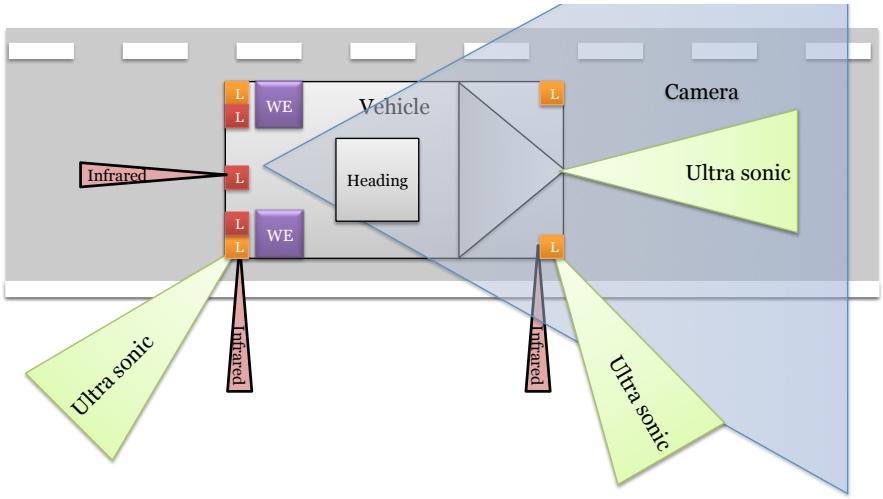
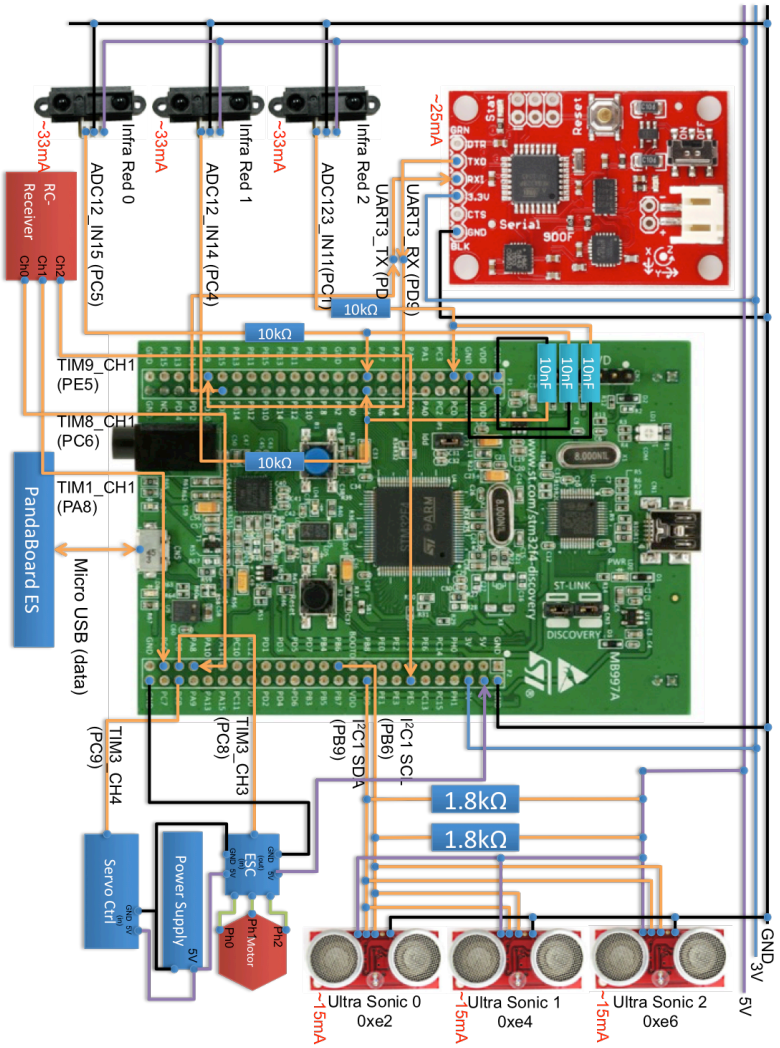
Accelerating Regression Testing for Scaled Self-Driving Cars with Lightweight Virtualization – A Case Study

