A Compiler-Centric Infra-Structure for Whole-Board Energy Measurement on Heterogeneous Android Systems

Junio Cezar Ribeiro da Silva[1], Fernando Magno Quintão Pereira[1,2], Michael Frank[3] and Abdoulaye Gamatié[2]

***Abstract*— A heterogeneous architecture combines, within the same hardware, different kinds of processors, with the goal of delivering fast execution at lower energy budgets. Because energy is such an important parameter of the efficiency of these architectures, much effort has been put into the implementation of techniques to measure the power dissipated by programs that they run. Yet, a vast majority of publications reporting experimental results produced on heterogeneous architectures either resort to simulations or rely on hardware counters; physical measurements are rare. In this paper, we introduce an apparatus – hardware and software – that we have been using to measure energy consumption in Odroid-based big.LITTLE architectures running the Android execution environment. This infra-structure is affordable and reliable. To demonstrate its viability, we show how we can use it to build oracles, i.e., a map that assigns different parts of a program to the hardware configuration that minimizes its energy consumption.**

I. Introduction

Modern computer architectures are becoming each day more heterogeneous [1]. Heterogenity emerges through the combination, within the same hardware, of several different processors, such as big/LITTLE multi-core Central Processing Units (CPUs) [2], [3], Graphics Processing Units (GPUs) [4] and Digital Signal Processors (DSPs) [5]. The heterogenous hardware design allies two opposing goals: time and power efficiency. To run faster, processors tend to spend more power. In a homogeneous architecture, the rate of power dissipation remains high, even when the processor is doing work that is not computationally intensive. On the other hand, in a heterogeneous system, we can allocate to each job the hardware configuration that best fits it, in terms of time and energy. This observation has motivated much work to reduce the energy consumed by programs [6], [7].

Yet, measuring the power/energy consumption of these processors with regard to the execution of a given code segments is a nontrivial task. The use of industrial power measurement tools for the energy evaluation of fine-grained code segments is a tedious job. The use of such tools requires careful synchronization to match energy results with segments of code. To circumvent this shortcoming, some hardware development boards provide developers with embedded power sensors, allowing them to extract the power profile of programs. As an example, the Odroid XU-E board, which integrates the Exynos5 Octa big.LITTLE chip, contains four separate current sensors to probe in real-time the power consumption of big cores, LITTLE cores, GPU and DRAM. Nevertheless, such sensors also have shortcomings of their own. First, they are not available in every processor. For instance, the Odroid XU4, which descends from the XU-E, does not provide them. Second, said sensors, when present, do not account for the entire board, but to individual components of it. Depending on the intention of the experimentalist, such compartimentalization can be an advantage; however, many researchers and users are concerned about the total energy budget of the board, including its peripherals.

This paper describes a reliable methodology to measure energy in embedded boards, with a focus on big.LITTLE architectures running Android systems. We describe the hardware and software components necessary to measure energy in programs that execute in the Android Runtime Environment. These programs can be written either in Java or Kotlin programming languages. Our methodology lets us measure the energy consumption of very small code segments. To this end, we use a circuit that can be activated via signals produced by the program. Such signals are triggered by instrumentation that we can insert automatically at particular program points, using the Soot bytecode analyzer [8], or that we can insert manually, at the discretion of the programmer. We focus on Android because we perceive a lack of infra-structure to carry out fine-grained energy measurement in this platform. Typical experiments report energy for the entire program [9], [10]. Attempts to get around this limitation tend to sacrifice precision. For instance, Leal *et al.* [11] resort to a combination of power meter, camera and image recognition to measure the energy spent by specific methods within an application.

To demonstrate the benefits that emerge with this new apparatus, in Section III, we show how to build a *Time-Energy Oracle* for a typical Android application that finds routes on a map. This application has different phases, which include network access, memory initialization and parallel processing. A time-energy oracle is a map that associates each program phase with the hardware configuration that best serves it in terms of time, energy, or the ratio time/energy. As we shall discuss in Section III, such an oracle brings several insights on the nature of the big.LITTLE architecture, and its relation with the programs that it supports. The ability to instrument programs, and relate code parts with energy

*This work has been partly supported by the French ANR agency under the grant ANR-15-CE25-0007-01, within the framework of the CONTINUUM project, by Google Research Award in Latin America and by CNPq.

[1]Department of Computer Science, Universidade Federal de Minas Gerais, Brazil {juniocezar,fernando}@dcc.ufmg.br

[2]Laboratory of Informatics, Robotics and Microelectronics (LIRMM), Montpellier, France abdoulaye.gamatie@lirmm.fr

[3]LG Electronics michael@pil.net

978-1-5386-3344-1/17/$31.00

and time numbers, allow us to take syntax into consideration when carrying out optimizations. This is in contrast with several previous work [6], [7], in which programs are treated as a black box, and only the state of the hardware is taken into consideration in energy minimization efforts.

II. The Measurement Apparatus

The technology described in this paper, henceforth called AndroidLeap, is based on the JetsonLeap framework [12], [13], which, in turn, expands the AtomLeap framework [14]. JetsonLeap measures energy consumed by programs written in C, and running on the Nvidia Tegra TK1 board. The AtomLeap framework is a hardware-only apparatus to carry out energy measurements in the Intel Atom board. AndroidLeap works in commodity Odroid boards, for programs written in Java or Kotlin, running on the Android Runtime Environment. The physical measurement apparatus used by AndroidLeap, i.e., power meter and circuitry, is the same as in the JetsonLeap framework. On the other hand, the software stack, which includes the library to interface with the hardware and the instrumentation framework, is substantially different, as it is adapted to Android.

