Selective Event Processing for Energy Efficient Mobile Gaming with SNIP

Prasanna Venkatesh Rengasamy, Haibo Zhang, Shulin Zhao, Anand Sivasubramaniam, Mahmut T Kandemir, Chita R Das
The Pennsylvania State University.
{pur128,huz123,suz53,axs53,mtk2,cxd12}@psu.edu

***Abstract*—Gaming is an important class of workloads for mobile devices. They are not only one of the biggest markets for game developers and app stores, but also amongst the most stressful applications for the SoC. In these workloads, much of the computation is user-driven, i.e. events captured from sensors drive the computation to be performed. Consequently, event processing constitutes the bulk of energy drain for these applications. To address this problem, we conduct a detailed characterization of event processing activities in several popular games and show that (i) some of the events are exactly repetitive in their inputs, not requiring any processing at all; or (ii) a significant number of events are redundant in that even if the inputs for these events are different, the output matches events already processed. Memoization is one of the obvious choices to optimize such behavior, however the problem is a lot more challenging in this context because the computation can span even functional/OS boundaries, and the input space required for tables can takes gigabytes of storage. Instead, our Selecting Necessary InPuts (SNIP) software solution uses machine learning to isolate the input features that we really need to track in order to considerably shrink memoization tables. We show that SNIP can save up to 32% of the energy in these games without requiring any hardware modifications.**

I. Introduction

Gaming is an important application domain with an estimated 2.3 billion users playing games on their mobile devices [1]. Games are extremely demanding in terms of their hardware impositions, performance requirements, and user interactions. When running a game, the CPU needs to constantly interact with a diverse set of hardware units – main memory, codecs, SD cards, GPUs, displays and the network – to interactively respond in real-time to a continuous stream of user inputs coming from input sensors such AR/VR wands, screen, accelerometer, gyroscope, camera, etc. Even if one were to offload much of the main processing to a back-end server on the cloud, these hardware units still consume a significant amount of energy, often leading to a drain on the limited battery capacity in the mobile device. It has been noted that heavy-weight games can drain the entire battery of a mobile device in a couple of hours [2]. This paper looks to develop a holistic solution to reduce end-to-end energy consumption of the entire mobile device rather than a piece-meal solution for any single component.

The main idea of our solution is *selective event processing*, rather than reacting to every event. Gaming applications are inherently event driven, with user input continuously (generated by numerous sensors) driving the computation. The applications need to prepare and react to each such event, which can result in significant energy consumption. Instead, if we are selective about which event will really impact the game behavior, we could avoid unnecessarily (re-)processing thousands of events. Such redundant processing can happen due to two classes of events: (i) *Repeated Events*: When the exact same events keep recurring, the consequent actions/impact are also usually repetitive. For instance, if a game registers for a screen swipe or a button press event (with the OS), and during execution, the user keeps pressing the same button again and again, the application may not need to react to every subsequent press. Since user inputs are highly complex, one may expect we do find a significant number of repeated events. However, our study shows that there were only around 2-5% of such repeated event executions across a spectrum of 7 games. Upon closer examination, we find that "exact" repetition has a lower probability, as opposed to close enough inputs/events that eventually result in the exact/same game behavior. This leads to the next category of (ii) *Redundant Events*: These events, though they may not exactly match to prior occurrences, they still do not impact the application execution when they occur. For instance, a game that reacts to rotation/gyroscope events for some windows of execution (say for switching from portrait to landscape), may need to react only for significant movement of the device as opposed to minor movements (which can be largely ignored). Our characterization of 7 different top chart games from the Play Store show that, anywhere from 17% to 43% of the events processed fall in the latter redundant category, not needing any processing at all. Our solution is intended to avoid processing both kinds of events, which can result in multiple hardware component energy savings.

One of the previously proposed techniques for dealing with redundancy/repetition is memoization. Essentially, we identify frequently executing computations for which the input values repeat, and maintain a table mapping these input values to the corresponding output produced by the computation. Subsequently, when the same input occurs for this computation, the entire processing can be "snipped" by simply substituting it with the output from the table. This popular technique has drawn applicability at the instruction level (to ease functional unit pressure) [3, 4], or even at functional levels [5] to reduce computation. Our SNIP – Selecting Necessary InPuts – solution is similar to this strategy with the following key differences: (i) We do not stop at single Instructions or even functional granularity for memoization. Instead, our solution tries to snip the entire sequence of instructions, which could potentially span multiple functions and even application-OS boundaries, that are driven by the event we are targeting to avoid processing; (ii) Apart from reducing the overhead of

fetching and executing these instructions, SNIP also avoids the overheads when certain parts of the computation (in event processing) are offloaded to accelerators/IPs on the mobile SoC; and (iii) As pointed above, if we are to stick to exactly matching inputs, the scope for optimization is relatively small. Instead, we also include the Redundant Events in our memoization, where even if the inputs do not exactly match, we can still snip those computation to produce an output that is no different than performing the entire computation. SNIP, thus, goes well beyond prior techniques to identify and cut-short computations for redundant events.

However, such an ambitious goal has considerable challenges. Memoization requires look-up tables, which are indexed by the inputs to find the corresponding output. When we move to such large granularity for memoization, the number of inputs becomes enormous – spanning not just registers, but also numerous memory locations. Further, each of these can have a wide spectrum of values that we need to look up the table. In fact, we find that required table sizes for the games under consideration can run into gigabytes, defeating the whole purpose of memoization. Hence, conventional approaches to doing this will simply not work.

To address the above challenges, SNIP uses the following software based approach:

- *Pre-processing/Profiling:* Mobile applications go through extensive debugging and testing, before getting deployed in the App store. We suggest another phase of profiling, which could happen either before the application is deployed (as part of the testing phases by different users), or continuously from the usage by different users as they play their games. The purpose of such profiling is to collect event data that is needed to build lookup tables as is discussed next. Thus the app is distributed along with its profile data when downloaded.
- *Shrinking the Table:* Thousands of input locations, each containing possibly millions of values, make memoization tables hard to construct. We use a machine learning technique called Permutation Feature Importance (PFI) [6, 7] which is essentially a feature extraction utility that is very useful for our needs. It trains on the input-output data for each event, to identify only a subset of "essential inputs" that is needed for a bulk of memoization needs. After this stage, numerous input possibilities will get mapped to a small subset of output values/actions. By capturing only the necessary inputs (and their values), SNIP requires less than 1% of the original table that would have been needed, while still sufficing to succinctly capture a significant number of memoization opportunities. Hence, the term SNIP, for Selecting Necessary InPuts. SNIP can be run on the cloud, not needing the mobile devices limited resources.
- *Looking up the Table and Memoization:* Note that, once compressed, we can no longer simply lookup the table with the original inputs. Instead, we compose a hash of the input for looking up the table. The output is anyway in "non-compressed" form, which we can pick up to replace the corresponding computation.

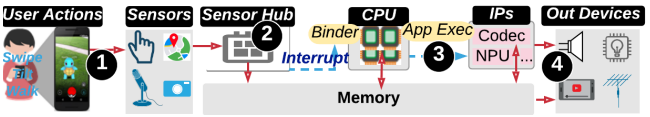


Fig. 1: Example game execution in a smart phone. The user generated events are captured at sensors, to be processed at both CPUs and IPs and finally produces the outputs back to the user.

- *Correctness issues:* Since it is a lossy compression of the table, we can get into erroneous behaviors. Although important, gaming is not a mission critical domain (unlike financial, healthcare, etc.). At the same time, we still would like to reduce errors as much as possible. Hence, we allow some over-ride mechanisms (i) Developer over-rides - where the developer can over-ride with necessary input fields that cannot be left out in the compressed form, and (ii) Continuous learning – where the PFI adapts to changes in user behavior by continuously learning and updating the necessary input fields.

