BrezeFlow: Unified Debugger for Android CPU Power Governors and Schedulers on Edge Devices

⋆Alexander Hoffman, †Anuj Pathania, ⋆Philipp H. Kindt, +Samarjit Chakraborty, †Tulika Mitra
⋆*Chair of Real-Time Computer Systems, Technical University of Munich*
†*School of Computing, National University of Singapore*
+*Department of Computer Science, University of North Carolina at Chapel Hill*
Email: alex.hoffman@tum.de

***Abstract*—Power management is quintessential to the successful deployment of edge devices, such as smartphones, in power-, thermal-, and energy-constrained environments. Governors and schedulers operate system sub-routines for power management at the edge. There exist several tools for debugging power issues in *Android* applications. However, there exists no tool to identify and classify inevitable misdecisions by power managers, given their often inefficient underlying heuristics. In this work, we introduce the first tool – *BrezeFlow* – designed for unified (scheduling and frequency scaling) power debugging of CPU power managers on *Android* edge devices. *BrezeFlow* enables kernel developers to evaluate designs of their power managers retrospectively with closed-source applications in real-world scenarios based on any user-defined strategy and thereby gain insights for better future governor designs. *BrezeFlow* detected an average of 815 misdecisions per second for the commonly deployed duo, *ondemand* governor and *Completely Fair Scheduler*, on mobile edge devices running popular applications.**

I. Introduction

Modern *Android*-based edge devices, such as smartphones, are the amalgamation of heterogeneous multi-core hardware architectures, hardware management logic, and multi-threaded applications. While the computational capacity of CPUs on mobile devices has been growing exponentially, the batteries supplying these devices have not been able to keep up. Furthermore, dissipating power (heat) from edge devices has become a limiting factor [1]. Therefore, effective on-chip power management techniques are crucial in achieving respectable battery life and safe operating temperatures for edge devices, such as smartphones, vehicle entertainment systems, etc.

Many state-of-the-art edge devices employ heterogeneous multi-core CPUs built using the *ARM big.LITTLE* architecture, which incorporates two heterogeneous CPU clusters – a *Big* cluster with high-performance, high-power cores and a *Little* cluster with low-performance, low-power cores. The CPUs share the same instruction set architecture (ISA) but significantly different micro-architectures. Therefore, they can run threads from the same application using heterogeneous multi-processing (HMP) but with very contrasting power-performance thread execution characteristics.

The majority of mobile devices now run the *Android* operating system (OS) on top of device-customized *Linux* based kernels. Governors are the OS subroutines primarily responsible for managing the power for the CPUs. Governors, in heterogeneous multi-cores, are responsible for scaling the operational voltages and frequencies of both clusters using dynamic voltage and frequency scaling (DVFS), and the power-gating of clusters. Optionally, they can also influence the allocation of threads on to individual CPU cores.

Load balancing, through appropriate mapping of threads to heterogeneous CPU cores, also plays a role in deciding efficient DVFS configurations. In the default design, schedulers are the OS sub-routines responsible for thread management (load balancing) that operate independently from governors, and vice versa. A lack of synergy between a governor, managing only DVFS, and a scheduler, performing load-balancing, often leads to significantly inefficient power management [2]. Furthermore, the standard *Linux* kernel also employs simple heuristics within governors and schedulers, which are generally not optimized, especially when considered from the perspective of a specific application. Power management subroutines originating in the research community seek to integrate the role of governor and scheduler into one unified power manager, as this has proven effective [2], [3].

Determining highly efficient power management decisions at run-time requires perfect future workload predictions, but making such predictions are usually very difficult at run-time [4]. Therefore, misdecisions by power managers usually go unnoticed. This ignorance is even more apparent given that the metric commonly used in quantifying a power management strategy's efficacy is usually the gains over an antecedent strategy. There are preexisting tools, such as [5], to debug and optimize the power consumption of *Android* applications. However, there exists, to best of our knowledge, no tool which can automatically identify and classify misdecisions of power managers, such as the inefficient setting of a CPU's voltage and frequency, or the reallocation of a thread to decrease the maximum core-wise load of a CPU, in turn potentially allowing for more frequency reductions.