In this section, we explain briefly how AndroidLeap works. To this end, we shall use the program in Figure 1. This program is a rather artificial example; nevertheless, it lets us introduce the notion of an *oracle* – something that shall be done in Section III. Moreover, the proposed exampled is divided into five parts whose the power behavior of each part will be analyzed:

1) Function readMap reads a set of *queries*, which are data-structures describing geographic maps;
2) Function buildGraph transforms such queries into graphs, via the method buildGraph;
3) in a third phase, Function minimumSpanningTree finds the minimum spanning tree for each graph;
4) Function Network.write offloads this tree to a network;
5) finally, Function IO.write records the tree in persistent storage.

We have interposed short sleeping periods between each of these phases, to make it easier to study their time and energy behaviors.

A. Hardware

The AndroidLeap apparatus consists of two parts: a hardware and a software stack. The hardware, which Figure 2 shows, has tree components: (a) the Odroid XU4 board; (b) the measurement circuitry; and (c) a power meter. The combination of these components lets us run Android applications, collect and analyze power profiles, like the one illustrated in Figure 1. The next four paragraphs describe each one of these components, showing how we can couple all of them together.

a) Odroid XU4: This board is manufactured by *HardKernel*[1]. It is equipped with the Samsung Exynos 5422 CPU, an heterogeneous unit containing four ARM Cortex A15

```
for (query in queries) {
(1) Leap.rec{map = readMap(query)}
    Thread.sleep(1000)
(2) Leap.rec{graph = map.buildGraph()}
    Thread.sleep(2000)
(3) Leap.rec{tree = graph.minimumSpanningTree()}
    Thread.sleep(2000)
(4) Leap.rec{ack = Network.write(tree, server)}
    Thread.sleep(2000)
    when (ack) {
(5)   OK -> Leap.rec{IO.write(backup_fle, tree)}
      ERROR -> err.print("wrong connection")
    }
}
```

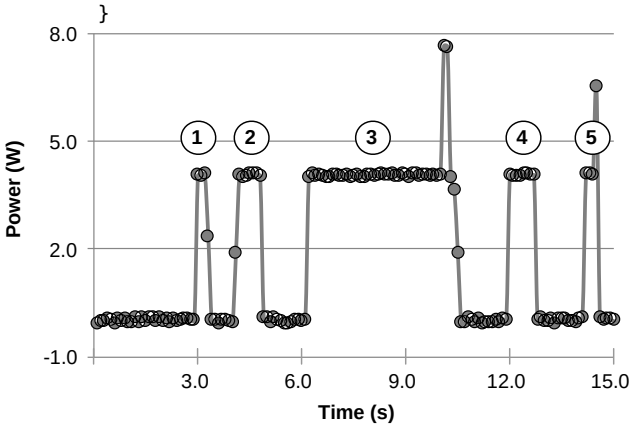


Fig. 1: Example of data produced by the AndroidLeap apparatus.

cores running at up to 2.0GHz and four ARM Cortex A7 cores running at up to 1.4GHz. The board also comes with a 30-pin *General Purpose I/O* (GPIO) port. We use specific pins of the GPIO port to send signals to our measurement circuit, which may enable or disable the measurement process in the power meter. The device is also capable of running the Android operating system. Given the present importance of energy optimization in the smartphone ecosystem, Android becomes an important requirement.

b) Power meter: The AndroidLeap setup uses the data acquisition (DAQ) device USB-6009 from National Instruments[2] to measure the power consumed by the Odroid XU4 board. This power meter offers two terminal blocks, one for digital I/O signals and another for handling analog I/O signals. In our apparatus, analog I/O is used to probe the voltage between the two leads of the shunt resistor present in the measurement circuit. Once we collect such value, we can read the instantaneous current flowing through the resistor. Additionally, we can derive the instantaneous power consumed by the board, and we can calculate the energy spent during a period of time.

c) Measurement Circuit: This component, displayed in the center of Figure 2 and diagrammed in Figure 3, works as a bridge between the program under evaluation and the power meter. This circuit contains a relay, which we can switch via signals produced by a special library of our own craft (see

[1]http://www.hardkernel.com/main/products/prdt_info.php

[2]http://www.ni.com/en-us/support/model.usb-6009.html

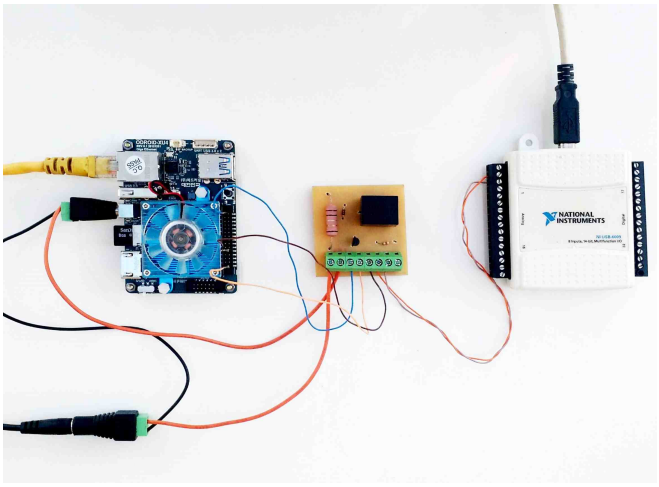


Fig. 2: AndroidLeap: hardware components.

function Leap.rec in Figure 1). Internally, our library uses the board's GPIO pins (#1 #2 and #3) to send such signals to the circuit. Whenever the relay is in its off position, no measurement is recorded by the power meter. This state gives us the valleys in Figure 1's chart. Thus, a simple integral on the curves produced by the power-meter gives us the energy consumed by the events of interest. In the absence of the switch, the integral can no longer be used, or else we would capture the energy spent due to actions external to the program under evaluation. To illustrate the importance of this switch, Figure 4 shows the power profile of our running example, with the interference of AndroidLeap. The noise between the different program phases is clearly visible in this new curve.