The entire SNIP framework, comprising the profiler and compiler which snips the computation using the reduced tables, is completely implemented in software for Android OS. We have experimentally evaluated this approach using 7 most popular games from the Play store on a Google Pixel XL phone with Snapdragon 821 SoC. Results show a 32% average energy saving, which is a significant portion of battery drainage on mobile devices. Even if the profiler is not able to do a perfect job in the first attempt (to avoid errors), within a handful of usages and continuous profiling, we are able to make the executions almost completely error free.

II. Overview of Gaming Workloads

Gaming workloads perform event-driven computations to react to various user actions, gestures, etc., and render the resulting output to the user. For example, Fig. 1 shows a user playing a typical Augmented Reality (AR) game [2] on a phone, where the user swipes, tilts and walks with the device. The user's objective is to capture the various objects that are augmented into the scene that is captured continuously in the phone's camera (and simultaneously processed and displayed on its screen). To achieve this, the game uses the input data (walking, tilting, swiping, camera feeds, etc.) to process and respond back to the user. Under the hood, the gaming device captures the three events below continuously: (i) swipe action is captured using a series of touch events on the screen; (ii) tilt is captured using a series of gyro events; and (iii) walk is captured using a series of both the camera feed and the GPS position. To understand the implications of such events in the hardware, we next walk through the example in Fig. 1 and illustrate what happens in the hardware.

A. What happens in the hardware?

Towards better performance and energy efficient executions, the apps running on contemporary System on Chip (SoC) designs leverage a combination of compute units (such as general purpose CPU, GPU cores) and domain-specific accelerators/IPs (such as encoders, decoders, neural networks,

image processors), to take advantage of the spectrum of performance and energy efficiency tradeoffs offered by them. To understand how these components get orchestrated and work together during execution, we next delve into what happens in the underlying hardware during application execution. As depicted in Fig. 1, the event generation begins with the user interacting with the device. As the user interacts (e.g., swipes, walks with the phone, etc.), the corresponding sensors are read by the sensor hub (step 2 in Fig. 1), and the values of the sensors are subsequently passed on to the CPU as interrupts. The OS framework for these interrupts (e.g., SensorManager in Android [8]) processes these raw sensor values into high-level events (e.g., swipe, tilt, etc.) – that are further passed onto the game execution at the CPU through shared memory between the sensor hub's runtime and the game workload execution (step 3 in Fig. 1). This is accomplished using the Binder framework in Android [9]. The workload execution at the CPU subsequently processes these events using a sequence of event handler functions in CPU as well as accelerator/IPs and after processing, renders the outputs back to the user (e.g., display "pokemon is captured" on the screen).

In short, the CPU cores initiate and manage all the event handling and initial processing, and it subsequently offloads the heavier tasks such as frame/audio rendering, storing and batching events etc., to domain-specific microphone, display controller, codecs, GPUs and sensor hubs.

B. Characterizing the game executions

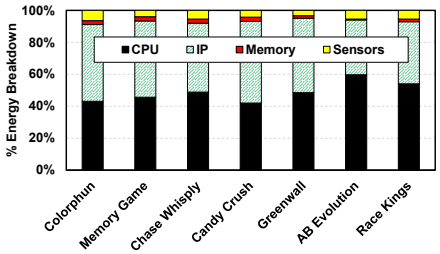


Fig. 2: Both CPU and IPs consume almost equal amount of energy.

Since there are multiple components that interact closely during game executions, we first study the normalized energy breakdown in these components (grouped as sensors, memory, CPUs and IPs) in a modern Pixel XL class phone hardware in Fig. 2. As seen, the sensors and memory consume very small portion of the total energy ($< 10\%$), while the rest is split more or less equally between the CPU and IPs. The major components of energy consumption are from the CPU and IP executions, where the CPU consumes 40% to 60% of the total energy, and the IPs also consuming 34% to 51% of the total energy.

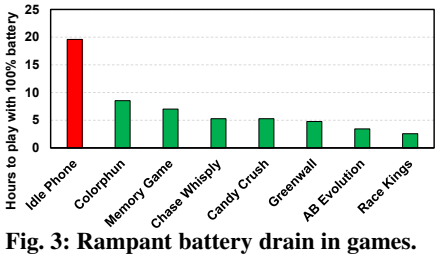


Fig. 3: Rampant battery drain in games.

While it may seem similar across all the games in the x-axis, they are actually sorted in the order of complexity of playing the game – as evident in their battery drain characteristics (same x-axis ordering in Fig. 3). For example, lightweight games such as Colorphun [10] involves an occasional touch event from the user and even this drains the battery fast ≈ 8.5 hours (vs. ≈ 20 hours for idle phone). It gets much worse as the game play gets more and more complex: AR (Chase Whisply [11]), 3D graphics games (Race Kings [12]), etc. the battery drains from a 100% charge to 0% in ≈ 3 hours ($6\times$ faster than the idle phone).

Combining the above two observations, to solve this rampant battery drain problem, we look into the "whole" SoC execution rather than optimizing for an individual component.

C. Opportunities, drawbacks, and challenges

Since the SoC is for general purpose executions, a particular game execution may not need all its hardware. In turn, we can use this domain knowledge to optimize execution. For the example in Fig. 1, the hardware execution starts from the sensors generating raw values, to the output generation at the display/speakers, etc. The opportunities are:

At the sensors: Each of the sensors have a range of values it can generate for an external user interaction. For example, a gravity/rotation sensor has value limits from 0° to 360° its x (α), y (β) and z(γ) rotation angles, that captures the accurate way in which the device is currently held by the user. However, the execution may only require whether the device is held in landscape ($\beta > 90^\circ$) or portrait mode ($\beta < 90^\circ$), and not care about the rest of the details at all. In such scenarios, as discussed in [13], one could employ a low fidelity mode for the sensors to save energy.

However, the drawback of such an optimization is that our workloads do not consume much energy at the sensors itself (Fig. 2). Optimizing at this level could result in very small energy benefits overall.

When processing an event and generating output: After the sensor values are obtained at the OS, the app event handler is invoked with the corresponding sensor data packed into an event object as arguments. To actually leverage the domain-level behavior, we first need to understand when/which the event values are useful/not useful *before processing them*. For example, a swipe-up in Fig. 1 may only be relevant when the game has a pokemon displayed for the user to swipe up and has no effect otherwise. To understand whether there is scope for such "wasted processing" in the game execution, we present the % of events that resulted in no change in the game state at all in Fig. 4.

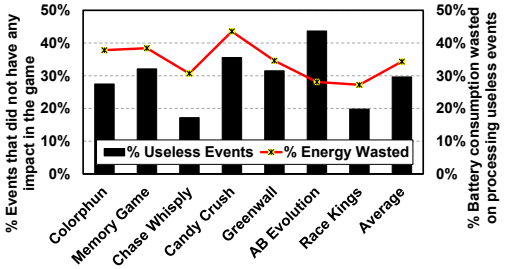


Fig. 4: % of user events captured in mobile games that resulted in the exact same output as current state after processing.