In this paper, we propose a power profiling tool – *BrezeFlow* – that can be used to quantify an application's execution in terms of the systems power management state against a set of user-defined misdecision classification strategies. This classification enables kernel developers to quantitatively evaluate the performance of their power managers. Misdecisions are identified retroactively with prescient knowledge and the help of a user-defined misdecision classification strategy.

BrezeFlow operates by extracting system performance and power information, then annotating interaction graphs from the execution of closed-source *Android* applications. Constituent tasks from an application represent the vertices in the interaction graph. A task slice is the intermittent execution of a single process from waking up until returning to a sleeping state. Edges represent inter-process communications (IPCs) between the tasks. Existing approaches derive similar graphs from the source code or the corresponding binary code [5]. In *Android*, significant portions of an application's functionality execute using system services. Therefore, neither the source code nor the binary code is useful for mining properties, such as the projected workload, required for debugging power managers. Furthermore, several task properties (such as the execution time) are subject to online power management decisions. Therefore, the necessary information can only be extracted from the active execution of a target application.

All user- and system-space tasks in *Android* communicate with each other exclusively using the *Binder* framework, through *Binder* transactions. Moreover, *Binder* transactions are always event-based. A task attempting to receive a *Binder* transaction gets blocked until the occurrence of the appropriate *Binder* event. Since all IPCs (including remote procedure calls (RPCs) and task synchronization) are realized using *Binder*, tracing these transactions will uncover the dependencies of the execution flow among different tasks. *BrezeFlow* exploits this to achieve an understanding of the inter-task dependencies, their execution/sleep cycles, and their communication behaviors. As a result, all information that is necessary for inferring power management decisions, in retrospect, is captured by jointly monitoring the scheduler, device drivers, and *Binder*.

BrezeFlow then compares the actual power management decisions taken to decisions that would have been in accordance with an alternate power management strategy provided by the user. This comparison is done in a time horizon that encompasses the target task and the tasks directly impacting it/being impacted by it. It flags a decision as a misdecision whenever a system configuration is available that would result in lower instantaneous energy consumption with the clairvoyant user-defined strategy. Governor misdecisions are the result of incorrect DVFS configurations or power-gating. Scheduler misdecisions result from incorrect load balancing. *BrezeFlow* can classify a misdecision as both a governor and a scheduler misdecision simultaneously. In the end, *BrezeFlow* generates a report which summarizes all the misdecisions for the user to analyze and derive insights. The insights provide a foundation for the development of more optimized power managers in the future.

Our Contributions: In this paper, we make the following contributions.

1) We propose the first tool – *BrezeFlow*– for the automatic extraction of interaction graphs on *Android* devices. *BrezeFlow* annotates the interaction graphs with power management misdecision information for easy visual analysis by a user.
2) *BrezeFlow* uses the interaction graph to debug the decisions of the governor, scheduler, or both. It classifies their decisions as misdecisions if they are found to be inefficient as per a clairvoyant user-defined strategy. Classification provides insights into shortcomings in existing designs for superior future designs.
3) We test and evaluate *BrezeFlow* on the popular *Exynos 5422* multiprocessor system-on-Chip (MPSoC) with *big.Little* heterogeneous multi-core CPU using the default *Android* governors and schedulers, as well as, with state-of-the-art unified *GameOptimized* governor from the research domain [2].

Open-Source Contributions: Source-code for *BrezeFlow* is available on *GitHub* [6].

II. Tracing Tasks and IPC in Android

In this section, we provide the background knowledge necessary for understanding the operations of *BrezeFlow*.

Binder Framework: Unlike in standard *Linux*, *Android* replaces all methods for IPC (such as semaphores, sockets, pipes, etc.) with the *Binder* framework in all application software. *Binder* is responsible for the handling of two-way communications, notifications, and inter-thread management signaling, both in the kernel- and user-space. It allows for the transmission of data, blocking and unblocking of tasks, RPC, and task synchronization. *Binder* employs a client-server communication model. A client task initiates communication by sending a `call` transaction via the *Binder* kernel driver. The client task is then put into a sleep state until a response is returned, leading to synchronous communication and task execution sequences. Compared to most other OSs, *Android* offloads a large portion of an application's workload to system services. As a result, inter-task dependencies in *Android* are relativity complex and are therefore of central importance for power management. Since *Binder* transactions control the sleep cycles of applications and most system services, tracing the scheduler and *Binder* uncovers the inter-task execution flow. *BrezeFlow* tightly integrates with the *Binder* framework.

