A LISA love story

Valentin Schneider
Steven Miao
19th September 2018
Outline

- LISA in a nutshell
- Kernel analysis
- Automated tests
- Future of LISA
- Questions
LISA in a nutshell
A brief description

- Python
- Umbrella project that bundles
  - devlib for device communication abstraction
  - TRAPpy for trace parsing
  - ... And a few others
- Eases experimentation with Linux targets (dev boards, android phones)
- Allows kernel analysis via trace (FTrace) analysis
- Comes with IPython notebooks for easy plotting

- Grab it here
Overview diagram

- **Target**
  - Storage
    - trace.dat
    - binaries
  - APIs for target interaction: SSH / ADB

- **Host**
  - Storage
    - trace.dat
    - binaries
  - devlib
    - ftrace
    - cpufreq
    - hotplug
    - ...
  - TRAPpy
  - LISA
    - Trace analysis
    - Kernel tests
    - Notebooks
    - dataframes
  - APIs for target interaction: SSH / ADB
  - Trace parsing
  - System analysis
Kernel analysis
What's a misfit task?

- big.LITTLE world
  - Asymmetric CPU performance
  - Triage needs to be done on which tasks run on the big CPUs

- Misfit patch-set
  - Authored by Morten Rasmussen
  - Flag task when its utilization is > 80% of its current CPU's capacity
  - Upmigrate if possible

- Misfit logic has been in Android EAS for a few years
Migration delays (HiKey960)

- Occasionally, it can take a long time to upmigrate
Debug by trace_printk()

- Fixed a few things (active balance, rd->overload...), but one very rare issue remained

- Digging around led me to notice high sd->balance_interval values:

  ```
  trace_printk("interval_debug: sd=%lx last=%lu now=%lu interval=%lu\n",
  *cpumask_bits(sched_domain_span(sd)),
  sd->last_balance, jiffies, interval);
  ```

- TRAPpy can parse such custom traces
  - This will give us a "interval_debug" dataframe with a "balance_interval" column
  - Dataframe can be used to get statistics, we can apply filters, and even get plots!
Balance interval issue

- Plot of `sd->balance_interval` of big CPU at DIE level

- Issue happens before the test workload even began!
Balance interval issue (cont.)

- During the setup phase, devlib uses some pinned busybox
  - Leads to the balance_interval of a given SD to inflate
    - LBF_ALL_PINNED logic
  - Leads to long delays when we would like that CPU to pull a misfit task
Visualizing kernel function call graph in LISA

• Flexible and extensible
  ○ Easy to add python modules and find example python code

• Py graphviz
  ○ Provides a simple pure-Python interface for the Graphviz graph-drawing

• Based on ftrace Function Graph Tracer
  ○ # trace-cmd start -p function_graph -g try_to_wake_up
  ○ A graph of function calls similar to C code
Visualize call graph

- Write a python class to parse graph tracer log
  - Based on example code to parse Dtrace call graph log file
  - Create kernel function call Tree
- Create graphviz dot file
  - Travel tree to dump callees
- Create callgraph from dot file
KVM performance analyses on hikey960

KVM on ARMv8.0-A

- KVM is an open source type-2 (hosted) hypervisor
- Integrated into host (Linux) OS
- Host kernel booted at EL2, but runs at EL1
Sysbench cpu and ram test

Sysbench test on host os and guest os
• Sysbench ram bandwidth test
• Sysbench cpu test
Sysbench IO test

- Sysbench fileio test
Profiling syscall average execution time

- Based on ftrace kernel function profile statistics
find_best_target pack or spread?

• Non latency sensitive tasks path
  ○ Select cpu with max spare capacity against capacity_origin (target_max_spare_cap = capacity_orig - new_util)
  ○ If we have two CPUs with the same max spare cap, we will chose second cpu
  ○ Next time, if the max spare cap is still same, will chose second cpu again

• It could be imbalanced on some cases
  ○ Fixed order for find_best_target

• run two tasks on two CPU VS packing both on a single CPU
Case assumption

- run two tasks on two CPUs at a lower OPP than packing both on a single CPU but running that CPU at an higher OPP

