Summer 2017

Stream processing optimizations for mobile sensing applications

Farley Lai
University of Iowa

STREAM PROCESSING OPTIMIZATIONS FOR MOBILE SENSING APPLICATIONS

by

Farley Lai

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

August 2017

Thesis Supervisor: Assistant Professor Octav Chipara

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Farley Lai

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the August 2017 graduation.

Thesis Committee: _____
Octav Chipara, Thesis Supervisor

_____
Sukumar Ghosh

_____
Ted Herman

_____
Alberto Segre

_____
Casare Tinelli

Dedicated to those in love with sensing the world

ACKNOWLEDGEMENTS

A Ph.D. program happens to be a long journey and mine is finally approaching the end. I heartily appreciate the research experience with my advisor, Prof. Octav Chipara, whose relentless exploration of the unknown always impresses me. His supervision refills motivations timely. His guidance sheds light on insights eventually. His optimism leads the inception of innovations to no limit. Dr. Chipara never misses opportunities to refine my writing and presentations in all academic occasions. It was off to an amazing start when we both were new to this department years ago. As a research collaborator, it was proved worth to have him to broaden my horizon in computer science for sure.

I would also like to especially thank my committee members, Prof. Segre, Prof. Herman, Prof. Ghosh, and Prof. Tinelli, for their valuable advice and support. Your criticism reminds me of overlooked details without exception in time. You enlightenment tremendously paves the way to my Ph.D. success. Every time interacting with you is no doubt a blast.

Present or past, everyone in the Mobile Systems Lab at the University of Iowa serves as source of inspiration. Leon hinted the ultimate way to present my work. Behnam gave constructive feedback on my academic performance. Marjan made the finest review of my write-ups. Shabih as a loyal colleague listened to my nonsense. Ryan showed me what perseverance is all about. Austin, Moosa, and Dhruv provided me with unlimited refreshment. I was so lucky to have you over the years.

Last but not least, our department staff, Catherine, Sheryl, and Matthieu deserve my cordial admiration. Catherine redefined hard working and productivity. Sheryl demonstrated the art of coordination. Matthieu turned impossibility into reality out of the blue. Without you all, I may not

pursue my Ph.D. in peace. Of course, my parents and family members remain the steady backup for everything. My homecoming is scheduled for you to be proud of sooner or later.

Still, too much is never enough. All good things must come to an end but my gratitude will go on and on.

ABSTRACT

Mobile sensing applications (MSAs) are an emerging class of applications that process continuous sensor data streams to make time-sensitive inferences. Representative application domains range from environmental monitoring, context-aware services to recognition of physical activities and social interactions. Example applications involve city air quality assessment, indoor localization, pedometer and speaker identification. The common application workflow is to read data streams from the sensors (e.g, accelerometers, microphone, GPS), extract statistical features, and then present the inferred high-level events to the user. MSAs in the healthcare domain especially draw a significant amount of attention in recent years because sensor-based data collection and assessment offer finer-granularity, timeliness, and higher accuracy in greater quantity than traditional, labor-intensive, data gathering mechanisms in use today, e.g., surveys methods. The higher fidelity and accuracy of the collected data expose new research opportunities, improve the reliability and accuracy of medical decisions, and empower users to manage personal health more effectively.

Nonetheless, a critical challenge to practical deployment of MSAs in real-world is to effectively manage limited resources of mobile platforms to meet stringent quality of service (QoS) requirements in terms of processing throughput and delay while ensuring long term robustness. To address the challenge, we model MSAs in dataflows as a graph of processing elements that are connected by communication channels. The processing elements may execute in parallel as long as they have sufficient data to process. A key feature of the dataflow model is that it explicitly capture parallelism and data dependencies between processing elements. Based on the graph composition, we first proposed CSense, a stream-processing toolkit for robust and high-rate MSAs. In

this work, CSense provide a simple language for developers to describe their sensing flow without the need to deal with system intricacy, such as memory allocation, concurrency control and power management. The results show up to 19X performance difference may be achieved automatically compared with a baseline using the default runtime concurrency and memory management.

Following this direction, we saw the opportunities that MSAs can be significantly improved from the perspective of memory performance and energy efficiency in view of the iterative execution. Therefore, we next focus on optimizing the runtime memory management through compile time analysis. The contribution is a stream compiler that captures the whole program memory behavior to generate an efficient memory layout for runtime access. Experiments show that our memory optimizations reduce memory footprint by as much as 96% while matching or improving the performance of the StreamIt compiler with cache optimizations enabled.

On the other hand, while there is a significant body of work that has focused on optimizing the throughput or latency of processing sensor streams, little to no attention has been given to energy efficiency. We proposed an accurate offline energy prediction model for MSAs that leverages the pipeline structure and iterative execution nature to search for the most energy saving batching configuration w.r.t. a deadline constraint. The developers are expected to visualize the energy delay trade-off in the parameter space without runtime profiling. The evaluation shows the worst-case prediction errors are about 7% and 15% for energy and latency respectively despite variable application workloads.