***Android* Kernel Tracing Frameworks:** Since the *Android* kernel is a derivative of the *Linux* kernel, it comes with existing Linux tools for tracing kernel events, supporting tracepoints. Tracepoints export kernel information to user-space, with almost negligible overhead, when triggered. Some of the tracepoints are already built into the kernel source code by default for key events such as context switches and *Binder* transactions. Kernel developers can add additional tracepoints by modifying the source code of the kernel itself or as a loadable kernel module (LKM). *BrezeFlow* consists of an LKM and a user-space application. It builds on top of the well-known *ftrace* tracing framework wherein it uses custom and in-built tracepoints for gathering information relevant to the construction of interaction graphs. By utilizing the highly efficient *ftrace* framework, *BrezeFlow* ensures that the traces extracted during runtime have a negligible impact on the execution characteristics of the system under observation.

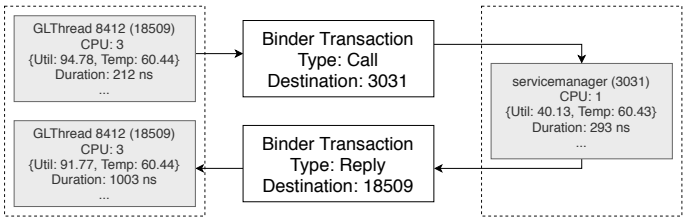

Fig. 1: An interaction graph showing an IPC between two tasks, working on behalf of the target application, providing insight into inter-task dependencies and execution flows.

III. Automated Interaction Graph Extraction

Online tracing of the *Android* kernel using *BrezeFlow* allows for accurate extraction of events from within the kernel with an almost negligible overhead as a result of using *ftrace*. *BrezeFlow* extracts task executions and their dependencies by tracing `sched_switch` and `binder_transaction` events. Core frequencies and other system metrics are extracted using custom tracepoints that leverage the devices' drivers to log runtime device metrics. *BrezeFlow* uses precise kernel timestamps to synchronize the events and the system metrics during off-line analysis. A kernel task periodically tracing the required non-event driven tracepoints, e.g., on-board power sensor values, allows for adjustment of performance (tracing overhead) vs. accuracy (reduced interval between measurements).

CPU events, such as `sched_switch` and `cpu_idle`, provide information on the context switches and utilization for an individual core. `sched_switch` events detail a context switch on the core wherein the process IDs (PIDs) of tasks that switches in and out. A task rarely executes on a CPU as a single block of work because of scheduling and interrupts. Since a task will leave the CPU with a state S (interruptible sleep) when its current execution cycle is completed, *BrezeFlow* successfully traces the complete task executions by tracing all task slices and then lacing them temporally. This trace provides timing constraints for the task that are known to guarantee the successful execution of the application.

As the *Binder* driver provides a tracepoint that details each `binder_transaction` event, traces show IPCs from a client task to a server task. Traces allow for the isolation of the application's task execution dependencies and the interactions of the tasks with the system services when analyzed off-line. This isolation is required when performing DVFS analysis as the reduction in frequency will increase execution times. Therefore, a valid frequency decrease must not violate the timing bounds of dependent tasks. *BrezeFlow* puts together all this information in the form of a human-readable interaction graph. Figure 1, for example, shows an interaction dependency between two tasks extracted using *BrezeFlow*.

IV. Debugging with BrezeFlow

BrezeFlow requires a debugging strategy with which to classify misdecisions, allowing for user-defined flexibility. This section provides an overview of the strategy we provide to show the functionality of *BrezeFlow* in this work. The strategy operates on prescient knowledge of the future for a limited time horizon. Provided with a more complex evaluation strategy, *BrezeFlow* can operate with an unbounded time horizon. However, even then, it does not know the best solution; and employs a heuristic that is expected to be highly efficient. Note that this strategy can be easily replaced by another in the tool.

