Python Implementation

Detect and Interpret Key Events in the Component Framework

Detect

- Internal breakpoints
  - no apparent execution stop
  - no screen notification

→ Python notification for framework events

Key Events

- New components, new binding
- Component execution trigger
- Message created, sent, received, ...
Python Implementation

**Detect and Interpret Key Events in the Component Framework**

**Detect**
- Internal breakpoints
  - no apparent execution stop
  - no screen notification
  → Python notification for framework events

**Key Events**
- New components, new binding
- Component execution trigger
- Message created, sent, received, . . .

**Interpret**
- Debug information (DWARF)
- API + Calling conventions
  → (almost\(^1\)) everything we need

\(^1\) some implementation-dependent bits still remain ...
Python Implementation

Debug Toolbox

**Function breakpoints**

Internal breakpoints triggered at the execution of a function

⇒ catch input, updated and output parameters

- `stop, do_after, data = prepare_before(self)`
- `stop = prepare_after(self, data)`

```python
NPM_instantiateComponent(&cmp1_handle, type1, nb_procs);
NPM_instantiateComponent(&cmp2_handle, type2, nb_procs);
NPM_instantiateFIFOBuffer(&fifo_handle, cmp1_handle, "src_itf", cmp2_handle, "dst_itf");
```
Python Implementation

Debug Toolbox

Function breakpoints

Internal breakpoints triggered at the execution of a function
⇒ catch input, updated and output parameters
• stop, do_after, data = prepare_before(self)
• stop = prepare_after(self, data)
  • gdb.execute("finish")
    "Thou shalt not alter the execution state of the inferior"
    (gdbdoc 23,2,2,20)
⇒ gdb.FinishBreakpoint instead

NPM_instantiateComponent(&cmp1_handle, type1, nb_procs);
NPM_instantiateComponent(&cmp2_handle, type2, nb_procs);
NPM_instantiateFIFOBuffer(&fifo_handle, cmp1_handle, "src_itf", cmp2_handle, "dst_itf");
Python Implementation

Debug Toolbox

**Function breakpoints**

Internal breakpoints triggered at the execution of a function

⇒ catch input, updated and output parameters

- stop, do_after, data = prepare_before(self)
- stop = prepare_after(self, data)
  - gdb.execute("finish")

"Thou shalt not alter the execution state of the inferior"

(gdbdoc 23,2,2,20)

→ gdb.FinishBreakpoint instead

```c
NPM_instantiateComponent(&cmp1_handle, type1, nb_procs);
NPM_instantiateComponent(&cmp2_handle, type2, nb_procs);

NPM_instantiateFIFOBuffer(&fifo_handle,
    cmp1_handle, "src_itf",
    cmp2_handle, "dst_itf");
```
Python Implementation

Debug Toolbox

User-level Multithreading

- threading implemented with longjmp/setjmp
  → invisible to GDB
REGISTERS = ($esp$, "$ebp", "$eip")

def save_current_thread():
    return [gdb.parse_and_eval(reg) for reg in REGISTERS]
REGISTERS = ("$esp", "$ebp", "$eip")

```python
def save_current_thread():
    return [gdb.parse_and_eval(reg) for reg in REGISTERS]

def switch_inactive_thread(next_):
    jmbuf = next_['context'][0]['__jmpbuf']
    gdb.execute("set $esp=%s" % jmbuf[JB_SP])
    gdb.execute("set $ebp=%s" % jmbuf[JB_BP])
    gdb.execute("set $eip=__longjmp")
    gdb.execute("flushregs")
```
REGISTERS = ("$$esp", "$$ebp", "$$eip")

def save_current_thread():
    return [gdb.parse_and_eval(reg) for reg in REGISTERS]

def switch_inactive_thread(next_):
    jmbuf = next_['context'][0]['__jmpbuf']
    gdb.execute("set $$esp=%s" % jmbuf[JB_SP])
    gdb.execute("set $$ebp=%s" % jmbuf[JB_BP])
    gdb.execute("set $$eip=__longjmp")
    gdb.execute("flushregs")

def reload_current_thread(stop_regs):
    for reg_name, reg_val in map(REGISTERS, stop_regs):
        gdb.execute("set %s=%s" % (reg_name, str(reg_val)))
Python Implementation

