Workload Shaping Energy Optimizations with Predictable Performance for Mobile Sensing

Farley Lai[†], Marjan Radi[†], Octav Chipara[†], and William G. Griswold[‡]
[†] Department of Computer Science, University of Iowa
[‡] Department of Computer Science and Engineering, University of California, San Diego
farley-lai@uiowa.edu, marjan-radi@uiowa.edu, octav-chipara@uiowa.edu, wgg@cs.ucsd.edu

***Abstract*—Energy-efficiency is a key concern in mobile sensing applications, such as those for tracking social interactions or physical activities. An attractive approach to saving energy is to shape the workload of the system by artificially introducing delays so that the workload would require less energy to process. However, adding delays to save energy may have a detrimental impact on user experience. To address this problem, we present Gratis, a novel paradigm for incorporating workload shaping energy optimizations in mobile sensing applications in an automated manner. Gratis adopts stream programs as a high-level abstraction whose execution is coordinated using an explicit power management policy. We present an expressive coordination language that can specify a broad range of workload-shaping optimizations. A unique property of the proposed power management policies is that they have predictable performance, which can be estimated at compile time, in a computationally efficient manner, from a small number of measurements. We have developed a simulator that can predict the energy with an average error of 7% and delay with an average error of 15%, even when applications have variable workloads. The simulator is scalable: hours of real-world traces can be simulated in a few seconds. Building on the simulator's accuracy and scalability, we have developed tools for configuring power management policies automatically. We have evaluated Gratis by developing two mobile applications and optimizing their energy consumption. For example, an application that tracks social interactions using speaker-identification techniques can run for only 7 hours without energy optimizations. However, when Gratis employs batching, scheduled concurrency, and adaptive sensing, the battery lifetime can be extended to 45 hours when the end-to-end deadline is one minute. These results demonstrate the efficacy of our approach to reduce energy consumption in mobile sensing applications.**

***Keywords*-Mobile applications, energy efficiency, programming, performance modeling**

I. Introduction

Mobile sensing applications (MSAs) are an emerging class of mobile applications (apps henceforth) that make inferences based on sensor data to provide users with advanced features and customization. For example, Moves uses motion sensors to recognize when a user is walking, cycling, or running to create a fine-grained record of their physical activities [1]. Similarly, Sociophone uses microphones to track face-to-face interactions and identify close social relations [2]. Unfortunately, MSAs can significantly reduce the battery life of a mobile phone due to their continuous operation and use of power-hungry resources such as cellular radio, Wi-Fi, GPS, or microphone. Developers must, therefore, implement complex application-level power management (PM) policies that coordinate the use of hardware resources to minimize energy consumption.

Best software engineering practice suggests focusing first on the functional aspects of an MSA and then refactor it to address performance concerns. Ideally, a developer should be able to incorporate a wide range of PM policies easily and evaluate their performance trade-offs. Today, such transformations are applied manually, and their impact is evaluated through extensive and time-consuming tests. To illustrate some of the challenges associated with this approach, consider an app that tracks a user's social interactions using speaker-identification techniques. The app collects audio samples, extracts features from these samples, saves them locally, and uploads them to an edge service determine the identity of speakers. A strategy that a developer may employ is to batch packets to improve the energy efficiency of wireless communication. However, network communication may occur at multiple locations in the app. Aside from having to apply the transformation at each location, it is possible that the transformation may introduce concurrency bugs that are challenging to debug when the uploads use multiple synchronizing threads. The underlying problem is that Android apps are difficult to refactor as they do not cleanly separate the functional aspects of the app from its concurrency and PM. Next, the developer has to perform several tests to quantify the impact that the batch size parameter has on the energy consumed and timeliness of data uploads. This step involves deploying the app and measuring its performance using different batch sizes under diverse operating conditions. As the PM policies increase in complexity, this approach becomes increasingly prone to errors, and the problem of configuring the parameters of a PM policy grows combinatorially with each new parameter.

The central contribution of this paper is ***Gratis*** — a novel programming paradigm that (1) simplifies the introduction of PM policies and (2) automates the configuration of PM policies. In this paper, we focus on *workload shaping energy optimizations*, a class of optimizations that save energy by controlling when hardware resources are used. Workload-shaping optimizations leverage that most of the operations performed by an MSA, including data collection, inference, and synchronization with remote services, are delay tolerant. Accordingly, we can save significant energy by introducing delays to increase the workload of the system artificially to allow the hardware to operate more efficiently. However, adding delays can negatively impact the user experience.

Therefore, our goal is to develop policies that *reduce energy consumption under soft end-to-end deadlines* that control the impact of energy optimizations on user experience.

Our approach builds on the insight that we must provide a clean separation between the functional aspects of an app and its run-time behavior to simplify the introduction of PM policies. We propose a novel programming model that combines a stream programming abstraction with PM policy that controls an app's execution. It is common to use stream programs to specify MSAs (e.g., [3]–[5]). However, in contrast to existing systems that execute components as soon as possible, the PM policy explicitly control the execution of an app. We have developed a coordination language for specifying parametric PM policies that control when components are used and their concurrency. PM decisions can be made based on the number of data frames in the queues of components or the time until the deadline of a data frame expires. The developed language is sufficiently expressive to specify a wide range of workload-shaping optimizations including batching, scheduled concurrency, and adaptive sensing. The benefit of our approach is that it enables a developer to specify sophisticated PM policies without having to modify an app's code.

A key challenge to writing effective PM policies is that their parameters must be configured to interactions of the app with its environment. Manually configuring the parameters of a policy is particularly challenging since there are many possible configurations, each having a different energy consumption and timeliness tradeoffs. We developed techniques to automatically configure parameterized policies into concrete policies where each parameter is assigned a fixed value. Central to our techniques is a specialized simulator that can estimate the energy consumption and delays of an app in a data-driven manner based on a set of traces collected in a diverse set of environments. The simulator is built on the insight that MSAs have *composable performance*, i.e., the delay and energy consumption of the entire app can be estimated accurately from performance profiles of its components. This insight allows us to build a simulator that is highly scalable and accurate even in when the app has dynamic workloads. Leveraging the simulator's scalability, we use standard optimization techniques (i.e., gradient descent and grid search) to instantiate PM policies. In combination, the new programming model and associated policy instantiation techniques, allow a developer to write PM policy iteratively, configure their parameters, and estimate the energy-delay trade-offs of PM policy without having to deploy an app.

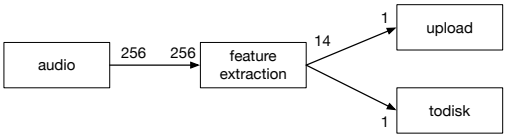
We have evaluated Gratis by implementing two MSAs for speaker identification (`SI`) and activity recognition (`AR`). We demonstrate the expressiveness of our coordination language by implementing workload shaping optimization that combines batching, scheduled concurrency, and adaptive sensing. Workload shaping policies can effectively reduce energy consumption. For example, the battery of a phone running the `SI` app lasts for only 7 hours when audio data is processed as soon as possible. In contrast, if the user is willing to tolerate a 60-second latency, the battery life is extended to 19 hours, a 2.7 times improvement. Adaptive sensing, which collects audio only when speech is detected, extends the battery life to 45 hours. The `AR` app shows similar trends. We have extensively evaluated the accuracy of the app simulator under a wide range of configurations. Our results show the average prediction errors for energy and delay are 7% and 15%, respectively. These results demonstrate the effectiveness of incorporating workload shaping-optimizations in MSAs automatically.