The measurement circuitry lets us use AndroidLeap to analyze small events within the program's code. By controlling the signals, we can determine the start and the end of discrete software events without the need to synchronize the power-meter's clock with the program's clock. To control such signals, we use a library, which we describe in the rest of this section. The switch time of our relay is approximately 0.5 milliseconds; hence, we estimate that we can measure reliably events comprising about 10^6 instructions at the maximum clocking rate of the Odroid XU4 board (2GHz).

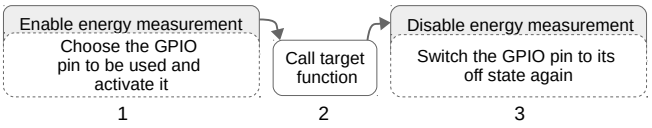
B. The Software Stack

AndroidLeap provides users with a small software library, which gives them the following functionalities:

- activate and deactivate energy measurement;
- determine a hardware configuration;
- fix the running frequency of a core;
- collect power measurement data.

This library is implemented in Kotlin, and can be used in code written in this programming language, or in Java. This library may be used directly by developers who want to perform interventions in available source code. We have also used it in tandem with the Soot compilation framework. Soot lets us instrument every occurrence of specific syntax that occurs in the program, such as calls to particular functions, or loops containing such calls, for instance.

d) Energy Measurement and Logging: users determine the duration of program events by manipulating the state of the data acquisition device used in our measurement apparatus. To mark the start and the end of events, users call specific functions to send signals through the GPIO pins available in the Odroid board. For example, if developers want to probe the energy spent by a specific function, they must perform three actions:



Lines 4 to 6 in Figure 5-(b) illustrate this sequence of actions. Due to interactions between the client application and the Android operating system, the target function might not transfer the control back to the caller. It is the responsibility of users to be aware of the application's lifecycle. In case of such "loss of control", energy measurement will not be disabled at the end of the event of interest, a fact that will prevent users from obtaining accurate results. In other words, the use of our apparatus fosters a programming style that is close to the well-known "Resource Acquisition is Initialization" (RAII) principle [15]. The Android programming API provides users with hooks that let them interpose checks along the lifecycle of an application to ensure that energy measurement resources are liberated, when no longer needed. Such resources can be released by calling the updateGPIOPin function whenever the state of an application is updated.

To generate energy reports, we have implemented a tool in C++ called CMeasure. This is the only part of our software

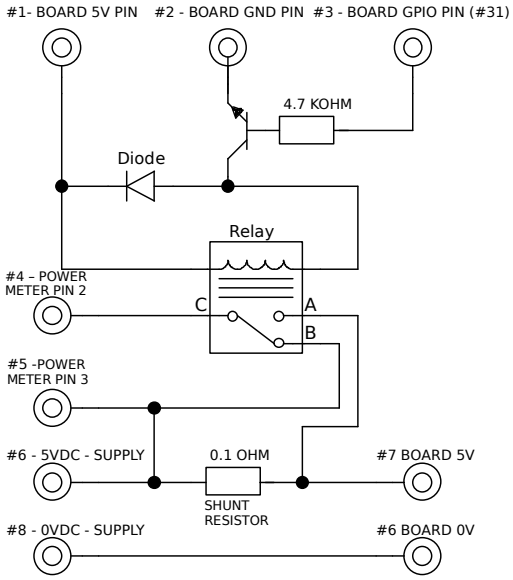


Fig. 3: Diagram of the measurement circuit of AndroidLeap. This circuit follows the design in JetsonLeap [13].

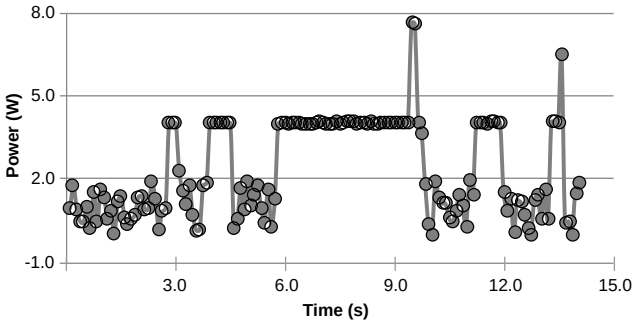


Fig. 4: Power profile with background interference.

stack that is not implemented in Kotlin or Java. CMeasure provides developers with an interface to communicate with the data acquisition device. It reads instantaneous power values, and performs an integral over these samples to yield final energy results. From this information, CMeasure produces tables that give us the energy profile of either specific program events, or the whole application.

e) The Interface with the Operating System: Our experience with different big.LITTLE systems tells us that interferences from the operating system complicate substantially the setup of reproducible experiments. Thus, to circumvent this problem, our software stack provides routines to fix the hardware configuration, i.e., the set of visible/hidden cores, and to switch on and off the operating system's scheduler. Figure 5 illustrates this capability. The code in the figure is an alternative version of the program seen in Figure 1. In Figure 5 we show how the user may define the set of visible cores on which the target function map.buildGraph is allowed to run.