Algorithm 1 Systematic analysis performed on each PID branch to evaluate the system's power configuration

```
1: procedure PROCESS GRAPH(InteractionGraph)
2:     for Task in Application PIDs do
3:         for Task in Tasks do
4:             if Task on big core then
5:                 cycles ← Task.cycles × migration_factor
6:                 for freq in f_min to f_max of little do
7:                     scaling_factor ← Task.little_freq ÷ freq
8:                     scaled_workloads ← little_workloads × scaling_factor
9:                     scaled_cycles ← cycles × scaling_factor
10:                    if scaled_workloads < 100 then
11:                        task_duration ← scaled_cycles ÷ CPU_cycles_per_sec
12:                        task_finish ← Task.start_time + task_duration
13:                        if task_finish < depender_start_time then
14:                            Fault Detected: Task can be moved to LITTLE
15:                            if freq ≠ Task.little_freq then
16:                                Fault Detected: DVFS possible w realloc
17:             if Task.core_freq ≠ min_core_freq then
18:                 if Lower max core util possible then
19:                     Fault Detected: Intra-cluster reallocation inefficient
20:                 for freq in f_min to f_max of current_core do
21:                     scaling_factor ← current_core_freq ÷ freq
22:                     scaled_workloads ← current_core_workloads × scaling_factor
23:                     if scaled_workloads < 100 then
24:                         Fault Detected: DVFS parameters can be reduced
```

It is a well-known fact for power management in *Android* devices that applications operate more power efficiently when they execute on cores close to their highest utilization at the lowest frequency necessary to sustain a given performance [7]. Therefore, *BrezeFlow* uses the core utilizations of the DVFS configuration set by the governor for a given system workload to quantify the efficacy of a governor. This quantification is done using an iterative time horizon that is bounded to each task slice and its directly dependent tasks' during evaluation.

An unbalanced load-balancing by a scheduler can force the governor to operate at a DVFS configuration with higher core frequencies than necessary to sustain the required performance. Furthermore, it is also a well-known fact in power management for edge devices that edge applications operate more power efficiently on *Little* cores than *Big* cores in a heterogeneous multi-core system [8]. Therefore, in the implemented evaluation strategy, *BrezeFlow* evaluates thread allocations in a heterogeneity-aware manner. It tries to enable further DVFS reductions through thread allocations that reduce peak core-wise utilizations.

Algorithm 1 provides the strategy within *BrezeFlow* used to extract potential sub-optimal system power management configurations. Through further analysis the identification of governor flaws can be simplified. *BrezeFlow*, by default, uses a migration factor (MF) model given by [8] to estimate the performance of a task slice on a *Little* core when originally executed on the *Big* core (Line 5). A utilization-frequency model given by [9] is used to estimate the performance of a task slice when executed at a lower frequency (Line 8). The integration of both models into *BrezeFlow* is modular, meaning

TABLE I: List of power managers used in evaluation.

Governor	*performance, powersave, ondemand, conservative, interactive*
Scheduler	*Completely Fair scheduler (CFS)*
Unified	*GameOptimized [2]*

TABLE II: List of applications used in evaluation.

Games	*Fruit Ninja, Candy Crush, Real Racing 3*
Benchmarks	*Seascape 3D*
Others	*Amazon* Shopping, *Chrome* Web Browser

they can be easily replaced with other system models (Line 9). Thus, the efficacy of *BrezeFlow* is sensitive to the accuracy of underlying models, such as [8] and [9].

The strategy within *BrezeFlow* flags a task execution on a *Big* core as a scheduler misdecision if migration to a *Little* core would have allowed the task to execute while still respecting the timing constraints of its dependent tasks (Line 14). For a reallocation to be valid, all task dependencies given by the edges in the interaction graph must remain valid in the alternate execution. Specifically, the task deadlines that are known from the traced context switches of the CPU. Thus, in the implemented strategy that does not cascade timing changes into future task execution. An execution is valid if the stretched execution duration of a task, due to a frequency change does not overlap dependent tasks' start times (Lines 13-16). It also flags an execution as a scheduler misdecision, if after a potential intra-cluster migration to reduce peak core-wise utilization (Line 19), the governor could have reduced the frequency of the cluster (Line 24). Thus, *BrezeFlow* flags an execution as a governor misdecision if the execution could have been run at a lower frequency without violating its extracted deadline. Similarly, if the governor fails to power down an empty cluster, this is also classified as a misdecision. Given user-defined metric thresholds, developers can thus more easily isolate situations where governor logic is failing.

