Existing Linux Kernel support for big.LITTLE?

Using existing Kernel features to control task placement in big.LITTLE MP systems running Android

Chris Redpath, ARM

chris.redpath@arm.com

Why do this?

- Discussed at Linaro Connect Q1.12
  - Scheduler Mini-Summit
- We ought to be able to achieve some workable solution
  - We have cgroups, hotplug, sched_mc
  - We have control over CPU task placement
- Decided to see how well we can do it!

The Architecture for the Digital World®
ARM®

What System?

- No hardware around to play with, so modelling is only option
  - Restricts what we can learn: which approaches are likely to be worth investigating on hardware
- An ARM RTSM Versatile Express model with a fictional logic tile
  - Has a Cortex A15 MP4 and a Cortex A7 MP4 CPU with coherent interconnect fabric between the two clusters
  - Same model as used for Linaro in-kernel switcher development
    - Very similar board support and boot code
    - Thanks to Dave Martin & Nicolas Pitre
- Linaro 12.04 Android release with some customisations
- Linux Kernel 3.2.3+ with Android support from Vishal Bhoj
  - Integrated some support from Vincent Guittot's kernel
    - sched_mc, Arch cpu_power & debugfs cpu_power controller

The Architecture for the Digital World®

Not tested..

- Hotplug
  - Not used at all here, but you'd probably need it if you wanted to freeze the kernel today
- Power savings
  - The model isn't cycle accurate so we can't even do rough estimates
- Performance (ish)
  - The model doesn't have any performance difference between big and little cores, so need to be careful with these results

The Architecture for the Digital World®
ARM®

Hypotheses to test

1. We can configure cgroups using cpusets to be helpful for static task partitioning
   - Use a separate cgroup for each cluster
2. Sched_mc can be used to concentrate tasks
   - Keeping short-lived tasks off the big CPUs naturally
3. User-side tools can help out the scheduler by moving tasks between groups
   - User side can make decisions based on what is happening
   - Android has a reasonably good mechanism already

The Architecture for the Digital World®
ARM®

Android cgroups

- Android already uses cgroups for task classification and cpuacct for task tracking
  - /dev/cpuctl/bg_non_interactive
    - Explicit background tasks (services etc.)
    - Low priority tasks (auto-background if thread priority low enough – ANDROID_PRIORITY_BACKGROUND)
    - Maximum 10% CPU usage
  - Root group /dev/cpuctl/
    - All other tasks
    - Unconstrained CPU usage
- Cgroup control implemented in thread priority setting code in a C library
  - used by both Dalvik code and C runtime code
  - system/core/libcutils/sched_policy.c

The Architecture for the Digital World®
ARM®

Existing cgroup hierarchy

- /dev/cpuctl/
  - Root group, load balancing turned on.
  - Has to have access to all CPUs
  - No changes to this group!
- /dev/cpuctl/bg_non_interactive
  - Restrict CPU affinity to Core 0
  - Remove CPU% age restriction
    - Core 0 likely to be in use for IRQs anyway
    - One little core out of the dual cluster setup is rather like a 7.5% restriction in practise and easier for me to think about ☺

The Architecture for the Digital World®
ARM®

New cgroups (1)

- /dev/cpuctl/default
  - Restricted to CPU0-3 (little CPUs)
  - Anything which is not ‘SCHED_BATCH’ (background) sched_policy goes in here
  - Load balancing enabled
  - Tasks report the same scheduler policy (SCHED_NORMAL) as those in root group
    - restrict Android changes to one file
  - Use our taskmove program to move all tasks from /dev/cpuctl/tasks to /dev/cpuctl/default/tasks early in boot
    - However some tasks don’t move
    - Others appearing later end up in the root group too

The Architecture for the Digital World®

New cgroups (2)

- /dev/cpuctl/fg_boost
  - Probably not Google's fg_boost reincarnated (couldn't find any code) but I liked the name
  - Restricted to CPU4-7 (big CPUs)
  - Load Balancing enabled
  - Tasks in this group report the same scheduler policy as those in root and default groups
  - Tasks with SCHED_NORMAL policy AND a priority higher than ANDROID_PRIORITY_NORMAL (i.e. <0) are placed in this group
  - I call this 'priority-based group migration'

- Here we are aiming to give access to the fastest CPUs to the most important (for responsiveness) tasks

The Architecture for the Digital World®

Hypothesis 1

1. We can configure cgroups using cpusets to be helpful for static task partitioning
   - Use a separate cgroup for each cluster

