"Deterministic replay support for Genode components"

Tokponnon, Mahoukpego Parfait ; Lobelle, Marc

Abstract

we are modifying Nova kernel to make it support temporal redundancy for user processes running Genode components. Every time a process releases the CPU, we manage to restart it from its last starting point and re-run the same portion of the process code. An practical application of this could be an implementation of a fault tolerant system. At the end of the two executions, one may compare their memory footprints to detect an eventual fault due for example to SEU. We are also interested in the performance penalty this would induce on the overall systems compared to existing redundancy models. In this talk, we will present how Genode components performances are impacted by this type of redundancy. Possible choices and remaining challenges for extending this model for multiprocessor environment will also be discussed.

Document type: Communication à un colloque (Conference Paper)

Référence bibliographique

Tokponnon, Mahoukpego Parfait ; Lobelle, Marc. Deterministic replay support for Genode components. FOSDE Micro-Kernel Dev Room (Université Libre de Bruxelles, 04/02/2017).

Available at: http://hdl.handle.net/2078.1/184187

[Downloaded 2019/04/29 at 02:10:19]
Dual Execution and Comparison For Genode Components Performance Penalty And Challenges

FOSDEM Micro-kernel Devroom, 04/02/17

Parfait Tokponnon
mahoukpego.tokponnon@uclouvain.be

Marc Lobelle
marc.lobelle@uclouvain.be
2 Outline

- Introduction to DWC
- Systematic process element replay
- Possible Usages and advantages compared to other fault tolerant techniques
- Genode deterministic Replay
  - Current state
  - Performance Impact
- Remaining works
Outline

- Introduction to DWC
- Systematic process element replay
- Possible Usages and advantages compared to other fault tolerant techniques
- Genode deterministic Replay
  - Current state
  - Performance Impact
- Remaining works
Execution replay
Introduction to DWC fault Tolerance

• DWC = Double execution With Comparison

• purpose: Detect transient errors and take actions to recover

• Double execution can happen
  • In parallel (simultaneously or with one execution slightly delayed) or in sequence
  • At instruction level or at set of instructions level

• To be effective, execution replay must be deterministic
  • Run the same code with the same initial data and environment

• Field of application: fault tolerant system, debugging, software verification, hardware testing …
Examples

- Primary-backup hypervisor based fault tolerance system (1)
- Virtual machine based security system: Revirt (2)
- Hardware assisted deterministic Replay: Capo (3)


6 Outline

• Introduction to Deterministic Replay (Dual Execution Replay)
• Systematic process element replay
• Possible Usages and advantages compared to other fault tolerant techniques
• Genode deterministic Replay
  • Current state
  • Performance Impact
• Remaining works
Our model: Systematic processing element replay

• Here, the execution replay is
  • applied to a set of instructions
  • is limited in time (< hundreds of µs), short enough so that it may not experience more than one error.

• The kernel is modified so that it systematically:
  • Divides any process in short “processing elements” (PE),
  • runs them twice and
  • compares the “result”:
    • OK: commit the result and start the next PE,
    • KO: restart the current PE
    • Unexpected exception during one of the executions: restart the current PE
Deterministic PE

- PE execution is **atomic and idempotent**: No interaction with the outside world.
- PE is delimited by IO, time dependent instructions (RDTSC), system calls, or any exception (page fault, protection fault, …) raised by the user process.
- Main goal:
  - Detect transient fault and correction techniques
OT Processing

• The “result” is composed of:
  • All modified memory pages \( (P_1, P_2, \ldots, P_m) \) and
  • User process related registers - UPRR (General Purpose Registers, RIP, SP, …)

• \( n \text{th} \) Processing Element is called \( e_n \)

• \( e_{n,i} \) (\( i \in \{1,2\} \)) is the \( i \text{th} \) execution of \( e_n \)

• \( P_{m,i} \) is the modified \( P_m \) during the \( i \text{th} \) execution of \( e_n \)

• \( P_{m,0} \) is the unmodified version of \( P_m \) before the first execution of \( e_n \)
OT Processing

- Before the $e_{n,1}$, save all UPRR $\rightarrow R_0$ and process memory to $PM_0$ (Pages 1, 2, ..., m)
  - Set process memory to Read-Only to keep trace of altered pages: will cause page faults

- During $e_{n,1}$, $PM_1$ (collection of all altered pages) is progressively constructed
  - At every page fault, the concerned page is replaced by a new page with same content and RW right and added to $PM_1$ ($P_{1,0} \rightarrow P_{1,1}, P_{2,0} \rightarrow P_{2,1}, ..., P_{m,0} \rightarrow P_{m,1}$): Copy $P_{j,0}$ to $P_{j,1}$
OT Processing

- At the end of $e_{n,1}$, and before starting $e_{n,2}$
  1. We replace all altered pages by new ones, but with RW right: $PM_{2}$ ($P_{1,0} \rightarrow P_{1,2}$, $P_{2,0} \rightarrow P_{2,2}$, ..., $P_{m,0} \rightarrow P_{m,2}$): **Copy $P_{j,0}$ to $P_{j,2}$ (No page fault is expected)**
  2. Save all UPRG $\rightarrow R_{1}$
  3. Flush the caches

- At the end of $e_{n,2}$, compare one by one all Pages $P \in PM$ ($P_{1,1}$ and $P_{1,2}$, $P_{2,1}$ and $P_{2,2}$, ..., $P_{m,1}$ with $P_{m,2}$) and all registers in UPRR
  - If comparison **OK**: Set $PM_{0}$ to $PM_{1}$ (or $PM_{2}$) and proceed to next OT
  - If comparison **KO**: restart the current OT
Implications