We observe that, in all the workloads, anywhere from 17% to 43% of event processing result in no output change at all, and that in turn wastes about 34% of energy in processing these events. For example, in AB Evolution game, the game play involves stretching a

catapult to release an object aimed towards a target. But, when the catapult is stretched to the maximum, no matter what the user swipe action is, it has no effect on the game. Thus, it leads to the highest useless events (43%). *If such useless events are identified before processing them, we can (i) save energy from not executing CPU and (ii) not invoke the accelerators.* To do so, we could use prior occurrences of events to track whether it resulted in output changes or not and then short-circuit subsequent occurrences. Such lookup table approaches have been studied in the past [3, 14] in the context of scientific computations. However, in the context of mobile game executions that are already draining battery, there are three main questions to determine its feasibility: *(i) are the inputs and outputs reasonably small to fit a lookup table?, (ii) are all the input/output locations known apriori?, and (iii) is there any dynamism/variations in loading inputs or generating outputs among repeating instances?*

III. Impracticality of Lookup Table Approach

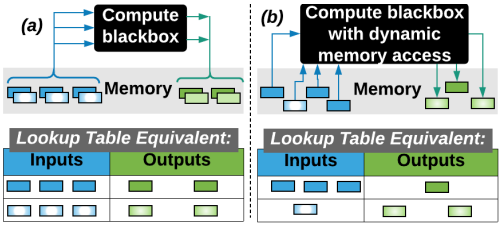


Fig. 5: Compute scenarios: (a) has deterministic locations for input/outputs; (b) has varying input/output locations and count.

Lookup tables store constant size inputs and outputs per record with the record itself consisting of the input/output values seen in prior executions. Using this history, future executions can index into a particular record based on the current input values, and get the outputs directly without actually executing the computations. For such an approach to be feasible, all the locations from where the inputs are loaded and outputs are stored should be known apriori (Fig. 5(a)). However, the various hardware components involved in the compute black box (CPU cores and IPs) in a mobile SoC often involve dynamic memory accesses during execution, as shown in Fig. 5(b) – where two instances of the computation (shown as different shades), consume varying number of inputs from varying locations, and also produce outputs to store into different locations.

To overcome the variability and still use a lookup mechanism, we consider the lookup table to contain the input values from all the possible input locations, i.e., union of all the input locations; and short-circuit the execution for all the possible outputs, i.e., union of all output locations. We study the impact of this approach for a sample game execution, AB Evolution [15], by plotting the *size* of the lookup table necessary for short-circuiting varying portions of the game execution in Fig. 6. Here, the x-axis plots the execution coverage in terms of the % of events weighted by the number of dynamic instructions each instance of the event processing executed – to account for the dynamism in context-sensitive processing.

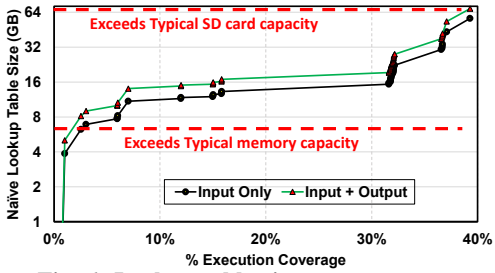


Fig. 6: Lookup table size vs. coverage.

As seen, even for short circuiting 1% of the execution, the lookup table grows to 5GB in size, while consuming the entire memory capacity (6GB) to short circuit 3%, and the entire SD card capacity (64GB) for short-circuiting 39% of the execution. This is because:

- Since this approach includes the union of all the input/output locations in each record, the sizes of the records are huge.
- In addition, the input values used by each event are not common and can have a wide range of values – resulting in millions of records in the lookup table. This is further exacerbated by the fact that games execute a large number of events, causing the lookup table to explode in volume.
- **Not utilizing the output redundancy:** As illustrated earlier in Fig. 4, up to 43% of outputs are exactly the same as prior executions. While even a one byte difference in the input record can potentially create a new entry in this lookup table approach, the outputs are still going to be the same for up to 43% of the events.

While prior works use lookup tables to optimize redundant output generations in other workload domains [3, 14], there is a clear contrast in games, where the table grows in both row size and number of rows. Thus, we next look into reducing both these aspects by exploiting the innate characteristics of input-outputs observed in game executions to identify the best heuristics to detect and short-circuit redundant computations.

IV. Input-Output Behavior of Event Processing

In order to overcome the above drawbacks of lookup table size, we need to answer the following questions: *what is in these huge input/output records?, are all these necessary to capture the redundant outputs in Fig. 4?, if not, what parts of input/output to keep? and what to trim down?* To answer, we define the categories of inputs the lookup table should contain, and use the definition to explore the feasibility of trimming the inputs/outputs of event processing, while minimizing errors.

A. Inputs Characteristics

For processing events, the execution not only takes the sensor events as input, but also the internal application state data from memory/storage, and external data from network/cloud. These three categories of input data respectively reside in different locations: Event Objects (In.Event), Previous Execution Output (In.History), and External Sources (In.Extern). To understand whether they are amenable for memoization or not, we next study their individual size/location characteristics in detail for an example execution of AB Evolution game in Fig. 7a by plotting the size spread of each of these categories in the x-axis and their cumulative % occurrences among various events processed during the game execution in y-axis.

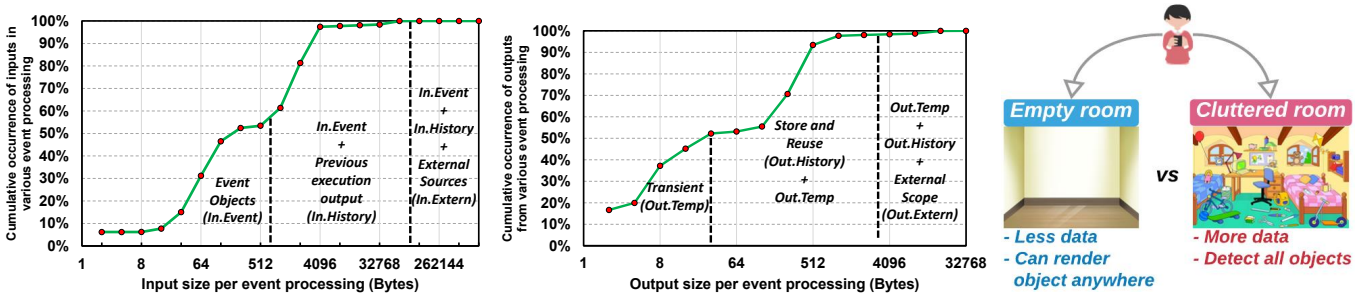


Fig. 7: Example characteristics of AB Evolution game illustrates that (a) the three types of inputs vary in sizes and vary for different instances of event processing; (b) likewise, the three types of outputs also vary similarly; and (c) the reason for such variations is the dynamism involved in these game executions.

Event Objects (In.Event): These contain the sensor values from user interactions and are passed as arguments to event handlers. Fig. 7a (x-axis) shows that the size of In.Events are relatively small and varies from 2 to 640 bytes (different event types have different sizes). While all event processing consume In.Event data, these inputs are also easily located using their object handles, and have fixed size for the same event type. For example, the event handler for detecting a change of swipes, always gets a MotionEvent object of a fixed size passed as argument to the handler – making the handler know its location in memory. **Previous Execution Output (In.History):** While In.Event data are instantaneous user interactions captured from sensors, the game needs the context involved in the execution progress. To understand the huge spread of sizes for In.History in Fig. 7a (600 bytes to 119 kB), we next use the example in Fig. 7c. Here, a user playing some AR game has two options to walk in. If it is an empty room, processing the camera feed will result in a plain surface to render the AR objects on top. Owing to the simplicity, the input data is also relatively small. On the other hand, if the room is cluttered with a lot of physical objects, the camera feed generates many options for rendering the AR objects, making the input size larger. This user input-based data size variation illustrates that this input cannot be found in static memory locations and, as seen in Fig. 7a, In.History is consumed as input in 47% of the execution.