The Architecture for the Digital World®
ARM®

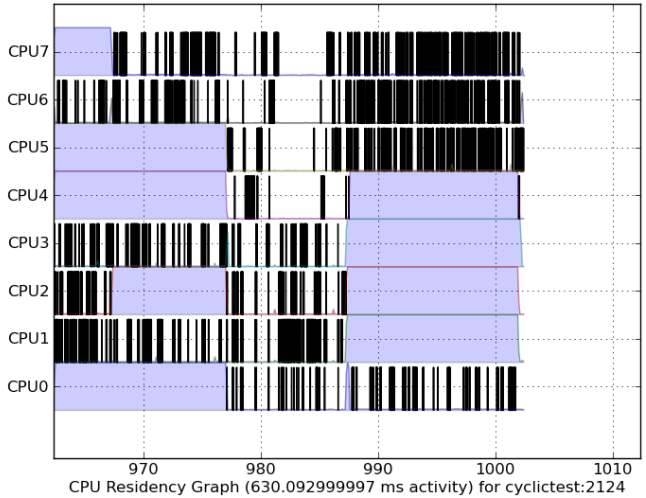
Synthetic Test Suite

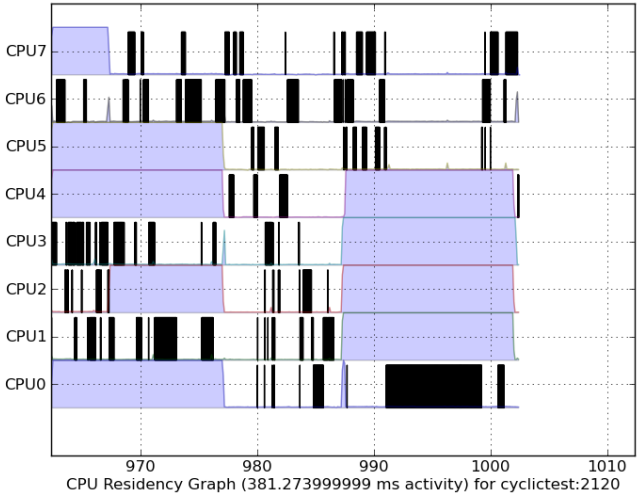
- 4 Sysbench Compute benchmark threads
  - Run 4 at once for 15s with a 10s break and then 4 more for another 15s
- 8 CyclicTest threads
  - All 8 run for the entire 40s use case
- Collect Kernel Ftrace output for each test case
- Variable options, driven with single test script - 18 variations tested

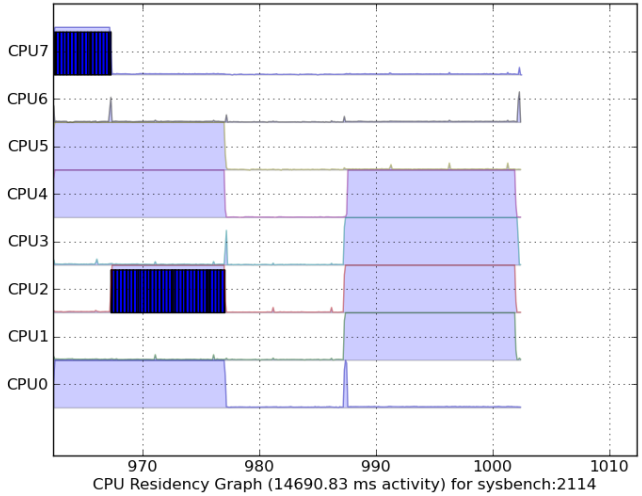
Test case	cpu_power	sched_mc	cgroups
1	modified	2	affinity
2	modified	1	affinity
3	modified	0	affinity
4	default	2	affinity
5	default	1	affinity
6	default	0	affinity
7	modified	2	No affinity
8	modified	1	No affinity
9	modified	0	No affinity
10	default	2	No affinity
11	default	1	No affinity
12	default	0	No affinity
13	modified	2	Optimised affinity
14	modified	1	Optimised affinity
15	modified	0	Optimised affinity
16	default	2	Optimised affinity
17	default	1	Optimised affinity
18	default	0	Optimised affinity

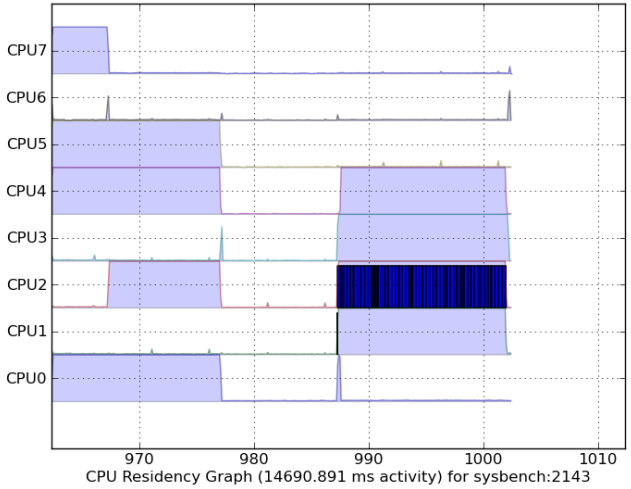
The Architecture for the Digital World®
ARM®