II. Problem Formulation

Workload shaping energy optimizations save significant energy by adding delays to shape the workload of an MSA. However, the amount of delay that may be added must be carefully controlled to ensure that it does not negatively impact the user experience. Through an example, we will illustrate workload shaping energy optimizations, introduce our PM specification language informally, and discuss the challenges of estimating the performance of PM policies. We will formalize these concepts in Section III.

Parametric PM Policies: Consider that a developer wants to minimize the energy consumption of the speaker identification (`SI`) app subject to the constraint that the data is uploaded to an edge service within 10 minutes. She implements the `SI` app as a graph of components that are responsible for collecting audio frames, extracting features from frames, and uploading them to the cloud service to determine the identity of speakers (see Figure 1a). She may incorporate a PM policy parametrized by the batch size (th_1) to improve energy efficiency by *batching* the writing of data to flash and its wireless transmissions. Additional energy may be saved using *schedule concurrency* which controls the concurrency of writing to flash and transmitting packets. The net effect of this optimization is that it consolidates periods of activity and sleep allowing the device to exploit deeper sleep states. Our insight is that this PM policy can be specified at the level of the stream graph by controlling when the app's components start their execution. The code of the PM policy can be specified and refined separately from the code of the app. We ensure that PM policies do not introduce concurrency bugs by enforcing that PM decisions to start a component depend only on local data associated with that component. This "correct-by-design" approach side-steps the difficulty of reasoning about the correctness of PM transformations in generic Android apps that extensively use multi-threaded and event-driven programming.

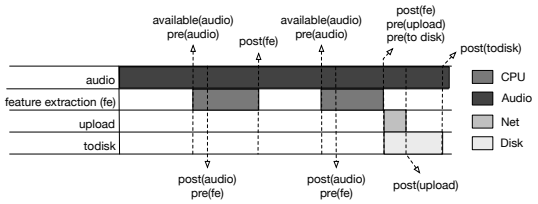
We developed a simple but expressive language for specifying parametric policies. The PM policy that integrates batching and scheduled concurrency which are two examples of workload-shaping optimizations (see Figure 1b). The basic primitives of the proposed language are events, handlers, and commands. The policy is executed in an event-driven manner as follows. When the Android framework reads 256 audio samples, Gratis inserts the frame containing the samples in the input queue of the `audio` component and calls the `available(audio)` handler. The `available(audio)` handler triggers the execution of the `audio` component. When its execution is completed, the `post(audio)` handler starts



(a) The basic `SI` app collects audio data (`audio`), extracts features (`feature_extraction`) from the samples, and uploads (`upload`) them to a remote server and saves them locally (`todisk`). The numbers indicate how many samples are produced/consumed by each component.

```
1:  E = { deadline : 10 min, th_1 : 2 }
2:  available(audio):
3:      execute(audio)
4:  post(audio):
5:      execute(feature_extraction)
6:  post(feature_extraction):
7:      if feature_extraction.num_output ≥ th_1:
8:          execute(upload)
9:          execute(todisk)
```

(b) A PM policy that incorporates batching and scheduled concurrency.



(c) Timeline of events generated using policy in Figure 1b. The amount resources uses for each component is shown.

Fig. 1. The basic version of the `SI` app that implements batching and scheduled concurrency.

the execution of the `feature_extraction` component. A typical pattern in Gratis policies is to trigger the execution of descendant components to be executed in a `post()` handler. This pattern occurs when components are executed sequentially. The `post(feature_extraction)` handler has two important aspects. First, the execution of `upload` and `todisk` is guarded by an if-statement. The consequent instructions are executed only when the number of frames in the output queue of the `feature_extraction` exceeds $th_1 = 2$. By configuring th_1, we can control the batching for the network (used by the `upload` component) and disk (used by the `todisk` component) in a data-driven manner based on the amount of data available in the queue of components. Second, the `upload` and `todisk` are scheduled to start concurrently. It is important to note that Gratis provides only coarse-grained control over resource usage. Gratis only controls when a thread becomes ready to be scheduled and not the precise interleaving of the threads. The operating system schedules the threads, sends packets, and stores data to disk.

Timing Semantics: An important consideration is how to capture timing in our system. In Gratis, we maintain a timestamp associated with each data frame[1]. When a component executes, it consumes some data frames from its input and produces some output samples. The output samples are timestamped with the minimum of the timestamps of the consumed samples. The end-to-end latency is the maximum difference between the time when a sample was produced until it was consumed. The introduction of delays will artificially increase the end-to-end latency. The developer constrains the end-to-end latency and bounds the impact on the user experience by specifying a soft end-to-end deadline. The end-to-end deadline is specified by setting the value of the global $deadline$ variable in the environment (see line 1 in Figure 1b).

Policy Instantiation: A unique property of the considered energy optimizations is that they provide predictable performance, i.e., their performance can be determined at compile time. It is difficult to automatically determine the impact that a PM policy on an Android app as it is challenging to determine the dependencies between components, their concurrency, and use of hardware resources from Java code. In contrast to generic Android apps, a unique property of Gratis apps is that they have composable performance: the overall performance of the app can be determined from the energy and delay profiles of its components. This property holds for two reasons. First, the stream program explicitly captures the dependencies between components, the number of frames consumed/produced, and we enforce that each component uses a single hardware resource. Second, the PM policies explicitly control the timing, the concurrency, and the amount of data processed by a component. As a result, a trace-driven simulator can determine when components are executed and the overlap in the use of hardware resources. If some components use the same hardware resource, they should use that resource in a fair manner (consistent with the behavior of the Linux scheduler). In contrast, if components use different hardware resources, they are executed concurrently. The simulator estimates the energy and delay based on the deterministic sequence of events generated by the PM policy (such as the sequence of events shown in Figure 1c).

III. Design

Gratis provides an intuitive and practical approach for specifying, evaluating, and configuring PM policies that implement workload shaping energy optimizations. Gratis allows developers to specify MSAs and their PM policies. A developer writes an app in the StreamIt programming language [6]. The stream program provides an implementation for each component and explicitly captures their dependencies and resource usage. We provide a coordination language for specifying a parametric PM policy that controls the execution of components at run-time. PM decisions are made either in a data-driven manner (based on the number of frames in the queue of a component) or in a time-driven manner (based on the time remaining until the deadline of a frame expires). The developers can refine the code and PM policies of an app independently.

The Gratis toolkit includes three components: a policy configuration tool, a translator, and a run-time scheduler. The policy configuration tool is used to determine concrete values for each parameter of a PM policy such that energy consumption is minimized and the end-to-end deadline constraints are

[1]We opt against maintaining a timestamp for each sample as it introduces a significant overhead.

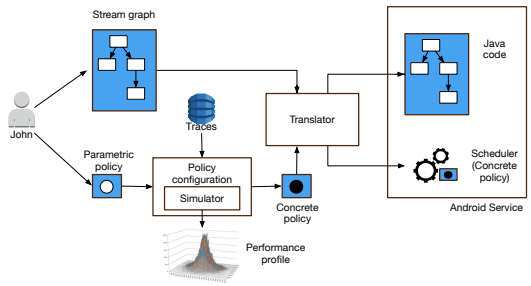


Fig. 2. A developer specifies an MSA as a stream graph and coordinates using a parametric policy. The policy configuration tool determines the concrete values for each parameter of a policy. A translator generates an Android service based on the stream graph and concrete policy.

satisfied. The tool proposes new concrete policies using either grid search or gradient descent techniques. The performance of each concrete policy is evaluated using a simulator based on the performance profile of components and previously collected traces. The translator includes a source-to-source compiler that translates the StremIt code in an Android service. Additionally, the translator generates an intermediary representation for a PM policy that can be executed by a scheduler a run-time. The scheduler enforces a concrete policy at run-time and manages the interactions with the Android framework. The remainder of this section discusses the language support, policy configuration, and translation, respectively.

