Utilization-aware Load Balancing for the Energy Efficient Operation of the big.LITTLE Processor

Myungsun Kim*†, Kibeom Kim†, James R. Geraci†, Seongsoo Hong*
*School of Electrical and Computer Engineering, Seoul National University, Seoul, Korea
†DMC R&D Samsung Electronics, Suwon, Korea
*{mskim, sshong}@redwood.snu.ac.kr, †{kb1020.kim, james.geraci}@samsung.com

***Abstract*—ARM's big.LITTLE architecture introduces the opportunity to optimize power consumption by selecting the core type most suitable for a level of processing demand. To take advantage of this new axis of optimization, we introduce processor utilization into the Linux kernel's load balancing algorithm. Our method improves the Linux kernel's ability to schedule tasks in an energy efficient manner without making it directly aware of the available core types. Experimental results show an energy consumption improvement over the standard Linux scheduler up to 11.35% with almost no reduction in performance.**

I. Introduction

ARM's big.LITTLE is a single-ISA heterogeneous multi-core processor that offers an architectural approach to provide high performance when needed and low power consumption when there is no demand for high performance computation [1]. Unlike previous multi-core processors, it contains two different kinds of cores: one designed for performance and the other designed for energy efficiency. Energy efficiency is achieved when running non-performance demanding applications on the energy efficient cores.

Unfortunately, the present Linux kernel with big.LITTLE architecture support neither distinguishes between the core types nor considers how heavily a core is being used when assigning tasks. To improve the Linux kernel's ability to assign tasks to cores in an energy efficient manner without having to make it directly aware of the available core types, we introduce a utilization-aware load balancing mechanism. This allows our system to minimize the ramping up of the host processor's frequency and helps increase the usage of the energy efficient cores on the big.LITTLE processor. Our experimental evaluation produced a maximum measured energy consumption reduction of 11.35% with minimal loss of performance.

Single-ISA heterogeneous multi-core processors are an active area of research [2]. Particularly, a number of researchers have looked at optimal ways of assigning tasks to these kinds of processors [3], [4]. Much research has focused on extracting the maximum computational performance with energy efficiency being a secondary benefit [5]–[7]. Recently, the focus has shifted to energy efficient performance.

Cochran et al. proposed a control technique to find the optimal DVFS and thread allocations for multi-core processors [8]. Their strategy is to assign groups of threads to the cores with minimum power consumption. They also utilize DVFS control to stay within a power budget. Li et al. proposed a scheme to avoid frequency over-provisioning [9].

Unlike the above mentioned approaches, we attempt to improve energy efficiency via a kernel mechanism of Linux. Our insight comes from the fact that the current Linux scheduler performs load balancing irrespective of core utilization. The DVFS module [10], however, uses processor utilization to adjust frequency and determines the operating core. This lack of communication between these two Linux kernel components can result in a suboptimal utilization of system resources and an over utilization of energy.

978-3-9815370-2-4/DATE14/©2014 EDAA

II. *big*.LITTLE architecture and Linux kernel

ARM's big.LITTLE processor contains two different types of processor cores: Cortex-A15 and Cortex-A7. Processors of the same kind are grouped together into clusters. Data can be shared between the clusters via the CCI-400 interconnect [11].

There are two major different operating modes for a big.LITTLE processor: multi-processing (MP) mode and switcher mode [12]. In MP mode, all eight cores are run at the same time. This maximizes the available compute resources. In switcher mode, each Cortex-A15 core is paired with a Cortex-A7 core. Only one core per A15/A7 pair can be running at any given time.

A. virtual CPUs and load balancing

Each cross-cluster A15/A7 pair appears as a single virtual CPU to the Linux kernel [12]. The four virtual CPUs make a set $S = \{\text{vCPU}_0, \text{vCPU}_1, \text{vCPU}_2, \text{vCPU}_3\}$.

The Linux kernel scheduler moves tasks among the virtual CPUs in S, assigns new tasks to them and moves tasks from the system-wide wait queue to the run queues of virtual CPUs. It uses the load balancing rules of Complete Fair Scheduling (CFS), which decides the target virtual CPU without regard to the physical core that backs the virtual CPU [13].