Synthetic Test 12, no modifications





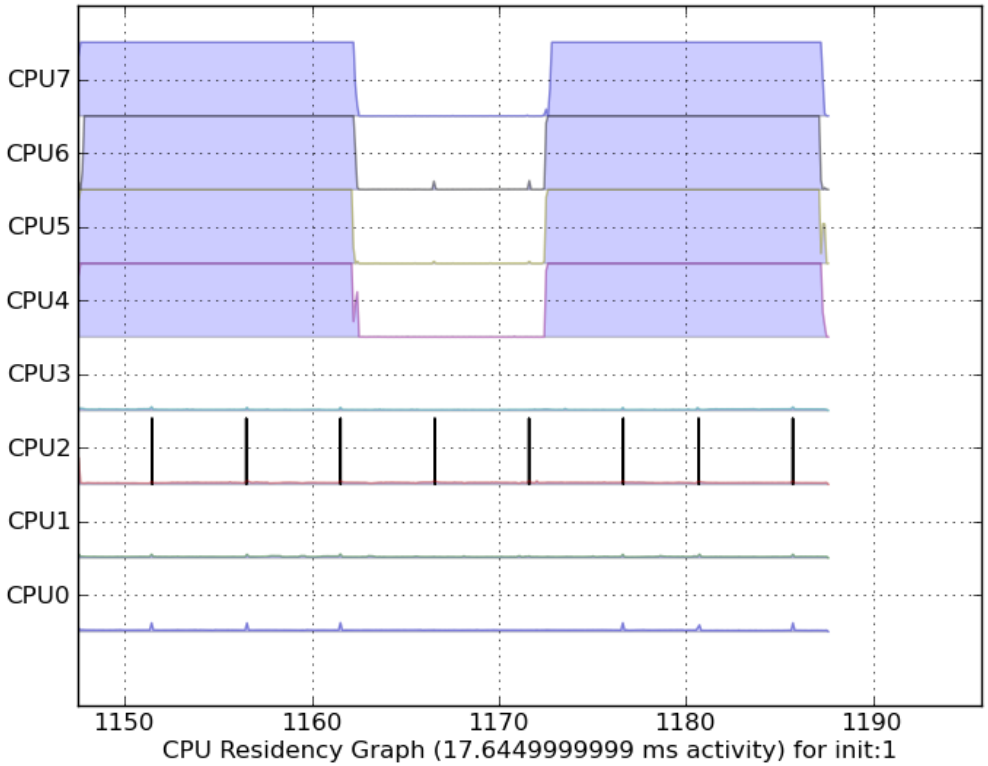




The Architecture for the Digital World®
ARM®

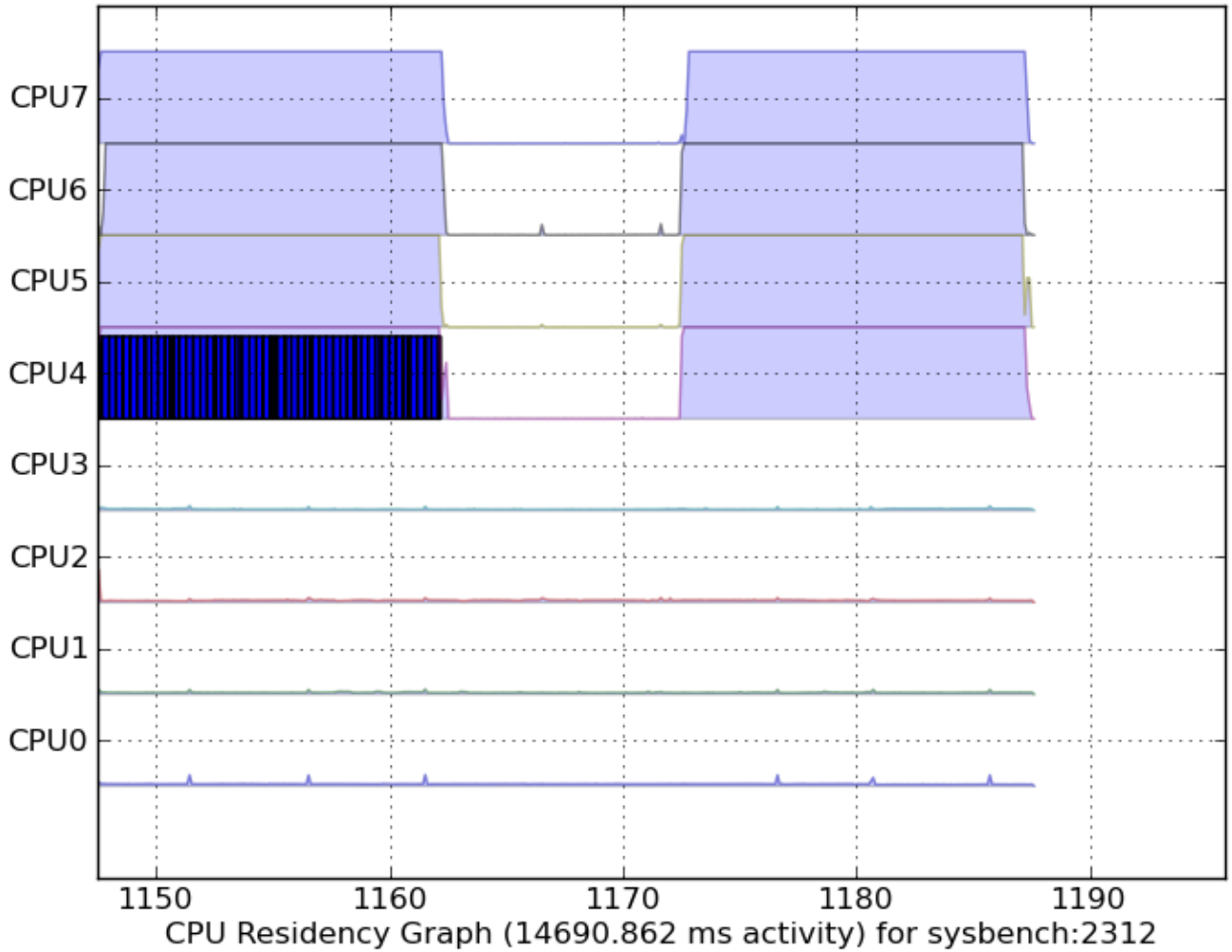
Synthetic Test cgroup Control

- Place sysbench threads in ‘fg_boost’ group (big CPUs)
- Place cyclictest threads in ‘default’ group (little CPUs)