A. Gratis Programming Model

The Gratis provides support to express MSAs as stream programs and control their behavior using a PM policy. Next, we formalize both aspects of the programming model.

1) Stream Programs: An app is structured as a graph of components[2] that are connected using FIFO queues. A component is the basic unit of a stream program. During its execution, a component reads frames from its input queue, performs computations based on the read data along with its internal states, and produces frames inserted into the output queue. A traditional synchronous dataflow model requires that a component produces and consumes the same number of frames in all its executions [7]. Gratis allows components to produce and consume a variable amount of data.

The key novelty of Gratis and the aspect where our work departs from previous work on dataflows is its model of computation. A traditional dataflow system assumes that input data is always available to be processed and the system should process the data as soon as possible. Therefore, in such systems, the primary concern is to manage the CPU efficiently to maximize the overall throughput. In sharp contrast, Gratis PM policy introduces delays to create workloads that may be processed more efficiently. Additionally, Gratis coordinates multiple hardware resources to achieve the desired energy-delay tradeoff.

[2]In StreamIt terminology, a component is called a filter. In this paper, we opt for the more general term of component since the proposed PM methodology readily extends to other stream programming systems.

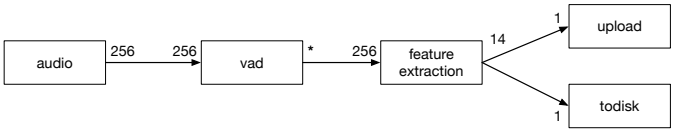
2) Policy Specification: The policy language controls when components are executed using four constructs: events, handlers, guarded commands, and an environment.

Events and Handlers: Gratis may generate and handle three types of events: data available, execution, and timeout. The app's scheduler registers to be notified when data is available either from a sensor or a socket. The scheduler identifies the components that are interested in receiving this data, inserts it in their input queues, and calls their `available()` handlers. The `available` event is the only event triggered externally by the underlying Android framework. The execution and timeout events are generated internally by Gratis . Gratis generates a `pre(A)` event before starting the execution of a component `A` and a `post(A)` event after completing `A`'s execution. The timeout events are generated when the minimum slack of the frames in a queue falls below a configured threshold. Note that each handler is associated with a unique component called the handler owner.

Commands: The logic of a PM policy is implemented using guarded commands. The event handler contains a sequence of guarded commands. Gratis has two commands: `unsubscribe` and `execute`. A source component subscribes to receive data from sensors or a socket during its initialization. The `unsubscribe(A, d)` command stops the reception of these events by `A` for `d` seconds. This allows the device to sleep. The `execute(A)` command triggers the execution of A when there are sufficient frames in `A`'s input queue to execute at least once. After `A` starts executing, it may execute multiple times until the number of frames in its input is less than `A`'s consumption rate. The `execute(A)` command is idempotent, i.e., if `A` is already executing, the command has no effect. If multiple `execute()` commands are issued within the same handler (as it the case in lines 8 – 9 of Figure 1b), the components will be executed concurrently.

Commands may be guarded by conditional expressions that involve properties computed based on the states of a component's queues. Gratis exposes the number of frames in the input and output queues as `num_input` and `num_output`. Additionally, Gratis also exposes the minimum of the slack for the data frames in the input queues as `input_slack`. The execution of a component may be triggered in a data-driven manner when the number of frames exceeds a threshold. Alternatively, the execution of a component may be triggered in a time-driven manner when the minimum slack falls below a threshold. This case is handled using the timeout event handler `timeout(A, t)` when the value of `input_slack` of `A` falls below `t` seconds.

We limit both the scope and the complexity of the conditional expressions. An expression can only refer to data associated with the handler owner or in the global environment (described below). By forcing that components make decisions on local data, we avoid the potential of writing policies that may introduce concurrency bugs. Additionally, we limit the complexity of the boolean expressions to only include the variables described above. This ensures that a PM policy can be executed efficiently at run-time.



(a) The app collects audio data (`audio`), determines whether the audio includes speech (`vad`), extracts features (`feature_extraction`) from the samples that include speech, and uploads (`upload`) to a remote server and saves them locally (`todisk`). The numbers indicate how many samples are produced/consumed by a component. A * indicates that the rates are variable.

```
1:  E = { deadline : 10 min, th1 : 2, th2 : 30 sec,
              th3 : 1024, sleep_duration : 10 sec }
2:  available(audio):
3:      execute(audio)
4:  post(audio):
5:      execute(vad)
6:  post(vad):
7:      unsubscribe(audio, sleep_duration)
8:      if vad.num_output ≥ th1:
9:          execute(feature_extraction)
10: timeout(feature_extraction, th2):
11:     execute(feature_extraction)
12: post(feature_extraction):
13:     if feature_extraction.num_output ≥ th3:
14:         execute(upload)
15:         execute(todisk)
```

(b) A PM policy that incorporates batching, scheduled concurrency, and adaptive sensing optimizations.

Fig. 3. The advanced version of the `SI` app that implements batching, scheduled concurrency, and adaptive sensing.

Environment: The app interacts with the PM policy through its execution environment. An execution environment is a dictionary that maintains the policy parameters. The policy parameters can be used as part of the guarded commands. By default, the dictionary includes the $deadline$ variable, which specifies the end-to-end deadline. We provide a simple interface to allow the values of the variables to be read and modified from the StreamIt code.

Example: In Figure 1b, we present a PM policy that saves energy by combining batching and scheduled concurrency. Next, we consider how additional energy may be saved using adaptive sensing (see Figure 3a). One of the challenges to supporting adaptive sensing in Gratis is that it introduces workload dynamics. A `vad` execution may generate either a frame containing speech data or no data. When the workload is dynamic, it is unclear what is the best strategy to configure the policy parameters. We may use small values for the th_1 and th_3 that control batching to ensure that even when the `vad` generates little data, the app will process it. However, this may not be energy efficient. Increasing th_1 and th_3 shall improve energy efficiency but could also cause a longer processing delay. Deadlines may be missed if the `vad` does not produce sufficient data to increase the number of frames in the queues of `vad` and `feature_extraction` components beyond th_1 and th_3 respectively. A better approach to handling this situation is to use a `timeout` handler. The timeout handler can be used to trigger the execution of the app based when the slack falls below a threshold. In our example, the execution is triggered when the minimum slack of the frames in the input queue of `feature_extraction` falls below $th_2 =$ 30 sec. The policy shows how our coordination language can be used to express a policy that combines batching, scheduled concurrency, and adaptive sensing.