The system's wait queue contains tasks that have been assigned to a virtual CPU, partially executed, and have entered the wait state for some reason. Run queue $\text{Q}_{run}(s)$ of virtual CPU $s \in S$ is a set of tasks and is used to hold tasks that are scheduled to run on s.

CFS uses the loads on the virtual CPUs for assigning and moving tasks. It compares the loads and tries to equalize them. The load on virtual CPU $s \in S$ is defined by

$$\text{load}(s) = \sum_{\tau_i \in \text{Q}_{run}(s)} W(\tau_i) \tag{1}$$

where $W(\tau_i)$ is the weight of task τ_i [13].

B. Linux kernel 'switcher' and core utilization

To take advantage of the two different core types that can back each virtual CPU in switcher mode, a switcher was

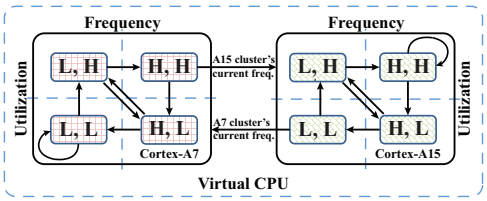

Fig. 1. A two-state DVFS state transition diagram for a big.LITTLE virtual CPU running in CPU switcher mode.

TABLE I. STATE MEANINGS FOR FIG. 1

	Low (L)	High (H)
Frequency	< max core frequency	= max core frequency
Utilization	< down-threshold	> up-threshold

introduced into the Linux kernel [12]. The Linux kernel uses the switcher to decide which physical core to actually run the virtual CPU's tasks on. It also decides when to move the tasks between the cores in the pair.

The switcher has two distinct operating modes: cluster mode and CPU mode. In cluster mode, all virtual CPUs must be backed by the same physical core type. Even if one virtual CPU requires its Cortex-A15, all the others will be backed by their A15. Whereas in CPU mode, each virtual CPU can be backed by either of the physical cores in its pair without regard to how the others are operating. Because CPU mode turns off the Cortex-A15 when it is not in use, it represents an opportunity for energy efficient usage of the architecture. The switcher uses core frequency and utilization to decide which physical core in a pair to back the virtual CPU with.

The governor adjusts the core's frequency in response to utilization. Core utilization is $U = 100 * (Period - IdleTime)/Period$. *Period* is the amount of time between the adjacent governor epochs [10]. If, over the one *Period*, the governor observes a level of processor utilization that exceeds its up-threshold, 80% in the present big.LITTLE scheduler, it increases the core frequency to its maximum value. If the core utilization drops below the down-threshold, 70% presently, the governor will incrementally decrease the core frequency in an attempt to reduce power consumption. We exclusively use the switcher in CPU mode and the ONDEMAND governor.

C. Bi-level DVFS model in CPU switcher mode

To illustrate DVFS in the big.LITTLE context, we consider the following simplified bi-level DVFS system for the big.LITTLE architecture. In this system, there are only two presented frequency states {low, high}, or {L, H} for short. There are two core utilization states again {L, H}. Thus, each processor in an A15/A7 pair can be in one of four states which we represent with the notation (frequency, utilization) and enumerate as: {(L, L), (L, H), (H, L), (H, H)}. Arranging the potential states for each core in a square with {L, H} frequency on one axis and {L, H} utilization on the other leads to the state transition diagram seen in Fig. 1.

In practice, there are often many frequencies and utilization is measured as a percent, so meanings for the frequency and utilization L and H states are listed in Tab. I.

Fig. 1 shows that only when an A7 is in (H, H) does it switch over to the A15. The newly activated A15's frequency is set to the present frequency of the A15 cluster. The utilization of the newly activated A15 will be a function of the A15 cluster frequency and the tasks that it is required to run.

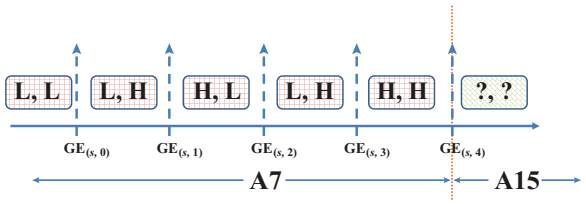

Fig. 2. Example movement from a Cortex-A7 to a Cortex-A15 at $GE_{(s,4)}$.