PUBLIC ABSTRACT

Mobile sensing applications (MSAs) are an emerging class of applications that process continuous sensor data streams to make time-sensitive inferences. Representative application domains range from environmental monitoring, context-aware services to recognition of physical activities and social interactions. Example applications involve city air quality assessment, indoor localization, pedometer and speaker identification. The common application workflow is to read data streams from the sensors (e.g, accelerometers, microphone, GPS), extract statistical features, and then present the inferred high-level events to the user. MSAs in the healthcare domain especially draw a significant amount of attention in recent years because sensor-based data collection and assessment offer finer-granularity, timeliness, and higher accuracy in greater quantity than traditional, labor-intensive, data gathering mechanisms in use today, e.g., surveys methods. The higher fidelity and accuracy of the collected data expose new research opportunities, improve the reliability and accuracy of medical decisions, and empower users to manage personal health more effectively.

Nonetheless, a critical challenge to practical deployment of MSAs in real-world is to effectively manage limited resources of mobile platforms to meet stringent quality of service (QoS) requirements in terms of processing throughput and delay while ensuring long term robustness. To address the challenge, we model MSAs in dataflows as a graph of processing elements that are connected by communication channels. The processing elements may execute in parallel as long as they have sufficient data to process. A key feature of the dataflow model is that it explicitly capture parallelism and data dependencies between processing elements. Based on the graph composition, we first proposed CSense, a stream-processing toolkit for robust and high-rate MSAs. In

this work, CSense provide a simple language for developers to describe their sensing flow without the need to deal with system intricacy, such as memory allocation, concurrency control and power management. The results show up to 19X performance difference may be achieved automatically compared with a baseline using the default runtime concurrency and memory management.

Following this direction, we saw the opportunities that MSAs can be significantly improved from the perspective of memory performance and energy efficiency in view of the iterative execution. Therefore, we next focus on optimizing the runtime memory management through compile time analysis. The contribution is a stream compiler that captures the whole program memory behavior to generate an efficient memory layout for runtime access. Experiments show that our memory optimizations reduce memory footprint by as much as 96% while matching or improving the performance of the StreamIt compiler with cache optimizations enabled.

On the other hand, while there is a significant body of work that has focused on optimizing the throughput or latency of processing sensor streams, little to no attention has been given to energy efficiency. We proposed an accurate offline energy prediction model for MSAs that leverages the pipeline structure and iterative execution nature to search for the most energy saving batching configuration w.r.t. a deadline constraint. The developers are expected to visualize the energy delay trade-off in the parameter space without runtime profiling. The evaluation shows the worst-case prediction errors are about 7% and 15% for energy and latency respectively despite variable application workloads.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

Mobile sensing applications (MSAs) emerge as the data collection front end for making timely inferences from continuous sensor data streams. Representative application domains range from environmental monitoring, context-aware services to recognition of physical activities and social interactions. Example applications involve city air quality assessment [2], indoor localization [3], pedometer [4] and speaker identification [5]. The common application workflow is to read data streams from the sensors (e.g, accelerometers, microphone, GPS), extract statistical features, and then present the inferred high-level events to the user. MSAs in the healthcare domain especially draw a significant amount of attention in recent years because sensor-based data collection and assessment offer finer-granularity, timeliness, and higher accuracy in greater quantity than traditional, labor-intensive, data gathering mechanisms in use today, e.g., surveys methods [6]. The higher fidelity and accuracy of the collected data expose new research opportunities, improve the reliability and accuracy of medical decisions, and empower users to manage personal health more effectively.

However, developing robust and high-rate MSAs is challenging because of high sampling rate, error-prone concurrency APIs, and low-level power management on modern mobile platforms such as Android. High sampling rate imposes heavy memory workload while the concurrency APIs and low-level power management tend to cause programming errors and energy bugs [7] at runtime. In view of the performance improvement in our initial work due to static memory allocations, the memory management for MSAs is worth further investigation to achieve both high performance and reduced resource usage at the same time. A MSA can be broken down into the sensing and the

stream processing phases. While the sensing phase is straightforward, the stream processing can be arbitrarily complex and compute intensive. The synchronous dataflow (SDF) is the conventional model of computation for stream processing where the program is represented as a stream flow graph (SFG) with processing units as nodes and FIFO communication channels as edges. A node is schedulable for execution if its input is ready. Unfortunately, naive communication channel implementations based on copy-by-value semantics may incur significant performance overhead. It is generally hard to capture the whole program memory behavior but static analysis is possible when programs are written in a SDF language, allowing the stream compiler to optimize the program by transforming the program structure.