V. Experimental Evaluation

Experimental Setup: We employ an *Odroid XU3* development board in this work. The board contains an *Exynos 5422* MPSoC with an *ARM big.Little* CPU running *Android* 7.1 and *Linux* kernel 3.10.9. We evaluated six different governors alongside the default Linux scheduler (CFS) on the *Odroid XU3* board with six different applications using *BrezeFlow*. Table I lists all governors and schedulers used in the evaluation. Besides evaluating all default governors and schedulers available on the *Odroid XU3*, we also evaluate a state-of-the-art power manager [2] that unifies the role of governor and scheduler. We employ a large and varying array of organic workloads by running a range of *Android* applications such as email, e-commerce, games, and benchmarks in our evaluation. We evaluate several scenarios within an application. Table II lists all of the applications used in the evaluation.

BrezeFlow produces a large quantity of data every second due to the tracing of all context switches on an eight-core CPU. Limiting the test duration to four seconds reduced the interaction graph processing time and thereby allowing for a greater quantity of tests to be performed. Nevertheless, *BrezeFlow* can potentially generate and process interaction graphs for much longer test runs.

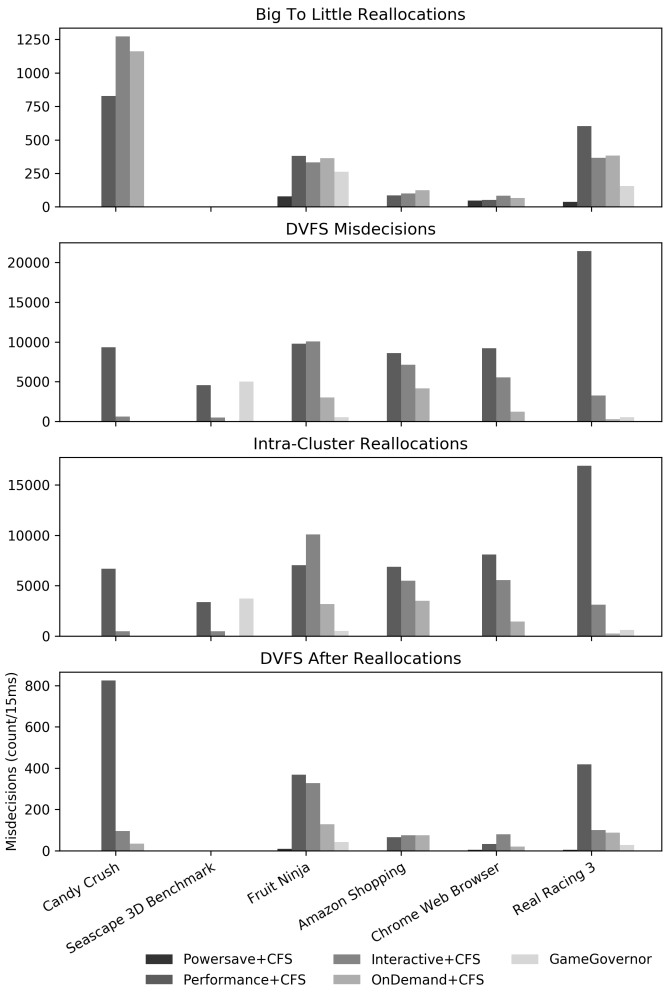

Fig. 2: Amount of various CPU governor and scheduler misdecisions observed when running various *Android* applications.

Results: Figure 2 shows the misdecisions recorded for each application and governor tested. Misdecisions are classified into inter-cluster reallocations, wrong cluster-wise DVFS, intra-cluster reallocations that reduce maximum core-wise utilization, and inter-cluster reallocations that would enable DVFS optimizations. The extremely large number of errors recorded for the *performance* governor was to be expected due to its complete lack of load balancing and its overexertion of the hardware. Similarly, *powersave* sets the core frequencies to their minimum without power gating the big CPU leading to the visible big-to-little reallocation misdecisions and no DVFS misdecisions due to the extremely restrictive management of the hardware. There is no room for any power optimizations, although the entire system configuration is inefficient from a quality of service (QoS) perspective under *powersave*.