<table>
<thead>
<tr>
<th>CPU</th>
<th>Capacity Orig</th>
<th>cpu_util</th>
<th>New_util</th>
<th>target_max_spare_cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 0</td>
<td>500</td>
<td>100</td>
<td>562 (100 + (1024-100)*50% = 562)</td>
<td>continued via “new_util &gt; capacity_orig”</td>
</tr>
<tr>
<td>CPU 1</td>
<td>500</td>
<td>100</td>
<td>562</td>
<td>continued via “new_util &gt; capacity_orig”</td>
</tr>
<tr>
<td>CPU 2</td>
<td>1024</td>
<td>100</td>
<td>562</td>
<td>462</td>
</tr>
<tr>
<td>CPU 3</td>
<td>1024</td>
<td>100</td>
<td>562</td>
<td>462</td>
</tr>
</tbody>
</table>
Case assumption cont.

- A task with task_util 10 and boost value 50 comes, CPU 3 is still selected.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Capacity_orig</th>
<th>cpu_util</th>
<th>New_util</th>
<th>target_max_spare_cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>500</td>
<td>100</td>
<td>517 (10 + (1024-10)*50% = 517)</td>
<td>continued via “new_util &gt; capacity_orig”</td>
</tr>
<tr>
<td>1</td>
<td>500</td>
<td>100</td>
<td>517</td>
<td>continued via “new_util &gt; capacity_orig”</td>
</tr>
<tr>
<td>2</td>
<td>1024</td>
<td>100</td>
<td>517</td>
<td>507</td>
</tr>
<tr>
<td>3</td>
<td>1024</td>
<td>200</td>
<td>517</td>
<td>507</td>
</tr>
</tbody>
</table>
Create real test cases and analyze with LISA tools

- Create 10% percent cpu duty rt-app tasks for each big core(4, 5, 6, 7 on hikey960)
- Create small boosted delayed tasks
- Add trace_printk message for case A, B, C code path
  ○ trappy.register_dynamic_ftrace("eastestc", "casec")
- CPU residency plot
Filter for new booted small task

In [27]:
```python
select=(df.task == 'stp_6')
df[select]
```

Out[27]:
```
<table>
<thead>
<tr>
<th>__comm</th>
<th>__cpu</th>
<th>__line</th>
<th>__pid</th>
<th>capacity_orig</th>
<th>new_util</th>
<th>pid</th>
<th>target_cpu</th>
<th>target_max_spare_cap</th>
<th>task</th>
</tr>
</thead>
<tbody>
<tr>
<td>stp_4</td>
<td>7</td>
<td>31189</td>
<td>6692</td>
<td>1024</td>
<td>588</td>
<td>6594</td>
<td>7</td>
<td>436</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>7</td>
<td>42223</td>
<td>0</td>
<td>1024</td>
<td>544</td>
<td>6594</td>
<td>7</td>
<td>480</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>7</td>
<td>43158</td>
<td>0</td>
<td>1024</td>
<td>578</td>
<td>6594</td>
<td>5</td>
<td>446</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>7</td>
<td>43162</td>
<td>0</td>
<td>1024</td>
<td>578</td>
<td>6594</td>
<td>7</td>
<td>446</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>7</td>
<td>44242</td>
<td>0</td>
<td>1024</td>
<td>536</td>
<td>6594</td>
<td>4</td>
<td>488</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>7</td>
<td>44243</td>
<td>0</td>
<td>1024</td>
<td>536</td>
<td>6594</td>
<td>5</td>
<td>488</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>7</td>
<td>44244</td>
<td>0</td>
<td>1024</td>
<td>536</td>
<td>6594</td>
<td>6</td>
<td>488</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>7</td>
<td>44887</td>
<td>0</td>
<td>1024</td>
<td>812</td>
<td>6594</td>
<td>4</td>
<td>212</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>4</td>
<td>45618</td>
<td>0</td>
<td>1024</td>
<td>598</td>
<td>6594</td>
<td>4</td>
<td>426</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>4</td>
<td>46301</td>
<td>0</td>
<td>1024</td>
<td>626</td>
<td>6594</td>
<td>4</td>
<td>398</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>4</td>
<td>49098</td>
<td>0</td>
<td>1024</td>
<td>628</td>
<td>6594</td>
<td>4</td>
<td>396</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>4</td>
<td>50646</td>
<td>0</td>
<td>1024</td>
<td>558</td>
<td>6594</td>
<td>4</td>
<td>466</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>4</td>
<td>51416</td>
<td>0</td>
<td>1024</td>
<td>561</td>
<td>6594</td>
<td>7</td>
<td>463</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>4</td>
<td>52527</td>
<td>0</td>
<td>1024</td>
<td>565</td>
<td>6594</td>
<td>4</td>
<td>459</td>
<td>stp_6</td>
</tr>
<tr>
<td>&lt;idle&gt;</td>
<td>4</td>
<td>53184</td>
<td>0</td>
<td>1024</td>
<td>601</td>
<td>6594</td>
<td>4</td>
<td>423</td>
<td>stp_6</td>
</tr>
<tr>
<td>stp_6</td>
<td>1</td>
<td>53577</td>
<td>6594</td>
<td>1024</td>
<td>780</td>
<td>6594</td>
<td>4</td>
<td>244</td>
<td>stp_6</td>
</tr>
</tbody>
</table>
```
CPU residency
Automated tests
Tests overview