In additional to performance concerns, another critical challenge to making MSAs practical in real-world deployment is to effectively manage the limited energy resources of mobile platforms to meet stringent quality of service (QoS) requirements in terms of processing throughput and delay. While there is a significant body of work that has focused on optimizing the throughput or latency of processing sensor streams, little to no attention has been given to energy efficiency. To facilitate latency constraints and asynchronous processing, we extends the stream language to specify end-to-end deadlines and graph partitioning. Different graph partitions execute in separate threads and the synchronous queue size between partitions controls the buffering latency. Unfortunately, previous work such as [8, 9] based on runtime profiling is infeasible to search for energy efficient batching configurations due to time-consuming state space exploration. Worse, the accuracy of runtime energy profilers relies on the OS resource usage updates which are not timely in less than 10ms to save runtime overhead. This makes existing power models infeasible for high-rate MSAs with short energy consuming operations. If intensive logging is incorporated to capture the execution timing, the runtime overhead is likely to outweigh the energy savings. As a result,

	CSense [43] (IPSN 14')	ESMS [10] (EMSOFT 15')	Gratis (under review)
MoC	- agent-based	- synchronous dataflow	- hybrid coordination model
Annotations	- explicit memory allocation - domain constraints - frame type constraints	- explicit I/O rates - predefined memory access	- explicit I/O rates - domain constraints - deadline - sensing intervals
Analyzability	- trackable memory usage - threading boundaries	- precise memory behavior characterization	- processing delay estimation - energy usage prediction
Optimizations	- static memory pooling - queue specialization - flow analysis	- memory usage - memory performance	- batching - scheduled concurrency - selective sensing - adaptive sensing

Table 1.1. Research overview.

we are in need of an offline solution to fast evaluating the parameter space for an energy efficient configuration without real-time profiling.

The goal of this dissertation is to develop optimization tools to tackle the aforementioned technical challenges and deliver efficient runtime MSA solutions. A research overview that summarizes my Ph.D. research work in terms of models of computation (MoC), annotations, analyzability, and optimizations is given in table 1.1. Our initial work CSense [43] adopts a flexible agent-based model that allows concurrent components to operate at dynamic rates. Rich annotations provided by CSense include explicit memory allocation sources and sinks, domain threading hints, and frame size constraints. The CSense toolkit leverages the additional information from the annotations to prevent memory leaks and automate threading for synchronization safety at compile time. The resulting performance is improved with static memory pooling, hybrid synchronization primitives, and flow analysis that allocates efficient frame sizes w.r.t. frame type constraints. The evaluation of CSense indicates the memory performance likely serves as the bottleneck of sensing applications. However, the whole program behavior is difficult to analyze for further optimizations because of concurrent component executions. We consequently considered the SDF model where each component declares its processing rates statically and must access the I/O channels

using pre-defined memory operations. Despite the copy-by-value semantics imposed by the SDF on data exchange, the entire execution follows a cyclo-static schedule that may repeat infinitely. This facilitates the static analysis of the memory access per component and the characterization of the whole program memory behavior through simulating the static schedule once. Our following work ESMS [10] stands for Efficient Static Memory Management for Streaming and demonstrates the exploitation of location and temporal sharing opportunities to reduce the memory usage while eliminating unnecessary memory operations to improve the performance. In the latest work, we proposed Gratis that builds on a hybrid MoC combining SDF and CSense concurrent domains to address the energy efficiency of MSAs. The programmer is allowed to specify the deadline constraints and sensing intervals for duty cycling. The programming model supports application power management (PM) policies that incorporate workload shaping techniques such as batching, scheduled concurrency, and adaptive sensing. A simulation based approach is developed to analyze the concurrent execution timing and estimate the energy-delay trade-off based on individual domain performance profiles. This results in predictable performance for fast PM policy space exploration at compile time.

The remainder of the thesis is organized as follows. A theoretical background of stream processing models of computation is given in chapter 2. A high level description of the devised solutions focusing on the novel aspects is presented next. This section is concluded by outlining the specific research contributions in this dissertation.

1.1 Long Term Robust High-Rate Mobile Sensing Applications

Robust and high-rate mobile sensing applications require a new programming model that supports flexible application configuration, a high-level concurrency model, memory management, and compiler analysis as well as optimizations. We address the requirement by proposing a stream-

processing toolkit for developing robust and high-rate mobile sensing application in Java. The toolkit compiler includes a novel flow analysis that optimizes the exchange of data across components from an application-wide perspective. A mobile sensing application benchmark indicates that flow analysis may reduce CPU utilization by as much as 45%. Static analysis is used to detect a range of programming errors including application composition errors, improper use of memory management, and data races. We identify that memory management and concurrency limit the scalability of stream processing systems. We incorporate memory pools, frame conversion optimizations, and custom synchronization primitives to develop a scalable runtime. CSense is evaluated on Galaxy Nexus phones running Android. Empirical results indicate that our runtime achieves 19 times higher steam processing rate compared to a realistic baseline implementation.We demonstrate the versatility of CSense by developing three mobile sensing applications.

1.2 Static Memory Management for Mobile Sensing Applications

Memory management is a crucial aspect of mobile sensing applications that must process high-rate data streams in an energy-efficient manner. Our work is done in the context of synchronous dataflow models in which applications are implemented as a graph of components that exchange data at fixed and known rates over FIFO channels. We show that it is feasible to leverage the restricted semantics of synchronous dataflow models to implement efficient stream processing engines. Specifically, our memory optimization approach includes two components:

- We use abstract interpretation to analyze the complete memory behavior of a mobile sensing application to identify opportunities for sharing data across components and to determine the live ranges of exchanged samples. The static analysis is precise for a majority of considered stream applications.