The *interactive* governor was found to make more misdecisions than the *ondemand* governor for all situations because of its greater reluctance to decrease frequency after a frequency

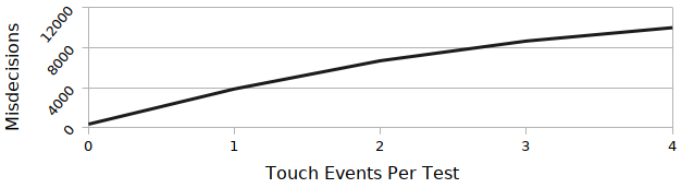

Fig. 3: Near linear increase between detected misdecisions of the *interactive* CPU governor and touch events when running *Chrome* web browser on *Android*.

Algorithm 2 *ondemand* Governor Algorithm

```
1:  procedure PROCESS GRAPH(InteractionGraph)
2:      for every CPU in the system do
3:          loop Every X milliseconds
4:              Get utilization since last check
5:              if Utilization > UP_THRESHOLD then
6:                  Increase frequency to MAX
7:          loop Every Y milliseconds
8:              Get utilization since last check
9:              if Utilization < DOWN_THRESHOLD then
10:                 Decrease frequency by 20%
```

ramp. Both *ondemand* and *interactive* governors' responses in the *Amazon* app, created large amounts of misdecisions due to the large workloads from the web traffic requests and system service communications generated as a result of the user's interaction with the application. Figure 3 highlights this by showing the number of screen scroll events and their impact on power management through the background workload spikes that induce frequency ramps. As the games tested do not produce rapidly changing workloads during playing, they exhibit significantly fewer misdecisions when compared to applications such as *Chrome* and *Amazon Shopping*.

The research *GameGovernor* [2] performed better than the standard governors by employing a bin packing first fit task allocation strategy. This strategy caused spikes in intra-cluster allocation misdecisions due to the overhead-aware task allocation performed by the algorithm. These spikes show that there is a significant number of inefficient decisions in short time horizons. Similarly, the workload prediction employed by the governor caused an increase in task-local DVFS misdecisions. Determining the overall efficiency of the governor for larger time horizons in consideration with task migration overheads would require further analysis.

Case Study: Previous results show that *BrezeFlow* can identify a significant number of misdecisions for various power managers. To provide more insight into a use case, we present a case study that was performed to debug the *ondemand* governor. Algorithm 2 shows the high-level algorithm behind the *ondemand* governor [10]. *ondemand* ramps up the cluster frequency to the maximum when a new workload triggers the `UP_THRESHOLD` (Line 5). The frequency will then take some multiple of a kernel-defined period Y to settle on the minimum required frequency. However, it takes some time for *ondemand* to return to an appropriate lower long-term frequency (Lines 7 - 10). Therefore, *ondemand* spends a significant duration of time above the minimum required frequency.

Analysis with *BrezeFlow* shows that *ondemand* performs especially poorly in the execution of applications with short-duration and bursty workload. Figure 4 shows three such

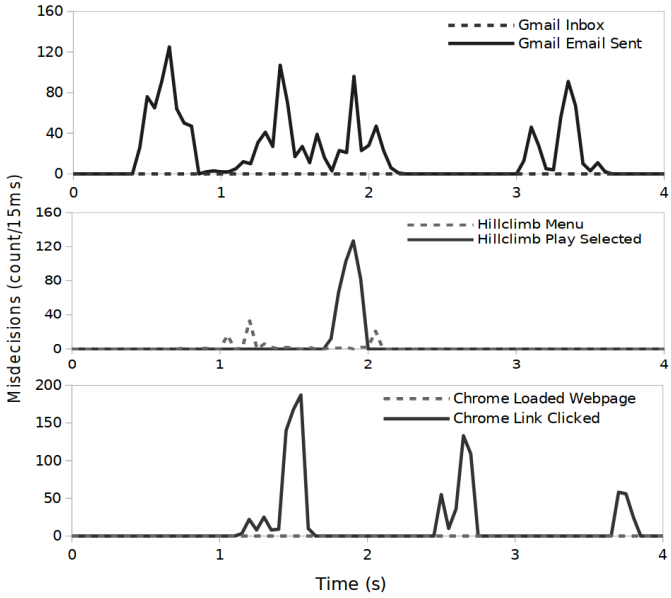

Fig. 4: Incurred power management misdecisions with interactive applications running on *Android* with *ondemand* governor.

