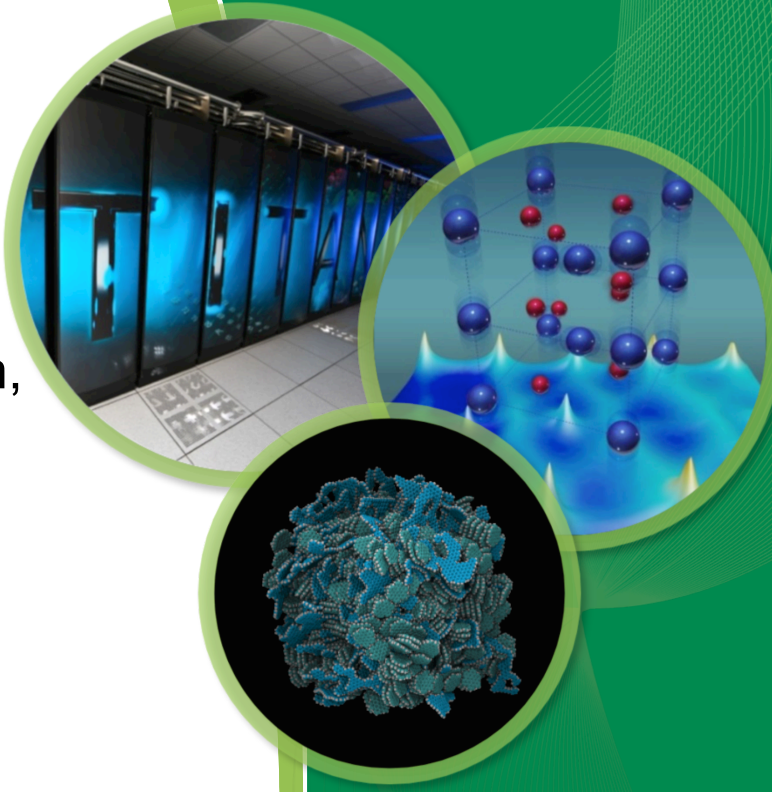
A Cooperative Approach to Virtual Machine Based Fault Injection

Thomas Naughton, Christian Engelmann, Geoffroy Vallée, Ferrol Aderholdt and Stephen L. Scott[2]

[1] Oak Ridge National Laboratory, USA
[2] Tennessee Tech University, USA



9th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds & Grids

Grenoble, France, August 23, 2016

OAK RIDGE
National Laboratory

Exascale Challenge: Resilience

- Current & next generation HPC systems[1]
  - Hardware failures occur, rates are debated
  - Future rates likely to be higher due to
    - Increased (commodity) component counts, and
    - Shrinking feature size (e.g., smaller transistors & radiation induced errors)
  - Fears of silent data corruption (SDCs)

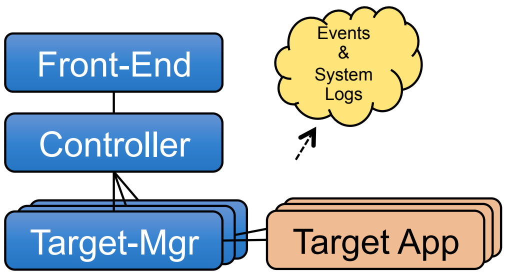
- Resilience Research in HPC
  - Work on fault/error models
  - Work on fault-tolerance mechanisms & resilient algorithms
  - Work on <u>tools for resilience investigations</u>

1. "Toward Exascale Resilience: 2014 update", *F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, M. Snir*, 2014, DOI: 10.14529/jsfi1401. http://superfri.org/superfri/article/view/14

OAK RIDGE
National Laboratory

Resilience Investigation Tools

- Fault injection (FI) tools to study effects of simulated errors
  - System Under Test (SUT) & FI Infrastructure (Controller, Injector/Mgr.)
  - Useful to keep SUT isolated from Controller
  - Useful to have SUT context for Controller/Injector