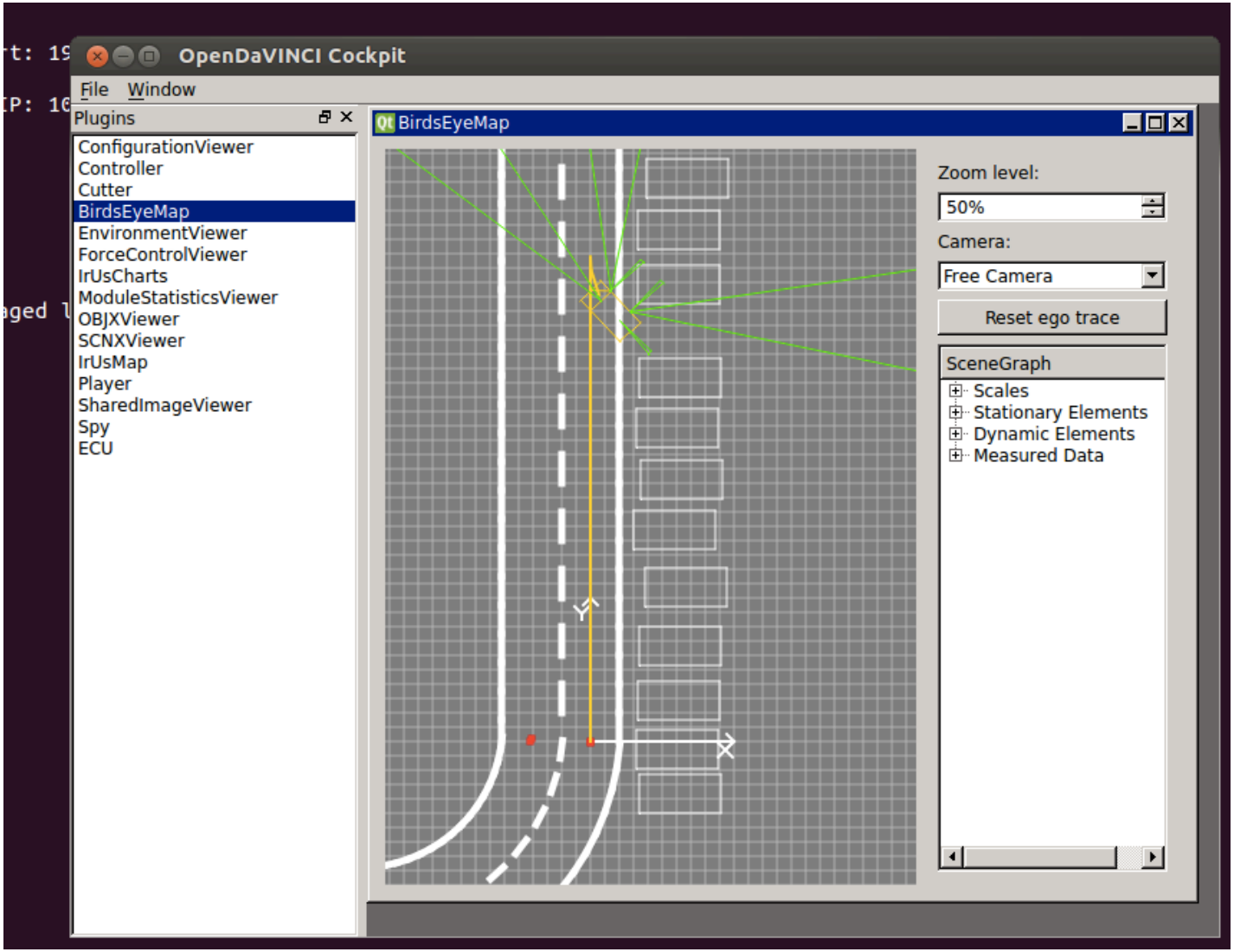
Christian Berger

Experimental Miniature Vehicle Fleet

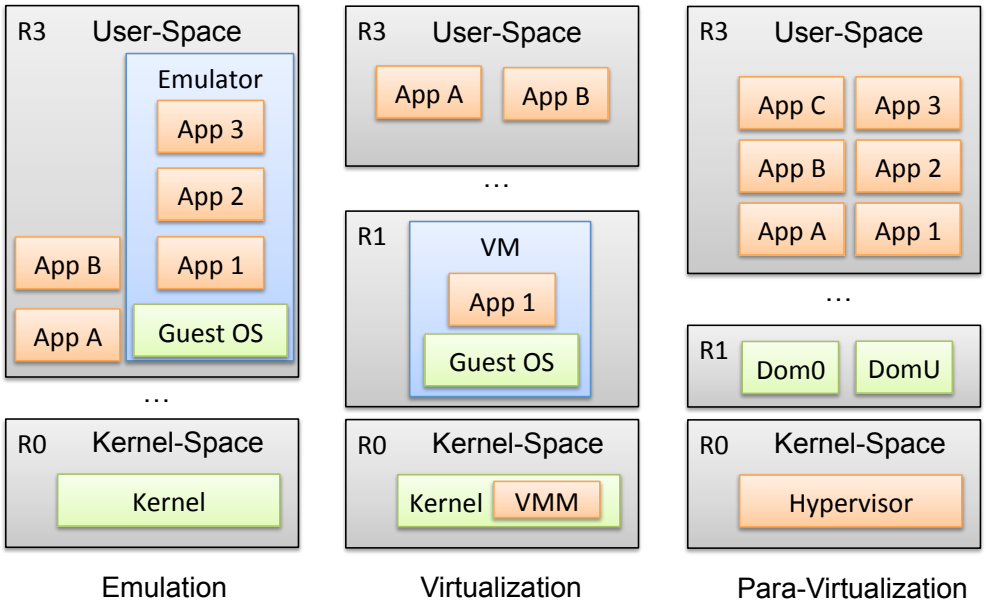




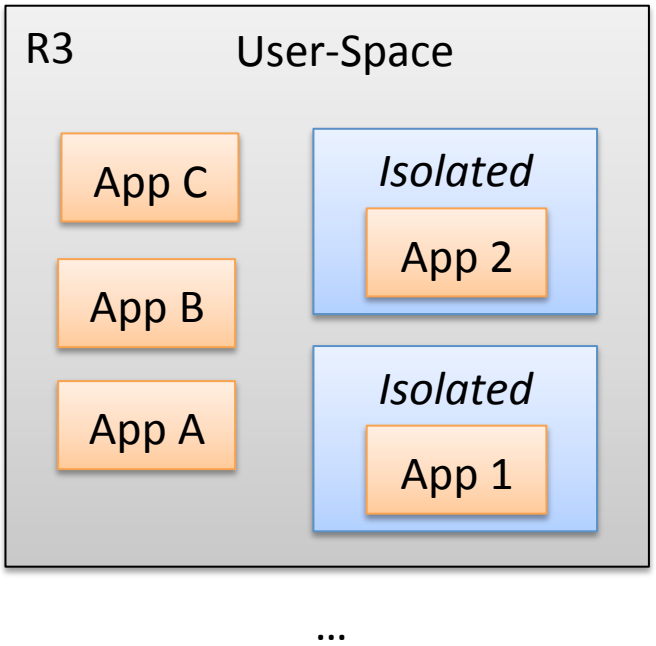
User's Perspective for Testing



Overview Virtualization Approaches



Emulation | Virtualization | Para-Virtualization



Docker

Concept:

- From Linux 2.6.25
- Shared kernel & process isolation
- Sharing or isolating system resources

➔ Application containers

Isolation:

- IPC
- NET
- UTS
- PID
- NS
- USER

Docker – User's Perspective

```
$ xhost +

$ sudo docker run --rm=true --net host -v /home/$USER/config:/opt/
data -w "/opt/data" -t -i seresearch/parking-demo:latest /opt/sim/
bin/odsupercomponent --cid=111 --freq=10 --managed=simulation_rt

$ sudo docker run --rm=true --net host -t -i seresearch/parking-
demo:latest /opt/sim/bin/vehicle --cid=111

$ sudo docker run --rm=true --net host -w "/opt/data" -t -i -v /
home/$USER/config:/opt/data seresearch/parking-demo:latest /opt/
sim/bin/irus --cid=111

$ sudo docker run --rm=true --net host -t -i seresearch/parking-
demo:latest /opt/sim/bin/boxparker --cid=111

$ sudo docker run --rm=true --net host -w "/opt/data" -t -i -e
DISPLAY=$DISPLAY -e QT_X11_NO_MITSHM=1 -v /home/$USER/config:/opt/
data -v /tmp/.X11-unix:/tmp/.X11-unix seresearch/parking-
demo:latest /opt/sim/bin/cockpit --cid=111
```