- Most of them are powered by **rt-app**
  - Makes it easy to create e.g. a 20% utilization periodic task

- **EAS behaviour**
  - Collection of different task compositions
  - Task latency, CPU selection are verified

- **Cpufreq**
  - Higher frequencies lead to more work done

- **Hotplug**
  - Target survives hotplug stress-test

- **Load tracking signals**

- **Misfit behaviour**
Migration test on HiKey960

- Tasks alternating between low utilization and high utilization phases
class EnergyModelWakeMigration(EASBehaviour):
    """
    One task per big CPU, alternating between two phases:
    * Low utilization phase (should run on a LITTLE CPU)
    * High utilization phase (should run on a big CPU)
    """
    task_prefix = "emwmn"

@classmethod
def create_rtapp_profile(cls, te):
    rtapp_profile = {}
    capacities = te.target.sched.get_capacities()
    bigs = [cpu for cpu, capacity in list(capacities.items())
            if capacity == cls.max_cpu_capacity(te)]

    start_pct = cls.unscaled_utilization(cls.min_cpu_capacity(te), 20)
    end_pct = cls.unscaled_utilization(cls.max_cpu_capacity(te), 70)

    for i in range(len(bigs)):
        rtapp_profile["{}{}".format(cls.task_prefix, i)] = Step(
            start_pct=start_pct,
            end_pct=end_pct,
            time_s=2,
            loops=2,
            period_ms=cls.TASK_PERIOD_MS
        )

    return rtapp_profile

How much utilization we want

The task composition itself
Statistical evaluation

- 03/08/2018 mainline integration results on HiKey960

<table>
<thead>
<tr>
<th>Failure rate evolution from 20180803_0000:</th>
<th>failure_old</th>
<th>failure_new</th>
<th>delta</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tests.eas.misfit.StaggeredFinishes.test_migration_delay</td>
<td>2.64%</td>
<td>6.72%</td>
<td>4.08%</td>
<td>2.59e-02</td>
</tr>
<tr>
<td>tests.eas.load_tracking.OneTaskCPUMigrationTest.test_utill_task_migration</td>
<td>0.00%</td>
<td>65.05%</td>
<td>65.05%</td>
<td>1.04e-79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>test/generic (LISA-test)</th>
<th>test_slack:</th>
<th>passed 372/372 (100.0%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tests.eas.generic.EnergyModelWakeMigration:</td>
<td>test_slack:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.EnergyModelWakeMigration:</td>
<td>test_task_placement:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.OneSmallTask:</td>
<td>test_slack:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.OneSmallTask:</td>
<td>test_task_placement:</td>
<td>FAILED 1/372 (0.3%)</td>
</tr>
<tr>
<td>tests.eas.generic.RampDown:</td>
<td>test_slack:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.RampDown:</td>
<td>test_task_placement:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.RampUp:</td>
<td>test_slack:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.RampUp:</td>
<td>test_task_placement:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.ThreeSmallTasks:</td>
<td>test_slack:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.ThreeSmallTasks:</td>
<td>test_task_placement:</td>
<td>FAILED 1/372 (0.3%)</td>
</tr>
<tr>
<td>tests.eas.generic.TwoBigTasks:</td>
<td>test_slack:</td>
<td>FAILED 10/372 (2.7%)</td>
</tr>
<tr>
<td>tests.eas.generic.TwoBigTasks:</td>
<td>test_task_placement:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.TwoBigThreeSmall:</td>
<td>test_slack:</td>
<td>passed 372/372 (100.0%)</td>
</tr>
<tr>
<td>tests.eas.generic.TwoBigThreeSmall:</td>
<td>test_task_placement:</td>
<td>FAILED 3/372 (0.8%)</td>
</tr>
</tbody>
</table>
• A new automated test suite for benchmarking Linux scheduler & EAS improvements on Android workloads.
  ○ WA is pretty good in running massive sets of experiments in a more portable way among different platforms
  ○ The LISA API will still be around but mainly for "interactive sessions"
• `wltests` is built on top of Lisa and Workload Automation (in-development version of WA v3)
  ○ WA3 actually has a feature to automatically download the APKs
  ○ Run `wltest` in a virtualenv environment
• It is intended to allow full evaluation of EAS/scheduler changes with real Android workloads
  ○ For example PELT vs. WAL comparisons)
WLTEST workflow