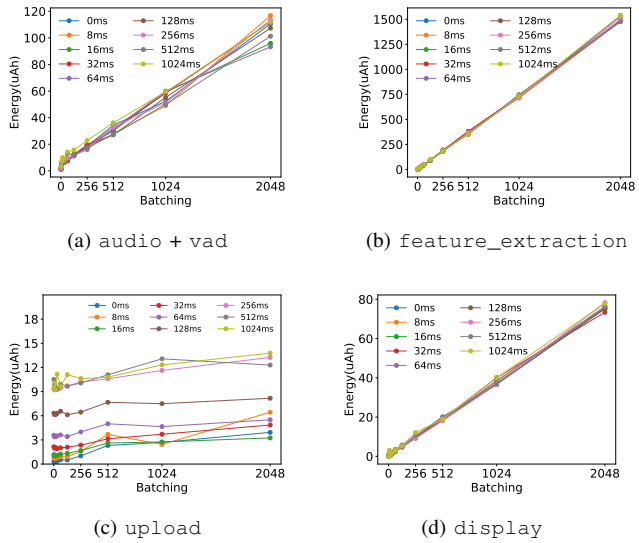
B. Evaluating PM Policies

An important feature of the considered energy optimizations is that they have predictable performance that can be determined at compile time. Next, we will develop techniques that assess the performance of a concrete policy in a computationally efficient manner using a small number of measurements. Our solution involves three steps: (1) the app is partitioned into multiple domains that are executed by independent threads, (2) a domain profiler constructs a performance profile for each domain, and (3) the simulator estimates the overall performance of the app based on the performance profiles and a set of traces. The set of traces capture all the interactions of the app with the environment by recording the timing and content of the `available` events. Since the PM policies are deterministic, this information is sufficient to replay the complete behavior of an app in a deterministic manner. Additionally, we assume that the MSA works in the background with minimal interference from other apps. Large-scale user studies support this observation [8].

Domain Partitioning: A naive approach to implementing PM policies is to have each component operate in a different thread. However, this method would incur significant overhead since the app would include many threads whose execution must be synchronized. To address these issues, we partition the app into *domains* such that all components that pertain to a domain use the same hardware resource. The components partitioned into the same domain are executed in the same thread. The constraint that all components of a domain use the same hardware resources ensures that the hardware resources can be controlled by starting/stopping the domains.

Domain Profiler: To create accurate performance profiles, we must address the following challenges: hardware resources have different energy/delay characteristics, and the resource usage of a domain may depend on its input. We address these challenges by performing measurements in which we control three parameters: batching, interim time, and data content. The batching parameter controls the amount of data that a domain to processes in an execution. The interim time controls the time between consecutive batch executions. The values of the inputs must be selected carefully when profiling dynamic domains. The compiler generates code to track the execution time of a domain. We measure energy consumption using a power meter.

The delay of components typically scales linearly with the workload regardless of the type of hardware resource used. In contrast, the energy consumed by a domain may scale linearly or non-linearly with the workload. For example, the energy consumed by the `feature_extraction` domain, which uses the CPU, scales linearly with the workload (see Figure 4b). For domains that use linear-scaling resources, it is sufficient to profile them with different batch sizes using a fixed interim value. Figure 4b shows that changes in the



(a) `audio` + `vad` (b) `feature_extraction`

(c) `upload` (d) `display`

Fig. 4. Energy consumption for a subset of domains of the `SI` app evaluated in Section IV.

interim have little impact on the energy consumed by the `feature_extraction`. In contrast, the energy consumed by network interfaces (e.g., Wi-Fi or cellular) scales non-linearly with the workload. For domains that use non-linear-scaling resources, we profile them with different batching *and* interim values. The reason why energy scales non-linearly is because the hardware resource remains in a high-power state for a while after its last usage [9]. For example, the energy consumed by `upload` varies significantly with the interim time as shown in Figure 4c. However, for a fixed interim, the energy consumed by `upload` can be approximated by a function that scales linearly with the workload.

The domains of a stream program may be either static or dynamic. The resources usage of a static domain depends only on the number of frames its processes and is independent of its input values. For example, computing audio features or uploading them introduce similar resource usage regardless of the values of the samples in frames. Empirical studies have shown that a majority of stream programs are composed of only static components. Additionally, even in stream programs that are dynamic, the majority of their components are static [10]. This is also the case for the `AR` and `SI` apps. It is sufficient to evaluate the performance of static domains can be evaluated using dummy data.

The input used to profile dynamic domains must be carefully selected to obtain an accurate profile. For example, the only dynamic domain in the `SI` app is the domain that includes the `audio` and the `vad` components. The performance of the domain may be divided into two clusters depending on whether speech is detected: the audio frames that contain no speech are dropped while those take contain speech are further processed to extract features. For each cluster, we create different performance profiles with different interim values. In our current implementation, the developer manually specifies the how tests pertaining to each cluster are generated. In the future, we will investigate automating this process.

As part of the app simulation, we must evaluate the execution time and energy consumption of domains for configurations for which we do not have direct measurements. Consider the case when the simulator wants to estimate the energy consumption of a domain given the state of its the input queue and the time from its previous invocation. We first compute the resource utilization for each frame in the input queue to determine the most frequent input cluster. We will use the performance measurements associated with the most frequent cluster to estimate the energy consumption. For each cluster, there are performance measurements for different batch and interims values. The time from the previous invocation is used as the interim in the performance profile. Referring to Figure 4c, when the is interim $i = 620$, we generate a dataset that approximates the behavior of the system at this interim based on the data collected for interims 512 and 1024 using linear interpolation. Then, the energy is estimated by fitting a linear function and evaluated given the number of frames in the component's queue.

App Simulator: The performance of the app is determined by simulating it in an event-driven manner according to its PM policy (see Algorithm 1). The input to the simulator is the trace of `available` events that are initially loaded into the queue of the simulator. The simulator estimates the delay via the Δ data structure. In response to an event, the simulator will call the `policy_handler` to execute the instructions associated with that event in the policy. If the guard of the instruction holds, a `pre` execution event will be inserted in the queue to be processed next. It is easy to ascertain whether the guard is true since guards are simple boolean expressions involving properties associated with the queues of a component. The `pre(d)` event indicates that domain `d` may be executed when there is sufficient data in its queue for at least one execution. If this is the case, the domain will be added to the `ready` data structure.

The ready data structure is a dictionary that maintains a mapping from hardware resources to a list of domains that are in execution. The core of the simulator is the `sim_execution` function that simulates the execution of the ready domains. The function executes the ready domains either until one is finished or until the time when the next event in the app will occur (provided as the next_event argument). Domains that use different resources are executed concurrently. In contrast, domains that use the same resource must share it in a fair manner. This is accomplished by cycling through the domains that are ready for a given resource using the `next_domain` function of the `ready` data structure. The execution of the domains in a fair manner approximates the behavior of the Linux kernel that implements a form of weighted fair queueing [11]. The result of the simulation is a timeline of when each domain starts and finishes executing. The energy consumption is evaluated at the completion of the simulation using the generated timeline.