- Levels of FI
  - Software / Hardware / Environment
  - Software Implemented Fault Injection (SWIFI)

- Virtualization & FI Tools
  - Good: Virtual Machines provide strong isolation, useful for FI Tools
    - SUT in Guest (in VM), Controller on Host (outside VM)
  - Challenge: VM+FI tools face “semantic gap” SUT/Controller

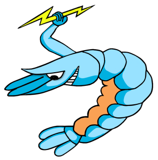
VMI: Virtual Machine Introspection

- Overview
  - Allows guest (VM) *internal state* to be exposed to *external viewer*
  - Example Viewers: Another VM, the VMM, or process on host
- Bridging the gap
  - Guest state is extensive (e.g., guest memory, device registers, etc.)
  - Guest state often difficult to understand due to “semantic gap”
  - Bridge this gap by incorporating memory map data
  - Example: Linux’s *System.map* file has kernel function/data addresses
- Uses
  - VMI techniques often used in computer security/forensics
  - Example: Intrusion detection & monitoring for VMs

VMI for our "friendly" case

- Coordinate VM+FI setup
  - VMI techniques to gain context
  - VMI harder if must assume **no** knowledge a priori
  - Simplify VMI techniques by using a "cooperative" approach
    - Share information about Guest/VM (assume some knowledge)
- Simplify VMI by sharing details of VM setup
  - Pre-configure tools with info on key kernel (guestOS) data structures
  - Add executable(s) to VM image
  - Provide symbol maps of guestOS & target application
- Example usage scenario
  - Run target application (inside VM)
  - Inject random errors into specific variable (from outside VM)

Virtual Machine Monitor

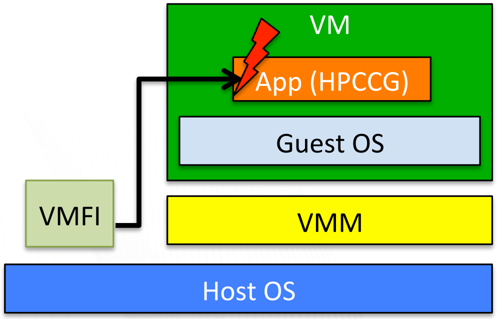


- Palacios VMM
  - Embeddable type-II hypervisor (Linux or Kitten)
  - Developed for use in HPC env. (e.g., Cray); also teaching tool/code
  - Developed at Northwestern, U of Pitt, Sandia Nat. Labs & U of NM

- Palacios enhancements – VM “kmem” patch
  - Extend Linux kernel module for Palacios on x86-64
  - Register VM memory device & create char devices for VM in host
    1. Used for VMI routines to walk process list
    2. Used to modify memory (data) of specific process
  - Routines for reading VM memory to support VMI methods

VM+FI Approach

- Guest level requirements
  - Launcher application (wrapper) to identify "target" process
    - Wrapper is parent for target
    - Manually take PID printed by wrapper for host VM+FI tool, or use host VM+FI tool to lookup instances of "wrapper" proc_name in GuestOS
- Host level requirements
  - Export Palacios VM's memory as char device in HostOS
  - Provide tool to walk kernel data structures of Guest OS from outside VM
  - Provide symbol maps
    - GuestOS (e.g., Linux system.map)
    - Wrapper App (e.g., wrapper.map)
    - Target App (e.g., HPCCG app-symbols.map)




```
guestVM:$ ./wrapper  ./HPCCG 100 200 100
hostOS:$ sudo ./kmem 198 wrapper.map  HPCCG.map rtrans 6 4 0
```

OAK RIDGE National Laboratory

HPCCG

- HPCCG – Conjugate Gradient Benchmark
  - Michael Heroux / Sandia mini-apps http://mantevo.org
  - Iterative algorithm supports serial & parallel (MPI, OpenMP) execution
  - HPCCG v1.0 - Self-contained C++ code