- We propose novel heuristics for memory allocation that leverage the graph structure of application to optimize data exchanges between application components to achieve not only significantly lower memory footprints but also increased stream processing throughput. We incorporate code generation techniques that transform a stream program into efficient C code.

The memory optimizations are implemented as a new compiler for the StreamIt programming language. Experiments show that our memory optimizations reduce memory footprint by as much 96% while matching or improving the performance of the StreamIt compiler with cache optimizations enabled. These results suggest that highly efficient stream processing engines may be built using synchronous dataflow languages.

1.3 Workload Shaping Energy Optimizations for Mobile Sensing

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 (PM) 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 a worst-case error of 7% and delay with a worst-case 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 and synthesizing 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 optimize energy consumption in mobile sensing applications automatically.

1.4 Research Contributions

This thesis makes the following research contributions detailed in the following chapters:

- **Automation of developing robust high-rate MSAs** : In Chapter 3, we describe CSense, a stream-processing toolkit for developing robust and high-rate MSAs on Android. CSense specifies a programming model allowing to express concurrency, component I/O port types, and explicit memory management operations. A novel flow analysis is included to leverage the type information and explicit memory management to perform application-wide optimizations. Application composition errors, leaks and races can be detected at compile time. CSense also eases the integration with components written in MATLAB for common signal processing. At runtime, platform-specific power management is automatically employed for long term background MSAs.

- **Static memory management and optimization for MSAs**: Chapter 4 describes our Efficient Static Memory Management for Streaming (ESMS) [10] compiler for efficient MSAs written in StreamIt. In particular, we developed a novel static analysis that characterizes the global memory behavior of a complete stream application. The static analysis can precisely identify the location and temporal reuse opportunities in most applications. We also proposed a novel layout algorithm that leverages the identified location and temporal reuse opportunities along with the application structure to optimize the memory layout. The code generation techniques transform a stream program into efficient C code that effectively uses the generated memory layouts.

- **Accurate offline prediction of energy delay trade-off for MSAs**: In Chapter 5, we present Gratis to accurately predict the energy and delay behavior of MSAs at compile time. We proposed a novel paradigm and coordination language for specifying PM policies that implement workload shaping energy optimizations. At runtime, a scheduler coordinates the execution of the app according to a PM policy. To accurately predict the energy-delay trade-off at runtime, we developed an app simulator that can estimate the energy and delays of an app accurately from a few measurements of its constituent components. Techniques for synthesizing policy templates and for configuring their parameters are demonstrated to facilitate the PM state space exploration. The combination of these tools provides an automated solution for PM in MSAs.

CHAPTER 2

BACKGROUND

2.1 Models of Computation

A model of computation (MoC) formally specifies the behaviors and properties of a system for developers to make sensible design decisions. In the following, we will explore different models of computations that execute dataflows in parallel but have different characteristics in terms of determinacy, expressivity, and memory requirements. Determinate systems guarantee the order of output given fixed input regardless of the parallel scheduling. Expressivity shows the class of programs the language or model can express and may raise concerns about non-determinism in the system and termination of executions. The memory requirements may be bounded or unbounded and system designers have to ensure the implementation consumes no more than available resources.

The models of computation for stream processing can be traced back to the dataflow architectures in 1970s. In contrast with the control flow based von Neumann architecture where the execution of a program compiled as a sequential instruction stream follows a program counter, a dataflow program is modeled as a directed graph that has functions as nodes and communication channels as edges. The functions can be invoked simultaneously given the availability of required inputs. The graph explicitly exposes the parallelism of a program. Therefore, it can simplify program analysis and optimizations. In the following, we will present two representative dataflow models: Kahn Process Networks (KPNs) [11] and Synchronous Data Flow (SDF) [12]. The two models differ in their properties which affect our ability to build scalable stream processing systems.

2.1.1 Kanh Process Networks

The requirements of KPNs are simple but effective to guarantee desirable properties such as determinacy for parallel programming. A KPN is a network of processes connected by possibly unbounded FIFO channels. No channels are shared across processes. A process always succeeds writing to an output channel but may block waiting on an empty input channel.

A FIFO channel has a sequence of tokens, each of which represents one or more consecutive samples in the channel. A process may have multiple input and output channels. An input sequence from *multiple* input channels is represented as a sequence of tuples, each tuple containing a token per channel. The output sequence is modeled similarly. In KPNs, a process may consume and produce an arbitrary number of tokens from and to its input and output channels.

Each process is a *continuous* function which maps a possibly infinite input sequence of tuples to an output sequence of tuples. We note that KPNs consider continuous functions defined on ordered sets rather than the special case of real numbers. To formally define the *continuity* of processes, an input or output sequence is represented as an increasing *chain* of sequences $\chi = \{X_0, X_1, \ldots, X_n\}$ that captures all the possible input or output sequences in an ordered set. The variable X_i is a sequence of i tuples. Accordingly, X_0 is an empty sequence denoted by $\perp$ and X_n is an infinite sequence as $n \to \infty$. A *source* process without input is modeled as consuming an empty sequence while a *sink* process without output produces an empty sequence. χ forms a prefix ordering $\sqsubseteq$ where X_i is the prefix of X_j if $i \leq j$, i.e., $X_0 \sqsubseteq X_1 \sqsubseteq \ldots. \sqsubseteq X_n$. The property holds because channels are FIFO, i.e., χ is a complete partial order (cpo) if and only if for all subsets in χ, there exists an upper bound in χ denoted by $\sqcup\chi$ such that each element in the subset is a prefix of the upper bound. To be specific, X_n is the upper bound in χ and $\sqcap\chi$ is the empty sequence $\perp$ which serves as the prefix of any sequence and is the lower bound of χ.