External Sources (In.Extern): This is the data received from outside the scope of an application execution. For example, data from the cloud, network, etc., are not within the scope of the application executing inside the phone. We observe from Fig. 7a that In.Extern input is only used in the $< 0.05\%$ of the events, as most of the images/audio, etc. are read from external sources only a limited number of times during execution and are stored in memory for future (becomes In.History). Note that, the audio, images etc. are also huge in size and thus consume 1MB size of inputs in those instances of execution.

To summarize, In.Event can be used to index into the lookup table because of their ubiquitous occurrence (53%) in event processing and their fixed-size and fixed-location property. On the other hand, In.History and In.Extern do not have a fixed size and also are not statically located in the memory. Therefore, it is impractical for using them in lookup tables.

B. Shrinking the table using event data to lookup

Stemming from the previous characterization, we next study the effects of trimming the naive lookup table by using only the input fields from In.Event categories for indexing the table. Since this scheme only uses In.Event data to index into the lookup table, there is a chance that it can lead to wrong outputs. For example, if a swipe up event is generated in the example in Fig. 1, it may lead to increase in a user's game score output only when the Pokemon object is displayed (In.History category). When the object is not displayed, the swipe up event will not result in any change. By considering only the swipe event (In.Event data) to index into the lookup table, and not using the knowledge of whether a pokemon is being displayed or not (no In.History data), the memoization may sometime lead to correct outputs and erroneous outputs some other times as well.

To understand the impact of such a scheme on both execution coverage and erroneous outputs, we present the differences in employing In.Event based memoization with respect to the naive lookup table approach in Fig. 8(a). As seen, this scheme is clearly advantageous – as the lookup table size is very small (1.5% or 290 MB) compared to the approach explored in Sec. III (19GB in Fig. 6) for short-circuiting 27% of the execution. This lookup table can easily fit in the memory, and can even reside in the software [16].

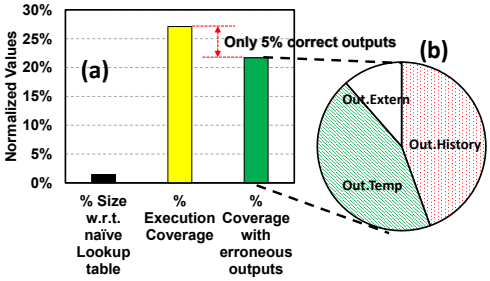


Fig. 8: (a) Using only In.Event objects for input records, the AB Evolution game's lookup table is smaller, but produces erroneous outputs; (b) The breakdown of erroneous outputs.

While this can help in improving the overall energy efficiency, this scheme can match more than one possible outputs for 22% of the total execution. Since there is no way of knowing which of these possible outputs is correct without additional input data, namely the In.History and In.Extern inputs that are not known prior to execution, this scheme

cannot be realized for short-circuiting redundant events. Thus, the clear advantage of implementing a much smaller lookup table by only indexing the In.Event data is not possible as it can result in erroneous outputs.

Note that, similar to different input categories, outputs also belong to the categories below. Fig. 7b shows that there are three categories of outputs:

- **Out.Temp:** Temporary responses from a game to the user such as a displayed frame block, vibrate/haptic feedback etc., are categorized as Out.Temp. Even if this category of outputs is short-circuited to a wrong output value, the execution itself does not get affected except for the particular user reaction. This could still go unnoticed by the user and can result in expected correct execution progress. Note in Fig. 7b that, these outputs are usually $<$ 64 bytes in size. For example, there may be a tile in the displayed frame for the user that is wrong due to an erroneous output from this In.Event based lookup table. Since 60 frames or higher [17, 18] are displayed per second in these devices, one frame's tile being wrong will have little to no impact on the user as well (displayed for $<$ 16ms – while the user's reaction time is $\approx 10\times - 20\times$ slower [19]).
- The other two categories, **Out.History and Out.Extern** compliment the In.History and In.Extern respectively, i.e., Out.History outputs are used as inputs in subsequent event processing and Out.Extern are outputs sent to the cloud/network, etc. Therefore, if we short-circuit either of the **Out.History or Out.Extern** outputs wrong, the execution itself risks becoming erroneous as these outputs are used subsequently as inputs to future executions. Thus, as long as the erroneous results of this approach is not in these two output categories, it could still be a useful tool for identifying and short-circuiting redundant executions, albeit with wrong Out.Temp outputs.

Fig. 8(b) shows the breakdown of erroneous outputs produced as a result of this scheme and as seen, 44% of the erroneous executions are Out.Temp, and so, even if it has errors, it will only lead to minimal quality degradation to the user. On the other hand, the remaining 56% of erroneous executions fall into the other two output categories (Out.History and Out.Extern) being wrong, and so, the scheme cannot be viable to short-circuit redundant event processing.

We next analyze how such erroneous executions can be avoided by augmenting this mechanism to still take advantage of a relatively small lookup table.

V. Selecting Necessary InPuts (SNIP)

Motivated by the fact that $\approx$ 600 bytes of In.Event fields of the 1MB input data are enough to short-circuit 14% of the execution, we further investigate *whether there are other "influential" input fields from In.History and In.Extern categories that can help avoid erroneous outputs or not?*. Since there is no specific fixed location or fixed size known for these most-influential/necessary input fields to identify, we actually need to search the mega bytes of inputs that determine the correct outputs when short-circuiting executions. Therefore, finding necessary input fields could involve much complex techniques such as scouring through gigabytes of profile data. We next address this problem with our proposed SNIP approach.

A. Identifying necessary inputs

While dataflow analysis techniques such as [4, 20] traverse through the dataflow graphs within a function or basic blocks and find the necessary inputs for every output, such schemes do not scale well to analyze executions spanning multiple function calls, OS, and IP invocations, that are a common occurrence in mobile game executions. Fortunately, scouring big data to identify necessary fields are well known in the machine learning domain, where mature techniques such as Permutation Feature Importance (PFI [6, 7]) have already been employed. In the context of identifying necessary input fields, the PFI takes the lookup table described in Sec. III, and trims it down by identifying a subset of input fields that is most influential in accurately short-circuiting the output fields. Towards achieving this, PFI searches through different random permutations of input fields and measure the % output fields that resulted in erroneous values. By repeating this process for different permutations of input fields, it identifies the permutation of input fields (usually a small subset of the input fields), that results in the least erroneous outputs.

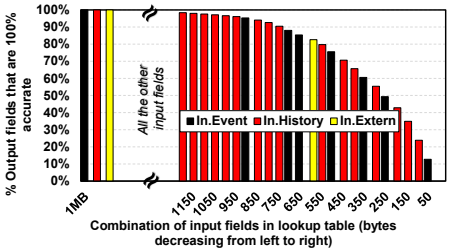


Fig. 9: Permutation Feature Importance [6] identifies the most influential input fields from all input categories.

Our goal is to trim down the number of input fields to the bare minimum, while being able to short-circuit without errors. Towards this, Fig. 9 demonstrates an example execution of the PFI approach to identify necessary input fields for AB Evolution game, where it starts with the complete set of input fields (left most bar in the x-axis = 1MB size) to short-circuit all the output fields with 100% accuracy, akin to the naive lookup table approach. Moving from left to right in x-axis, PFI iteratively trims the input fields further and further with not much loss of accuracy among the outputs short-circuited (y-axis) at first – where just 1% of output fields are erroneous even with the input fields getting trimmed down to 1200 bytes. After 1200 bytes however, the error rate rapidly increases – approximately 1% for every $\approx$50 bytes of trimming. These 1.2kB constitute the most necessary input fields for this application.