Figure 5 (b) shows the body of the rec function, illustrating what our library does. Such library lets us collect the identification of the caller's thread, which is defined as tid in line 1. Using this value, we bind the caller's thread and its potential children threads to the set of cores specified by the user. This binding is made possible through system calls to the Linux kernel tool *taskset*[3]. The CPU affinity that we set up via the routine setAffinity will be honored by the Linux scheduler. In other words, it will not allow the thread to run on any other CPU. In line 4 of Figure 5 (b), power measurement is enabled by means of a signal sent to the GPIO #31 (pin 22, according to the Odroid XU4 block diagram[4]). After user-defined code is executed in line 5, power measurement is disabled via a new signal to the GPIO port, seen in line 6 of Figure 5 (b).

f) Automatic Instrumentation: In order to support the automatic instrumentation of programs, an extension to the Soot framework has also been implemented [8]. Soot allows us to remove from the developer the burden of manually determining which parts of the code should be monitored. As a result, our extension enables the task of automatically selecting program events and generating energy usage profiles based on pre-defined criteria. For example, we can use Soot to automatically instrument a target program to collect energy results for particular function calls or other set of instructions, such as *loops*, regardless of where they are present in the program's source code.

[3]http://man7.org/linux/man-pages/man1/taskset.1.html
[4]http://odroid.com/dokuwiki/doku.php?id=en:xu4_hardware

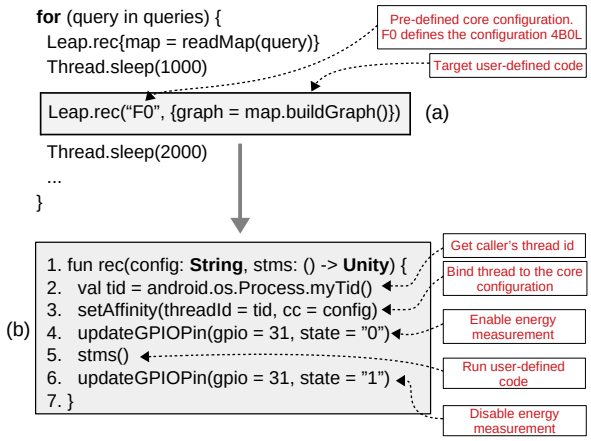


Fig. 5: (a) Running statements with AndroidLeap measurement interface. (b) Steps followed by our library in order to run code snippets on specific cores. The configuration format 4B0L (*xByL*) is introduced in Definition 3.3.

III. Time-Power Oracles

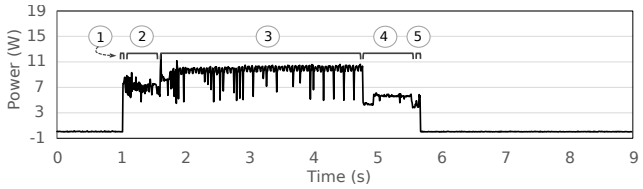
In this section, we demonstrate how AndroidLeap can be effectively used. To this end, we chose to apply it in the construction of an *oracle* for the program earlier seen in Figure 1. The definition of oracle requires another concept, which we shall call *Program Phase*, and that we state below:

Definition 3.1 (Program Phase): A program phase is a continuous sequence of events, observed during the program execution, sharing common power and time behaviours, and corresponding to a fixed text within the program code.

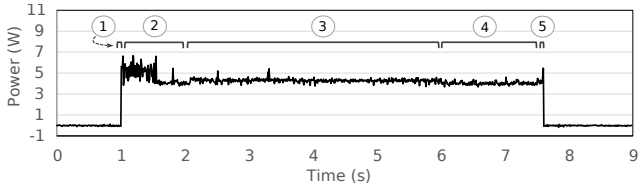
As an example, we can recognize five program phases within the loop in the program seen in Figure 1. If this loop runs for N iterations, then we shall observe $5 \times N$ different phases. Power and runtime behavior tend to be homogeneous within a program phase, and might differ among different phases. As an example, Figure 6 shows the power profile of the program in Figure 1, when executing under different configurations (see Definition 3.3). We have highlighted the different phases that we have observed in the execution of one iteration of the main loop in the program. We emphasize that this notion – program phases – has been left loose on purpose: we do not enforce any minimum degree of uniformity or heterogeneity to characterize phases, as this precision is immaterial to the application of AndroidLeap that we introduce in this section. From this rather loose definition of phases, we introduce the notion of oracle:

Definition 3.2 (Oracle): An oracle is a function that maps program phases to hardware configurations, to minimize either power consumption or runtime of the whole program.

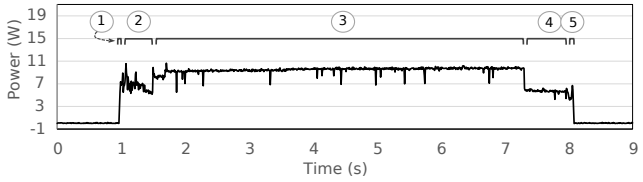
Definition 3.2 asks for the notion of a *hardware configuration*. This concept is stated in Definition 3.3. Our notion



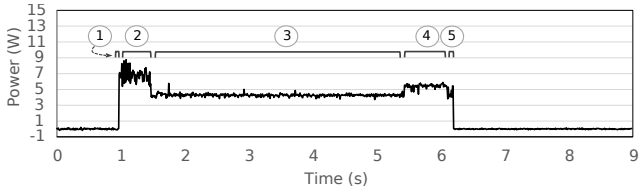
(a) 4B4L configuration



(b) 0B4L configuration



(c) 4B0L configuration



(d) Oracle

Fig. 6: Power consumption details for 4B4L, 0B4L, 4B0L and oracle. Circled numbers 1 to 5 indicate the same program phases as described in Figure 1

of hardware configuration does not account for the mapping between processes and cores. In our context, only the relation between visible and hidden cores is important.

