CECS

CENTER FOR EMBEDDED & CYBER-PHYSICAL SYSTEMS

Using HSFMs to Model Mobile Gaming Behavior for Energy Efficient DVFS Governors

Jurn-Gyu Park, Hoyeonjiki Kim, Nikil Dutt, Sung-Soo Lim

Center for Embedded and Cyber-Physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

{jurngyup, dutt}@ics.uci.edu, {hyjg1123, sslim}@kookmin.ac.kr

CECS Technical Report 16-02

Jun. 24, 2016

Using HSFMs to Model Mobile Gaming Behavior for Energy Efficient DVFS Governors

Jurn-Gyu Park, Nikil Dutt
School of Information and Computer Science
University of California, Irvine, CA, USA
jurngyup, dutt@ics.uci.edu

Hoyeonjiki Kim, Sung-Soo Lim
School of Computer Science
Kookmin University, Seoul, South Korea
hyjg1123, sslim@kookmin.ac.kr

ABSTRACT

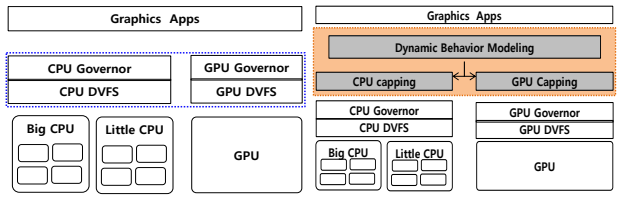
Contemporary mobile platforms use software governors to achieve high performance with energy-efficiency for heterogeneous CPU-GPU based architectures that execute mobile games and other graphics-intensive applications. Mobile games typically exhibit inherent behavioral dynamism, which existing governor policies are unable to exploit effectively to manage CPU/GPU DVFS policies. To overcome this problem, we present HiCAP: a Hierarchical Finite State Machine (HFSM) based CPU-GPU governor that models the dynamic behavior of mobile gaming workloads, and applies a cooperative, dynamic CPU-GPU frequency-capping policy to yield energy efficiency adapting to the game's inherent dynamism. Our experiments on a large set of 37 mobile games exhibiting dynamic behavior show that our HiCAP dynamic governor policy achieved substantial energy efficiency gains of up to 18% improvement in energy-per-frame over existing governor policies, with minimal degradation in quality.

Keywords

Model-based design; Power management policies; DVFS; Integrated GPU

1. INTRODUCTION

Mobile games are an increasingly important application workload for mobile devices. These games often demand high performance while rapidly depleting precious mobile battery resources, resulting in low energy efficiency. The recent trend towards Heterogeneous Multi-Processor Systems-on-Chip (HMPSoC) architectures (e.g., ARM's big.LITTLE with integrated GPU) attempt to meet the performance needs of mobile devices, and rely on software governors for power management in the face of high performance. Contemporary commercial mobile platforms (e.g., SAMSUNG Galaxy S6 and NEXUS 6) deploy separate governors for CPU and GPU DVFS power management (Figure 1(a)), but miss opportunities to coordinate power management between the CPU and GPU domains for improved energy efficiency. Some recent research efforts have proposed integrated CPU-GPU DVFS policies [17] [9] for a small set of mobile games, assuming a fairly static workload behavior (dotted box in Figure 1(a)). However, mobile games generally exhibit highly dynamic behaviors through a varying mix of CPU and GPU workloads; existing software governors are unable to exploit this behavioral dynamism to further improve energy efficiency while delivering acceptable user experience for mobile gaming. To address these issues, we present HiCAP (Figure 1(b)): a hierarchical FSM (HFSM) based integrated CPU-GPU DVFS governor that: 1) naturally models the application's dynamic behavior using the HFSM model, and 2) uses a simple, cooperative CPU and GPU frequency capping technique to achieve improved energy efficiency.



(a) Contemporary Governors (b) HiCAP Governor

Figure 1: CPU-GPU Mobile Governors

Many mobile games have multiple levels of play with complex stages, and thus the games themselves are typically designed hierarchically [11] [19]. Furthermore, the class of hierarchical FSMs provide an excellent abstraction for behavioral modeling of many complex embedded systems [3] [13]. Therefore we believe HFSMs offer a natural and intuitive modeling abstraction for capturing the behavioral dynamism of a game, for applying effecting DVFS policies.

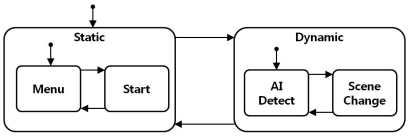
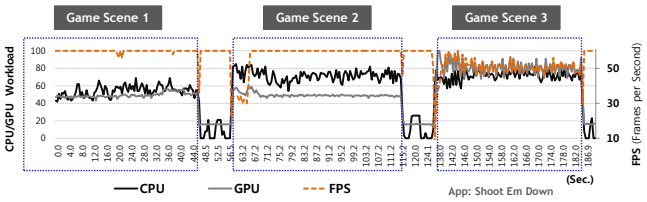


Figure 2: HFSMs in Game Design.

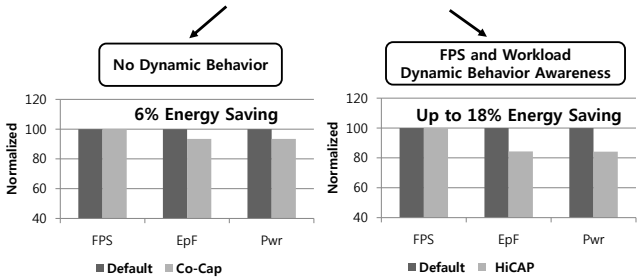
For instance, Figure 2 shows the top-level HFSM of a typical game that proceeds from the game set-up (*Static* superstate) to game execution (*Dynamic* superstate); of course during execution the *Dynamic* superstate will contain a complex hierarchy of game levels, stages and conditions to support various features of individual games. This motivates the use of HFSMs in our HiCAP governor model.

This paper is the technical report of [ISLPED'16]
HiCAP: Hierarchical FSM-based Dynamic Integrated CPU-GPU Frequency Capping Governor for Energy-Efficient Mobile Gaming.
DOI: http://cecs.uci.edu/files/2016/6/CECS-TR-16-02.pdf

Now consider Figure 3, where use a sample game *ShootEm-Down* to motivate HiCAP's need for HFSM modeling, and HiCAP's opportunity to improve energy efficiency by exploiting game dynamism. Figure 3(a) captures the dynamism for different game scenes/actions, showing the delivered Frames-per-Second (FPS) (dotted orange) [1], and the CPU (black) and GPU (gray) workloads, normalized to their respective maximums. This figure clearly shows the need for HFSMs to capture the game's behavior hierarchically: *Game Scene 1* is fairly static, whereas *Game Scenes 2* and *3* are highly dynamic, but with varying FPS, CPU and GPU workloads.