To understand what category of inputs constitute these necessary input fields, we also color-code the category of inputs that got trimmed down from the previous input permutation to the left, that resulted in the corresponding decrease in the % of outputs that can be short-circuited with 100% accuracy. The right most bar belongs to In.Event category, indicating that just 50 bytes in In.Event category can short-circuit 12% of the output fields with 100% accuracy. Similarly, PFI also automatically identifies around 1kB of input fields from In.History

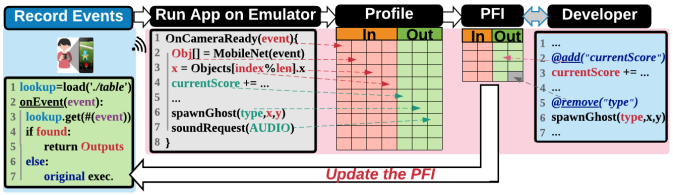


Fig. 10: Overall flow of the proposed methodology

category at various points of x-axis to be necessary for short-circuiting the execution. It also identifies some of the fields from In.Extern category as necessary. In total, these fields only represent $\approx$ 1.2kB of data (approximately 0.2% of the total input fields), and can predict 99% all of the outputs in the event processing with 100% accuracy. On the other end of the spectrum, using PFI approach can disregard all of the remaining 99.8% of the input fields with only 1% of the output fields with erroneous values. And, in order to short-circuit the 1% of output fields with 100% accuracy, we also need all the remaining input fields in the lookup table. As mentioned earlier, we can still tolerate erroneous executions if all the 1% of the erroneous output fields belong to the Out.Temp category.

Towards using PFI for picking all necessary inputs, we need to ensure that SNIP to address the challenges below:

- **Minimizing overheads at the mobile phone:** PFI trains on profile data that typically exceeds the total storage capacity of a mobile phone (Fig. 6). Thus, we need to employ mechanisms with minimal overheads to transfer the profile data to an offline cloud and only get the necessary input fields back from the cloud.
- **Dealing with correctness from profile:** Since the necessary input combination produced by PFI can also have certain a % of erroneous output fields (for just 1% of the outputs), if those fields belong to Out.History category, it can subsequently cause the whole execution to be erroneous.
- **Dealing with correctness at runtime:** Since PFI operates on profiled data, it is not clear if the profile captures all the scenarios/input variations that can occur during execution. In case of an insufficient profile, PFI can miss learning some of the necessary input fields, which can result in a higher % of erroneous outputs.

B. Methodology

Fig. 10 shows the overall flow of the proposed methodology with the following steps:

Record and send events to cloud: The first step in SNIP is to record the different inputs and outputs of event processing observed when the user is playing a game. This can be done either during the rigorous testing phases involved in app development [21] or continuously when users play the game. We describe the latter approach in Fig. 10. As recording the input-output of event processing is data intensive (Fig. 6), SNIP records only the event inputs and send them to the cloud. To trace these event data, future android versions can instrument the Binder instances from Android HAL to dump all the events into a background service similar to logcat [22]. In our experiments, we use the debug features of Android Studio to record these events of the phone connected to an Android Studio [23] host. Since camera is not a part of the sensorhub, we leverage the existing screen record features in the debuggers to capture the camera feed.

Run app on AOSP Emulator and build the profile: At the cloud, we use an offline profiler based on the AOSP emulator setup [23] running the game app with input events from the previous step in Fig. 10. In order to capture the input-output behavior accurately, the recorded events are fed in the same manner as if the user is playing the game once again in the emulator using additional tools such as [21, 23, 24]. During this emulation, we dump the input and outputs consumed by the event processing (across various game execution threads) by instrumenting the emulator to record memory traces [24, 25] along with additional information about the source of the data accesses (e.g., CPU instruction, IP, sensor hub, etc.). To achieve this, we use various tools in the existing Android framework such as heap profiler [26] to capture the whole memory dump and function call execution trace from Android Studio. On top of the traces, we use a set of analysis tools such as apat [27], hprof and dmtrace [23] to process the memory dumps and obtain the exact input and output addresses, the data in those addresses (from the memory dump) and the call stack.

PFI gets necessary inputs: Once the input-output data is available from the previous step, SNIP runs the PFI technique on the profiled data to get the necessary input fields. To ensure correctness during execution, SNIP allows two options:

Option 1: Developer intervention: This option is useful when PFI is applied on the testing phase of app development, where the necessary input fields from PFI, can be fed back to the app developers for corrections as shown in Fig. 10. As seen, the necessary input fields are mapped to the source code's variables using the additional information tracked during the above profile phase, and the app developers can fine tune the necessary inputs by adding more necessary inputs and/or marking Out.Temp variables that can tolerate errors.

Option 2: Continuous learning: Instead of statically fixing the necessary inputs for an app during the development stage, SNIP also facilitates continuous learning by just looping through the initial steps (without developer intervention stage) by recording events, building the profile and developing a PFI based lookup approach repeatedly when the user is playing the game. This option is more generic than the developer intervention, as it allows to fix any short comings due to insufficient profile data. We will demonstrate in Sec. VII that this continuous learning can effectively adapt and control the erroneous executions based on user behaviors.

User feedback vs. Options 1 and 2: While SNIP already could use developer feedback to decide on whether an erroneous output field will impact the execution, user experience rating can also be taken as feedback to understand this impact as examined in [28]. In this paper, we have limited the scope to energy efficiency, without unduly impacting user experience. We will conduct user experience studies in the future.

Using the lookup table during execution: After the PFI based lookup table is built, it contains only the necessary

inputs and is subsequently sent to the device as an over-the-air update, along with additional code instrumentation as shown in Fig. 10. As seen, the lookup table is loaded as a hash table during app initialization. During execution, on any event, the table is indexed with the event hash-code and if hit, all the other necessary inputs are loaded and compared against the corresponding important input entries in the lookup table. If the comparisons lead to a match, the execution is directly short-circuited. Else, process the event as baseline.

Thus, SNIP could achieve minimal overhead, and also tunes towards a user's game play to potentially short-circuit all the redundant event processing. We next evaluate whether SNIP could actually be beneficial to game executions or not (in comparison to baseline and state of the art), and if not, how to minimize the short comings.

VI. Experimental Setup

A. Game Workloads

We consider a mix of both open source and off the shelf games from Play store with a mix of input data characteristics as described below, to study the effects of SNIP on a wide spectrum of game workload execution behaviors. All these apps are consistently ranked as top games in Play Store [29].

Simple Touch based games: Simple In.Event based games such as Colorphun [10] and Memory Game [30] involve the user to touch specific places on the display to score and make forward progress. These games are also light on graphics and compute components, and mainly use CPU and display for most of their execution.

Swipe based games: Games such as Candy Crush [31] and Greenwall [32] (open source version Fruit Ninja [33] game) involve swipe as the major In.Event input, and also have more animations in the game outputs compared to the simple touch based games. These games also display more components on the screen compared to touch based games, with which the users can interact, performs animated reactions, and more computations in each event processing as well.

Multi In.Event games: AB Evolution or Angrybirds Evolution [15], Chase Whisply [11] and Race Kings [12] games have much more complex In.Event objects involving drag, tilt, and multi-touch events. Unlike the above four games, these games also use 3D rendering in the screen with the GPU heavily involved to process the events, that involve heavy physics computations [34]. In addition to other IPs, ChaseWhisply also uses the camera feed continuously to render AR objects in it and display them to the user.