workloads – the sending of an email, starting of a game, and clicking of a link in a web browser. For each misdecision, *BrezeFlow* produces a detailed description of the misdecision as well as a snapshot of the system's power state.

Analysis of the web browser's workload showed that the clicking of a link that blanks the web browser's internal window and creates background network workloads generated several misdecision spikes. Misdecisions were a reaction for each action causing the `UP_THRESHOLD` threshold value in Algorithm 2 to be reached. This reaction caused the frequency of the *Little* core to be ramped up to its maximum, causing large amounts of inefficient DVFS decisions as well as causing some tasks to be migrated to the *Big* cluster despite the *Little* cluster's capacity. This migration was a commonly observable characteristic of workload spikes. The spikes in misdecisions were also observed to decrease over time, which adheres to the recessive nature of the *ondemand* governor's algorithm.

Similarly, the misdecision spike shown in the *Hillclimb* graph was the result of a large load created between the game's event, *Mali*, *OpenGL*, and *SurfaceFlinger* system service threads. Similarly in the sending of an email, *BrezeFlow* showed that changing the displayed window, playing an "email sent" sound, and the network activity of sending the email incited several spikes due to interaction between *Google Mobile Service, Android Audio Server, OpenGL*, *Mali*, and *SurfaceFlinger* system service threads. The scheduling of this relatively low load application was handled well with even thread loads across the system's cores such that the system was able to spend most of the time between DVFS spikes at its minimum operating frequency.

VI. Related Work

Energy-efficient application development is dependent on tools that aid in the automation of power debugging. Authors of [11] were the first to develop an automatic test framework,

based on the evaluation of event traces for energy-consuming bugs, such as the improper management of network resources, background services, and wakelocks. Similarly, authors of [12] used the tracing of system calls to identify energy bugs. These works build upon the extraction and processing of *Android* activity graphs from the application's source code or binary. Activity graphs are perfect for debugging applications. However, they are too high-level to debug power managers due to their abstraction of system-space information.

Nevertheless, there exist several works that trace the execution flow of an application from the perspective of its interaction with the system. Tools were being used to deconstruct the execution flow of embedded applications to optimize hardware-software co-design of embedded systems [13]. However, state-of-the-art *Android* applications and edge devices are now too complex for application of these dated tools long before *Android* OS. Kernel and *Binder* traces have been previously used to develop the *Appscope* framework [14]. *Appscope* was primarily designed for measuring the energy consumption of an application. Authors of [15] were one of the first to use the extraction of task execution and IPCs information for the characterization of *Android* applications. However, their methodology lacked the means and details necessary for debugging decisions of power managers for state-of-the-art heterogeneous multi-core processors.

Power management for heterogeneous multi-/many-cores is an active area of research [16]. Most *Android* devices ship with a group of default heterogeneity-aware power managers – governors [10] and schedulers [17]. However, most of these default power managers operate with naive heuristics, which are grossly inadequate to manage sophisticated state-of-the-art *Android* applications. Nevertheless, it has not stopped researchers from proposing sophisticated power managers that unify the role of governor and scheduler in one manager. Research power managers for heterogeneous multi-cores employ sophisticated heuristics from control theory [18] to price theory [19]. They are also often application-specific such as for mobile gaming [2] and web browsing [20]. Due to their application-specific design, they usually are only used by kernel developers to design power managers for application-specific edge devices. *To best of our knowledge, we are the first to propose a tool to debug CPU power managers designed for edge devices with Android OS.*

VII. Conclusion

In this work, we introduce the first automated tool – *BrezeFlow* – that is capable of quantifying the quality of CPU power governors and schedulers (power managers) on edge devices with *Android* OS. Evaluations on a real-world heterogeneous multi-core CPU show that *BrezeFlow* can identify and classify misdecisions for several different power managers from both industry and academia. *BrezeFlow*, on average, flagged 2,100 misdecisions per second for the evaluated power managers. We also present a case study demonstrating the use of *BrezeFlow* to gain insights into the shortcomings of an existing power manager, which can then be used for improving future designs.