http://goo.gl/Qs5jts

Docker – Under the Hood

- Docker's concept simply relies on one new Linux syscall: clone(2)

- clone(2) is similar to fork(2):
  - Creates a new child process
  - But allows to specify the process context for the new child process regarding sharing **or** isolating of
    - Network devices
    - Hostnames
    - File system
    - Inter-process communication (shared memory, semaphores, …)

- Warning: "clone() is Linux-specific and should not be used in programs intended to be portable." (cf. man page)

The clone(2) System Call

- Consider the following test suite:

```
testCase1() {
    m = createSharedMemory("mem");
    assert(m is valid);
    write letters A .. J to m;
    read from m and compare written letters;
    release(m);
}

testCase2() {
    m = createSharedMemory("mem");
    assert(m is valid);
    write letters K .. S to m;
    read from m and compare written letters;
    release(m);
}
```

The clone(2) System Call

- Parallelizing the test suite with fork(2):

```
testCase1() {...}
testCase2() {...}

void runInParallel_fork() {
    pid_t child = fork();
    assert(child > -1);
    if (child == 0) { testCase1(); }
    else { testCase2(); }
}
```

- Behavior:

```
$ testSuite
Semaphore could not be created, errno: 2
testSuite: /home/msv/GITHUB/Mini-Smart-Vehicles/sources/
OpenDaVINCI-msv/examples/example8/Example8.cpp:56: void
examples::testCase1(): Assertion `memClient->isValid()'
failed.
```

The clone(2) System Call

- Parallelizing the execution of test cases using today's unit test environments:
  - CxxTest: not supported
  - Google Test (cf. http://goo.gl/mm2Q1F):
    - **Does Google Test support running tests in parallel?**

      Test runners tend to be tightly coupled with the build/test environment, and Google Test doesn't try to solve the problem of running tests in parallel. ...
    - **Why don't Google Test run the tests in different threads to speed things up?**

      It's difficult to write thread-safe code. Most tests are not written with thread-safety in mind, and thus may not work correctly in a multi-threaded setting.

      If you think about it, it's already hard to make your code work when you know what other threads are doing. It's much harder, and sometimes even impossible, to make your code work when you don't know what other threads are doing (remember that test methods can be added, deleted, or modified after your test was written). If you want to run the tests in parallel, you'd better run them in different processes.
  - gtest-parallel (parallelizing Google Test with Python): uses fork(2) to parallelize execution... (cf. http://svn.python.org/projects/python/trunk/Lib/subprocess.py)