A process in KPNs is a continous function $F : \chi \rightarrow \Psi$ that maps an input sequence in the increasing chain χ to an output seuqecne in another increasing chain Ψ such that the following property holds:

$$F(\sqcup\chi) = \sqcup F(\chi)$$

Informally, it says the upper bound of input to the function is the upper bound of output of the function. This implies the function is order-preserving, namely, *monotonic*. A function *F* is monotonic if given $X_i \sqsubseteq X_j$, $F(X_i) \sqsubseteq F(X_j)$ holds. It is straightforward to verify that $F(\sqcup\{X_i, X_j\}) = F(X_j) = \sqcup\{F(X_i), F(X_j)\}$ implies $F(X_i) \sqsubseteq F(X_j)$. Therefore a continuous function must be monotonic. The intuition behind monotonicity allows the process in KPNs to consume and produce sequences of tokens from and to channels incrementally. Support an input sequence $X = x_1.x_2$ such that x_1 is the prefix of X. $F(x_1) \sqsubseteq F(X)$ by continuity implies that the consumer process of F is able to incrementally process $F(x_1)$ and then the remaining output induced by x_2.

However, a monotonic function may not be continuous. In this case, it is possible for a process to wait for an *infinite* input sequence before producing any output. Consider the monotonic function F defined in Eq.(2.1) and the increasing input chain χ. In this case, $\sqcup\chi = \lim_{n\to\infty} X_n$ is infinite so $F(\sqcup\chi)$=(v). Unfortunately, $\sqcup F(\chi)$ can only produce $\perp$ because the process needs to block reading indefinitely to determine if the input is an infinite sequence. This behavior may cause non-determinism and justifies why KPNs require processes to be continuous functions. In practice, we may think processes in KPNs are monotonic and never block reading infinite input

forever.

$$F(X) = \begin{cases} \perp; \text{ if } X \text{ is finite} \\ (v); \text{ for some integer } v \text{ if } X \text{ is infinite} \end{cases} \tag{2.1}$$

Determinacy. An important property of KPNs known as the Kahn principle is that KPNs are *determinate* if given the input and internal sequences, the final output sequences are uniquely determined regardless of the process scheduling policies. This principle holds because all the processes are continuous functions on cpos and by induction, a finite composition of processes is also a continuous function. By the fixed-point theorem, there exists a least fixed point solution to the system of equations formulated from the functions in the KPN. To derive such a solution, start with $\perp$ as initial input sequences and iterate until the output sequence does not change. Specifically, the least fixed solution is the least upper bound on the cpo. Nevertheless, it is still possible to violate the semantics of KPNs without caution and cause non-determinism. This usually happens when dataflow models are implemented in an imperative host language that allows processes to communicate over shared variables, test if channels are empty, and call non-deterministic system APIs. A dataflow compiler should alert programmers the potential violation of determinacy.

Boundedness of Channels. Another concern in KPNs is whether channels have bounded capacity. A practical implementation of KPNs would require bounded memory requirements. In the next section, we will describe how we can limit the semantics of the MoC to ensure bounded channels. However, in the following we describe a solution proposed by Parks [13] proposed a dynamic scheduling approach to guarantee bounded channel buffers. In his approach, the channel capacity begins with at least one and each process would block for writing to a full channel. The scheduler then schedules enabled processes to execute. A process is enabled if its input is ready as required by KPNs. As expected, the system may reach an artificial deadlock when *all* the

processes are blocked, some of which on a full FIFO. In this case, the scheduler increases the lowest full channel capacity until the deadlock is resolved. However, implementations of KPNs based on Parks' approach may result in incomplete output since only global deadlocks are considered. Geilen and Basten [14] have improved the scheduling despite local deadlock cycles. On the other hand, if unbounded channel capacity is allowed, the fairness of scheduling should be concerned to ensure enabled processes execute infinitely often. Otherwise, unfair scheduling may lead to low throughput even though the output is still determinate.

Expressivity. KPNs have been shown to be Turing-complete in [15] given boolean tokens, initial tokens in the channels of feedback loops, memoryless processes as well as conditional constructs such as the Switch and Select. A stateful process can be transformed to be memoryless by passing its states through a self-loop channel and reading back in the next invocation. The Switch and Select implement the if-then-else conditionals in common imperative languages.

Though KPNs are powerful in the sense of expressivity, the termination and channel buffer boundedness are undecidable in general. Moreover, the runtime overhead due to the dynamic scheduling could be undesirable for high performance computing. In the next section, we introduce the synchronous data flow which models more restrictive semantics of KPNs but allows to decide the termination and derives static schedules with bounded memory at compile time.