Debug Toolbox

**User-level Multithreading**

```plaintext
(gdb) info processors
#1 Processor DMA 1 // user-level threads
#2 Processor 1 Cluster 1 // <-> simulated processors
* #3 Processor 2 Cluster 1
  #4 Processor 1 Cluster 2
...

(gdb) info components
#1 Host // component not scheduled
* #2 Component A1 // current component
  #3 Component A2
~ #4 Component B1 // component not schedulable
~ #5 Component B2 // <-> no execution context
```
Python Implementation
Debug Toolbox

User-level Multithreading

(gdb) component 3
[Switching to sleeping Component A2 #3]
(gdb) where
#0 0x47bb07a0 in __longjmp () from /usr/lib/libc.so.6
#1 0xf7fe3f20 in contextSwitch (old, new)
#2 0xf7fe406d in schedule_next_execution_context ()
#3 0xe7eb7838 in schedNext ()
...
#9 0xdd55e23d in outputBuffer_fetchNextBuffer (...)
#10 0xdd5d26c8 in rtmMaster (...)
#11 0xdd5d307d in thread_main (...)
...
Python Implementation

Debug Toolbox

User-level Multithreading

- far from being perfect
- no coordination with GDB thread capabilities
- user-level thread debugging is possible with Python
- a Thread_db library (e.g., *User-Level Thread_db*) could make it more standard and reliable

ULDB: a debugging API for user-level thread libraries, K. Pouget et al, MTAAP 10
Python Implementation

Entity Tracking

On framework function breakpoint:

1. identify operation and parameters
   - which function?
     gdb.Breakpoint.location
   - API for parameters
   - cmp_py = lookup_table[handle]
Python Implementation

Entity Tracking

On framework function breakpoint:

1. Identify operation and parameters
   - Which function?
     - `gdb.Breakpoint.location`
   - API for parameters
   - `cmp_py = lookup_table[handle]`

2. Identify active component
   - Based on current thread/processor
Python Implementation

Entity Tracking

On framework function breakpoint:

1. Identify operation and parameters
   - Which function?
     \[ \text{gdb.Breakpoint.location} \]
   - API for parameters
   - \[ \text{cmp\_py = lookup\_table[handle]} \]

2. Identify active component
   - Based on current thread/processor

3. Update internal state accordingly, e.g.,
   - Create a component/link object
   - Move a message btw components
   - …
Python Implementation

Entity Tracking

On framework function breakpoint:

1. Identify operation and parameters
   - Which function?
     - \texttt{gdb.Breakpoint.location}
   - API for parameters
     - \texttt{cmp\_py = lookup\_table[handle]}

2. Identify active component
   - Based on current thread/processor

3. Update internal state accordingly, e.g.,
   - Create a component/link object
   - Move a message btw components
   - ...

4. Check user breakpoints/catchpoint
Agenda

1. Component Debugging Challenges
2. Component-Aware Interactive Debugging
3. Feature Details
4. Python Implementation
5. Conclusion
Conclusion

- Debugging dynamic component application is challenging
- Lack of high level information about components framework

Our work: bring debuggers closer to the component model
  - better understanding application behavior
  - keep focused on bug tracking
Conclusion

- Debugging **dynamic** component application is challenging
- Lack of **high level information** about components framework

**Our work:** bring debuggers closer to the component model
  - better understanding application behavior
  - keep focused on bug tracking

**Proof-of-concept:** GDB and its Python interface
  - interface good enough to build real improvements in Python
  - a few missing bits contributed to the project
    - `gdb.FinishBreakpoint`
    - multiple breakpoint hits
    - `gdb.selected_inferior()`
Conclusion

- Debugging **dynamic** component application is challenging
- Lack of **high level information** about components framework

- **Our work**: bring debuggers closer to the component model
  - better understanding application behavior
  - keep focused on bug tracking

- **Proof-of-concept**: GDB and its Python interface
  - interface good enough to build real improvements in Python
  - a few missing bits contributed to the project
    - `gdb.FinishBreakpoint`
    - multiple breakpoint hits
    - `gdb.selected_inferior()`

- Going further programming-model aware debugging
  - OpenCL
  - Dataflow execution model
  - ...