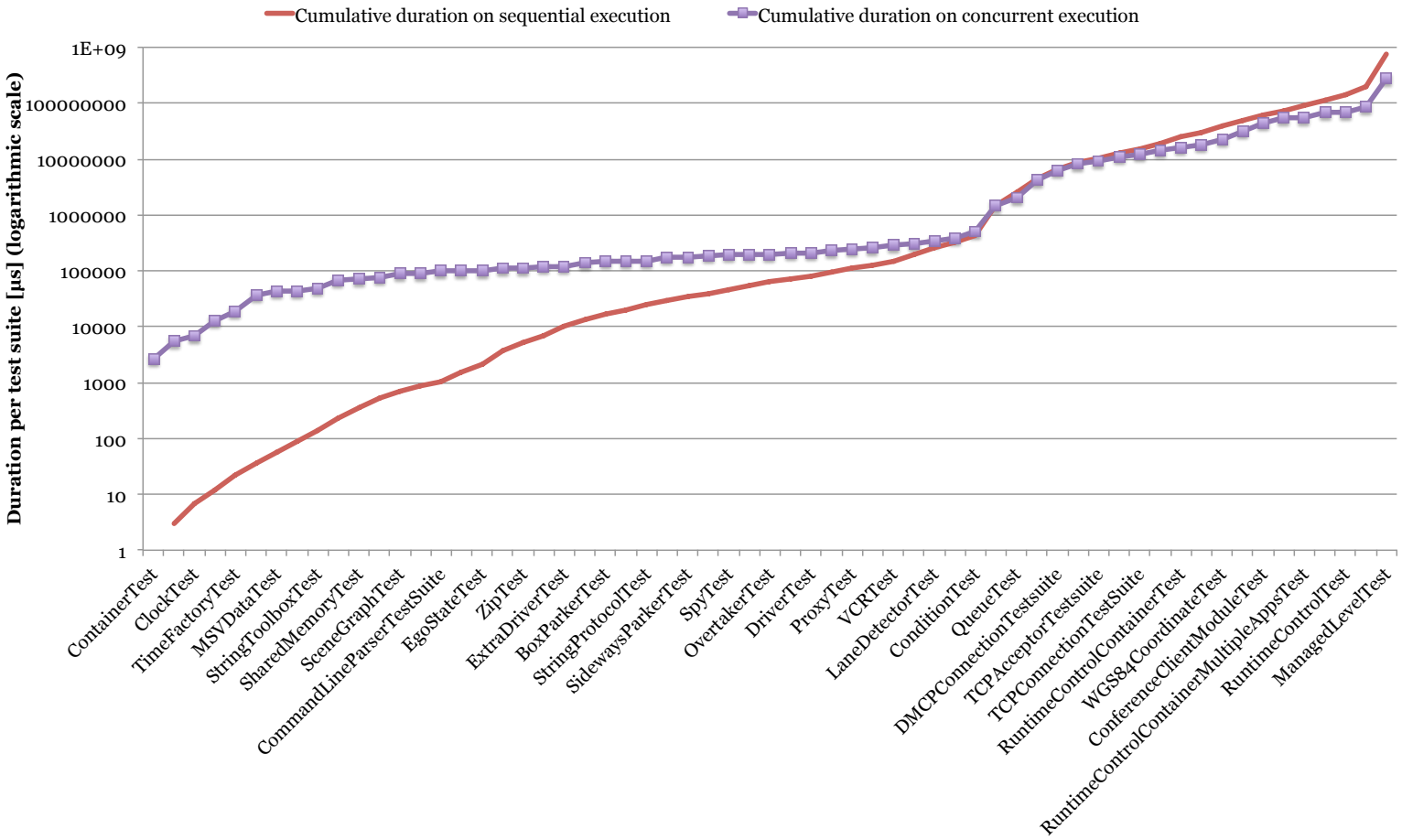
The clone(2) System Call

- Better: Create a “light-weight” virtual machine **per** test case:

```
testCase1() {...}
testCase2() {...}

void cloneWrapper(void *arg) {
    int id = *(int*)(arg);
    switch (id) {
        case 1: testCase1(); break;
        case 2: testCase2(); break;
    }
}

void runInParallel_clone() {
    pid_t child1, child2;
    char *stack1 = (char*)malloc(STACK_SIZE);
    char *stack2 = (char*)malloc(STACK_SIZE);

    int id = 1;
    child1 = clone(cloneWrapper, (*stack + STACK_SIZE), CLONE_NEWIPC | CLONE_NEWNET |
CLONE_NEWNS | SIGCHLD, (void*)id);

    child2 = clone(cloneWrapper, (*stack + STACK_SIZE), CLONE_NEWIPC | CLONE_NEWNET |
CLONE_NEWNS | SIGCHLD, (void*) ++id);
.
    waitpid(child1, NULL, 0);
    waitpid(child2, NULL, 0);
}
```

The clone(2) System Call

- Implemented a clone(2) wrapper for CxxTest to create a light-weight virtual machine **per** test case:



Summary & Conclusion

- Docker allows light-weight virtualization by sharing and isolating system resources using clone(2) system call
- clone(2) system call can also be embedded into Unit Test environments to safely isolate test cases from each other for parallel execution
- My examples are available here:

  https://hub.docker.com/u/seresearch/

  http://goo.gl/AOwj7U