The Architecture for the Digital World®
ARM®

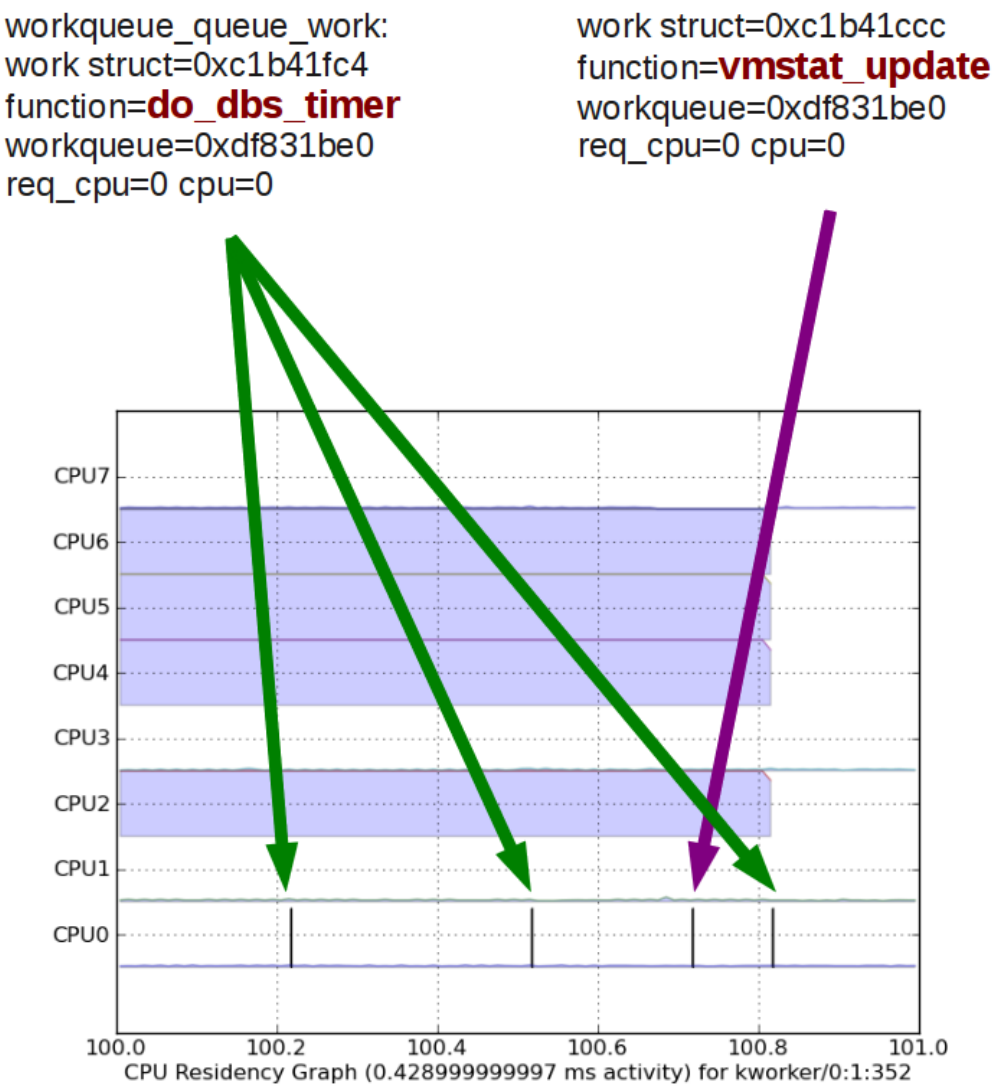
Cgroup control - sysbench



The Architecture for the Digital World®
ARM®

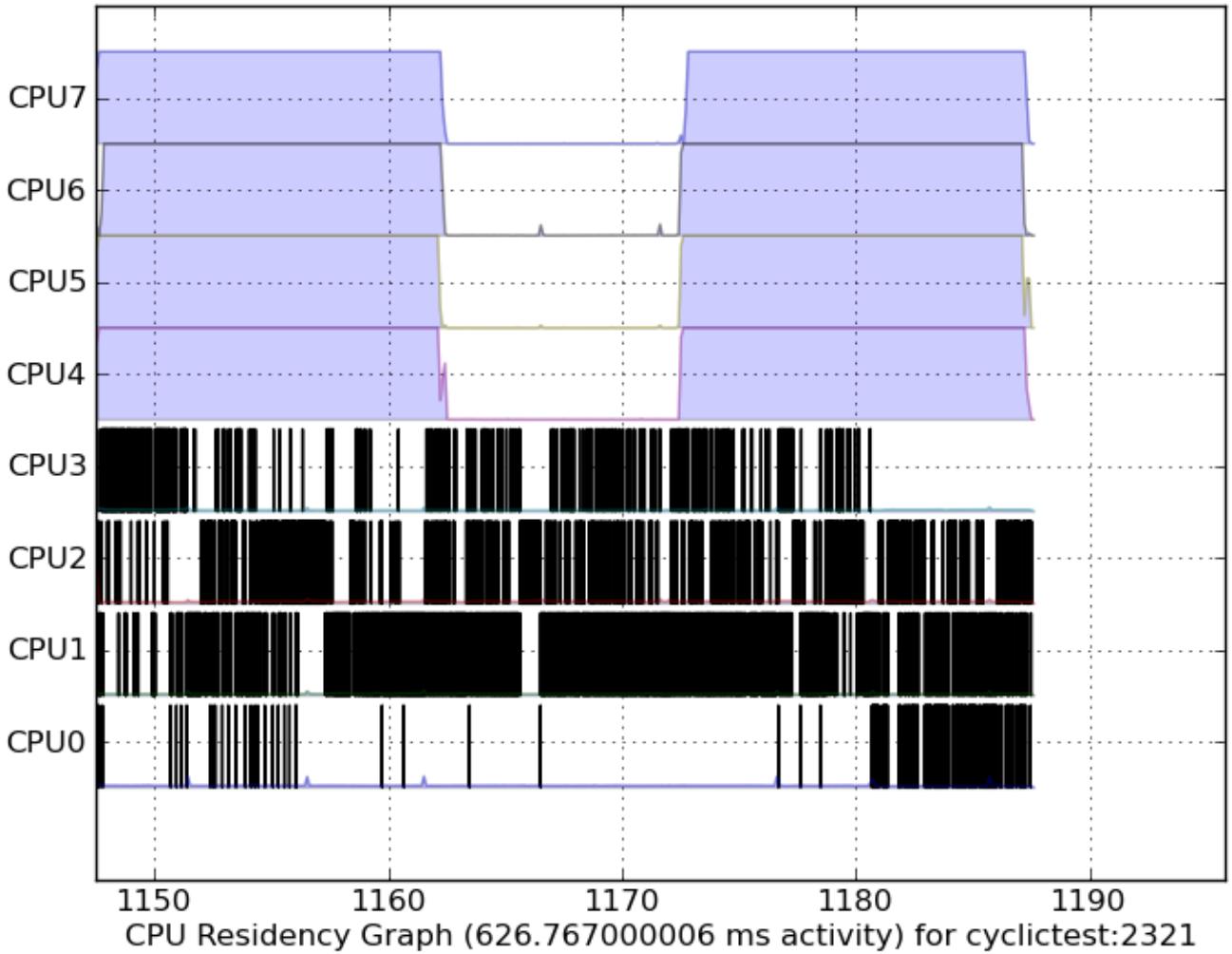
Persistent kworker thread activity

- Seen on all cores in all tests
- ondemand cpu governor calls do_dbs_timer
  - this is the cause of most calls
  - ~every 0.3s for each CPU
- vm stat generation calls vmstat_update



The Architecture for the Digital World®
ARM®

Cgroup control - cyclictest