B. System Setup

All our studies and experiments are conducted in a Pixel XL class mobile device, that has a Qualcomm Snapdragon 821 SoC [35] containing Quad-core Kryo CPUs, a 4 GB LPDDR4 memory and a 32 GB internal storage, and is powered by a 3450 mAh battery. Using this hardware for our experiments has two specific objectives, namely, (i) measure the energy consumption of the different hardware components in the app execution; and (ii) record the event data during app execution to subsequently send to cloud to follow the steps in Sec. V. We next describe the system setup to achieve the objectives.

Measuring the energy at hardware components: While there are multiple ways of measuring energy at the hardware [36–38], we use Qualcomm's Trepn power monitor app [37] installed in the phone, that can tap into any process or the whole system execution to collect detailed stats on battery consumption, CPU, memory, GPU, and other hardware usage. To record individual components' energy consumption, we deploy specific microbenchmark apps to use only specific components, namely, CPU, CPU+memory, display, sensors, camera, audio and video codecs and measure their power consumption using the Trepn app. With this system, any game's events recorded (using the process described next) can be plugged-in with the power consumption of the different components to get a detailed time series view of component level energy consumption in the course of execution.

To compute the duration taken to drain a 100% charged battery, we also make use of the above system to measure the game play's power consumption behavior over a duration of $\approx$ 5-10 minutes, to calculate how long the execution will take to consume 3450 mAh (100% battery capacity).

Recording the events in game execution: In order to record all the events occurring during execution (as described in Sec. V-B), we customize the Android OS to log all the event data occurring in the execution. In our experiments, we connect the phone to an Android Debugger [39] client and collect the event logs directly and use it for building the profile with the setup described earlier. However, we envision that the system will be able to transfer the event logs to cloud from any smartphone in the future. While capturing all the sensor activities and events are straightforward (by instrumenting binder threads), capturing the camera feed is a special case because of how the hardware is built in modern SoCs [35]. For tracking camera events, we run a screen record process that simultaneously record the camera feed into a video file that are sent to the cloud for building the PFI. In the future, the screen record feature in upcoming Android versions [40] can be leveraged to accomplish this across all apps. We next use this experimental setup to study the effectiveness and drawbacks of SNIP.

VII. Results

Example Code	Max CPU [3, 41, 42]	Max IP [43]	SNIP
Event			
$CPUFunc_1()$	Yes	No	Short-circuit
$CPUFunc_2()$	Yes	No	the whole
$CPUFunc_k()$	Yes	No	execution
$IP_1()$	No	Yes	
$CPUFunc_{k+1}()$	Yes	No	
$IP_2()$	No	Yes	
Output			

TABLE I: Example Code in Games and what parts can be optimized by the prior works.

To evaluate the benefits from SNIP, we next list comparison points from prior works and best case scenarios, that specifically test the different aspects of our proposal namely, optimizing only the CPU part [14, 42], optimizing only the IP part [43] and the lookup table overheads. They are:

- **Max CPU:** To study the effects of short-circuiting the CPU computations alone using prior approaches such as [3, 14, 42] for game executions, and also understand the energy gap between optimizing just for CPU execution (example

in Table I) vs optimizing the whole SoC in SNIP, we present the Max CPU scheme. Note that, this scheme also assumes quantifies the maximum benefits from techniques such as [42] which assumes all data to be known apriori (recall from Fig. 5(a)), whereas the game execution needs our proposed lookup table solution to find all inputs apriori.

- **Max IP:** Prior approaches such as [43] show that IPs can be switched to sleep states when they are idle. This scheme studies the impact of such techniques in game executions (example in Table I) and also quantifies the gap between short-circuiting just the IP calls vs the whole event processing in SNIP.
- **SNIP:** This is our proposed technique, where both CPU and IPs can benefit from avoiding redundant event processing and hence both IPs and CPUs can save their energy.
- **No Overheads:** This scheme follows the exact same optimization steps of SNIP. In addition, it does not incur any overheads in terms of lookup table costs and comparisons on each input event processing and shows the scope for future optimizations in this domain.

A. Energy benefits and overheads of SNIP approach

We next discuss the energy benefits from the schemes described earlier and the reason for these benefits in Fig. 11. First, Fig. 11a shows the energy benefits for all the schemes w.r.t baseline execution, where we observe that both Max CPU and Max IP have limited energy benefits in games, where Max CPU can save 0.5% (Chase Whisply) to 13% (Race Kings), and Max IP saves 0.7% (Memory Game) to 9% (Candy Crush) in terms of energy. In contrast, by taking both SNIP can benefit anywhere between 24% (Race Kings) to 37% (AB Evolution) of the event processing energy, translating to an extra battery life of 1.6 hours on an average and a maximum of 2.6 hours in Colorphun game. This benefit is mainly from the better opportunity to short-circuit the event processing end-to-end instead of optimizing only certain parts of the execution as shown in the example code in Table I. For example, Max CPU can only optimize repeated $CPUFunc_i$ and not the IP_i calls and Max IP can optimize for only the IP_i invocations. Quantitatively, Fig. 11b shows the % of executions that can be short-circuited by each of the above schemes. As seen, Max CPU and Max IP could potentially short-circuit a maximum of 26% and 15 % of the execution for Colorphun but the energy gains from Colorphun is just 0.6% and 5% respectively. This is primarily because Colorphun game is already a light weight application, and even the overheads for looking up the necessary inputs (Fig. 11c) compares 7.5kB of data on every event. On the other hand, SNIP can potentially short-circuit anywhere between 40% (Race Kings) to 61% (Candy Crush) of the execution with an average scope for short-circuiting 52% of the execution – that translates to 32% average energy savings (or 1.6 hours of extra battery life).

Note that, SNIP approach also has additional overheads as seen in Fig. 11c, where it needs to load a lookup table (memory operations) and compare against each and every necessary input for that event (Comparisons $\times$ PFI Input Size) in the table in order to find when to short-circuit the execution. In order to measure this overhead, we also present SNIP scheme without any overheads from these comparisons to save additional energy of anywhere from 1% (Colorphun, Greenwall, and AB Evolution) to 3% (Race Kings) with the exception of 12% in Memory Game – due to the high amount of comparisons for each event processing. On an average, the overheads in SNIP approach can consume 3% of the execution energy – indicating the PFI based Selecting Necessary InPuts scheme to be viable, software based alternative compared to the traditional memoization approaches.

B. Continuous learning to avoid developer intervention

The above analysis assume the profile and the developer instrumentation to accurately capture all necessary input fields for the execution to result in correct execution. However, in practical purposes prior studies such as [44] show that users generate vastly different events/inputs and it is important for any event learning approach such as PFI in SNIP to fine tune the learning continuously (Option 2 in Sec. V-B).

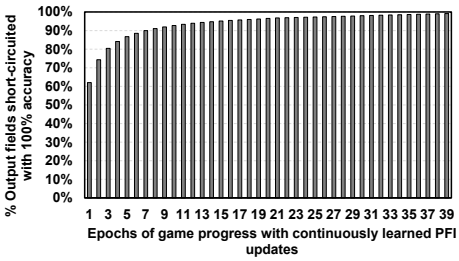


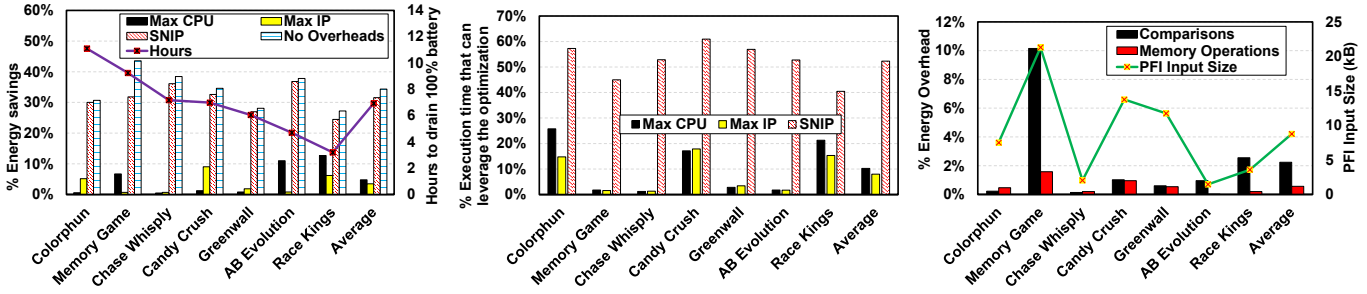
Fig. 12: Avoiding developer intervention is possible with adaptive, continuous learning.