Example: To clarify the behavior of the simulator, consider

```
time = 0
queue = the trace of available events and the associated data
ready = a mapping from hardware resources to domains that use that
    resource and are ready to run
Δ(d, data, num_frames) = latency for domain d and input data
while time < sim_time do
    (time, event, data) = queue.pop()
    switch event do
        case sample: do
            policy_handler(time, event, data, queue)
        case pre(d): do
            policy_handler(time, event, data, queue)
            if d.num_input ≥ d.min_input then
                ready[d.resource()].append(d)
                d.duration = Δ(d, d.input, d.num_input)
                d.data = data
        case post(d): do
            policy_handler(time, event, data, queue)
    sim_execution(ready, time, queue.peek().time, queue)
Procedure policy_handler(time, event, data, queue)
    // Execute the instructions of the handler associated with the
       event
    for instr in handler(event) do
        if instr.guard() then
            queue.schedule(time, pre(instr.target), data)
Procedure sim_execution(ready, time, next_event, queue)
    // Execute the domains that are ready such that domains using
       different resources run independently and those sharing
       resources use them fairly
    tick = 5 ms
    new_event = False
    while (time ≤ next_event) and (new_event = False) do
        for resource in ready do
            d = ready[resource].next_domain()
            d.duration -= tick
            d.num_input -= d.input(d.data)
            d.num_output += d.output(d.data)
            if domain.duration ≤ 0 then
                new_event = True
                ready.remove(d)
                queue.schedule(time, post(finished), None)
        time = time + tick
```

Algorithm 1: Pseudo-code for the app simulator. The output of the simulation is used to assess the energy consumption and the end-to-end delay.

the case when two domains, Domain 1 and Domain 2, require the CPU for 10 ms and 15 ms respectively. An additional domain, Domain 3, requires the network for 7 ms (see Figure 5). Since the network and the CPU resources are independent, the execution of the Domain 1 along with Domain 2, and Domain 3 is concurrent. As a result, Domain 3 finishes after 7ms. In contrast, Domain 1 and Domain 2 uses the same resource. Accordingly, they must alternate using the CPU to share the CPU fairly. The Domain 1 and Domain 2 share the CPU for 15 ms until Domain 1 finishes executing. Domain 2 continues to execute for another 10 ms until it finishes. Technically, Domain 3 execution still uses the CPU but the time is relatively shorter than CPU domains (< 5 ms). The actual network transfer may happen later with some delay after

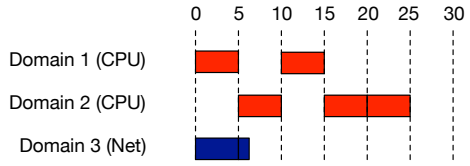


Fig. 5. Simulation of three domains. Domain 1 and 2 use the CPU and have delays of 10 and 20, respectively. Domain 3 uses the network. Domains 1-2 and domain 3 execute in parallel since they use different hardware resources. Domains 1 and 2 use the CPU fairly.

Domain 3 completes its execution. Therefore, the execution of Domain 3 essentially makes I/O requests to the OS. We ignore the tiny amount of CPU sharing to simplify the simulation while still capturing precise execution interims.

C. Configuring PM Policies

The PM policy language that we have developed can express a broad range of workload shaping energy optimizations. However, it does not address the problem of configuring the parameters of a PM policy. The policy parameters have a significant impact on the performance of an app and determine the amount of energy that may be saved. To overcome this challenge, we have developed a tool for automatically configuring a PM policy.

The input to the configuration tool is a parametric policy. The configuration tool starts by analyzing the policy and classifying its parameters as controlling either data-driven or time-driven parameter. A data-driven parameter makes PM decisions based on the state of the queue. We identify the batching parameters by inspecting the guards of the `execute` commands. In contrast, time-driven parameters control the timeouts, which are essential for handling dynamic workloads. We identify the timeout parameters by inspecting the second argument of the `timeout` handler.

The overall strategy to configure the policy parameters is to first configure the data-driven parameters and then the time-driven parameters. We have developed two configuration approaches that build on grid search and gradient descent, respectively. The grid search exhaustively iterates over all possible batching configurations using a grid of possible values for each batch parameter. As we consider each configuration, we maintain the solution that provides the minimum energy consumption and meets the end-to-end deadline. For each batching configuration, we use the simulator to determine its energy consumption and the maximum end-to-end latency. If the maximum end-to-end latency of the considered configuration is within the end-to-end deadline and the energy consumption is the best solution evaluated thus far, we update the best solution with this configuration. If the maximum end-to-end latency exceeds the deadline, it may be possible to reduce the latency of the configuration by tuning the time-driven parameters. We proceed with this step if the considered configuration has better energy consumption than the current best solution. We iteratively decrease the time-driven parameters until the deadline is met or the energy consumption

becomes worse than the current solution.

The grid search is usually computationally feasible for apps that have a few domains. This is possible because of the simulator, which is invoked to evaluate the performance of each configuration, is scalable as shown in Section IV-D. In addition, the grid search can mostly evaluate multiple configurations in parallel. We ensure that the updates of the best solution are atomic. Nonetheless, a gradient descent based search is more computationally efficient despite potentially suboptimal solutions due to local minima. The gradient descent search works by initially setting the data-driven parameters to the smallest value. Then, we evaluate the impact of increasing each parameter by a fixed amount. Note that these operations can be done in parallel. Similar to the grid search method, we attempt to reduce the latency of the considered configurations if they exceed the deadline. In the next iteration, we select the configuration that provides the best energy consumption and meets the end-to-end deadline. The optimization stops when the reduction in energy consumption falls below a threshold.

D. Prototype Implementation

We have developed a source-to-source compiler that transforms a StreamIt program into an Android service that runs in the background. The result of the compilation process is a complete Android project. This project can be referenced from Android apps that typically provide a user interface for controlling the service. The compiler translates each component type in a StreamIt program into a Java class. The compiler also generates the code necessary for the Android service implementation. This code implements the standard APIs for Android services and manages the instantiation of the stream program. Further, the service provides an additional interface for loading PM policies and modifying their parameters. The functionality common to Gratis apps is included in a runtime library that provides support for managing sensors, network communication, and the event system used for PM.

The compiler partitions the StreamIt program into domains. The first step is to determine what resources are used by each component. The compiler will generate an error if a component uses more than one hardware resource. Next, the partitioning process proceeds greedily. We create the initial domain that includes the source of the stream program. The immediate successors of the source are added to the domain if they use the same hardware resources. Otherwise, a new domain is created, and the process is started recursively with the component requiring a different hardware resource as the source of the new domain. Each domain will be executed as a different thread and manages a power lock. The power lock is acquired when the domain starts executing and released when the domain completes its execution.

The exchange of data between components is managed using FIFO queues. The compiler differentiates the exchange of frames between components in the same domain and those in different domains. Since components pertaining to the same domain run in the same thread, their queues do not need to be synchronized. In contrast, the data exchange between domains must be synchronized. The compiler generates code to instantiate the appropriate type of queues at run-time. The size of the queue is configured using the app simulator to determine the peak value observed during simulations. Note that by pre-allocating memory for each queue, we avoid the overhead of garbage collection that previous stream engines have shown to be significant.

The execution of the domains is managed by a scheduler. A nice property of the proposed coordination language is that it associates event handlers with specific components and queues. Accordingly, a queue maintains a set of guarded commands that it needs to evaluate. The guarded commands are evaluated when data is inserted into the queue which changes the values `num_input`, `num_output`, `input_slack`, and `output_slack`. When a guard is true, the scheduler generates a `pre` event before executing the domain, and a `post` event after the execution completes [12], [13].

IV. Experiments

The goal of this section is to evaluate the efficacy of the proposed PM methodology for developing energy-efficient MSAs. We are interested in answering the following questions:

- Can Gratis save significant energy using workload shaping policies? If so, what optimizations are most effective?
- How accurate are the performance predictions?
- Can the parameters of Gratis PM policies be configured effectively and efficiently?