The Architecture for the Digital World®
ARM®

Hypothesis 1 – TRUE

- We can configure cgroups using cpusets to be helpful for static task partitioning
- The tasks in a group will balance nicely
- Cgroups can be used for all situations where we know which threads need to execute on which cluster

The Architecture for the Digital World®
ARM®

Hypothesis 2

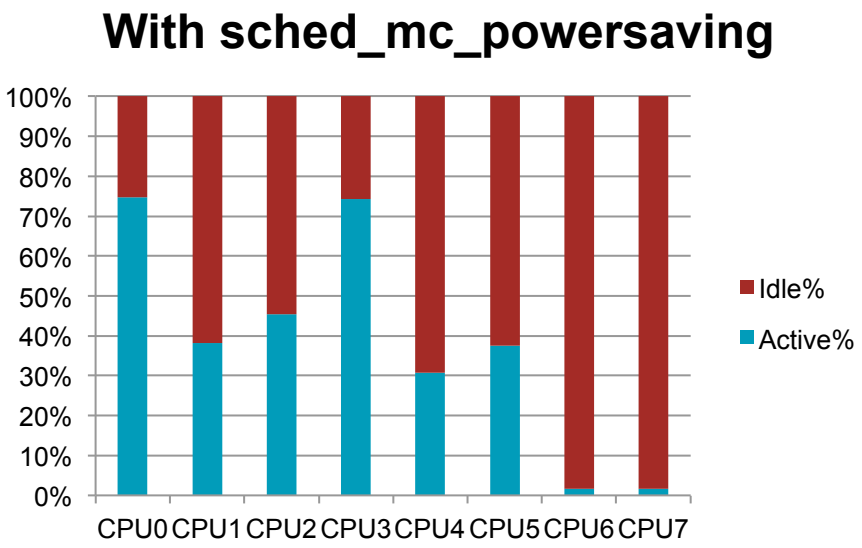
- Sched_mc can be used to concentrate tasks
  - Keeping short-lived tasks off the big CPUs naturally