- Usage: `test_HPCCG nx ny nz`
  - Input parameters: 3Dimensions Size (x, y, z)
  - Configurations: Max-Iterations & Tolerance
    - Example: `max_iterations = 150, tolerance = 0.0` (*always run to max*)

- Relevance
  - Identified by Heroux & Dongarra as possible new Top500 metric[1]
  - Previous work[2] has shown iterative algorithms resilient to some errors

1. "Toward a New Metric for Ranking High Performance Computing Systems", J. Dongarra and M. A. Heroux, Sandia Report SAND2013-4744, June 2013. http://www.sandia.gov/~maherou/docs/HPCG-Benchmark.pdf
2. "Soft error vulnerability of iterative linear algebra methods", G. Bronevetsky and B. de Supinski, . Proc. of International Conference on Supercomputing (ICS), 2008, http://doi.acm.org/10.1145/ 1375527.1375552

Evaluation – Guest Application Errors

- HPCCG – Setup
  - Select matrix size to fit available memory size of VM and to keep wallclock times low to speed testing
  - Input params: nx = 100, ny = 200, nz = 100
  - Max_iterations = 150

- HPCCG – Changes for testing
  - Tolerance = 0.0000001
    - Change 'tolerance' to non-zero (0.000001) to allow less than max_iters
  - Move 'rtrans' to global symbol for HPCCG() function
  - Statically linked and run in serial mode

Evaluation – Guest Application Errors (2)

- Fault Injections – simulate course-grained data corruption
  - Inject errors into key variable in algorithm (‘rtrans’)
    - Identified by manual inspection
    - Made global for VM+FI utility to locate target address in guestOS
  - Run application 30 times with & without errors injected
  - Inject random value between 1..100 at 1 sec intervals into target
- Results
  - All runs complete without crash
  - Observe output for *Final Residual* (‘normr’) printed at the end
  - Non-error runs: HPCCG output was deterministic for all runs
  - Error runs: HPCCG output showed slight perturbations
    - Slight deviations in ‘normr’ result, <u>very</u> small but STD ≠ 0

Observations

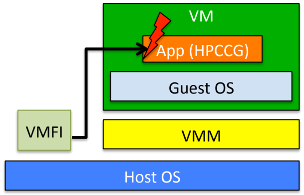
- Advantages of VM+FI Approach
  - Offers strong isolation between target/host environments
  - VM provides configurable environment (OS, Libs, Apps, etc.)
  - VM captures full application, help with repeatable experiment setup
  - VM snapshots can be helpful for speeding FI campaigns
- Disadvantages of VM+FI Approach
  - Adds challenge of “semantic gap”, but VMI techniques minimize issue
  - Increased level of complexity for testing environment/tools
    - May be necessary overhead if target is system software (high impact crash targets)
- Remarks on reproducibility
  - Pro: VMs capture application setup (reproducible)
  - Con: Host setup not captured (less reproducible)

Related Work

1. Aderholdt et al.
   – Used VMI methods to implement efficient VM checkpointing
2. Suesskraut et al.
   – Used VMs to speed FI campaigns (snapshot pre-injection)
3. DeBardeleben et al. (F-SEFI)
   – Extended QEMU's dynamic translation layer to corrupt instruction operands (e.g., `FMUL`)
   – Supports random or per-function basis & single or multi-bit errors, and injections based on deterministic or probabilistic basis
4. Le & Tamir (Gigan)
   – SWIFI using Xen, to harden ReHype hypervisor
   – Injectors inside VM equivalent to injectors outside VM
     - *ReviewerNote*: Fidelity of program-level SWIFI, not fully mirror HW vulnerabilities. So, be mindful of potential overestimations for bit-flips. Koopman similar warning on FI for Dep. Benchmarking
5. Li et al. (BIFIT)
   – Instrument binary (PIN) based on profile of HPC applications (no virtualization)
   – Observed global data had significant influence on app output/execution-state