Definition 3.3 (Hardware Configuration): Given a big.LITTLE architecture formed by B big cores and L LITTLE cores, a hardware configuration is an element in the $(B \times L)$ space, which we denote by $xByL$, meaning that we have x big cores and y LITTLE cores visible to a target process or program phase.

The Odroid XU4 board contains 24 possible hardware configurations, which are formed by the set $\{xByL | x, y \in \{0, 1, 2, 3, 4\}\} - 0B0L$. Notice that we remove, from the universe of possible configurations, the setting $0B0L$, which has no visible core. Continuing with our example, Figure 6 (d) shows the power profile for an oracle built for one iteration of the loop in the program in Figure 1. This oracle minimizes energy, i.e., the integral of the power outline. Visual inspection of the four different outlines in Figure 6 reveal that the area under the power curve seems, indeed, smaller in the oracle. In the rest of this section, we explain how we have used AndroidLeap to build such an oracle.

A. Constellations

In the rest of this paper we shall rely on the concept of a *execution constellation* as a useful tool to build oracles. We define constellations as follows:

Definition 3.4 (Constellation): Given a program P, its input I, and a big.LITTLE architecture A with $B \times L - 1$ possible configurations, a constellation for the triple (P, I, A) is a function that maps each possible configuration $xByL$ of A onto a pair (J, T). This pair consists of the energy J and runtime T observed during the execution of P, fed with input I, in the configuration $xByL$. When convenient, we can also consider mappings from configurations to pairs (W, T), involving time T and average power W.

Figure 7 shows two different constellations from running the program in Figure 1 with all possible 24 hardware configurations of the Odroid board. The constellation on the top relates time and the average power spent by the program, while the one on the bottom relates time and the total amount of energy consumed during that execution. These charts are generated automatically by AndroidLeap. To this end, we instrument the loop in Figure 1, so as to acquire the power profile of its entire execution. Instrumentation also lets us change the current configuration automatically; hence, repeating the experiment as many times as we deem it necessary to obtain results with high confidence. In this example, results are averages of ten executions. We never repeat the same configuration between samples to ensure a fair

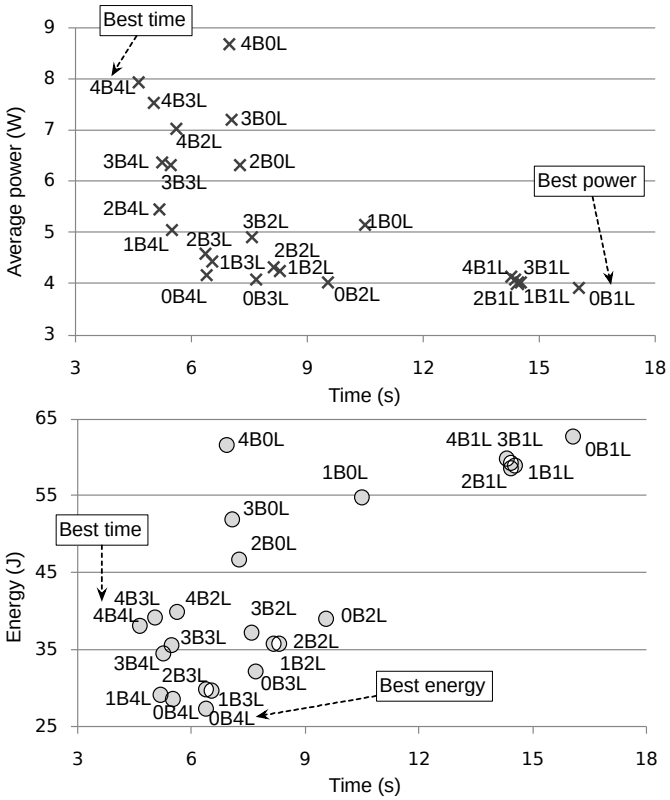


Fig. 7: (Top) Power vs. execution time for program seen in Figure 1. (Bottom) Energy vs. time for the same program.

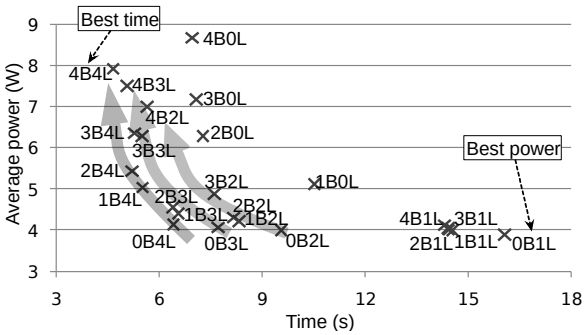


Fig. 8: The arrows show how average power increases, if we fix the number of LITTLE cores, and vary the number of big cores.

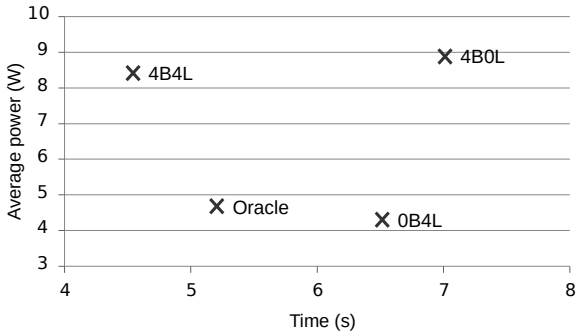


Fig. 9: Power vs. Execution time: oracle versus three other configurations.

comparison. The numbers reported in Figure 7 correspond exclusively to the code regions of interest; i.e., program events, other than those produced within the loop in Figure 1, bear no influence in the constellation that we have built.