(a) Different Types of Game Dynamism



(b) No Dynamic Behavior (c) Dynamic Behavior

Figure 3: Sample Mobile Game (ShootEmDown)

Figures 3(b) and (c) highlight HiCAP's opportunity for energy savings by exploiting this game's dynamic behaviors. When the game's FPS and CPU/GPU workload dynamism are not captured (Figure 3(b)), a state-of-the-art static-modeling DVFS capping technique [16] achieves a modest (up to 6% improvement). However, the game's execution exhibits various levels of dynamism, and our HiCAP governor is able to achieve significant (up to 18%) improvement in energy efficiency over state-of-the-art static modeling governors by exploiting the dynamism in the game's FPS and the CPU/GPU workload, clearly demonstrating the potential for improved energy efficiency.

Our paper makes the following specific contributions:

- We propose a Hierarchical FSM (HFSM) based dynamic behavior modeling strategy for mobile gaming
- We present HiCAP: a cooperative CPU-GPU governor that deploys a simple maximum frequency-capping methodology exploiting the HFSM for dynamic DVFS
- We present experimental results on a large set of 37 real mobile games with dynamic behaviors, showing significant energy savings of up to 18% in Energy-per-Frame (EpF) with minimal loss in FPS performance.

[1]Note that the delivered Frames-per-Second (FPS) is a typical metric for gaming user experience, with a maximum FPS of 60 in the Android system.

2. RELATED WORK

DVFS is a traditional energy conservation strategy that has been applied for both CPUs and GPUs in desktop and mobile space. For graphics-intensive applications such as mobile games, typically frames per second (FPS) and energy per frame (or FPS per watt) are used as performance and energy consumption metrics, respectively. Some efforts have begun analyzing graphics-intensive rendering applications (e.g., 3D games) in mobile devices: Gu *et al.* [5], [4] proposed CPU graphics rendering workload characterization and CPU DVFS for 3D Games, under the assumption that mobile devices such as PDAs and mobile phones do not have integrated mobile GPUs.

With the emergence of high performance integrated mobile GPUs, several research efforts have proposed integrated CPU-GPU DVFS governors: Pathania *et al.*'s [18] integrated CPU-GPU DVFS algorithm didn't consider quantitative evaluation for energy savings (e.g., per-frame energy or FPS per watt); their next effort [17] further developed power-performance models to predict the impact of DVFS on mobile gaming workloads, but did not model the games' dynamic behaviors; and Kadjo *et al.* [9] used a queuing model to capture CPU-GPU-Display interactions across a narrow range of games exhibiting limited diversity in CPU-GPU workloads.

In another direction, frequency capping techniques have been proposed as a simple yet effective strategy for DVFS. Li *et al.* [10] initially introduced frequency capping for energy efficiency and Sahin *et al.* [20] proposed a QoS-aware frequency capping technique for thermal management policies, but their approaches are restricted to only CPU governor. Most recently, our previous work [16] proposed a coordinated CPU-GPU maximum frequency capping technique and applied it to a range of mobile graphics workloads, but assumed static characterization of each application, and did not consider dynamic behaviors for different CPU/GPU graphics workloads and the Quality of Service (QoS) requirement.

FSMs and Hierarchical FSMs [3] were proposed as an effective semantic model to capture complex system behavior, with many variants such as Statecharts [7] providing modeling support for complex embedded systems [13] in both hardware and software. Malkowski *et al.* [12] proposed a phase-aware adaptive hardware selection technique (FSM-implementation) employing DVFS, prefetchers and hardware performance counters for both reducing power consumption and performance improving. Moreover, many embedded systems – particularly for streaming multimedia – have combined dataflow with HFSM models to enable tasks such as runtime resource allocation and dynamic task mapping (e.g., [8]).

To the best of our knowledge, our HiCAP approach is the first to introduce a hierarchical FSM-based dynamic behavior modeling strategy for mobile gaming considering QoS requirements; and the first to exploit the dynamics of CPU/GPU graphics workload variation through a simple maximum frequency capping strategy for energy efficiency on modern HMPSoC platforms by avoiding frequency over-provisioning while delivering acceptable user QoS.

3. APPROACH

3.1 Preliminaries

We begin by defining terminology used in this paper.

CPU/GPU Workload: Mobile workload variations are

typically quantified using a cost metric that is a product of the utilization and frequency [2] [18]. Accordingly, as shown in Equation(1), we deploy the normalized CPU and GPU costs [16].

$$Normalized\ Cost = \frac{Curr\_Util. \times Curr\_Freq.}{Max\_Util. \times Max\_Freq.} \quad (1)$$

CPU/GPU Cost Matrix and Quadrants: To model the dynamics of game behavior across the CPU and GPU domains, we develop a CPU/GPU Cost Matrix using the normalized CPU (y-axis) and GPU (x-axis) costs, where the dominance of that component (CPU- or GPU-) increases along each axis. Figure 4 shows our set of 37 gaming applications that populate different entries in this matrix, corresponding to the CPU/GPU intensity of its workload. This matrix has 4 major quadrants: No CPU-GPU dominant, CPU dominant, GPU dominant and CPU-GPU dominant workloads [16].

CPU \ GPU Cost	0–20	20–40	40–60	60–80	80–90	90–100
0–20		Dream Bike Sky Castle 2	Bonsai Bench	Extreme Motorbike		
20–40	Armored Car	Anomaly2 Low	Hercules Anomaly2 Normal	Epic Citadel	Anomaly2 High Implosion	
40–60	Fast and Furious 7	Madden NFL 15 Micro-bench 102	Frontline Dday Godus ShootEm Down 5 Micro-bench 110 Micro-bench 111	3D Mark - Normal		
60–80		Q3 Zombie Map 1 Terminator Genisys Micro-bench 100 Micro-bench 101	Micro-bench 112 Dino Gunship ShootEmDown A	ShootEmDown D Train Simulator		
80–90		Micro-bench 103	Micro-bench 113	RoboCop Edge of Tomorrow		
90–100	Assassin Creed : P	Turbo FAST (Net) Q3 Zombie Map 2 Q3 Zombie Map 4				A Total Set : 37

Figure 4: A set of benchmarks.

Frequency Over-Provisioning and Frequency Capping: Existing software governors often over-provision frequency, which provides energy-saving opportunities by simply capping the maximum operating frequency without loss of quality [16].

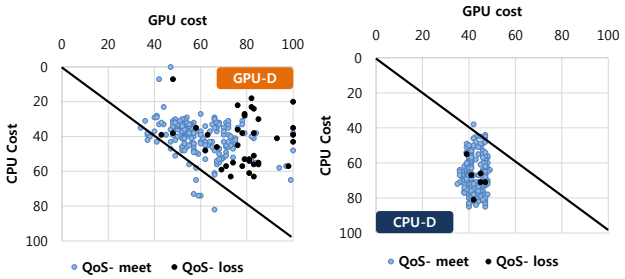
Saturated Frequency Look-up Tables: We can enter saturated (capped) frequencies (determined statically [16] or dynamically) in each entry of the CPU-GPU cost matrix, resulting in a CPU/GPU saturated frequency look-up table that covers the entire cost matrix. This saturated frequency look-up table can then be used for determining the new CPU and GPU maximum frequency settings by the runtime governor.

Having discussed preliminaries, we now describe HiCAP's approach for using HFSMs to model dynamic behavior for mobile gaming (Section 3.2), and HiCAP's heuristics for configuring CPU/GPU saturated frequencies through a runtime frequency-capping methodology (Section 3.3).