Fig. 12 shows the PFI detection can adapt to recover itself from erroneous short-circuits by re-learning the user behavior continuously. We plot the different instances of the same user playing a sample game in x-axis, and plot the % erroneous output fields in y-axis as a result of short-circuiting using SNIP without developer intervention for AB Evolution game. In this experiment, we artificially keep the initial few iterations of the profile to be insufficient for PFI to not capture all the necessary input fields of the subsequent execution. Therefore, we also observe the initial few iterations of short-circuiting using PFI to be approximately 40% erroneous for the first few instances of execution. However, as more and more instances of user events get to the cloud, a more accurate lookup table is built. Thus, after a few initial bad runs, the % of erroneous output fields get to $< 0.1\%$ in just 40 training epochs.

To avoid developer intervention (Option 1 in Sec. V-B), one could train the PFI model and test on subsequent event records till a confidence threshold is reached. This could ensure that the user will only start experiencing PFI based short-circuiting when the % of erroneous output fields is negligible. As a future extension, the profiler can also direct the mobile phone to "clear" the PFI lookup table if it detects the error rate to worsen (although not observed in our experiments), as well as receive user feedback on the quality of execution – to even "turn off" SNIP feature.

C. Overheads and limitations

For SNIP to be effective, the mobile phone should just



(a) (b) (c)

Fig. 11: (a) Energy benefits using various schemes; (b) The % execution that can leverage each of the optimizations; (c) The overheads in SNIP are due to the extra energy spent at the CPU and memory for looking up the table before each event processing.

capture and send the user events to the server. Therefore, at the client side, the event collection overhead is negligible. However, at the backend, processing 2 minutes game play to get a sleek lookup table could take around 2 days of processing in a 48 core, 64GB memory Xeon E5 class server. In this process, the lookup table size gets shrunk from 100s of GBs to 600MB. While this back-end overhead is costly, techniques such as federated AI [45] can be explored as future research directions for reducing the backend overheads as well as performing collective learning.

VIII. Related Work

We next discuss the related works to SNIP below:

A. Memoization

Prior works such as [3, 14, 46] have built look up tables (both in hardware and software) for short-circuiting such repeated computations in the CPU execution contexts for scientific workloads. For example, [3, 14] use hardware table to short circuit data flow graphs based on register information, [16, 47] uses a software based compiler and runtime memoization engine, [42] replaces frequently executed hot codes with a trained CNN engine to approximately short circuit the execution. However, as the example code in Table I illustrates, these prior works cannot be directly adopted to fully exploit short-circuit the end to end execution from event generation and all the nested function calls crossing app/OS/IP invocations occurring in a mobile game execution.

B. Mobile SoC Optimizations

In mobile SoC, many prior works such as [13, 43, 48] target optimizations towards a single component such as CPU [49–51], video codecs [52, 53], sensors [54, 55], interconnects [48], memory [56], neural/vision processing [57] and battery [58, 59], and while considering the whole SoC, works such as [56] aim to meet the QoS requirements of frame based apps by reorganizing the IP scheduling. The most related work to our proposal is [43] where individual IPs are aggressively switched to sleep states when they are idle, to save energy. While SNIP also aims to conserve energy of the whole SoC, it creates more opportunities by exploiting the significant occurrences of redundant event processing and snips the computation to get the outputs directly.

C. ML based Optimizations

ML is emerging as a useful tool to optimize different parts of the system such as prefetchers [60], branch predictors [61], approximating executions [42] and scheduling [58, 59] in the recent years. Particularly techniques such as [58, 59] are already implemented in mobile phones, and they focus on better user interactions, manage the battery as per user behavior, etc. by training appropriate ML models for them. However, Android battery optimization techniques are not domain specific and just learn to suspend/kill idle threads in the whole system. SNIP exploits the redundant events in these games to bring additional gains on top of the existing energy savings from the Android battery optimization techniques.

IX. Conclusions

Although gaming is a widely popular domain of applications in mobile phones, these applications drain the battery much faster than many other classes of applications. This is primarily because of the user-driven interactive mode of operation, where the generated events continuously stress the SoC. In this paper, we propose a software solution, called SNIP, to minimize the energy consumption by exploiting the repetitive nature of inputs and outputs. While memoization can identify and short-circuit redundant events, the event processing involves multiple function calls spanning to even OS and IP invocations and hence, the lookup table size becomes prohibitively large. Our proposed solution SNIP uses a machine learning technique on the execution profile, to trim down the lookup table size by keeping only a small subset of necessary inputs needed to generate correct outputs. The complete SNIP design consists of a lightweight event tracker at the smartphone, a cloud based offline profiler, a Permutation Feature Importance (PFI) module to trim down the lookup table size, and subsequent compiler based code instrumentation to leverage the PFI lookup table during execution. We have implemented and evaluated SNIP approach on a Pixel XL phone and observe that we can save 32% energy in 7 popular games while being almost completely error free.

X. Acknowledgements

This research is supported in part by NSF grants 1763681, 1629915, 1629129, 1317560, 1526750, 1714389, 1912495, and a DARPA/SRC JUMP grant.

References

[1] statista, "Number of active mobile gamers worldwide from 2014 to 2021 (in millions) ," "https://www.statista.com/statistics/748089/number-mobile-gamers-world-platform/", 2018, [Online].

[2] I. Niantic, "Pokémon GO," https://play.google.com/store/apps/details?id=com.nianticlabs.pokemongo, 2019.

[3] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," *SIGARCH Comput. Archit. News*, pp. 194–205, 1997.

[4] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam, "SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 40, no. 5, pp. 95–105, Oct. 2006.

[5] A. Moshovos and G. S. Sohi, "Microarchitectural innovations: boosting microprocessor performance beyond semiconductor technology scaling," *Proceedings of the IEEE*, pp. 1560–1575, 2001.

[6] L. Breiman, "Random forests," *Machine learning*, pp. 5–32, 2001.

[7] A. Fisher, C. Rudin, and F. Dominici, "All Models are Wrong but many are Useful: Variable Importance for Black-Box, Proprietary, or Misspecified Prediction Models, using Model Class Reliance," *arXiv e-prints*, p. arXiv:1801.01489, Jan 2018.

[8] Android, "android.hardware.SensorManager," https://developer.android.com/reference/android/hardware/SensorManager, 2019, [Online].

[9] Google, "Binder," "https://developer.android.com/reference/android/os/Binder", 2019, [Online].

[10] P. Srivastav, "A Simple Color Game in Android," https://github.com/prakhar1989/ColorPhun, 2019.

[11] tvbarthel, "Chase Whisply - Beta," https://play.google.com/store/apps/details?id=fr.tvbarthel.games.chasewhisply, 2019.