A. Methodology

We have developed mobile apps that implement two common tasks in mobile sensing: tracking the user social interactions using speaker identification techniques (`SI` app) and recognizing the user physical activities from motion sensors (`AR` app). We have evaluated the two apps using several workload shaping optimizations that combine batching, scheduled concurrency, and adaptive sensing. Our experiments focus on evaluating the performance of each app in isolation. This decision is motivated by three factors: (1) Empirical studies show that only a few apps typically run in the background so there will be minimal interference with the app. (2) The OS provides mechanisms such as *cgroups* to provide isolation between groups of processes and control resource isolation. Accordingly, it is reasonable to focus on the function of an app in isolation. (3) The results from using a single app at a time are realistic for mobile phones that run a small number of MSAs in the background and for wearable and IoT devices (e.g., smartwatches) that usually run one or a small number of apps.

The `SI` app collects audio samples of type float at 44KHz. The app is partitioned into four domains responsible for reading audio frames, extracting audio features, uploading them to a remote server, and displaying them. `SI` computes fourteen Mel-frequency Cepstral Coefficients (MFCCs) [14] from the collected frames. MFCCs have been extensively used for speaker identification and speech recognition on mobile phones [2], [15], [16]. The app may use either static or

adaptive sensing. In the case of static sensing, SI works in duty cycles by alternating between reading audio for a period of P seconds and sleeping for $3P$ seconds. Adaptive sensing is implemented using a voice activity detector to determine whether a frame contains speech. The voice activity detector is based on the algorithm proposed by Moattar et al. [17]. When no speech is detected, the app stops sampling for 11.1 seconds. Otherwise, the app continues collecting audio data.

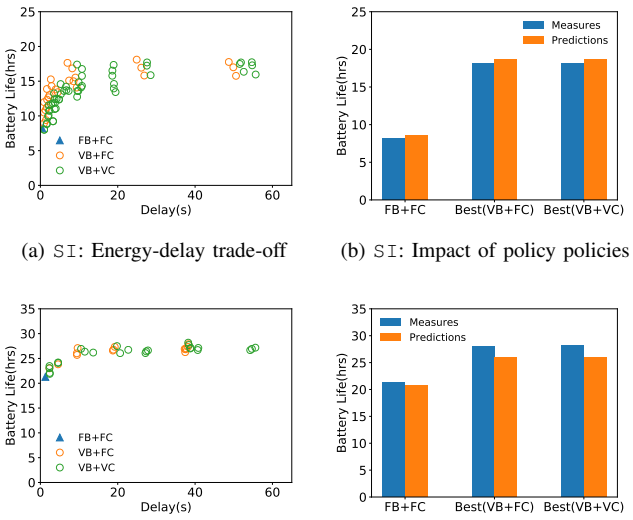
The AR app collects samples from the accelerometer at 225Hz that are aggregated into frames of size 128. Similar to the SI app, the AR app is partitioned into four domains responsible for collecting acceleration readings, extracting features, uploading to the server, and displaying the features. The app extracts the energy and the entropy for each axis of the accelerometer. We have evaluated the app using both continuous and adaptive sensing. Adaptive sensing is implemented by determining whether the collected frames include any motion. If no motion is detected, the app stops collecting samples for 18 seconds.

The SI and AR apps may be configured to run in real-time or use a previously recorded trace file. We use the latter capability for the adaptive sensing experiments to evaluate the performance of PM policies in a consistent manner. The performance of the SI file was evaluated using several long audio recordings. The audio was collected in the homes of older adults as part of a previous study. Similarly, the performance of AR is evaluated using long acceleration traces. The traces were obtained from a publicly available dataset.

The experiments were performed on Nexus 6 mobile phone running Android 5.0.2. The phone used Wi-Fi to connect to an access point located in the same room. The SI and AR apps have been tested using several PM policies configured with different parameters. Each experiment took five minutes during which we measured the processing latency and the energy consumption. The processing latency was measured directly by the application calling the standard Java API `nanoTime()`. Power consumption was measured using an external power meter from Monsoon Solutions [18]. The energy consumption was calculated by summing up the instantaneous power measurements. Based on the computed energy consumption, we estimate the battery life of the phone when assuming a capacity of 3000 mAh.

B. Gratis Extends Battery Life

In this section, we evaluate the ability of Gratis to save energy using different PM policies. We start by considering energy optimizations that combine batching (B) and scheduled concurrency (C) without adaptive sensing. We generated several PM policies that we instantiated with different parameter values. The policy FB + FC uses fixed batching and fixed concurrency. Fixed batching indicates that the domains process data beyond minimum thresholds. This baseline represents the energy consumption of a stream program that is executed without any energy optimizations. Fixed concurrency indicates that the domains execute one after another as soon as possible without seeking to overlap hardware access.



(a) SI: Energy-delay trade-off (b) SI: Impact of policy policies

(c) AR: Energy-delay trade-off (d) AR: Impact of policy policies

Fig. 6. The energy-delay trade-off for SI and AR when using *static sensing*. Batching significantly improves energy efficiency. Combining batching with scheduled concurrency provides little additional improvement.

This baseline shows the performance a naive execution of a stream program would have. The policy VB + FC uses batching but fixes concurrency as described above. We have evaluated the policy by configuring each domain with the following batch sizes: 1, 16, 32, 256, 512, 1024, and 2048. The line VB + VC includes the results from multiple policies that use batching and controlled scheduling. Some of the policies overlap sensing with feature extraction or feature extraction with network upload. We use a total of 8 policies with different scheduled concurrency settings.

Figures 6a and 6c show the energy-delay trade-off when SI and AR use static sensing. The figures show that significant energy savings can be achieved by controlling the energy-delay trade-off in an MSA. For example, the SI app can run for 7 hours when data is processed as soon as possible by setting the end-to-end deadline to zero. In contrast, when the end-to-end deadline is increased to 60 seconds, the battery life is extended to almost 19 hours, a 2.7 times improvement in battery life. The AR app is less energy intensive. The phone can run AR for 20 hours when the end-to-end deadline is zero. Setting the deadline to 60 seconds extends the battery life to 27 hours, which is a 1.35 times improvement. Part of the reason for the better battery life of the AR app is that the Nexus 6 includes a specialized co-processor for motion sensing. However, even with the use of specialized hardware, there is still a benefit of using workload shaping optimizations. The energy-delay trade-off shows that increasing the deadline yields diminishing energy savings. In our apps, most of the energy savings can be obtained if the user is willing to tolerate a delay of about 10 seconds.

Figures 6b and 6d plot the best configurations for the three classes of policies that we consider for SI and AR respectively. We plot both the performance measured directly and the performance estimated using the app simulator. The figures indicate that the policies that incorporate batching

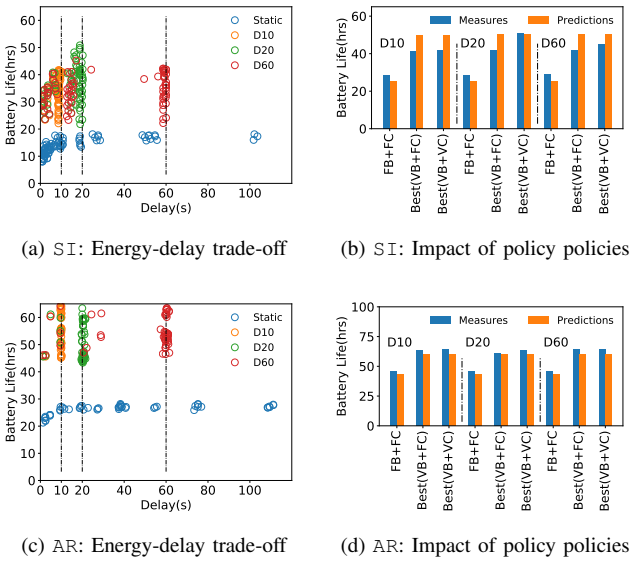
provide significant energy savings over the baseline. To our surprise, including scheduled concurrency did not provide significant energy savings over the best batching policy. Scheduled concurrency provides no additional improvement for SI. However, the best policy for AR is the one that overlaps the execution of the audio and feature_extraction components concurrently with the upload component. This policy extends the battery life by an additional 13.2 minutes over Best(VB+FC) (note that this is not visible in the figure due the magnitude of the y-axis). The figures show that the app simulator predicts the energy consumed by both apps with reasonable accuracy. We will analyze the accuracy of the simulator in more details in the next subsection.