3.2 HFSM-based Dynamic Behavior Model

3.2.1 *Game Dynamism Analysis*

Although each game may be situated in a specific quadrant of the CPU/GPU cost matrix (Figure 4), during runtime the game will exhibit dynamic behavior that covers a dynamic footprint of the CPU/GPU cost matrix. For instance Figure 5 shows the dynamic footprints of two benchmarks (*3DMark-Normal* and *Dino-Gunship*) during a 500ms execution snapshot, with the blue dots representing states that satisfy QoS (labeled as *QoS-meet*) while the black dots represent states that are unable to meet the QoS requirement (labeled as *QoS-loss*). Furthermore, we also note that the CPU-GPU workload dynamism can be characterized as being CPU-dominant (*CPU-D*) or GPU-dominant (*GPU-D*), if the CPU/GPU workload can be characterized quantitatively. For the specific examples in Figure 5, we note that the footprint of the *3DMark-Normal* game (Figure 5(a)) is mainly located in the right-side of the diagonal line, which implies that GPU cost is higher than CPU cost (i.e., this is a GPU-dominant workload). Similarly, the game *Dino-Gunship* (Figure 5(b)) has the footprint of a CPU-dominant workload. In summary, using this strategy we can characterize each game's dynamic behavior.



(a) 3DMark-Normal (b) Dino-Gunship

Figure 5: Footprints of Game Dynamism

3.2.2 *HFSM-based Modeling*

We use hierarchical FSMs (HFSMs) to model each game's dynamic behavior, as shown in Figure 6. We use a formalism similar to StateCharts, where each state represents a specific dynamic behavioral state of a game application, and the state may be further refined into another lower-level FSM. We call the inside FSM the sub-state and the outside FSM the super-state in the StateCharts language [13].

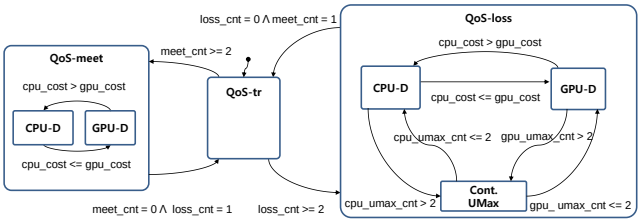


Figure 6: Hierarchical Finite State Machine

Our HFSM model starts with two super-states at the highest level labeled *QoS-meet* and *QoS-loss*, and each super-state has two sub-states labeled *CPU-D* and *GPU-D* as shown in Figure 6. During design optimization, new states can be added heuristically to this model. For example, while meeting a QoS requirement, let's assume that two states are fluctuating continuously (e.g., repeated *QoS-meet* and *QoS-loss*) for a certain amount of time. In this situation, the output of each state also fluctuates continuously, resulting in performance or power overheads. To improve this situation, a new QoS-transient (*QoS-tr*) state can be designed, that generates a new output. Furthermore, in terms of CPU/GPU workload (cost) – i.e., multiplication of frequency and utilization – a continuous maximum utilization (*Cont-UMax*) may result in FPS degradation in the *QoS-loss* state. Because this may further degrade performance, we can design a new state to deal with this specific situation. Also note that due to the hierarchical structure of the HFSM, sub-states such as *CPU-D*, *GPU-D*, or *Cont-UMax* are in turn also super-states at a lower level, consisting of

sub-states themselves; detailed additional sub-states are not described here, but can be found in Section 3.4.

The HFSM in Figure 6 has switching conditions shown as annotations on the HFSM edges. We partition the execution time of applications into every epoch (Tepoch = 500ms) [2]. At the start of each epoch, using FPS and workload data captured during the previous epoch, our HFSM determines whether to make the transition or not, and takes the appropriate action. The QoS-target is changeable by users, but for this study we use the maximum FPS (60 in Android system) to compare directly the default and related governors. We now provide a detailed description of our switching heuristic using Figure 6 and Algo. 1.

The HFSM is initialized to the *QoS-tr* super-state (the default state among the super-states). If the average-FPS of the previous epoch meets the QoS-target for the past two states (*meet_cnt* >= 2), then the FSM transitions to the *QoS-meet* state (line 2). In this state, if the average-FPS loses the FPS-target for the first time, then the FSM transitions to the *QoS-tr* state (line 17), not directly to *QoS-loss* state. However, if the average-FPS loses FPS-target two times consecutively (*loss_cnt* >=2), then the FSM transitions to the *QoS-loss* state (line 8). We use two counters (*meet_cnt* and *loss_cnt*) to detect the *QoS-tr* state.

For sub-state transitions, the *QoS-loss* super-state can be further refined into *CPU-D*, *GPU-D*, and *Cont-UMax* sub-states according to various conditions: if CPU cost is larger than GPU cost, the *CPU-D* sub-state will be the new active state (line 9); if GPU cost is bigger than or equal to GPU cost, the *GPU-D* sub-state will be the new active state (line 11); and if CPU/GPU maximum utilization counters (*cpu_umax_cnt* or *gpu_umax_cnt*) are bigger than specific thresholds, the *Cont-UMax* sub-state will be the new active state (line 14). In a hierarchical FSM model, same sub-states such as *CPU-D* and *GPU-D* can be located in different super-states (line 3 and 5).

Until now, we have not considered outputs (reactions) generated by our hierarchical FSM. Once the two transitions in super- and sub-state are triggered, possible reactions include the generation of events and/or assignments to variables. Using the HFSM model, we assign new appropriate CPU/GPU saturated frequencies as HFSM outputs, which are then applied in the next epoch.

3.3 Frequency-Capping

We adopt a CPU and GPU frequency capping heuristic for the following reasons: 1) Simplicity: adding a capping module on top of the default (or custom) CPU/GPU governors is easier than complicated integrated CPU/GPU governors. 2) Portability: capping is easily adaptable to newer platforms. 3) Efficacy: the previous work [16] with the technical report [15] has shown overall improvements for different types of graphics workloads. 4) Elimination of wasteful frequency over-provisioning: commercial CPU governors typically scale frequency using utilization based thresholds. In other words, if CPU utilization is greater than or equal to a certain threshold, the governor sets the highest frequency from the corresponding frequency [10]. This is also similar in GPU governor but more conservative. However, in both cases these approaches may result in frequency over-provisioning, since the frequency is set to a higher level even when the target QoS has been met at maximum utilization. This results in wasted power.