[12] H. Games, "Race Kings," https://play.google.com/store/apps/details?id=com.hutchgames.racingnext, 2019.

[13] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "RedEye: Analog ConvNet Image Sensor Architecture for Continuous Mobile Vision," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016, pp. 255–266.

[14] S. Balakrishnan and G. S. Sohi, "Exploiting value locality in physical register files," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2003, pp. 265–276.

[15] R. E. Corporation, "Angry Birds Evolution," https://play.google.com/store/apps/details?id=com.rovio.tnt, 2019.

[16] H. J. M. M. van GoghBrian C. Beckman, "Automatic and Transparent Memoization," 2012, US Patent 8,108,848 B2.

[17] OnePlus, "Never Settle – OnePlus," https://www.oneplus.com/, 2019.

[18] Oculus, "Play the Next Level of Gaming," https://www.oculus.com, 2019.

[19] A. Jain, R. Bansal, A. Kumar, and K. Singh, "A comparative study of visual and auditory reaction times on the basis of gender and physical activity levels of medical first year students," *International Journal of Applied and Basic Medical Research*, p. 124, 2015.

[20] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The Load Slice Core microarchitecture," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 272–284.

[21] Android, "Monkeyrunner," https://developer.android.com/studio/test/monkeyrunner/index.html, 2017.

[22] A. Studio, "Logcat command-line tool," https://developer.android.com/studio/command-line/logcat, 2019.

[23] Google, "Android Studio," https://developer.android.com/studio/index.html, 2017.

[24] S. J. Patel and S. S. Lumetta, "rePLay: A Hardware Framework for Dynamic Optimization," *IEEE Transactions on Computers*, pp. 590–608, 2001.

[25] N. Chidambaram Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. Kandemir, and C. R. Das, "GemDroid: A Framework to Evaluate Mobile Platforms," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2014.

[26] Google, "View the Java Heap and Memory Allocations with Memory Profiler," https://developer.android.com/studio/profile/memory-profiler, 2017.

[27] Madvay, "Android analysis tools," https://github.com/madvay/android-analysis-tools, 2015.

[28] Y. Zhu, M. Halpern, and V. J. Reddi, "The Role of the CPU in Energy-Efficient Mobile Web Browsing," *IEEE Micro*, pp. 26–33, 2015.

[29] Android, "Top Charts – Android Apps on Google Play," https://play.google.com/store/apps/top/category/GAME, 2019.

[30] R. Kushnarenko, "Simple and Beautiful Memory Game for Kids," https://github.com/sromku/memory-game, 2019.

[31] K. L. Aragon, "Candy Crush Saga," https://play.google.com/store/apps/details?id=com.king.candycrushsaga, 2019.

[32] T. Messick, "A Weirdly Addictive Arcade-style Android Game, Where you Fling Fruit at a Wall," https://github.com/awlzac/greenwall, 2019.

[33] H. Studios, "Fruit Ninja," https://play.google.com/store/apps/details?id=com.halfbrick.fruitninjafree, 2019.

[34] U. Engine, "What is Unreal Engine 4," https://www.unrealengine.com, 2019.

[35] Qualcomm Inc, "Qualcomm Snapdragon 821 Mobile Platform," https://www.qualcomm.com/products/snapdragon-821-mobile-platform, 2019.

[36] "Inspect energy use with profiler," https://developer.android.com/studio/profile/energy-profiler, 2019.

[37] Qualcomm Inc, "Trepn Power Profiler," https://developer.qualcomm.com/forums/software/trepn-power-profiler, 2019.

[38] Android, "android.os.BatteryManager," https://developer.android.com/reference/android/os/BatteryManager, 2019.

[39] Android Studio, "Android Debug Bridge (adb)," https://developer.android.com/studio/command-line/adb, 2019.

[40] A. Authority, "Android Q has a native screen recorder, but it's pretty rough for now," https://www.androidauthority.com/android-q-screen-recorder-965837/, 2019, [Online].

[41] A. Sodani and G. S. Sohi, "An Empirical Analysis of Instruction Repetition," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998, pp. 35–45.

[42] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012, pp. 449–460.

[43] N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Domain knowledge based energy management in handhelds," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2015, pp. 150–160.

[44] Google, "Quick, Draw! The Data," https://quickdraw.withgoogle.com/data, 2019.

[45] G. A. Blog, "Federated learning: Collaborative machine learning without centralized training data," https://ai.googleblog.com/2017/04/federated-learning-collaborative.html, 2017.

[46] D. A. Conners and W. W. Hwu, "Compiler-directed dynamic computation reuse: rationale and initial results," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1999, pp. 158–169.

[47] Y. Ding and Z. Li, "A compiler scheme for reusing intermediate computation results," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004, pp. 277–288.

[48] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Vip: Virtualizing ip chains on handheld platforms," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 655–667.

[49] Prasanna Venkatesh Rengasamy and Haibo Zhang and Nachiappan Chidhambaram Nachiappan and Shulin Zhao and Anand Sivasubramaniam and Mahmut Kandemir and Chita R Das , "Characterizing Diverse Handheld Apps for Customized Hardware Acceleration ," in *In Proceedings of IEEE International Symposium on Workload Characterization*, Oct 2017.

[50] P. V. Rengasamy, H. Zhang, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Critics critiquing criticality in mobile apps," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 867–880.

[51] D. Boggs, G. Brown, B. Rozas, N. Tuck, and K. S. Venkatraman, "NVIDIA's Denver processor," *HOTChips*, pp. 86–95, 2011.

[52] N. Ickes, G. Gammie, M. E. Sinangil, R. Rithe, J. Gu, A. Wang, H. Mair, S. Datla, B. Rong, S. Honnavara-Prasad, L. Ho, G. Baldwin, D. Buss, A. P. Chandrakasan, and U. Ko, "A 28 nm 0.6 V Low Power DSP for Mobile Applications," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, 2012.

[53] Haibo Zhang and Prasanna Venkatesh Rengasamy and Shulin Zhao and Nachiappan Chidambaram Nachiappan and Anand Sivasubramaniam and Mahmut Kandemir and Ravi Iyer and Chita R. Das, "Race-To-Sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds ," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Oct 2017.

[54] M. Norris, B. Celik, P. Venkatesh, S. Zhao, P. McDaniel, A. Sivasubramaniam, and G. Tan, "Iotrepair: Systematically addressing device faults in commodity iot," in *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 2020, pp. 142–148.

[55] S. Zhao, P. V. Rengasamy, H. Zhang, S. Bhuyan, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, and C. Das, "Understanding energy efficiency in iot app executions," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 742–755.

[56] Haibo Zhang and Prasanna Venkatesh Rengasamy and Nachiappan Chidambaram Nachiappan and Shulin Zhao and Anand Sivasubramaniam and Mahmut Kandemir and Chita R. Das, "FLOSS: FLOw Sensitive Scheduling on Mobile Platforms," in *Proceedings of the Design and Automation Conference (DAC)*, 2018.

[57] N. Chandramoorthy, G. Tagliavini, K. Irick, A. Pullini, S. Advani, S. A. Habsi, M. Cotter, J. Sampson, V. Narayanan, and L. Benini, "Exploring Architectural Heterogeneity in Intelligent Vision Systems," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2015, pp. 1–12.

[58] Android, "Power management — Android," https://developer.android.com/about/versions/pie/power, 2018.

[59] ——, "Doze on the Go..." https://developer.android.com/about/versions/nougat/android-7.0.html, 2016.

[60] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2019, pp. 399–411.

[61] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2001, pp. 197–206.