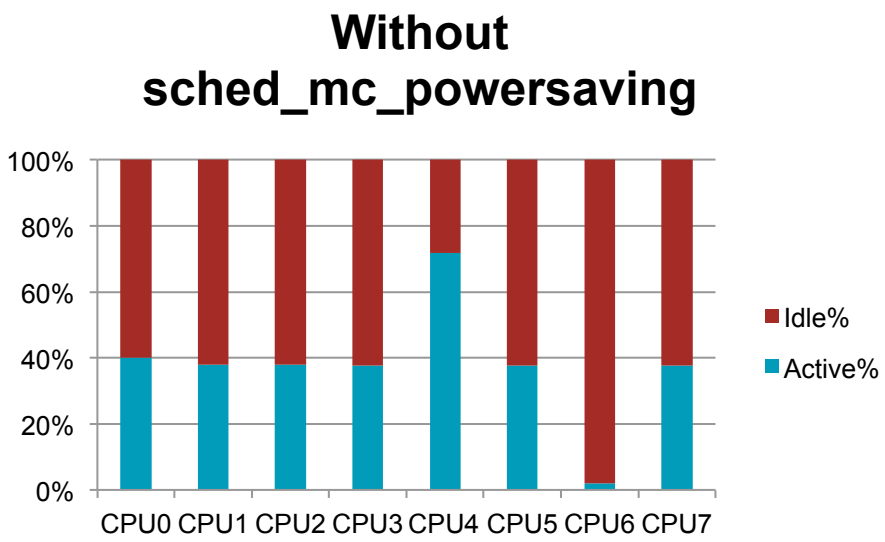
- Compare two cases:
  - Test case 10
    - Sched_mc_power_savings set to 2
    - Cpu_power set to 1024 for all cores
    - Cgroups have no affinity
  - Test Case 12
    - Sched_mc_power_savings set to 0
    - Cpu_power set to 1024 for all cores
    - Cgroups have no affinity

The Architecture for the Digital World®
ARM®

It does change spread of tasks

- With powersave enabled:
  - Tasks are spread over fewer cores
- Without powersave
  - Tasks spread fairly evenly amongst cores
- However...

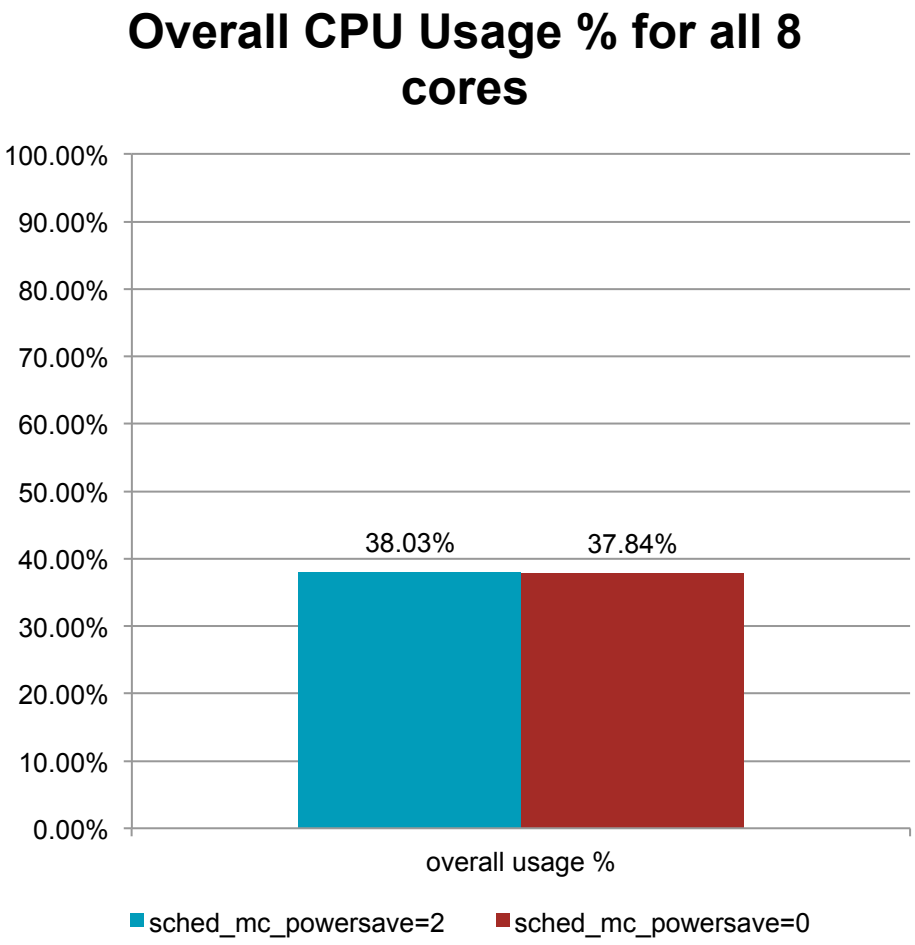




The Architecture for the Digital World®
ARM®

It doesn't change much overall

- Average CPU usage actually increases slightly in this case
  - Difference is generally within a few percent either way
  - Can only achieve power saving if we had been able to enter a deeper sleep state on the one core we vacated
  - Caveat: model performance & no power management!
  - Although we vacated a big core this time, that varies
- Difference is not convincing on model, needs evaluating on HW