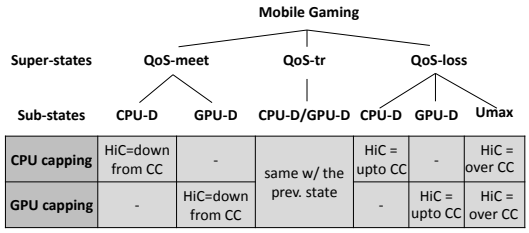


Figure 7: Different Capping Policies as Outputs

Algorithm 1 The HFSM Heuristic Pseudo-code

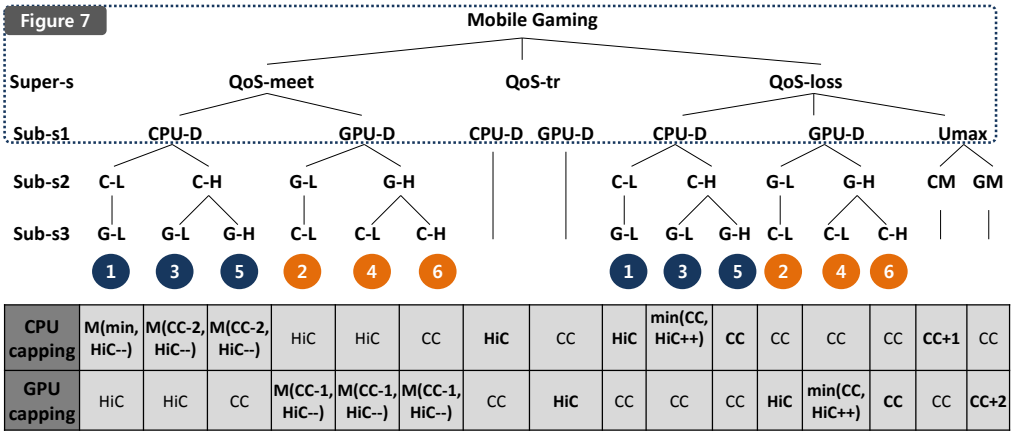
```
    Every epoch (Tepoch)
 1: Calculate average FPS, CPU and GPU Costs per epoch

 2: if meet_cnt >= 2 then                                  ▷ QoS-meet
 3:     if cpu_cost > gpu_cost then                          ▷ CPU-D
 4:         HiC = CPU Freq-Cap down from CC
 5:     else                                                 ▷ GPU-D
 6:         HiC = GPU Freq-Cap down from CC
 7:     end if
 8: else if loss_cnt >= 2 then                             ▷ QoS-loss
 9:     if cpu_cost > gpu_cost then                          ▷ CPU-D
10:         HiC = CPU Freq-Cap up to CC
11:     else                                                 ▷ GPU-D
12:         HiC = GPU Freq-Cap up to CC
13:     end if
14:     if cpu (or gpu)_umax_cnt > 2 then                    ▷ UMax
15:         HiC = CPU or GPU Freq-Cap over CC
16:     endif
17: else                                                   ▷ QoS-tr
18:     Same Frequency-Cap with the previous epoch
19: end if
20: Newly updated CPU and GPU HiC LUTs for next epoch
```

To configure appropriate saturated frequencies, we design dynamically changing CPU and GPU HiCAP (HiC) look-up tables (LUTs), which determine the new CPU and GPU maximum frequencies of cost quadrants for the next epoch. We label the upper-bounded frequencies from Co-Cap [16] as CC.

By exploiting the hierarchical property of HFSMs (Figure 7), we apply different capping policies in each sub-state. For example, the capping policies between *QoS-meet QoS-tr*, and *QoS-loss* state are fundamentally different. In the *QoS-meet* state, lower saturated frequencies than CC are set because QoS-target is already met (line 4 and 6). However, in the *QoS-loss* state, almost similar frequencies to CC are required to have competitive performance (line 10 and 12). In particular, for continuous maximum utilization in *QoS-loss* state, temporarily higher frequency than CC is used to prevent FPS reduction (line 15). In the *QoS-tr* state, the same saturated frequencies of the previous state (line 18) are used to reduce fluctuation of outputs (i.e., overheads). According to the characterization of different types of graphics

[2]We tried different epoch periods. In our previous work [16], we used an epoch of 100ms which is same with the GPU governor period. However, when we consider an average-FPS of the epoch in this work, 100 ms is too short to detect average-FPS of the epoch (i.e, the maximum frame number per 100ms is 6 (60 FPS); if three frames are lost consecutively due to intensive workloads during the epoch, the average-FPS will be 30 FPS; we use 500ms epoch to set/compare fine-grained FPS-targets such as 50 and 40 FPS, rather than 1000ms epoch which has more fine-grained FPS control but may occur responsiveness problem in performance.



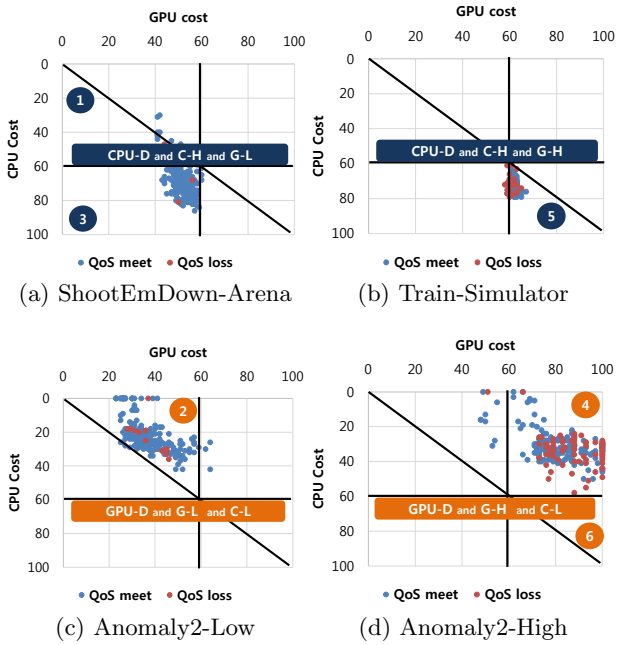
CPU capping	M(min, HiC--)	M(CC-2, HiC--)	M(CC-2, HiC--)	HiC	HiC	CC	HiC	CC	HiC	min(CC, HiC++)	CC	CC	CC	CC	CC+1	CC
GPU capping	HiC	HiC	CC	M(CC-1, HiC--)	M(CC-1, HiC--)	M(CC-1, HiC--)	CC	HiC	CC	CC	CC	HiC	min(CC, HiC++)	CC	CC	CC+2

Figure 8: Detailed HFSMs and Capping Policies.

workloads (i.e., for CPU (GPU) dominant workloads, energy saving within minimal FPS decline mainly results from CPU (GPU) capping), we mainly scale CPU (GPU) saturated frequency for *CPU-D* (*GPU-D*) sub-states.

3.4 Detailed HFSMs

In the previous sections, we described the fundamental HFSM-based dynamic behavior model and the frequency-capping policies in the first sub-states. However, as we mentioned in Section 3.2.2, the sub-states such as *CPU-D*, *GPU-D*, and *Cont-UMax* are also super-states consisting of sub-states themselves. Therefore, in this section, we describe the full three-depth sub-states, the detailed frequency-capping policies, and the reasons for the heuristics.



(a) ShootEmDown-Arena (b) Train-Simulator

(c) Anomaly2-Low (d) Anomaly2-High

Figure 9: Detailed Footprints of Game Dynamism.