2.1.2 Synchronous Data Flow

Similar to KPNs, the Synchronous Data Flow (SDF) models a program as a directed graph. However, SDFs require the production and consumption rates of processes during an invocation to be known statically. Process states are maintained through self-loops. The execution flow of the SDF is data-driven: the push model that starts from upstream sources to downstream sinks in a topological order. There are many derivatives and variants of SDFs (e.g., StreamIt [16] and

Lustre [17]) that adopt similar semantics for generating static schedules with bounded memory allocations and allocating initial tokens to handle feedback loops. The remainder of the section will cover these aspects of SDFs.

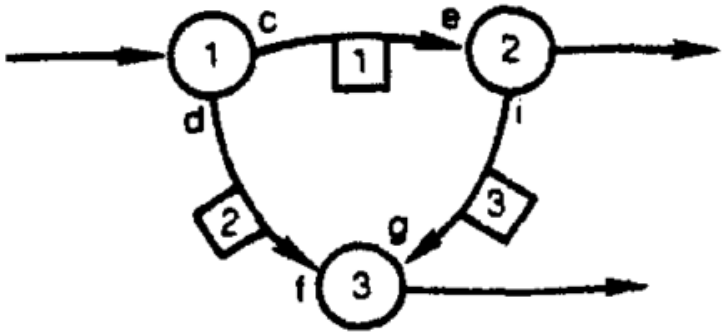


Figure 2.1. A SDF graph with nodes and edges numbered in order and letter rates.

Periodic Static Schedules. Assuming a single processor architecture, it is possible to derive a periodic admissible sequential schedule (PASS) [12] from a SDF graph and then adapt the schedule for multiprocessor architectures. A PASS is periodic to allow for processing infinite streams of data by executing the program repetitively. It utilizes a finite amount of memory and is sequential for a single processor system. The idea is to simulate process invocations and track the number of tokens in the channel buffer to find such a schedule. The schedule makes sure the buffer size in the number of tokens is the same as the beginning of executing the schedule.

To begin with, a *topology matrix* based on a SDF graph with rows representing channels and columns representing processes. An entry in the topology matrix corresponding to the number of tokens consumed or produced by a process. Each positive value indicates that the process produces data while a negative value indicates a process consumes data. Fig. 2.1 shows a simple SDF with

nodes and rate labeled edges. The topology matrix Γ of the SDF in the figure is given blow

$$\Gamma = \begin{pmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & i & -g \end{pmatrix}. \tag{2.2}$$

To simulate the invocation of a process, Γ is multiplied by an indicator column vector. The vector $v_n(i)$ indicates the process i is executed at time n by setting the i_{th} entry in v_n to one and setting all other entries to zero. In our example, $v_n(i)$ may take one of the following values:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \text{ or } \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \text{ or } \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{2.3}$$

depending on which process is scheduled. A process is *schedulable* if there is sufficient data to consume in its input channel(s). This ensures each time a process in invoked, the corresponding channel buffer never underflows. We track the number of tokens available in a channel using the column vector $b(n)$. The number of tokens in a channel after invoking a process may be computed using the following equation:

$$b(n+1) = b(n) + \Gamma v_n(i).$$

Suppose the initial buffer bize is $b(0)$, the simulation iteratively executes processes until the channel buffer size $b(n) = b(0)$ for some $n > 0$. Clearly, the channel buffer usage is bounded and equal to the maximum number of tokens on each channel during the simulation. Additionally, the sequence of invocations recorded in v_t for $0 \leq t \leq n$ constitutes one feasible PASS.

However, in some cases, the simulation never ends when there is no iteration such that $b(n) = b(0)$. This situation occurs when the rates are inconsistent. The authors in [18] proved that the necessary condition for a SDF graph to have a PASS is

$$\text{rank}(\Gamma) = s - 1,$$

where s is the number of processes. The proof is by induction on the rank of Γ. It starts with a two-node tree with rank one. Whenever a node and an edge are added to connect the new node, Γ is expanded to have a new column with a nonzero entry in the new row for the new node and edge. In this way, rank(Γ) is always equal to the number of nodes minus one given the SDF graph as a tree. Next, consider the actual SDF graph with more edges which only add rows to Γ without decreasing its rank and thus the following equation holds.

$$s - 1 \leq \text{rank}(\Gamma) \leq s$$

Since the total buffer size change is equal to $\Gamma\Sigma_{t=0}^{n} v_t$ which should be zero for boundedness, the nullity of nonzero $\Sigma_{t=0}^{n} v_t$ can only be one by the rank-nullity theorem and rank$(\Gamma) = s - 1$ as a result. If rank$(\Gamma) = s$ for a given SDF graph, any schedule will result in either deadlock or unbounded buffer size. That implies a particular channel has its buffer size either decreasing or increasing unboundedly. After determing the existence of a PASS, it is sufficient to find the schedule by simulating the process invocations. If there is a deadlock before the buffer size is restored, some additional delays are necessary to add to the initial buffer size $b(0)$. This concludes the sufficient condition for the static schedule.