VIII. Acknowledgement

This work was partially funded by Singapore Ministry of Education Academic Research Fund TI 251RES1905.

References

[1] S. Pagani, L. Bauer, Q. Chen, E. Glocker, F. Hannig, A. Herkersdorf, H. Khdr, A. Pathania, U. Schlichtmann, D. Schmitt-Landsiedel, *et al.*, "Dark Silicon Management: An Integrated and Coordinated Cross-Layer Approach," *it-Information Technology*, vol. 58, no. 6, pp. 297–307, 2016.

[2] N. Peters, D. Füß, S. Park, and S. Chakraborty, "Frame-Based and Thread-Based Power Management for Mobile Games on HMP Platforms," *IEEE 34th International Conference on Computer Design (ICCD)*, pp. 169–176, 2016.

[3] P.-H. Tseng, P.-C. Hsiu, C.-C. Pan, and T.-W. Kuo, "User-Centric Energy-Efficient Scheduling on Multi-Core Mobile Devices," *51st Design Automation Conference (DAC)*, pp. 1–6, 2014.

[4] B. Dietich, N. Peters, S. Park, and S. Chakraborty, "Estimating the Limits of CPU Power Management for Mobile Games," in *35th IEEE International Conference on Computer Design (ICCD)*, pp. 1–8, IEEE, 2017.

[5] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting Energy Bugs and Hotspots in Mobile Apps," *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-November-2014, pp. 588–598, 2014.

[6] A. Hoffman, "BrezeFlow." github.com/alxhoff/BrezeFlow, 2020.

[7] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, "Integrated CPU-GPU Power Management for 3D Mobile Games," *51st Design Automation Conference (DAC)*, pp. 1–6, 2014.

[8] A. Pathania, S. Pagani, M. Shafique, and J. Henkel, "Power Management for Mobile Games on Asymmetric Multi-Cores," *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 243–248, 2015.

[9] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra, "Power-Performance Modelling of Mobile Gaming Workloads on Heterogeneous MPSoCs," *52nd Design Automation Conference (DAC)*, p. 201, 2015.

[10] Venkatesh Pallipadi & Alexey Starikovskiy, "The Ondemand Governor," in *Linux Symposium Volume Two*, 2006.

[11] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting Energy Bugs and Hotspots in Mobile Apps," *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-Nove, pp. 588–598, 2014.

[12] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with eprof," *European Conference on Computer Systems (EuroSys)*, pp. 29–42, 2012.

[13] K. S. Vallerio and N. K. Jha, "Task Graph Extraction for Embedded System Synthesis," *International Conference on VLSI Design*, vol. 2003-Janua, pp. 480–486, 2003.

[14] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "Appscope: Application Energy Metering Framework for Android Smartphone using Kernel Activity Monitoring," *USENIX Technical Cconference*, pp. 387–400, 2012.

[15] S. Han, Y. Yun, and Y. H. Kim, "Profiling-Based Task Graph Extraction on Multiprocessor System-on-Chip," *Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 510–513, 2016.

[16] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends," *50th Design Automation Conference (DAC)*, pp. 1–10, 2013.

[17] J. Kobus and R. Szklarski, "Completely Fair Scheduler and its Tuning," *Whitepaper*, 2009.

[18] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical Power Management for Asymmetric Multi-Core in Dark Silicon Era," *50th Design Automation Conference (DAC)*, pp. 1–9, 2013.

[19] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, "Price Theory Based Power Management for Heterogeneous Multi-Cores," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 161–176, 2014.

[20] N. Peters, S. Park, S. Chakraborty, B. Meurer, H. Payer, and D. Clifford, "Web Browser Workload Characterization for Power Management on HMP Platforms," *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pp. 1–10, 2016.