As shown in Figure 8, two additional depths of sub-states (Sub-s2 and Sub-s3) in addition to the first-depth sub-states (Figure 7) were added for the detailed HFSMs based on the detailed footprints of game dynamism (Figure 9). For example, *CPU-D* sub-states in Figure 8 can be refined into CPU-Low (*C-L*) and CPU-High (*C-H*) sub-states (Sub-s2); In addition, the *C-H* state can be refined again into GPU-Low (*G-L*) and CPU-High (*G-H*) sub-states (Sub-s3); if CPU (GPU) cost is less than a specific threshold (we call it *cost_th* and use empirically 60 [16] for the categorization), it is the *C-L* (*G-L*) state, otherwise *C-H* (*G-H*) state. Therefore, the *CPU-D* sub-states have the three leaf states: *C-L*/*G-L* (the number one area), *C-H*/*G-L* (the number three area), and *C-H*/*G-H* (the number five area). (Like the *CPU-D* sub-states, the *GPU-D* sub-states can cover the number two, four, and six areas in Figure 9).

Algorithm 2 The Heuristic in QoS-meet and CPU-D

```
1: if cpu_cost < cost_th then                    ▷ C-L and G-L
2:     HiC_cpu = max(min_cpu, HiC_cpu – –)
3:     HiC_gpu = HiC_gpu
4: else
5:     if gpu_cost < cost_th then                ▷ C-H ad G-L
6:         HiC_cpu = max(CC_cpu - 2, HiC_cpu – –)
7:         HiC_gpu = HiC_gpu
8:     else                                      ▷ C-H and G-H
9:         HiC_cpu = max(CC_cpu - 2, HiC_cpu – –)
10:        HiC_gpu = CC_gpu
11:    end if
12: end if
```

With the added dynamic behaviors, more detailed frequency capping policies are also needed. As an example, we describe one instance of the *CPU-D* sub-state in the *QoS-meet* super-state using Figure 8 and Algo. 2. First, the *C-L*/*G-L* state is already *QoS-meet* state and has low CPU/GPU workload; the most aggressive power saving policy – CPU capping down to the minimum frequency (line 2) and GPU capping using HiC LUT [3](line 3) – is applied. Second, the *C-H*/*G-L* state is already *QoS-meet* state and has high CPU- and low GPU- workload; a less aggressive power saving policy compared to the policy of the *C-L*/*G-L* state – CPU capping down to a specific frequency of CC_cpu-2 (line 6) and GPU capping with HiC LUT (line 7) – is applied. Finally, the *C-H*/*G-H* state is *QoS-meet* state but has

[3]Note that HiC LUTs are always lower than or equal saturated (capped) frequencies except the *CONT-UMax* state compared to CC LUTs. This is because CC LUTs [16] have fixed upper-bounded CPU and GPU saturated frequencies which were configured statically using a large training set and evaluation set; HiC LUTs are changing dynamically down from/up to CC LUTs except the *CONT-UMax* state.

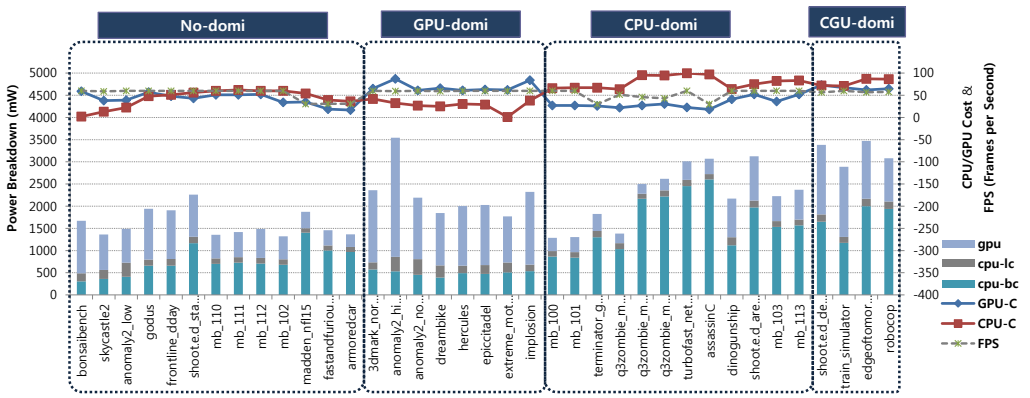


Figure 10: Characteristics of Benchmark Set.

high CPU and GPU workloads; a relatively performance-demanding policy compared to the previous policies – CPU capping down to a specific frequency of CC_cpu-2 (line 9) and GPU capping with CC LUT (line 10)– is applied.

Note that we capped down to the minimum CPU frequency only for *QoS-meet/CPU-D* state. Alternatively, we set a two-level lower CPU frequency from CC (CC_cpu-2) as a new available minimum frequency (line 6 and 9) to minimize unpredictable FPS degradation. In addition, the reason that we use only a one-level lower GPU-frequency from CC in the *GPU-D* sub-states is related to governor characteristics on ODROID-XU3. Unlike the big CPU cluster frequency levels (from 1.2 to 2.0Ghz, 9 levels with relatively high minimum frequency), the GPU governor supports from 177Mhz to 543Mhz (6 levels with very low minimum frequency). With the consideration of CPU/GPU governor characteristics, a different heuristic is applied in each leaf state. Similar to the *QoS-meet/CPU-D* state, the other CPU-D and *GPU-D* sub-states can be interpreted using the frequency capping heuristics in Figure 8; *QoS-loss* sub-states have opportunistically more performance-demanding (requiring higher frequency) policies compared to the *QoS-meet* sub-states, up to min(CC, HiC++) or CC LUTs.

Algorithm 3 The Heuristic in Cont-UMax

```
1: if cpu_umax_cnt >= cu_th then      ▷ CPU Cont-UMax
2:         HiC_cpu = min(CC + 1, max_cpu)
3:         HiC_gpu = CC_gpu
4: endif
5: if gpu_umax_cnt >= gu_th then      ▷ GPU Cont-UMax
6:         HiC_cpu = CC_cpu
7:         HiC_gpu = min(CC_gpu + 2, max_gpu)
8: endif
```

Unlike the other sub-states, only in the *Cont-UMax* sub-state (continuous maximum utilization state), temporarily one-/two-level higher frequencies than the CC LUTs (line 2 and 7 in Algo. 3) are assigned to reduce unpredictable FPS degradation. We tested different thresholds for *cu_th* and *gu_th*; an relatively interactive *cu_th* threshold (2) and a more conservative *gu_th* (3) threshold were finally determined heuristically based on the governor characteristics and empirical data.

In summary, CPU and GPU saturated (capped) frequencies are dynamically changing according to the two lookup tables: CC and HiC LUTs. CC LUTs are fixed upper-bounded CPU and GPU saturated frequencies which were configured statically using a large training set and evaluation set [16]; HiC LUTs are configured dynamically according to the dynamic behavior states and the described specific policies. To achieve competitive FPS and energy saving at the same time for different types of graphics workloads (No-, CPU-, GPU-, and CPU-dominant workloads), each heuristic policy in each leaf state was applied by our HFSM-based model using heuristic thresholds based on the footprints of game dynamism and empirical results observation.