The Architecture for the Digital World®
ARM®

Hypothesis 2 – partially TRUE

- Sched_mc does concentrate tasks
  - Usefulness for power depends on hardware configuration

- Sched_mc does not appear to be very useful as an automatic 'little aggregator'
  - Might not be missing much but..
  - Combination of Sched_mc & asymmetric packing options does not result in migration of tasks towards lower-numbered cores
    - No result in this slide set

The Architecture for the Digital World®
ARM®

Hypothesis 3

- User-side tools can help out the scheduler by moving tasks between groups
- Test with Browser application
- Using thread priorities as an indication of important tasks, place threads in either big or little groups as appropriate
  - Hook into Android's priority control
- I call this..

The Architecture for the Digital World®
ARM®

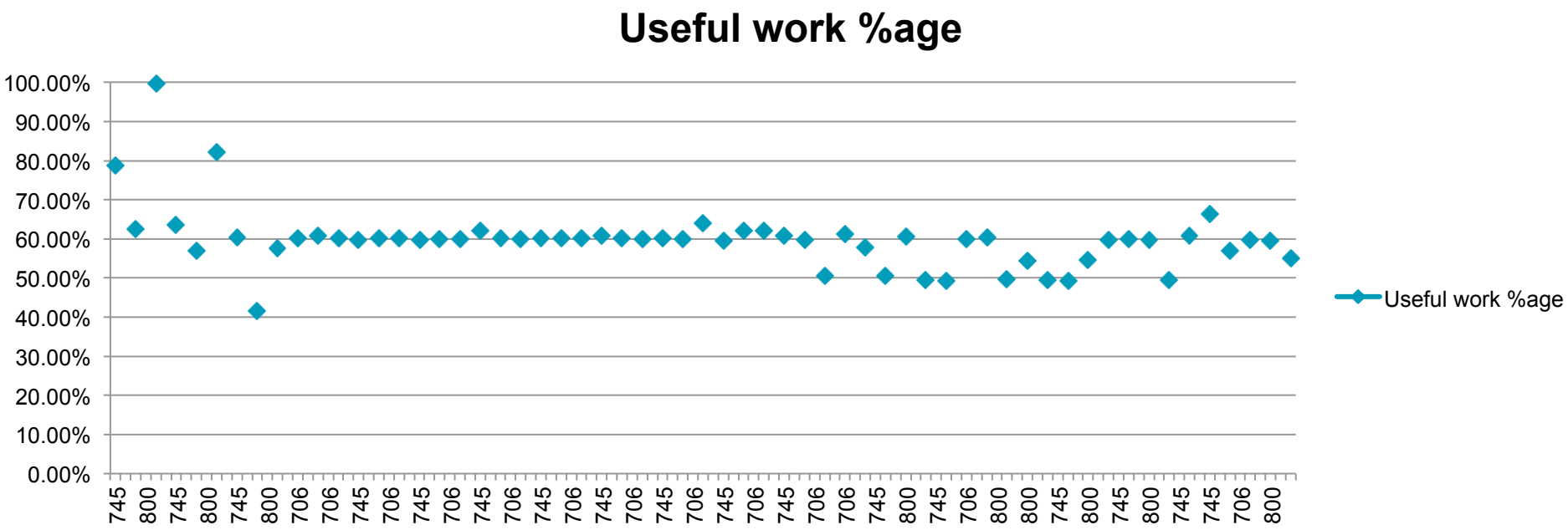
Priority-Based Group Migration

- Android documentation says that foreground applications receive a priority boost so that they get more favourable CPU allocations than other running applications
- So use this boosted priority to move threads between clusters!
- Easiest option to do that:
  - Move threads into a different cgroup
  - Assign cpuset.cpus on the groups to separate the clusters, as we described earlier

The Architecture for the Digital World®
ARM®

Priority-based Group Migration?