LISA console

Autobuild & auto flash device

WA3 autotest

Extract test result wa_output

LISA post process(data visualization)

test rounds
Integrate wltest to customer’s autotest framework

• Wltest is cool, but customer has their autobuild and autotest framework
  ○ They don't need kernel autobuild step
  ○ WA is easy to integrate to their autotest framework

• Compare test results with sched-evaluation-full notebook
  ○ Wltest results collect and compare against kernel sha1 id
  ○ Can't compare same kernel with different configuration
Result analysis against test tags

workloads:
- name: exoplayer

classifiers:
tag: test1

workload_parameters:
format: "mov_720p"
duration: 10
Result analysis PELT vs WALT
Future of LISA
Upcoming changes

- Python 3
  - Python 2 end of life is getting near
- Statistical approach to results processing
- New API for kernel tests
  - Lots of time lost circumventing the API
  - Lessons have been learned, good time for a redesign
  - Also opportunity to make it more CI-friendly
- Documentation facelift
  - Ease of use must be improved
2018 roadmap

● August
  ○ Executor API rework
  ○ Test execution (prev. nosetest) rework

● September
  ○ Port existing tests to the new API
  ○ Begin Python 3 migration work

● October
  ○ Internal release
  ○ Get some mileage (humans + CI)

● November
  ○ External release, new branch on the official repo
  ○ After some time (~6 months), this becomes the default branch

● More tests!

● A few links
  ○ [WIP branch](#)
  ○ [WIP documentation](#)
**API**

**Base classes**

```python
class CapacitySanityCheck(TestBundle):
    """
    A class for making sure capacity values make sense on a given target
    :param capacity_work: A description of the amount of work done on the
target, per capacity value ([{capacity: work}])
    :type capacity_work: dict
    """

def __init__(self, res_dirl, capacity_work):
    super(CapacitySanityCheck, self).__init__(res_dirl)

    self.capacity_work = capacity_work

classmethod
def __from_target(cls, te, res_dirl):
    with te.target.cputest.use_governor("performance"):
        sysbench = Sysbench(te, "sysbench", res_dirl)

        cpu_capacities = te.target.sched_list_capabilities()
        cpu_work = ["{capa : sys.maxint for capa in cpu_capacities.values()}"
                    for cpu in cpu_capacities.keys()]

        # We could save the work done on each CPU, but we can make
        # things simpler and just store the smallest amount of work done
        # per capacity value.
        cpu_work = cpu_work[capa] = min(capus.work[capa], sysbench.output.nr_events)

        return cls(res_dirl, cpu_work)

def test_capacity_sanity(self):
    """
    Assert that CPU capacity increase leads to more work done
    """
    sorted_capacities = sorted(self.capacity_work.keys())
    res = ResultBundle(True)

    for capacity, work in self.capacity_work.items():
        res.add_metric("Performance @{} capacity", format(capacity), work)

    for idx, capacity in enumerate(sorted_capacities):
        if self.capacity_work[capacity] == self.capacity_work[sorted_capacities[idx]]:
            res.passed = False

    return res
```
Questions