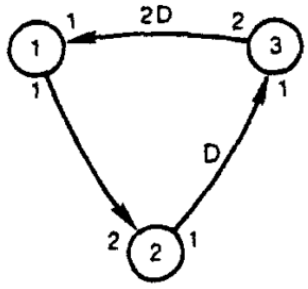


Figure 2.2. An example SDF graph for constructing a PAPS.

Parallel Schedules. Based on the PASS, constructing a periodic admissible parallel schedule (PAPS) is straightforward. For simplicity, assume the time to execute a SDF process takes an integral number of time slots. This is realistic given processors with manageable pipelining. Otherwise, some inter-process synchronization may be required. Suppose there are M processors, each of which is assigned a static schedule ψ_i to run such that for each process, the total number of invocations on all processors is a multiple of the number of invocations in a PASS, namely,

$$\Sigma\psi_i = J \cdot \phi$$

where ϕ is the corresponding PASS and *J* is some positive integer called the blocking factor to scale up the PASS for better performance. Fig. 2.2 gives an example SDF graph with a minimum PASS $\{1, 1, 2, 3\}$ for executing processes 1, 1, 2, and 3 in one period. For $J = 1$ and $J = 2$ and given $M = 2$, two possible PAPS can be constructed as follows:

$$\psi_1 = \{3\}, \psi_2 = \{1, 1, 2\} \text{ for } J = 1$$

$$\psi_1 = \{3, 1, 3\}, \psi_2 = \{1, 1, 2, 1, 2\} \text{ for } J = 2.$$

Assume it takes one time slot to execute process 1, two slots for process 2 and three slots for process 3. The makespans on both processors are illustrated in Fig. 2.3. Apparently, it is not efficient for $J = 1$ since processor 2 needs to wait for processor 1 to complete (the shadow area) while for $J = 2$, processor 2 is allowed to continue executing and both processors wait no time for each other until the end of one period of the PAPS. Given an appropriate *J*, finding the optimal partition is a well-known assembly line problem in operations research. Though it is NP-complete [19], there are efficient approximation algorithms such as [20] for such constructions.

Asynchrony. In some cases, expressing the entire program in a single SDF graph is not feasible due to the existence of asynchronous processes whose production and consumption rates are

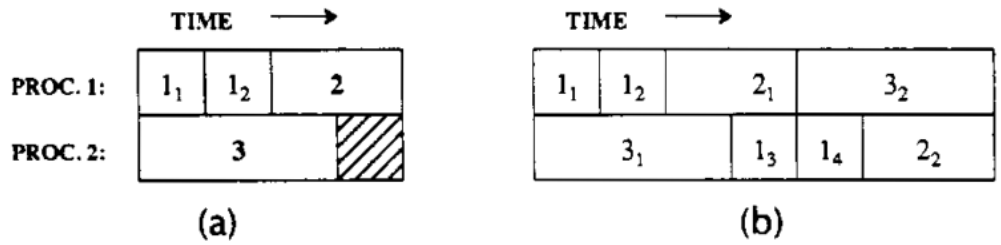


Figure 2.3. Makespans on two processors for $J = 1$ and $J = 2$ in (a) and (b).

data dependent. This implys the existence of internal *Switch* and *Select* logics which accept control inputs and change the dataflow paths at runtime. Nonetheless, a practical solution is to partition the program into separate SDF subgraphs connected by build-in Switch and Select components which are compiled as conditional statements on a single processor architecture and inter-process communication constructs on multiprocessor systems to direct the dataflow paths. In this way, each SDF subgraph simply follows its own static schedule without concerning how dataflow is routed outside the subgraphs or changing its semantics and is therefore free from non-determinism.

So far, we have gone through the main line of dataflow computational models for stream processing including the semantics, scheduling and possible non-determinism. In the next section, several optimizations based on static analyses of process semantics, channel buffer management, and dynamic scheduling will be explored. Related work and their implementations will be introduced with only essential differences from SDF.

CHAPTER 3

LONG TERM ROBUST HIGH-RATE MOBILE SENSING APPLICATIONS

3.1 Introduction

Mobile phones are capable sensing platforms that include multi-modal sensors, increasing computational and memory resources, and versatile networking capabilities. Their capabilities have enabled a new generation of mobile sensing applications (MSAs). We are interested in using mobile phones to transform how healthcare professionals collect information regarding a patient's physiology, physical activities, and social interactions. Results of recent studies on mobile health systems have shown the feasibility of collecting medical records with higher resolution than is possible through manual data collection methods [6, 21–23]. However, experience has also shown us that the development of MSAs is particularly time demanding and challenging as significant time is spent on ensuring that the system operates robustly within the resource constraints of the embedded platform.

The development of MSAs faces several challenges that are poorly addressed by existing operating systems like Android:

Concurrency: MSAs must handle data processing and asynchronous events concurrently: sensors are sampled, data is uploaded to servers, and the system responds to user interactions or changes in the environment. Such systems are difficult to implement correctly using low-level concurrency primitives such as threads or events. Thus, MSAs require a flexible concurrency model that supports static analysis to detect bugs.

High Frame Rates: MSAs collect data from one or more sensors at high rates (e.g., 44100Hz for processing audio). The collected data frames must be processed in real-time or within a few

seconds from collection and may involve expensive signal processing operations (e.g., FFT). Supporting such high rates is difficult due to the limited resources available on mobile phones.