Figure 8 shows how power and time evolve when the number of big cores increases, given a fixed number of LITTLE cores. We use arrows to highlight this trend. As an example, let us consider the configuration $0B2L$. If we keep adding cores; hence, gradually moving from $0B2L$ towards $4B4L$, we see that we tend to reduce runtime at the expense of a steep increase in energy consumption. This steep increase in energy is due to the nature of big cores, which are time efficient, but power-hungry. If we consider configurations composed of only a single LITTLE core, then increasing the number of big cores reduces the execution time without a noticeable boost in power consumption.

Sometimes, the best configuration in terms of time is not necessarily the best configuration in terms of energy. To demonstrate this statement, Figure 7 (right) shows a constellation relating the time and the energy consumed during that time, for the region of interest in our example. The arrows show how this relation, energy vs time, varies as we vary the number of LITTLE cores available in each configuration. The fastest configuration, in our example, was $4B4L$, which uses every core available. However, the most energy efficient was $0B4L$, which uses only the LITTLE cores. Therefore, in this example, the rush-to-idle approach does not reduce energy consumption, as it is usually the case, in DVFS-based systems [16].

B. The Construction of an Oracle

The constellations in Figure 7 show the runtime and the power (or energy) for the entire program. To build an oracle, we need to perform a similar analysis, but restrict it to program regions, instead of the entire program. AndroidLeap enables us to carry out such study, whose goal is to select, for each region of interest, the most energy-efficient configuration for it. This process is interesting as long as the best configuration varies among regions. Intuitively, this is the case. For instance, the execution of compute-intensive code regions will be achieved on the high-performance big cores, while IO-intensive code regions should rather be executed on LITTLE cores, since CPUs are likely to be under-used.

In our example, the best configurations, in terms of energy, for different program phases, were $1B0L$ and $0B4L$. The former configuration fits regions 1, 2, 4 and 5 in Figure 1. The latter suits region 3, which runs in parallel. This observation leads to the following oracle: $r_1 \mapsto 1B0L, r_2 \mapsto 1B0L, r_3 \mapsto 0B4L, r_4 \mapsto 1B0L, r_5 \mapsto 1B0L$. The oracle is more energy efficient than $0B4L$, which was shown to be the best configuration in the constellation seen in Figure 7. This efficiency remains, even when we consider the overhead of the instrumentation necessary to change the hardware configuration. Figure 9 shows this result, considering the four configurations earlier seen in Figure 6: 4B4L (the most performance-efficient), 0B4L (the most power-efficient), 4B0L (the most power-hungry) and the oracle.

Figure 6 helps us to analyze how power dissipation varies over time. The 4B4L configuration (Figure 6a) is the fastest; however, it is also the one that dissipates the most power. The 0B4L configuration (Figure 6b) is the one that dissipates least power, yet, it takes the longest time to terminate. The 4B0L configuration (Figure 6c) shows degradation in both execution time and power consumption; hence, bringing no gain compared to the previous two configurations. The oracle leads to a better compromise between power and time, as it leverages core heterogeneity (Figure 6d).

Figure 6 also shows a rather surprising fact: the 4B0L configuration dissipates more power, not only because the big cores run at a higher frequency, but also because its overall execution time is longer than the other scenarios. In this case,

TABLE I: Results for the three core configurations 4B4L, 4B0L, and 0B4L running the toy applications A, B and C.

Config.	Time (s)			Energy (J)			Prog.
	min	max	mean	min	max	mean	
4B0L	4.9	5.09	4.98	57.77	60.56	59.21	
0B4L	9.98	10.13	10.06	41.93	42.95	42.66	A
4B4L	4.08	4.41	4.26	50.75	52.96	51.68	
4B0L	9.04	9.27	9.16	96.42	96.94	96.75	
0B4L	8.8	9.40	8.98	60.28	61.84	61.38	B
4B4L	13.83	17.84	15.08	113.58	121.11	123.8	
4B0L	1.57	1.85	1.69	10.17	14.36	13.08	
0B4L	2.9	2.99	2.95	16.38	28.00	18.79	C
4B4L	1.45	1.79	1.60	9.30	12.22	10.33	