Only when an A15 is in (L, L) does the switcher migrate all tasks to the A7. The startup state of the newly activated A7 is a function of the A7 cluster frequency and the task mix in its run queue.

Fig. 2 illustrates how the state transitions might occur with time on virtual CPU $s \in S$. Each governor arrival epoch is marked with a vertical dashed arrow and denoted by $GE_{(s,x)}$ where x is the governor arrival epoch number. At each governor epoch, the governor looks at the processor utilization between it and the previous governor epoch. It then decides whether or not to adjust the clock frequency or to migrate across processor types.

For example, between $GE_{(s,3)}$ and $GE_{(s,4)}$, both clock frequency and utilization are high, so at $GE_{(s,4)}$, the core changes to the A15. The state that it transitions into must be determined at runtime, so it is marked with double question marks (?, ?). The assigned frequency will be the present A15 cluster's frequency, or if there isn't a live A15, the lowest DFVS policy allowed A15 frequency.

III. MOTIVATION AND PROPOSED SOLUTION

The present Linux scheduler is CPU utilization agnostic; therefore, it runs the chance of running a task unnecessarily on a high-frequency core, unnecessarily increasing the core frequency or causing unneeded A7 to A15 transitions. These all cause the processor to consume more energy than necessary.

For example, consider two virtual CPUs, s_1, $s_2 \in S$, each backed by their A7 cores and both in the (L, L) state as of the last governor arrival epoch. Also, $\text{load}(s_1) < \text{load}(s_2)$ and $U(s_1) > U(s_2)$.

To the present Linux scheduler, the heavily utilized s_1 core appears to be the better system to assign a new high priority task to because its load is lower than that of s_2. However, assignment to s_1 has a higher chance transitioning into the (L, H) state and causing the measured utilization at the next governor epoch to be H. This would cause the governor to increase the clock frequency during the subsequent epoch interval and thereby increase energy usage.

To address this issue, we present our utilization aware load balancing algorithm. It makes use of either of two estimators. For each virtual CPU, these estimators develop an estimate based on the assumption that the task to be assigned will be moved from the wait queue to that virtual CPU's run queue.

A. Utilization-aware load balancing

Our algorithm, specified in Fig. 3, attempts to balance tasks on the run queues of the virtual CPUs and operates when inserting new tasks or moving tasks off the wait queue. It has two steps. First, as seen in Lines 5-6 of Fig. 3, it moves tasks among the cores to level out their loads. Our method

```
1: IPC = 1.25
2: Task_ν ← DEQUEUE(Q_wait)
3: CPU_idlest ← arg_{s∈S} min load(s)
4: CPU_busiest ← arg_{s∈S} max load(s)
5: while load(CPU_busiest) > load(CPU_idlest) ∗ IPC do
6:     Task_popped ← DEQUEUE(Q_run(CPU_busiest))
7:     ENQUEUE(Q_run(CPU_idlest), Task_popped)
8: end while
9: CPU_util ← arg_{s∈S} min E[U(s, Task_ν)]
10: ENQUEUE(Q_run(CPU_util), Task_ν)
```

Fig. 3. Pseudo code for utilization based load balancing. Inserting a task from the wait queue is based on expected utilization as in Line 9.

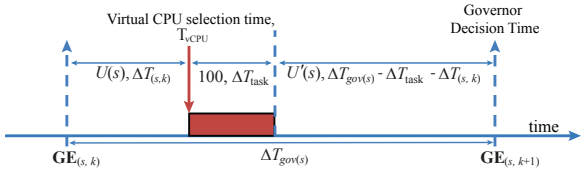

Fig. 4. Important times between interarrival epochs $\text{GE}_{(s,k)}$ and $\text{GE}_{(s,k+1)}$.

removes the Linux kernel scheduler's active and event driven load balancing, so this is the only time our system attempts to load balance the virtual CPUs.