- Which threads qualify in the browser? Only UI Threads
  - Priority boost only happens for threads involved in drawing or input delivery – in our case mostly surfaceflinger and binder
  - Boost also only occurs for short periods of time – average 2.6ms, minimum 530us, max 120ms
  - Average Migration Latency is 2.9ms, minimum 768us, max 240ms



The Architecture for the Digital World®
ARM®

Browser Launch Conclusions

- As it happens, the responsiveness boost delivered to applications does not work the way I assumed..
  - I'd hoped that the application as a whole would get boosted but instead individual functions are boosted
  - Due to the multi-process UI model in Android, this priority boost ripples across a number of threads as the UI activity is processed which multiplies latency
  - Priority boosts are in place for short periods of time (generally milliseconds at a time) which means that we have to be careful about latency

- However we are able to move UI threads back and forth between big and little clusters from the Android userspace code with very minimal changes

The Architecture for the Digital World®
ARM®

Runtime Cluster Migration Issues

- Since the priority boost is only in place for a short time, the latency of this migration becomes very important
  - If priority was raised for all foreground application threads while the app was in the foreground, latency would not be high enough to worry about
  - Even with this latency, when loading and rendering the test page over 15s we spend a total of 36ms doing migration – but this is only for 60 migration events
  - If we are only hitting 60% useful time (i.e. migration is taking 40% of the time spent) then we are unlikely to benefit from this approach
  - Migration needs to be below 25% of the time spent for this approach to guarantee performance benefits
  - The numbers are likely to change a lot on hardware

The Architecture for the Digital World®
ARM®

Hyopthesis 3 – TRUE BUT!

- Latency might be a problem, need to evaluate on hardware

- On the plus side:
  - 40% of this latency comes from changing the cgroup of the task which could perhaps be improved in the kernel
  - We could make changes to ActivityManager to give better indications about the threads owned by the foreground application
  - Also do not assume that the scheduler would have similar latencies if it were making the decisions – no reason for that to be true – this code path goes right through sysfs and cgroups before it gets anywhere near scheduling

The Architecture for the Digital World®
ARM®

Try to Minimise Migration Latency

- An idea I had:
  - When the system is not heavily loaded, we don't need to use the big cores – the little cores are still quite powerful in their own right.
- To test this I hacked up a solution involving CPUFreq
  - Modify the sysfs code to allow /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq to be polled
  - Write a dummy CPUFreq driver for my platform with 2 operating points
  - Install the ondemand governor so that the cpu would 'switch' between these two dummy operating points
  - Monitor current frequency of CPU0 in sysfs
    - On high frequency, set affinity of fg_boost group to big CPUs
    - On low frequency, set affinity of fg_boost group to little CPUs
  - Leave Android assigning threads to fg_boost when the priority is high enough

The Architecture for the Digital World®
ARM®

Results pending

- Initial investigation shows that ondemand governor keeps the cpu frequency at the highest level for pretty much all the time we are interested in
- fg_boost group is consequently on big CPUs all the time
- Some tweaking of thresholds might be necessary
- I suspect latency of ondemand governor + userside-polling + user-side cpuset affinity changes will be relatively large
  - Want to evaluate exactly what it does cost on real hardware

The Architecture for the Digital World®
ARM®

An alternative

- Watch for specific applications starting and force their thread affinity using cgroups

- Looks possible but this is in the very early stages
  - Probably need a few more alterations in activity manager etc.
  - And remove my current ones ☺
  - Easy to identify PIDs belonging to specific Android apps from proc fs and Android's data directories

The Architecture for the Digital World®
ARM®

Recap

- It's easy to control the placement and balancing of threads when you know which thread is which
  - You have to know your CPU layout to set up the cgroups
- If you want to do it dynamically, Android already does a little of it
  - Only 'UI' threads are currently picked out and identifiable
  - Need to make more invasive changes to increase to include more of the 'current application' threads
- Latency is relatively high, so might be an issue
  - Model is simulating 400MHz, so the 2.4ms average latency might be closer to 0.9ms
  - May be ok especially if 'boosted' threads are boosted for longer

The Architecture for the Digital World®
ARM®

Q&A

- Questions and comments gratefully received!

- Next steps..

The Architecture for the Digital World®
ARM®

Next Steps 1

- Evaluate priority-based cluster migration on hardware
  - Latency will be important
  - Most threads belonging to an application don't reach sufficiently high priority – but are they important enough from performance point of view?

- Evaluate real impact of cgroup change latency on hardware
  - Model only gives an indication that there is latency which needs to be investigated
  - In the model, latency is split 40/60 between group change latency and thread reschedule latency

The Architecture for the Digital World®
ARM®

Next Steps 2

- Attempt to change Android's runtime cgroup task management so that the priority boosting applied to tasks is not so short-lived
  - Make changes in ActivityManager to more clearly indicate which application is in the foreground
  - Introduce a new scheduler policy to Android instead of hanging off priorities. All threads belonging to 'foreground' application would have 'fg_app' scheduler policy independent of short-term priority boosting.
  - Threads would be placed in this policy when application comes to front and removed when application is not on screen.

The Architecture for the Digital World®
ARM®

Next Steps 3

- Try out ‘Semi-Dynamic high priority applications’
  - A user-side layer to manage specific applications

- For a specific application want to designate as important:
  - Identify app UID from /data/data/<app name>
    - com.android.browser is app_11 on Linaro Android
  - Monitor /proc folder periodically to look for processed owned by the right UID
  - When we find it..
    - List contents of /proc/<pid>/tasks
    - Place threads in ‘big’ group
- Would need to modify the priority-based group assignment so it didn’t clash

The Architecture for the Digital World®
ARM®