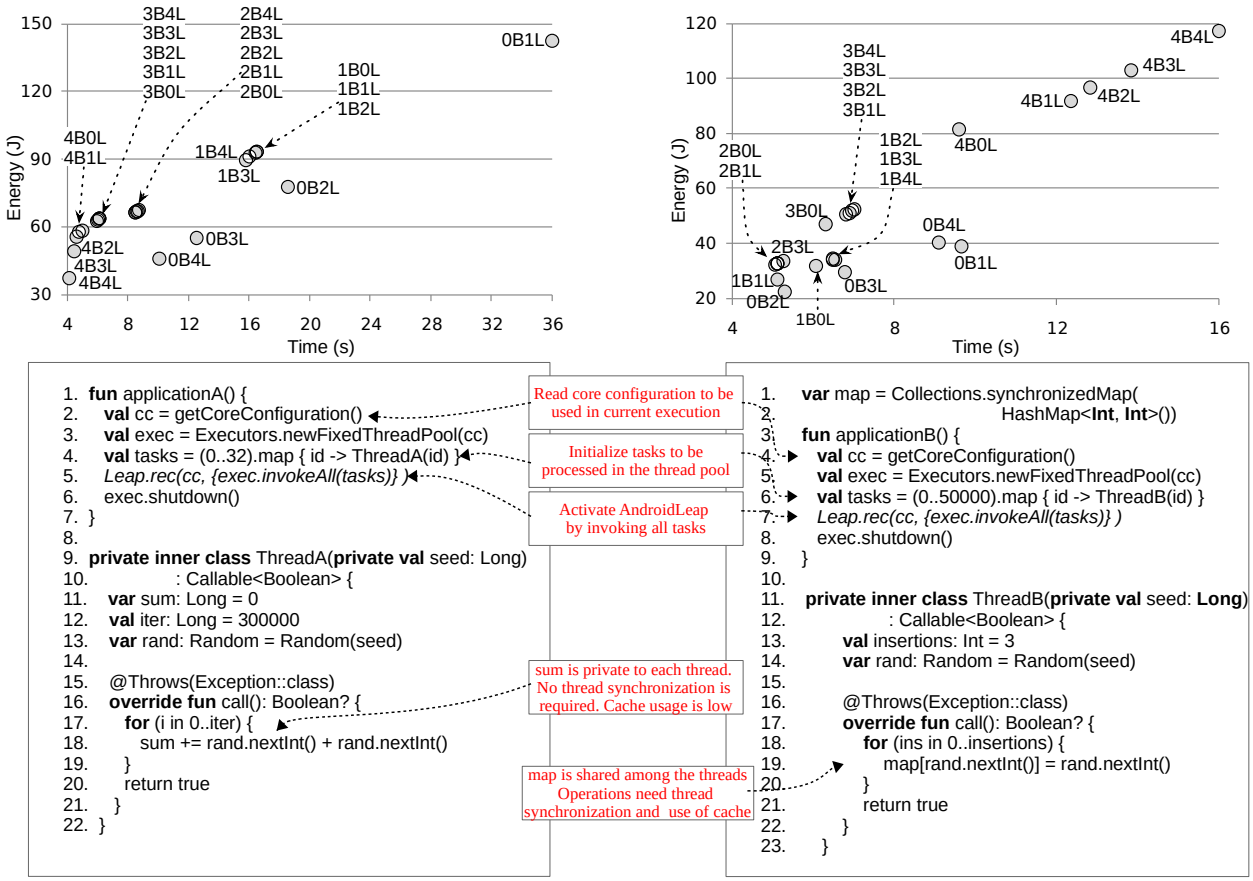


Fig. 10: Constellations for programs A (left) and B (right) from Table I. The code of each application is reproduced below its respective constellation.

the culprit is synchronization: in a scenario marked by heavy synchronization among threads, the extra speed of the big cores does not pay off for the power that they dissipate. To further investigate this concern, we analyzed the execution of three different parallel programs, whose time vs energy behavior is reported in Table I:

- In program A, each thread is assigned the task of summing up randomly generated numbers. In this program, no thread synchronization is required. In addition, cache memory usage is very low, as each thread generates random numbers on the fly and sums them up.
- In program B, each thread inserts elements in random positions of a single shared hash-table. Here, there is an evident need for thread synchronization, as many of them will try to concurrently update the same data-structure. Cache usage is also heavy.
- In the last program, C, each thread inverts large lists. In this case, we see heavy use of cache memory; however, there is no thread synchronization, as each thread operates on individual lists.

We have used the AndroidLeap apparatus to analyze the time and power profiles of these three programs. Table I shows the result of this experiment, and Figure 10 shows time-energy constellations for programs A and B. For program A, the $4B4L$ configuration is on average the fastest, as it is the most parallel. Yet, the $0B4L$ configuration provides the lowest energy consumption. For program B, the synchronization overhead between faster big cores and slower LITTLE cores penalizes both the runtime and energy consumption of the 4B4L configuration. In this case, $0B4L$ appears as the most energy-efficient configuration in this synchronization-intensive program. Finally, for program C, which is more compute- and cache-intensive, configurations with big cores (associated with bigger cache size for higher locality) provide the best results.

IV. Related Work

The main inspiration of this work is JetsonLeap, an apparatus used to perform power measurements on the Nvidia Tegra TK1 board. JetsonLeap, in turn, was inspired by AtomLeap. The main contribution of JetsonLeap on top of AtomLeap is the ability to record precisely power consumed in very fine-grained events. As mentioned in Section II, AndroidLeap departs from JetsonLeap in the choice of platform, including board and operating system. None of the results presented in this paper could be produced with the original JetsonLeap approach. Nevertheless, in all these "Leap-systems", power is measured via physical devices, instead of estimated via analytical models. Notice that this fact –acquisition of physical power– is not a contribution per se, as different research groups have done it consistently. Yet,

in the absence of fine-grained synchronization, researchers much either focus on peak consumption [17], on large programming events [18], or resort to power models based on hardware performance counters.

This last power measurement approach, based on hardware performance counters, seems to be the technique most adopted among researchers. For instance, Walker *et al.* [19] have presented a methodology for defining run-time power models based on performance monitoring counters (PMCs) for mobile and embedded devices. They rely on statistical analysis to derive a limited number of PMC events from a larger set collected from executions of programs. Events are carefully selected, so as to avoid using repeated information in power estimates. Walker *et al.* reported average errors of 3.8% and 2.8% for ARM Cortex-A7 and Cortex-A15 using the Exynos 5422 chip. This methodology is as precise as the hardware counters allow it to be; however, porting it to other architectures, if at all possible, requires non-negligible effort.