An effective approach to saving energy is to reduce the workload of the system by introducing adaptive sensing. We have integrated adaptive sensing in the previously generated policies. The policies were generated using end-to-end deadlines of 10, 20, and 60 seconds respectively. The adaptive sensing policies use timeout handlers to trigger the domain executions once the slack falls below a threshold. The slack thresholds were configured by the policy configuration tool.

Figures 7a and 7c show the energy-delay trade-off when adaptive sensing is used. For comparison, we include the static sensing data from the previous experiment. In contrast to the static sensing case, most of the configurations are concentrated around the deadline given sufficient batching. This is because of the timeout handler that triggers the execution of components when the slack of the frames in their queues falls below a threshold. It is easy to see that adaptive sensing significantly increases the battery life of the phone. This is because the apps intelligently determine when it is necessary to remain awake. Adaptive sensing is effective in extending the battery life by 7 and 3 times for SI and AR respectively. Figures 8d and 7d plot the best configurations for the three classes of policies that we previously defined when adaptive sensing is used. As in the static sensing case, we observe that batching significantly increases the battery life of the app. However, policies with different scheduled concurrency may provide some additional energy savings for adaptive sensing. When SI has a deadline of 10 seconds, batching increases the lifetime from 29 to 41 hours. Similar differences can be observed for the other deadlines. If the user is willing to increase the end-to-end deadline from 10 to 60 seconds, the phone's battery life may be extended by 4 more hours with combined scheduled concurrency. Note for SI with a deadline of 20 seconds, incorporating scheduled concurrency can extend the battery life by additional 8 hours over the best batching. AR has a similar behavior to SI with a smaller improvement of scheduled concurrency that extends the battery life by 2 hours given a deadline of 20 seconds.

C. Gratis Apps Have Composable Performance

We have evaluated the accuracy of the app simulator by comparing the difference between the predicted and measured performance. Figure 8 plots the accuracy of the predictions for the considered apps. Overall, the average error for energy



(a) SI: Energy-delay trade-off (b) SI: Impact of policy policies

(c) AR: Energy-delay trade-off (d) AR: Impact of policy policies

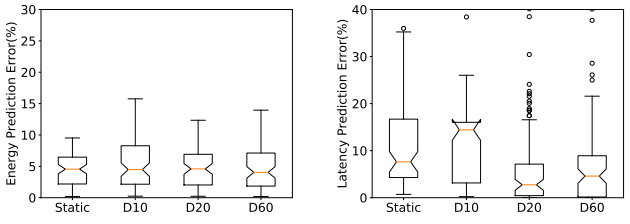
Fig. 7. The energy-delay trade-off for SI and AR when using *adaptive sensing*. Adaptive sensing significantly reduces energy consumption relative to static sensing. Batching significantly improves energy savings.

predictions is 7%. The error for latency is 15% on average. We have tracked the main source of the inaccuracies to be the Android system alarm wake-up delay which could delay the delivery of events by as much as 2 seconds. The SI latency error is larger in the cases of static sensing and adaptive sensing with a short deadline because the measured end-to-end delays are small such that a small variation could lead to large errors in percentage. Hence, this shows that our assumption that the performance of Gratis apps is composable is reasonable.

We remark that the overall performance accuracy is slightly better in the static sensing case than adaptive sensing. This is not surprising since in the adaptive case, we also have to cope with variations in user input. Perhaps more interestingly, there is a significant dependency between the errors and the deadlines when adaptive sensing is used. The reason for this is because when deadlines are tight, the fraction of the time that contributes to the end-to-end latency is dominated by the time required to execute the domains. As the deadline is increased, the time that we artificially inject for workload shaping starts dominating the end-to-end latency. Therefore, it is easier to estimate the delays that we introduced than predicting the domain execution time based on average performance profiles.

D. Gratis App Simulator is Scalable

A key ingredient to the effectiveness of the policy configuration tool is the scalability of the simulator. The simulator is invoked to evaluate the performance of each policy. The most demanding use of the simulator is to simulate a policy that employs adaptive sensing. Figure 9 plots how the simulation time increases with the length of the trace used by the simulator. The performance of the simulator mainly depends on the batching parameters used by the policy. The figure plots the best-case and the worst-case simulation time for a given



(a) SI: Energy prediction errors (b) SI: Latency prediction errors

(c) AR: Energy prediction errors (d) AR: Latency prediction errors

Fig. 8. Accuracy of energy and delay predictions for the app simulator

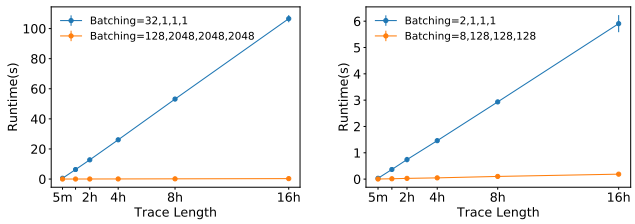
App	Search Method	Best(VB + FC)	Best(VB + VC)
SI	Grid	14717 μAh / 254 s	14717 μAh / 7166 s
	Gradient	14952 μAh / 56 s	14952 μAh / 66 s
AR	Grid	220 mAh / 183 s	220 mAh / 2022 s
	Gradient	221 mAh / 78 s	221 mAh / 88 s

trace length. The simulator can process traces that are several hours long in a few seconds.

Table IV-D shows the results of configuring the PM policies for SI and AR using adaptive sensing, batching, and controlled scheduling. The configuration tool used traces of 10 minutes and 5 hours for SI and AR, respectively. We report both the energy consumption and the total simulation time using the grid search and the gradient descent method. Configuring all the policies with varying batching and scheduled concurrency for SI requires nearly 2 hours when the grid search is used. This time is reduced to 1 minute or so by using the gradient descent method with merely 1.6% more energy consumption of the best PM policy. Similarly, configuring the policies for AR requires 33.7 minutes when using the grid search. This time is reduced to 1.47 minutes by using the gradient descent search with less than 1% energy consumption difference from the identified best PM policy. These results show that it is computationally feasible to configure PM policies automatically and that the gradient descent search method provides an effective approach to reducing configuration time.