• This involves to:
  • Copy 3 times, word by word up to 10 memory frames, 4 kB each,
  • Compare, word by word, up to 10 memory frames, 4 kB each.
    • The working sets vary usually from 0 to 10 frames, according to our tests
  • Flush the caches

• And all of these
  • In no more than certain time limit (200 µs for example) while
  • Fulfilling real time constraints of some applications.
• Introduction to Deterministic Replay (Dual Execution Replay)
• Systematic process element replay
• State of the concept
• Genode deterministic Replay
  • Current state
  • Performance Impact
• Remaining works
State of the concept

• Systematic processing element replay has already been applied to process running on bare metal (without OS) as fault tolerance technique against Single Event Upset in small embedded system\(^{(1)}\)

• On-going work by E.Assogba, to port to Operating System level

• We are trying to port it virtual machine support level as proof of concept to enable the use of any unmodified OS.

Limiting process execution time

- The process releases the CPU (traps or faults) before granted time limit is reached
  - Just restart the PE from its starting point
  - $e_{n,2}$ must normally be exactly the same as $e_{n,1}$
- The process exhausts its granted time
  - A timer interrupt is issued at time limit during $e_{n,1}$: N instructions have been executed then
  - $e_{n,2}$ runs with Performance monitoring interrupt armed on instruction counter overflow.
  - Make sure the same number of instructions is executed.
- Proceed to comparison phase.
- I/O instruction, MMIO and time dependent Instruction (eg. rdtsc) stop the PE
Outline

- Introduction to Deterministic Replay  (Dual Execution Replay)
- Systematic process element replay
- Possible Usages and advantages compared to other fault tolerant techniques
- Genode deterministic Replay
  - Current state
  - Performance Impact
- Remaining works
• When applying Systematic processing element replay to Genode framework, we are interested in the following concerns:

1. Will an OS, in a virtual machine, be run in this fashion while satisfying to its service constraints toward user processes?
2. What will be the overall overhead?
3. How long can we shorten the atomic execution (OT) time with a critical charge of work in the running virtual machine?
The implementation is not totally finished but some meaningful results are already available.

The second run is always shorter than the first (because no page fault is expected). This run may be considered as a normal Genode process execution.

**Fig1**: A correct OT execution with no cache flush

- $t_1$: first run
- $t_2$: second run
- $r$: time to restart - kernel
- $cc$: time to compare and commit
Results
OT execution (2/2)

Fig1: A correct OT execution with cache flush

$t_1$: first run
$t_2$: second run
$r_1$: first run treatment
$r$: time to restart - kernel
$cc$: time to compare and commit
$cf$: time to flush the caches
Outline

• Introduction to Deterministic Replay (Dual Execution Replay)
• Systematic process element replay
• Possible Usages and advantages compared to other fault tolerant techniques
• Genode deterministic Replay
  • Current state
  • Performance Impact
• Remaining works
Benchmark

• Benchmark execution not possible yet (virtual machine not supported yet)
  • Genode normal execution is approximated by the second run.
  • the overall performance penalty can be expressed by the ratio of the total execution time divided by the second run time.

\[
\tau = \frac{100 \times (t_1 + r1 + cf + t_2 + cc)}{t_2}
\]

• Current state only works for the Genode initialization phase.
• The system starting phase (initialization) is certainly the worse case since this time, processes are expected to make frequently a lot of system calls.
Performance penalty
When PE ends at system call or exception (1/2)

Overhead : 3400%
Total execution Time : 237 µs
Performance penalty
When PE ends at system call or exception (2/2)

Worse overhead distribution without cache flush

- First Run: 40%
- Restart Time: 28%
- Second Run: 19%
- verification & commit: 13%

Overhead: 527%
Total execution Time: 36 µs
Performance penalty
When PE stops after exhausting its granted time (1/2)

Worse overhead distribution

- First Run: 77%
- Restart Time: 9%
- Cache flushing: 4%
- Single Stepping: 7%
- Verification & commit: 1%

Overhead: 6221%
Total execution Time: 242 µs
Performance penalty
When PE stops after exhausting its granted time (2/2)

Worse overhead distribution without cache flush

- First Run: 32%
- Restart Time: 19%
- Second Run: 6%
- Single Stepping: 3%

Overhead: 263%
Total execution time: 56 µs
Overall performance overhead during the booting

- Normal Genode demo scenario Boot ≈ 14s (Lenovo x230, core i5, 8 GB)
- With cache flushing
  - Dual execution Mode : 7 min 40s
  - Performance penalty : 3285% with cache flushing
- Without cache flushing
  - Dual execution Mode : 16s
  - Performance penalty : 114%
Current issues

- Instructions counting
  - Until now we have not dealt yet successfully with all the peculiarities of the Intel instruction counter feature (compared to AMD).
  - Sometime, for the same Processing element, the number of instructions executed during the first and the second are not the same.

- Page fault appears randomly when the system is fully started (after the initialization phase) independently from the instruction counting problem
Outline

• Introduction to Deterministic Replay (Dual Execution Replay)
• Systematic process element replay
• Possible Usages and advantages compared to other fault tolerant techniques
• Genode deterministic Replay
  • Current state
  • Performance Impact
• Remaining works
Future work

• Understand the cause of page fault and correct the problem
• Optimize cache flush operation
• Make full virtual machine support
  • Run the Heeselicht scenario (Linux running In Genode running on DWC featured Nova Kernel)
  • Compile GCC and Linux kernel in the Linux virtual machine
  • Run some benchmarks in Linux Virtual machine

Thank You