Conclusion

- Cooperative approach for resilience investigation tool
  - Use strong isolation of VMs to isolate SUT from Controller
    - SUT in Guest (in VM), Controller on Host (outside VM)
    - Face “semantic gap” between SUT / Controller
  - Use Virtual Machine Introspection techniques to bridge “semantic gap”
    - Pre-configured utilities, symbol maps for OS and target app



- Demonstration of approach
  - Injected random errors into (serial) HPCCG benchmark
  - Keep injection tools (in host) isolated from SUT target (in VM)
- Next Steps
  - Gain better understanding of failure isolation properties
    - Virtual Machine based environments (e.g., type-I, type-II)
  - Alternate approach using different OS isolation mechanisms
    - OS virtualization environments (e.g., Linux containers)

Questions

- Oak Ridge National Laboratory Managed by UT Battelle, LLC under Contract No. De-AC05-00OR22725 for the U.S. Department of Energy.

- Research supported by US Department of Energy Office of Science, Advanced Scientific Computing Research (ASCR) program office, projects "*Hobbes OS and Runtime Support for Application Composition"* and "*Enabling Exascale System Design through Scalable System Virtualization*".

Auxiliary material

- Backup slides…

Additional Details (1)

- '`wrapper`' (inside VM)
  - Basic C program to launch (fork/exec/waitpid) an application

- Palacios VM "kmem" patch
  - Extends Linux kernel module for Palacios on x86-64
  - Register VM memory device & create char devices for VM in host
    1. Used for VMI routines to walk process list
    2. Used to modify memory (data) of specific process
  - Routines for reading VM memory to support VMI methods

OAK RIDGE National Laboratory

wrapper – target application launcher

```
Usage:
 ./wrapper <executable> [args]

Description:
 Wrapper utility to launch application and display useful information.
 Also, used a sentinel for locating the target process in the guest
 context, which is the child process of the wrapper utility.

Example:
  ./wrapper ./HPCCG 100 200 100
```

Fig. 1. Usage information for wrapper utility that runs within the guest VM context.

Additional Details (2)

- '`kmem`' tool (outside VM)
  - User-space tool to locate wrapper/target in VM
  - Implements VMI routines to walk page tables & process lists, etc.
  - Read/seek guestOS memory device using VMI methods
  - Embedded with key information for guestOS kernel layout
    - Example: Location of INIT_TASK, TASK_OFFSET, PID_OFFSET, etc.
  - Supports listing (finding) PID of the wrapper process (in VM)
    - Scans memory device, checking for any 'wrapper' processes
    - Walks process table in VM and prints PID for tasks with proc_name "wrapper"
  - Input parameters:
    - PID of wrapper process (in VM)
    - Symbol maps for: wrapper process & victim application (in VM)
    - Symbol name for target in victim application
    - Injection value (*what to write*), Number bytes (*how much to write*), Offset from target address (*additional bytes from target symbol*)

kmem - VM+FI Utility

```
Usage:
./kmem list
  --or--
./kmem <wrapper_pid> <wrapper_map_file> <victim_map_file>     \
       <target_symbol> <data_to_inject> <data_num_bytes>      \
       <offset_from_symbol>

 wrapper_pid                The pid of wrapper process residing in the guest
 wrapper_map_file           Mapping file for the wrapper process
 victim_map_file            Mapping file for the victim process
 target_symbol              The name of the symbol in the victim
                              process to inject a fault
 data_to_inject             What to inject into the victim process
 data_num_bytes             How many bytes to write
 offset_from_symbol         Any additional bytes (offset) from target symbol

Description:
  VMFI utility that can be used to LIST information about the guest context,
  or used to inject errors into a victim application running in the guest
  context.

Example:
  ./kmem 198 wrapper.map HPCCG.symmap rtrans 6 4 0
```

Fig. 2. Usage information for VMFI utility that runs on the host (outside VM).