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

We evaluated our HiCAP governor on the ODROID-XU3 [6] development board installed with Android 4.4.2 and Linux 3.10.9; Table 1 summarizes our platform configuration. The platform is equipped with four TI INA231 power sensors measuring the power consumption of big CPU cluster (CPU-bc), little CPU cluster (CPU-lc), GPU and memory respectively. The CPU supports cluster-based DVFS at nine frequency levels (from 1.2Ghz to 2.0Ghz) in CPU-bc and at seven frequency levels (from 1.0Ghz to 1.6Ghz) in CPU-lc, and GPU supports six frequency levels (from 177Mhz to 543Mhz).

Table 1: Platform Configuration

Feature	Description
Device	ODROID-XU3
SoC	Samsung Exynos5422
CPU	Cortex-A15 2.0Ghz and Cortex-A7 Octa-core CPUs
GPU	Mali-T628 MP6, 543Mhz
System RAM	2Gbyte LPDDR3 RAM at 933MHz
Mem. Bandwidth	up to 14.9GB/s
OS(Platform)	Android 4.4.2
Linux Kernel	3.10.9

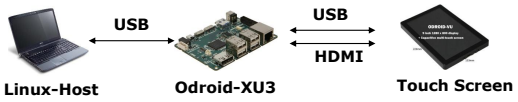


Figure 11: Experimental Setup.

Benchmark Set: As shown in Figure 4, we use 37 gaming applications covering different types of graphics workloads; many are derived from previously published papers ([14] [18] [17] [9] [16]). Due to rapidly changing heterogeneous platforms architectures and the continuing emergence of a plethora of mobile games [16], we need a wide range of games as representative benchmarks leaf exercise varying intensity of CPU and GPU workloads. Therefore, as shown in

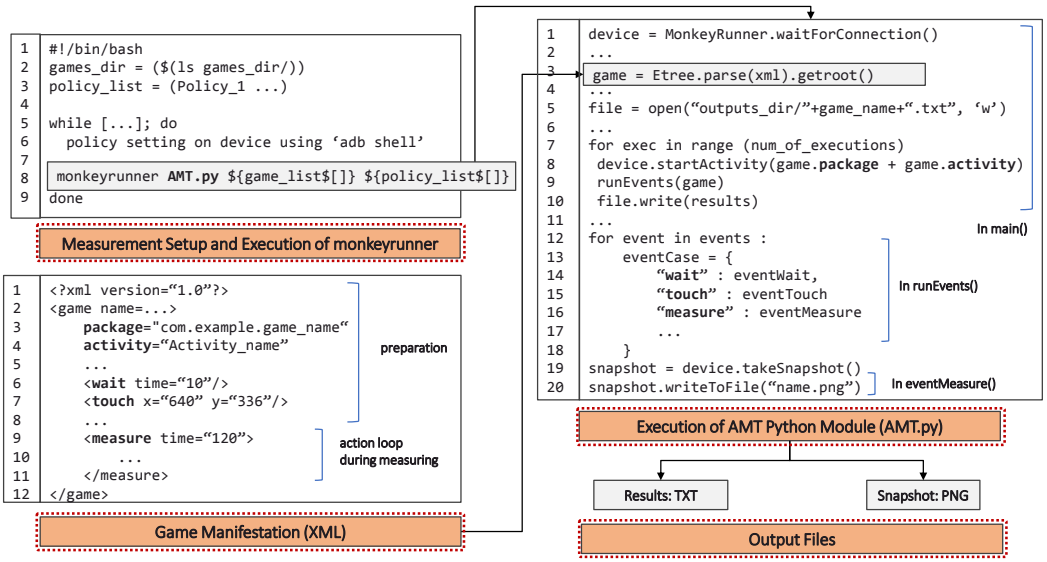


Figure 12: Automatic Measurement Tool.

Figure 10, we characterize the 37 benchmark set according to the four different CPU/GPU cost quadrants. The characteristics themselves also have important observations: the correlation between CPU/GPU cost and CPU/GPU power consumption is almost proportional; for CPU (GPU)-dominant benchmarks, average CPU (GPU) power consumption is 80 (68)% of the total (CPU-bc, CPU-lc and GPU) power consumption. For example, the CPU power of qzombie_map2 in CPU-dominant has up to 91% of the total power consumption; the GPU power of anomaly2_high benchmark in GPU-dominant has up to 76% of the total power consumption.

For comparison: We compare our HiCAP governor against the default and two state-of-the-art related governors ([17] and [16]). The default corresponds to the independent CPU and GPU governors in Linux (Interactive CPU governor and ARM's Mali Midgard GPU governor). The first state-of-the-art governor [17] (represented as PAT15) proposed an integrated CPU-GPU DVFS strategy by developing predictive regression power-performance models. The second state-of-the-art governor [16] (represented as Co-Cap16) presented a coordinated CPU and GPU maximum frequency capping technique using upper-bounded saturated frequencies configured in a static training phase.

Overhead: Our power manager was implemented in Linux kernel layer on top of CPU and GPU governors. The execution time of our manager per epoch is within 10us, which is totally negligible compared to the epoch period (500ms). Moreover, we did not observe any noticeable increase in average power consumption due to the power manager.

4.1.1 Automatic Measurement Tool

To measure a large set of mobile games automatically and quantitatively, there is a need for an automated framework to run games, capture key execution characteristics and ensure repeatability of experiments. Towards this end, we developed an automatic measurement tool for mobile games (AMTG) using Linux shell scripts, Python modules, and XML files. Figure 11 shows the hardware experimental setup, and all AMTG modules (Figure 12) are executed on the Linux-host with USB connection to the device (ODROID-XU3). Using this tool, we can capture the average values of FPS, total power (CPU-bc, CPU-lc, and GPU) and energy per frame (EpF) with repeated runs of each benchmark from the mobile device for a certain amount of time, and average their measurements.

The AMTG is based on the monkeyrunner tool [1] and composed of four major components: 1) Measurement Setup and Execution of monkeyrunner (Linux shell-scripts). 2) Game Manifestation (XML). 3) Execution of AMT Python Module. 4) Output files, as shown in Figure 12. The monkeyrunner tool provides an API (*MonkeyRunner.waitForConnection()*) for writing programs that control an Android device or emulator from outside of Android code. With monkeyrunner, we write a Python program that runs an installed Android application, sends input values to it, takes screenshots of its user interface, and stores screenshots on the host. Pseudo-codes in each module will describe main functions for each component.

Measurement Setup and Execution of monkeyrunner: In order to decide a directory which has games for measurements and candidate policies, a measurement setup process is required. Therefore, a specific directory and policies for comparison should be defined in this stage. For example, as shown in the pseudo code of Figure 12 (top-left), *games_dir* (Line 2) and *policy_list* (Line 3) were defined. After setting each candidate policy (Line 6), the monkeyrunner program is repeatedly executed with the three arguments (a Python module name, a game to measure and the corresponding policy) (Line 8).