V. Related Work

Researchers have developed a wide range of techniques for managing the trade-off between energy consumption and performance. These methods reduce the energy consumption of a sensing task by optimizing the subset of sensors that are used, the time when they are sampled, and the algorithms used to make inferences. For example, Kobe constructs offline efficient sensing pipelines by optimizing the features and classifiers



(a) Energy Prediction Errors (b) Latency Prediction Errors

Fig. 9. Simulation time for simulating SI and AR with adaptive sensing.

that are used [19]. Similarly, Orchestrator constructs multiple variants of a sensing pipeline and dynamically switches between these variants at run-time [20]. ACE saves energy by caching inference results across apps and by substituting the use of power-hungry sensors (e.g., GPS) with that of lower-power sensors (e.g., motion sensors) whenever possible [21]. In this paper, we focus on a broad class of PM policies that control the time when operations are performed subject to soft end-to-end deadlines. Our techniques save energy by shaping the workload so that it can be processed in a more energy-efficient manner without sacrificing sensing accuracy. The unique aspect of our work is the ability to estimate the impact of a PM policy on the energy and delay of an app at compile time.

Our solution leverages the use of high-level abstractions for writing energy-efficient programs. Several recent works have considered this approach. EnergyTypes allows developers to specify phased behavior and energy-dependent modes of operations in their application using a type system to dynamically adjust the CPU frequency and application fidelity at run-time to save energy [22]. EnerJ employs a type system for a developer to specify which data flows in their apps can be approximated to save energy and guarantees isolation of precise and approximate components [23]. Closer to our work, Tempus uses annotations that control when power-hungry operations are invoked [4]. A limitation of these approaches is the required deep understanding of power management, operating systems, and programming languages. More importantly, restructuring an app has an unpredictable impact on its energy consumption and delay. As a result, the developer must re-profile the app even when making minor changes.

The closest related works are the systems that use stream programs as a representation for mobile apps. Green Streams [5] and StreaMorph [24] focus dynamic voltage and frequency scaling (DVFS). Both papers recognize that executing streams as soon as possible results in energy inefficiencies. Green Streams addresses this problem by ensuring that components are executed at the same rate. StreaMorph further reduces energy consumption by compiling multiple versions of a stream program and switching between them at run-time. Unfortunately, applying DVFS for MSAs is usually ineffective because each invocation of an MSA component produces only a small amount of data that cannot be processed efficiently. Gratis provides a flexible and general mechanism for speci-

fying a wider range of PM policies that coordinate multiple hardware resources. Energy savings are the result of creating batches of data that can be processed in an energy-efficient manner. SymPhoney [3] is a stream execution engine that focuses on handling overload conditions due to interfering applications. In contrast, Gratis focuses on the more common situation when an app executes with minimal interference as a background service. Moreover, Gratis provides two additional improvements. (1) Gratis uses a simulation-based technique to determine the energy and delay of a PM policy in a computationally efficient manner. (2) Building on this property, Gratis optimizes the parameters of a PM policy to reduce energy consumption further.

VI. Conclusions

Gratis is a novel paradigm for incorporating workload shaping energy optimizations with predictable performance of MSAs. We presented an innovative programming model that combines stream graphs for specifying the functional aspects of an app and PM policies for controlling their runtime execution. We provide a coordination language that can express a broad range of workload shaping energy optimizations including those for batching, scheduled concurrency, and adaptive sensing. Gratis simplifies the introduction of PM policies by allowing developers to cleanly separate the functional aspects of an MSA from its power management. The developed programming abstraction also supports evaluating the performance of concrete policies at compile time and automatically configuring parametric policies. The key to these capabilities is an accurate and scalable simulator that is based on the observation that MSAs have composable performance.

We demonstrated that our approach is both flexible and expressive by incorporating workload shaping optimizations in two realistic apps. Our experimental results show that workload shaping optimizations can save significant energy consumption. For example, the `SI` app with static sensing can run for only 7 hours when data is processed as soon as possible. The battery life can be extended to almost 19 hours when the deadline is relaxed to one minute. The improvement is the result of applying batching and scheduled concurrency optimizations. Additional energy savings may be achieved by using adaptive sensing with combined scheduled concurrency to extend the battery life to 45 hours or more. The performance improvement for AR is equally impressive. AR can operate for 20 hours without optimizations. The use of batching and scheduled concurrency increases the battery life to 27 hours and even 60 hours with adaptive sensing. It is worth noting that the energy savings come with minimal cost to the developer. We have extensively evaluated the performance of our simulator. The simulator can predict the energy and delay with average errors of 7% and 15% respectively even when applications have variable workloads. The simulator is scalable simulating hours of traces in a few minutes. These results demonstrate that it is feasible to estimate the performance of MSA at compile time accurately and are the basis for efficiently configuring PM policies.

Acknowledgements

This work is supported in part by NSF CAREER Award CNS-1750155.

References

[1] (2017) Moves activity diary. [Online]. Available: https://www.moves-app.com/

[2] Y. Lee, C. Min, C. Hwang, J. Lee, I. Hwang, Y. Ju, C. Yoo, M. Moon, U. Lee, and J. Song, "Sociophone: Everyday face-to-face interaction monitoring platform using multi-phone sensor fusion," in *MobiSys*, 2013.

[3] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song, "SymPhoney: a coordinated sensing flow execution engine for concurrent mobile sensing applications," in *SenSys*, 2012.

[4] N. Nikzad, M. Radi, O. Chipara, and W. G. Griswold, "Managing the energy-delay tradeoff in mobile applications with Tempus," in *Middleware*, 2015.

[5] T. W. Bartenstein and Y. D. Liu, "Green streams for data-intensive software," in *ICSE*, 2013.

[6] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *ICCC*, 2002.

[7] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 24–35, 1987.

[8] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "Livelab: measuring wireless networks and smartphone users in the field," *SIGMETRICS Performance Evaluation Review*, 2011.

[9] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *EuroSys*, 2012.

[10] W. Thies and S. Amarasinghe, "An empirical characterization of stream programs and its implications for language and compiler design," in *PACT*, 2010.

[11] I. Molnar, "Modular scheduler core and completely fair scheduler," https://lwn.net/Articles/230501/, 2007.

[12] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "Xstream: A signal-oriented data stream management system," in *ICDE*, 2008.

[13] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett, "Design and evaluation of a compiler for embedded stream programs," in *ACM Sigplan Notices*, 2008.

[14] P. Mermelstein, "Distance measures for speech recognition, psychological and instrumental," *Pattern recognition and artificial intelligence*, vol. 116, pp. 374–388, 1976.

[15] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell, "The jigsaw continuous sensing engine for mobile phone applications," in *SenSys*, 2010.

[16] E. Miluzzo, C. T. Cornelius, A. Ramaswamy, T. Choudhury, Z. Liu, and A. T. Campbell, "Darwin phones: the evolution of sensing and inference on mobile phones," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 5–20.

[17] M. H. Moattar and M. M. Homayounpour, "A simple but efficient real-time voice activity detection algorithm," in *ESPC*, 2009.

[18] Monsoon Solutions, "Power Monitor laboratory equipment," http://msoon.com/LabEquipment/PowerMonitor/, 2017.

[19] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao, "Balancing energy, latency and accuracy for mobile sensor data classification," in *SenSys*, 2011.

[20] S. Kang, Y. Lee, C. Min, Y. Ju, T. Park, J. Lee, Y. Rhee, and J. Song, "Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments," in *PerCom*, 2010.

[21] S. Nath, "ACE: Exploiting correlation for energy-efficient and continuous context sensing," in *MobiSys*, 2012.

[22] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu, "Energy types," in *ACM SIGPLAN Notices*, vol. 47, no. 10, 2012, pp. 831–850.

[23] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *PLDI*, 2011.

[24] D. Bui and E. A. Lee, "Streamorph: A case for synthesizing energy-efficient adaptive programs using high-level abstractions," in *EMSOFT*, 2013.