The second step is to insert a new task into the run queue of the virtual CPU that has the lowest expected utilization at the next governor epoch after potential insertion, $E[U\,(s, \text{Task}_\nu)]$. Fig. 4 depicts the major times utilized in forming the estimates. In Fig. 4, $\Delta T_{(s,k)}$ denotes the elapsed time from the k^{th} governor epoch $\text{GE}_{(s,k)}$ to the virtual CPU selection time, T_{vCPU}, on virtual CPU s. T_{vCPU} is an absolute time and occurs simultaneously across all virtual CPUs. $\text{GE}_{(i,k)}$ is not necessarily equal to $\text{GE}_{(j,k)}$ for $i \neq j$, $\Delta T_{(i,k)}$ does not necessarily equal $\Delta T_{(j,k)}$ for $i \neq j$. $U(s)$ is the measured utilization of virtual CPU s over the interval $\left[\text{GE}_{(s,k)}, \text{T}_{\text{vCPU}}\right]$. $\Delta T_{gov(s)}$ is the time between governor epochs and $\Delta T_{gov(i)} \approx \Delta T_{gov(j)}, \forall i, j$. $U'(s)$ is the unknown utilization after task insertion.

B. Utilization based estimator

For our first estimate of utilization, $E_A\,[U\,(s, \text{Task}_\nu)]$, we assume that, in the second half of the governor period, the core will continue to be utilized in a manner similar to the way it had been utilized before the decision point and independent of whether or not the task is inserted into the core's run queue. That is, in Fig. 4, $U'(s) = U(s)$. We also assume, $\Delta T_{\text{task}} \ll \Delta T_{gov(s)}$, which leads to (2).

$$E_A\,[U(s, \text{Task}_\nu)] = U\,(s) \times \frac{\Delta T_{gov(s)} - \Delta T_{\text{task}}}{\Delta T_{gov(s)}} + \frac{100 \times \Delta T_{\text{task}}}{\Delta T_{gov(s)}} \approx U\,(s) \quad (2)$$

C. Heuristic augmented utilization based estimator

Our second estimate of utilization, $E_B\,[U\,(s, \text{Task}_\nu)]$, uses an estimate of ΔT_{task} to create a utilization over the period $\left[\text{GE}_{(s,k)}, \text{GE}_{(s,k)} + \Delta T_{(s,k)} + \Delta T_{\text{task}}\right]$. We again assume that $U'(s)$ matches the utilization in the first half of the period. However, this time, the first half is defined as the aforementioned interval and thus includes the time required by the task.

The new task's required time is estimated by taking the average of the actual previous sixteen task execution times. We call this estimate ΔT_{sma} where 'sma' stands for 'simple moving average' and we let $\Delta T_{\text{task}} = \Delta T_{\text{sma}}$. Since these

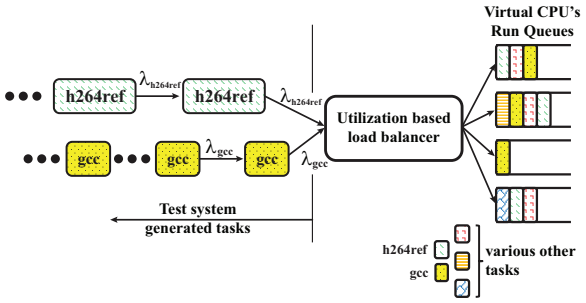

Fig. 5. Our test setup feeds h264ref SPEC benchmark and gcc SPEC benchmark tasks into our load balancer and then onto the various cores.

previous 16 runs can happen all on the same core or on a variety of cores, ΔT_{sma} is core independent, giving (3).

$$\begin{aligned} E_B[U(s, \text{Task}_\nu)] &= U'(s) \\ &= U(s) \times \frac{\Delta T_{(s,k)}}{\Delta T_{(s,k)} + \Delta T_{\text{sma}}} + \frac{100 \times \Delta T_{\text{sma}}}{\Delta T_{(s,k)} + \Delta T_{\text{sma}}} \end{aligned} \quad (3)$$

IV. Experiments

We employed two different types of experiments. First, we used a controlled laboratory benchmark test suite where we controlled the task birth-death rates. Second, we used software-decoded video playback to provide a real world demonstration of the QoS and energy efficiency of our method. An Exynos based Galaxy S4 was our testbed. It ran Linux kernel 3.4.0 and also our custom version of the same.