Game Manifestation: Before describing the Python module to run each game on a device, the manifestation of package name, activity name, and events should be defined in advance. This information is specified in an XML file (one XML file per game). The XML file (down-left) is composed of two parts: a preparation part and an action part. In the preparation part, the names of package and activity (Line 3 and 4), all positions for menu settings after loading times (Line 6 and 7) should be defined manually because each application has different package and activity names, settings and loading times. In the action part, we define the

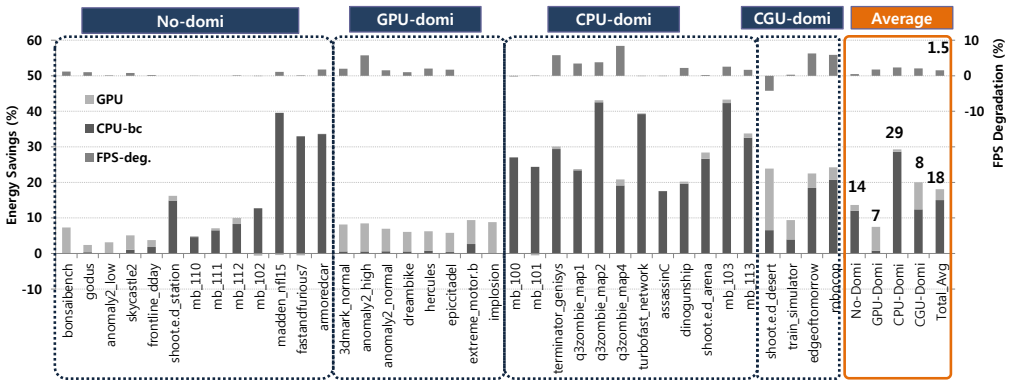


Figure 13: FPS and Energy Savings Comparison of The Default vs. our HiCAP.

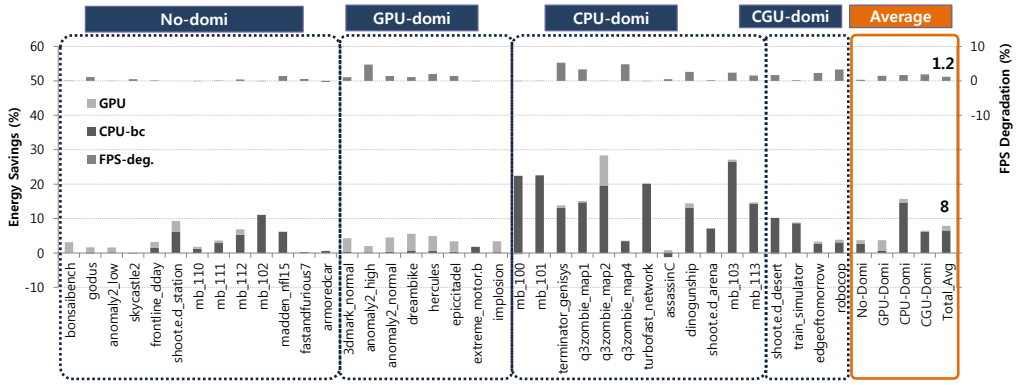


Figure 14: FPS and Energy Savings Comparison of Co-Cap16 [16] vs. our HiCAP.

measuring time (e.g., 120 seconds in this work) (Line 9) and can optionally add additional workloads periodically using touch, press or/and drag events in Line 10.

Execution of AMT Python Module: The Python module (AMT.py, top-right) connects the current device (ODROID-XU3) and returns a *MonkeyDevice* object (Line 1). After parsing the corresponding XML file (Line 3), it runs the package and activity of the game (Line 8) and executes the input events (Line 9), which are defined in the game manifestation file and are implemented for each event in this Python module (Line 12-18). Finally according to change of the *num_of_executions*, the number of measurements for each policy in each game will be defined.

Output Files: Output files are defined in the AMT Python module (Line 5 and 20). We generate two different types of output files: txt-based result files, png-based snapshot files. Each opened TXT file (Line 5) will be written by results (Line 10). The result files include the average FPS and power data during the measuring time in addition to all captured data such as CPU/GPU utilization, frequency and cost function etc; they can be obtained using '*adb shell dmesg*' and '*grep*' commands. The snapshot files have the start- and the finish-screenshot of the measuring time of each application, and can be used to evaluate correctness of each execution.

4.2 Results and Analysis

Figure 15 summarizes the average results of the benchmark in FPS and EpF. Our HiCAP manager improves energy per frame by 18%, 14% and 8% on average compared to the default, PAT15, and Co-Cap16 respectively, with negligible FPS decline of 1.5% and 1.2% on average compared to the default and Co-Cap16 respectively, and 2.9% better FPS than PAT15.

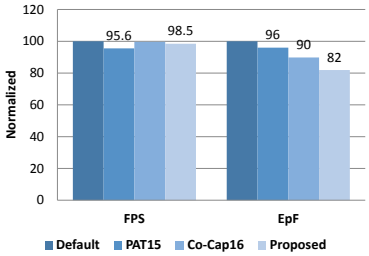


Figure 15: Average Results of the Benchmark Set

Comparison to the Default: Figure 13 shows CPU-bc (CPU) and GPU energy savings per frame and FPS degradation compared to the default. Our results using HiCAP show a significant combined CPU and GPU average energy savings of 18% with insignificant FPS degradation of 1.5% across all the benchmarks. On average, the CPU's contribution to the energy savings is 15%, while the GPU's energy savings contribution is 3%. However, the results are totally different for GPU-dominant benchmarks (the second category in Figure 13); the GPU's contribution to the energy saving is 7% while the CPU's contribution is 1% of the 8% total energy savings. For CPU-dominant benchmarks (the third category in Figure 13), we observe 29% CPU and 1% GPU contributions.

Comparison to Co-Cap16 [16]:Figure 14 shows CPU

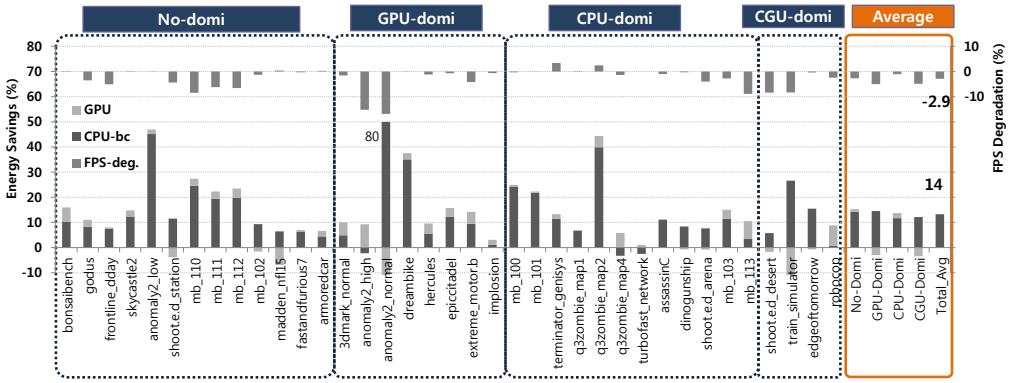


Figure 16: FPS and Energy Savings Comparison of PAT15 [17] vs. our HiCAP

and GPU energy savings per frame and FPS degradation compared to Co-Cap16. Our results using HiCAP show a significant combined CPU and GPU average energy savings of 8% with insignificant FPS degradation of 1.2% across all the benchmarks. On average, the CPU's contribution to the energy savings is 6.5%, while the GPU's energy savings contribution is 1.5%. For GPU-dominant benchmarks (the second category in Figure 14), the GPU's contribution to the energy saving is 3% while the CPU's contribution is 1% of the 4% total energy savings. For CPU-dominant benchmarks (the third category in Figure 14), we observe 15% CPU and 1% GPU contributions.