In addition to methodologies based on performance counters, developers also have access to models that provide general power consumption estimates for whole architectures, instead of focusing on the execution of individual programs. A canonical work in this direction is McPAT [20]. This tool relies on several low-level design parameters to provide power estimates: microarchitecture (frequency, vdd, in-order/out-of-order CPU, etc), circuit (SRAM / DRAM, Xbar, etc), technology (device in Low Standby Power or Low Operating Power modes, etc). However, we emphasize that while tools like McPAT have been used as part of the infrastructure to measure the energy consumed by programs [21], this usage is not its focus, in contrast to AndroidLeap. Finally, energy consumption estimation based on McPAT may present some limitation in accuracy [22].

V. Conclusion

This paper has presented AndroidLeap, a performance and power measurement methodology that targets the Odroid XU4 board, a system featuring eight ARM big.LITTLE cores. AndroidLeap consists of a hardware and a software stack that let us measure power consumption and execution time of programs running on the Android Environment. Our technique works at the granularity of program regions; that is to say, it lets developers analyze small program events such as loops and function calls. This is in contrast with competing approaches, often restricted to the granularity of entire programs. We demonstrated the benefits of our infrastructure by building a *Time-Energy Oracle* for a typical Android application with different computation phases. Such an oracle associates different phases within a program with the most energy-efficient hardware configuration for that phase. The fine-grained measurements enabled by AndroidLeap open several opportunities for the design and test of energy-efficient optimizations. In particular, the ability of assessing the impact of architecture configurations on the performance of code regions is key to resource allocation in energy-constrained devices.

References

[1] M. Zahran, "Heterogeneous computing: Here to stay," *Queue*, vol. 14, no. 6, pp. 40:31–40:42, 2016.

[2] R. Buchty, V. Heuveline, W. Karl, and J.-P. Weiss, "A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 7, pp. 663–675, 2012.

[3] P. Greenhalgh, "Big.LITTLE processing with ARM cortex-A15 & cortex-A7," San Francisco, CA, US, pp. 1–8, 2011. [Online]. Available: https://www.eetimes.com/document.asp?doc_id=1279167

[4] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, pp. 56–69, 2010.

[5] D. B. Williams and V. Madisetti, Eds., *Digital Signal Processing Handbook*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1997.

[6] R. Nishtala, P. M. Carpenter, V. Petrucci, and X. Martorell, "Hipster: Hybrid task manager for latency-critical cloud workloads," in *HPCA*. IEEE, 2017, pp. 409–420.

[7] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, and S. Gobriel, "Energy-efficient thread assignment optimization for heterogeneous multicore systems," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 1, pp. 15:1–15:26, 2015.

[8] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *CASCON*. IBM Press, 1999, pp. 13–.

[9] A. Canino and Y. D. Liu, "Proactive and adaptive energy-aware programming with mixed typechecking," in *PLDI*. New York, NY, USA: ACM, 2017, pp. 217–232.

[10] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *MSR*. New York, NY, USA: ACM, 2014, pp. 2–11.

[11] J. L. D. Neto, D. F. Macedo, and J. M. S. Nogueira, "A location aware decision engine to offload mobile computation to the cloud," in *Network Operations and Management Symposium*, 2016, pp. 831–838.

[12] T. Bessa, P. Quintão, M. Frank, and F. M. Q. Pereira, "JetsonLeap: A framework to measure energy-aware code optimizations in embedded and heterogeneous systems," in *Brazilian Symposium on Programming Languages*. Springer, 2016, pp. 16–30.

[13] T. Bessa, G. Gull, P. Q. ao, M. Frank, J. Nacif, and F. M. Q. ao Pereira, "JetsonLEAP: A framework to measure power on a heterogeneous system-on-a-chip device," *Science of Computer Programming*, vol. 33, no. 1, pp. 1–37, 2017.

[14] P. A. H. Peterson, D. Singh, W. J. Kaiser, and P. L. Reiher, "Investigating energy and security trade-offs in the classroom with the Atom LEAP testbed," in *Conference On Cyber Security Experimentation and Test*. USENIX, 2011, pp. 1–11.

[15] W. Weimer and G. C. Necula, "Exceptional situations and program reliability," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 2, pp. 8:1–8:51, 2008.

[16] T. Yuki and S. V. Rajopadhye, "Folklore confirmed: Compiling for speed = compiling for energy," in *International Workshop on Languages and Compilers for Parallel Computing*, 2013, pp. 169–184.

[17] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Determining application-specific peak power and energy requirements for ultra-low power processors," in *ASPLOS*. New York, NY, USA: ACM, 2017, pp. 3–16.

[18] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *OOPSLA*. New York, NY, USA: ACM, 2014, pp. 345–360.

[19] M. J. Walker, S. Diestelhorst, A. Hansson, A. Das, S. Yang, B. M. Al-Hashimi, and G. V. Merrett, "Accurate and stable run-time power modeling for mobile and embedded cpus," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 36, no. 1, pp. 106–119, 2017.

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*. New York, NY, USA: ACM, 2009, pp. 469–480.

[21] S. K. Rethinagiri, O. Palomar, R. Ben Atitallah, S. Niar, O. Unsal, and A. C. Kestelman, "System-level power estimation tool for embedded processor based platforms," in *RAPIDO*. New York, NY, USA: ACM, 2014, pp. 5:1–5:8.

[22] A. Butko, F. Bruguier, A. Gamatié, G. Sassatelli, D. Novo, L. Torres, and M. Robert, "Full-system simulation of big.little multicore architecture for performance and energy exploration," in *MCSoC*. IEEE Computer Society, 2016, pp. 201–208.