A. Benchmark test experiment and results

We used two CPU intensive benchmarks from the SPEC CPU 2006: h264ref and gcc [14]. Unlike typical Android applications, they rarely enter a sleep state, so once started, there is only a minimal chance they will enter the wait queue. While in mobile systems, to prevent any one application from dominating the usage of any particular virtual CPU, tasks are often moved from the virtual CPUs to the wait queue.

In order to test our load balancer, we developed the test setup shown in Fig. 5. This test system allows for the controlled generation of h264ref SPEC benchmark and gcc SPEC benchmark tasks. For h264ref, the number of encoding frames was set to two. For gcc, the file size to compile was fixed at 3.4KB. h264ref tasks arrived at the load balancer deterministically at a rate of $\lambda_{\text{h264ref}} = 1/3$ arrivals/second.

Once an h264ref task was assigned to a virtual CPU by the load balancer, it ran to completion and exited from the system. However, the task leaving the system was soon replaced by an identical task entering the system. Thus, we simulated a task moving from one of the run queues onto a virtual CPU, then into the wait queue and back onto one of the run queues.

λ_{gcc} was fixed over the course of an experiment and tasks arrived deterministically. λ_{gcc} was incremented between experiments by $10\frac{\text{arrivals}}{\text{second}}$ over the range of $[20, 80]\ \frac{\text{arrivals}}{\text{second}}$.

The average measured core-independent gcc service rate per virtual CPU, μ_{gcc}, was 14.3 gcc tasks/second or around 70ms/task. The system, therefore, can handle around 57.2 gcc tasks/second. The arrival rates of 50 arrivals/second or fewer represented an operating regime in which tasks were being recycled around the wait→run→wait loop with a finite

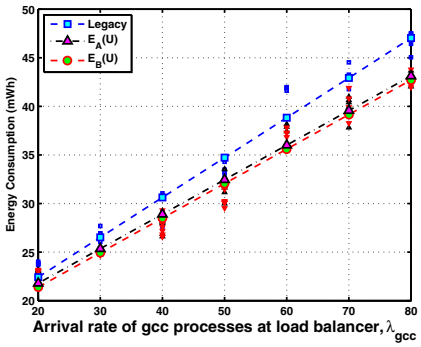

Fig. 6. Our estimators achieve lower absolute energy consumption and improve relative to the legacy load balancer as system activity increases.

TABLE II. BENCHMARK TEST EXPERIMENT RESULTS

λ_{gcc}	Data Kind	Legacy	$E_A(U)$	$\Delta_A\%$	$E_B(U)$	$\Delta_B\%$
20	Time (s)	36.40	36.44	**0.12%**	36.67	**0.75%**
	Current (mA)	591.87	577.74		570.06	
	Power (mW)	2359.81	2303.63		2272.79	
	Energy (mWh)	23.86	23.31	**-2.29%**	23.15	**-2.96%**
30	Time (s)	38.65	39.66	**2.62%**	39.84	**3.09%**
	Current (mA)	625.26	573.90		563.04	
	Power (mW)	2493.04	2288.24		2244.89	
	Energy (mWh)	26.77	25.21	**-5.83%**	24.84	**-7.18%**
60	Time (s)	50.88	52.84	**3.84%**	52.54	**3.25%**
	Current (mA)	743.24	636.81		638.29	
	Power (mW)	2963.27	2539.00		2544.80	
	Energy (mWh)	41.88	37.26	**-11.04%**	37.13	**-11.35%**

wait time. The 60 arrivals/second or greater rates represented tasks being both recycled and a steady stream of new tasks was being introduced into the system.

The energy consumption results for all of the collected data are shown in Fig. 6 where we have drawn regression lines for each of the systems through their respective data sets. The plot clearly shows that both of our estimators achieve better energy consumption performance than the legacy Linux system. Our estimators' performance improves relative to the legacy Linux system with increased system utilization.