Our results clearly show that HiCAP achieves significant improvements in energy efficiency over the default governor and Co-Cap16 [16]; and that these are fair comparisons, since we used the same ODROID-XU3 platform and similar games in our experiments. We also note that the results shown in [9] (using the Intel Baytrail SoC platform and executing a small set of No- and GPU-dominant benchmarks) are very different from ours: their work showed a larger savings from the GPU (13.3%) than the CPU (4.1%), because the GPU typically consumes more energy than the CPU during graphics applications. However unlike our HiCAP work, their experiments did not comprehensively cover a wide range of mobile games exhibiting dynamism. Indeed based on our comprehensive experiments and analyses, we observe that the energy savings for mobile gaming benchmarks are mainly dependent on characteristics of the benchmarks (i.e., GPU energy savings from GPU-dominant and CPU energy savings from CPU-dominant benchmarks) and platform characters such as CPU/GPU minimum/maximum frequency and the number of CPU/GPU frequency levels, in addition to the governor algorithm itself.

Comparison to PAT15 [17]: As shown in Figure 16, our HiCAP governor shows a significant average energy savings of 14%, with a concomitant FPS improvement of 2.9% across all the benchmarks compared to PAT15. Note that their work proposed linear regression-based prediction models and evaluated a small set of mobile games that have specific workload characteristics (mainly using CPU-dominant benchmarks). When we compared their approach on a larger set of varying graphics benchmarks, the performance (FPS) prediction was up to 20% worse than the default for some GPU-dominant benchmarks such as *Anomaly2 high* and *Anomaly2 normal* (the second category in Figure 16). In addition to the CPU-GPU frequency under-provisioning for some benchmarks with FPS degradation, their governor also allocates over-provisioned CPU/GPU frequencies for several benchmarks such as *Anomaly2 low* and *dreambike* which have no FPS degradation but worse energy savings than our HiCAP governor. Note that their approach [17] also had average energy savings of 16% with FPS degradation of 3% across CPU-dominant benchmarks compared to the default, but had 1% and 8% worse average energy savings with FPS degradation of 3% and 7% for No- and GPU-dominant benchmarks respectively. Based on these results analysis, we observe that characteristics of gaming benchmarks (CPU- or/and GPU-dominant and Low/High workloads on the evaluated platform) are very important factors and results of each proposal also are dependent on the characteristics of benchmarks.

In summary, our HiCAP HFSM-based dynamic behavior model for mobile gaming is able to detect a specific state in terms of FPS-target and CPU/GPU workload; thus HiCAP is clearly able to eliminate CPU/GPU frequency over-provisioning using a simple frequency capping technique. Therefore, our HiCAP manager was able to achieve better results in energy saving, with minimal FPS degradation (and sometimes better FPS) compared to the default governor, as well as the most recent research efforts [17] [16].

5. CONCLUSION

In this paper, we proposed *HiCAP: a Hierarchical FSM-based Dynamic Integrated CPU-GPU Frequency Capping Governor for Energy-Efficient Mobile Gaming*. HiCAP exploits the inherent hierarchical behavior of mobile games through an HFSM modeling approach; and uses a simple-yet-highly-effective capping technique to eliminate wasteful frequency over provisioning present in previous governors. We analyzed a large set of 37 mobile games exhibiting dynamic behaviors and developed a hierarchical FSM model that captures the games' dynamism. We then configured the CPU/GPU saturated frequency at run-time using a simple frequency-capping methodology as outputs of the HFSM state transitions. Our experimental results on the ODROID-XU3 platform across these 37 mobile games show that our HiCAP governor improves energy per frame by 18%, 14% and 8% on average compared to the default, PAT15 [17], and Co-Cap16 [16] governors respectively, with little FPS decline of 1.5% on average compared to the best performance (the default), and negligible overhead in execution time and power consumption. We believe HiCAP presents an intuitively natural way to model and exploit the dynamic behavior of mobile games, and is easily portable across newer

mobile platforms. The work presented here is our first step, with ongoing and future work addressing: 1) Proposing a scientific methodology for dynamic behaviors such as classification and regression machine learning algorithms for comparison to this heuristic, and 2) integration of dynamic thermal management in addition to cooperative CPU/GPU power management techniques. Finally, while our HiCAP methodology was targeted mainly for mobile games, we believe it can also be applicable for various other classes of CPU-GPU integrated graphics applications.

6. REFERENCES

[1] Android. monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[2] Y. Bai and P. Vaidya. Memory characterization to analyze and predict multimedia performance and power in embedded systems. In *ICASSP*, 2009.

[3] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.*, 18(6):742–760, 1999.

[4] Y. Gu and S. Chakraborty. Power management of interactive 3D games using frame structure information. In *VLSI Design*, Jan. 2008.

[5] Y. Gu, S. Chakraborty, and W. T. Ooi. Games are up for DVFS. In *DAC*, July 2006.

[6] Hardkernel. Odroid-xu3. http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu3.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program*, 8:231–274, 1987.

[8] H. Jung, C. Lee, S. haeng Kang, S. Kim, H. Oh, and S. Ha. Dynamic behavior specification and dynamic mapping for real-time embedded systems: Hopes approach. *ACM Trans. Embedd. Comput. Syst.*, 13(4):135–161, March 2014.

[9] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz. A control-theoretic approach for energy efficient cpu-gpu subsystem in mobile systems. In *DAC*, 2015.

[10] X. Li, G. Yan, Y. Han, and X. Li. Smartcap: User experience-oriented power adaptation for smartphone's application processor. In *DATE*, 2013.

[11] M. L. Loper. *Modeling and Simulation in the Systems Engineering Life Cycle*. Springer-Verlag, London, pp. 75-81, 2015.

[12] K. Malkowski, P. Raghavan, M. Kandemir, and M. J. Irwin. Phase-aware adaptive hardware selection for power-efficient scientific computations. In *ISLPED*, 2007.

[13] P. Marwedel. *Embedded System Design*. Springer, B.V, pp. 42-52, 2011.

[14] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim. Quality-aware mobile graphics workload characterization for energy-efficient DVFS design. In *ESTIMedia*, Oct. 2014.

[15] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim. Cooperative CPU-GPU frequency capping (Co-Cap) for energy efficient mobile gaming. In *UCI Center for Embedded and Cyber-physical Systems TR*, 2015.

[16] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim. Co-cap: Energy-efficient cooperative cpu-gpu frequency capping for mobile games. In *SAC*, 2016.

[17] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra. Power-performance modelling of mobile gaming workloads on heterogeneous MPSoCs. In *DAC*, 2015.

[18] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. Integrated CPU-GPU power management for 3D mobile games. In *DAC*, June 2014.

[19] S. Rabin. *GAME AI PRO*. CRC Press, LLC, pp. 47-52, 2014.

[20] O. Sahin and A. K. Coskun. On the impacts of greedy thermal management in mobile devices. *IEEE Embedded Systems Letters*, 7(2), June 2015.