Results for representative arrival rates are displayed in Table II. The columns are as follows. 'λ_{gcc}' is the gcc inter-task arrival rate. 'Data Kind' is the type of data collected. 'Legacy' is for the result for the unmodified Linux Kernel. '$E_A(U)$' is the result from our load balancing algorithm employing the utilization estimator from Section III-B, '$\Delta_A\%$' the percent gain or loss between the legacy system and our new system with utilization estimator from Section III-B. '$E_B(U)$' holds results measured when using Section III-C's estimator, and '$\Delta_B\%$' is the percent change with the legacy system.

For all λ_{gcc}, both estimators show a reduction in current draw, power and energy usage. They also exhibit slight reductions in performance as measured by the increased run times.

B. Real world video decoding and results

We utilized the Android MX player in its software decoding mode. It ran 21 subtasks that moved around the system, between cores and between the wait queue and run queues, so as to fully demonstrate the efficacy of our mechanism under real world conditions. It decoded a 31 second long video clip of 1920x1080 full HD frames that had been compressed with the h.264 codec and streamed from the device's internal storage.

All three systems required two seconds of playback to achieve full playback frame rate and showed a tapering of performance in the last two seconds. These four seconds are excluded from the mean and standard deviation in Table III but are included in the other statistics. Per Table III, each of our estimators showed significant energy consumption reduction when compared to the legacy CPU mode Linux system. Furthermore, the $E_B(U)$ estimator had no drop in average frame rate and only a slight increase in frame rate instability.

TABLE III. SOFTWARE VIDEO DECODER RESULTS

	Legacy	$E_A(U)$	$E_B(U)$
Switcher	CPU	CPU	CPU
Time (s)	31.53	31.39	31.43
Current (mA)	690.78	650.46	644.28
Power (mW)	2754.34	2593.50	2568.94
Energy (mWh)	24.12	22.61	22.43
Energy $\Delta\%$	-	6.68%	7.53 %
Mean FPS	30.0	29.964	30.0
σ_{FPS}	0	0.429	0.385

V. CONCLUSION

We introduced a task and core agnostic utilization-aware load balancing method into the Linux kernel's big.LITTLE port. Tests showed our method's energy performance improvement over the present big.LITLLE architecture's Linux scheduler. Our method's energy efficiency and compute performance depend on the utilization estimator, so better estimators could produce greater gains. Complexity of any estimator, however, must be simple enough so as not to burden the scheduler with its compute time. This trade-off between estimator complexity and estimator performance make further research into utilization estimators a promising area of future work.

REFERENCES

[1] "Advances in big.little technology for power and energy savings." [Online]. Available: http://www.thinkbiglittle.com/

[2] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[3] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *High Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[4] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "Hass: a scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, Apr. 2009.

[5] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Computer Architecture, 2012 39th Annual International Symposium on*, pp. 213–224.

[6] L. Sawalha, S. Wolff, M. Tull, and R. Barnes, "Phase-guided scheduling on single-isa heterogeneous multicore processors," in *Digital System Design, 2011 14th Euromicro Conference on*, 2011, pp. 736–745.

[7] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *EuroSys*, 2010, pp. 139–152.

[8] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: adaptive dvfs and thread packing under power caps," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 175–185.

[9] X. Li, G. Yan, Y. Han, and X. Li, "Smartcap: User experience-oriented power adaptation for smartphone's application processor," in *Design, Automation Test in Europe Conference Exhibition*, 2013, pp. 57–60.

[10] "Cpu frequency and voltage scaling code in the linux(tm) kernel." [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[11] "Corelink cci-400 cache coherent interconnect." [Online]. Available: http://www.arm.com/products/system-ip/interconnect/corelink-cci-400.php

[12] M. Poirier, "In kernel switcher." [Online]. Available: http://events.linuxfoundation.org/images/stories/slides/elc2013_poirier.pdf

[13] "This is the cfs scheduler," May 2007. [Online]. Available: http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt

[14] "Spec's benchmarks and published results." [Online]. Available: http://www.spec.org/benchmarks.html