Reliability: MSAs are intended for long-term data collection from users in unpredictable environments. This operating regime, coupled with the need to provide a positive user experience, motivates a focus on bug prevention to reduce run-time errors.

Java Run-time Environment: Although Java increases programmer productivity and reduces programming errors (compared to C/C++), it also increases the complexity of implementing MSA efficiently. Efficient implementations must manually manage memory, select appropriate concurrency mechanisms, and integrate native implementations of expensive operations.

To address these challenges, we propose CSense, a stream processing (SP) toolkit for developing MSAs that provides a programmer the following capabilities:

- Consistent with SP approaches [16, 24–27], an MSA is modeled as a directed acyclic graph of components that encapsulate reusable user code. The programming model includes three novel features: (1) a simple but expressive concurrency model that handles concurrent operations and asynchronous events efficiently, (2) a flexible type system used to specify the types of inputs and outputs for components, and (3) the explicit inclusion of memory management operations as part of the component graph.

- Our compiler includes a flow analysis that leverages type information and explicit memory management to perform application-wide optimizations. The same information is leveraged by our static analysis to identify a range of programming errors including application composition errors, incorrect usage of the memory management system, and data races. Code generation techniques are used to integrate MATLAB functions (compiled into C code) as

CSense components.

- We designed a run-time environment that supports high-rate SP on the Dalvik VM. This required careful design of memory management and concurrency. The run-time environment tightly integrates with Android's power management system.

The primary novelty of CSense is the inclusion of type and memory management information as part of the programming model to facilitate static program analysis and optimizations.

Several frameworks aim to simplify various aspects of MSA development. CSense is closely related to efforts aimed at reducing the burden of resource management [26–29] and complementary to those that focus on integration with cloud services [30] or machine learning support [31,32]. SeeMon [28] selects an informative set of sensors to track a user's context. Coordinator [29] extends these capabilities to adapt application behavior in response to resource availability. JigSaw [26] provides customized pipelines for accelerometer, microphone, and GPS sensors. A limitation of these systems is that they only support MSAs that may be defined using constrained queries [28, 29] or customized pipelines [26]. In contrast, CSense provides a high-level stream programming abstraction suitable for a broad range of MSAs. Similar to CSense, SymPhony [27] provides a general programming model for MSAs, however, it emphasizes the problem of sharing resources across multiple MSAs that run on a single device. We do not focus on resource sharing as we are interested in mobile health applications where devices are dedicated to run a single application. More importantly, SymPhony does not support flexible concurrency, compiler optimizations, or static analyses.

The benefits of the CSense run-time and optimizations have been evaluated on mobile phones. Experiments show that the use of memory pools and lock-free synchronization improves

the peak SP rate by as much as 19 times over a realistic baseline implementation. Moreover, our frame analysis reduces the number of memory copies and allows components to be executed at different rates. We show that flow analysis can reduce CPU usage by as much as 45% in a realistic application.

We have used CSense to implement three MSAs: SpeakerIdentifier, ActiSense, and AudioSense. The three systems were selected because they produce different types of workloads and pose different system challenges. SpeakerIdentifier is a CPU-intensive application that processes speech samples to determine the identity of speakers. ActiSense requires high concurrency to predict patient activities from multiple accelerometers connected to a phone over Bluetooth. AudioSense [6] delivers electronic surveys and collects audio samples to evaluate the performance of hearing aids. The key challenge of AudioSense is to collect sensor data reliably during weeklong data collection sessions.

The remainder of the chapteris organized as follows. The programming model, compiler analysis, and optimizations are presented in the next section. The run-time environment that executes CSense applications is described in Section 3.3. Micro- and macro-benchmarks that show the benefits of flow optimization and run-time environment are provided in Section 3.4. The related work is reviewed in Section 3.5. Conclusions are included in Section 5.6.

3.2 CSense Design

CSense supports the development of MSAs that are robust and require high-rate SP. The building blocks of CSense are fine-grained components that encapsulate user functionality. An application is built by connecting components to form *Stream Flow Graph* (SFG). The SFG includes type and memory management information that facilitate static compiler analyses and optimizations.

The design of CSense is based on the following principles:

CSense builds on Java: CSense components are implemented as Java classes. The Android SDK provides programmers a rich set of reusable components which, when used in conjunction with object-oriented programming techniques, can significantly reduce development time. However, this approach has disadvantages: (1) supporting high-rate SP requires careful engineering and deep understanding of the operating system internals, (2) low-level concurrency primitives provide little support for writing of safe code, and (3) it is difficult for compilers to analyze and optimize an application globally when it is structured as loosely coupled Java components. CSense addresses these limitations.

Flexible, safe, and optimized applications: Applications are modeled as SFGs, which capture application-level properties including the flow of data between components, constraints on frame types and their sizes, and concurrency. SFGs support flexible configuration, program analysis for safety, and application-level performance optimizations.

Native code: Most stream operations can be implemented efficiently in Java. However, there are cases when native implementations would significantly reduce computational overhead. CSense components may be implemented in MATLAB and compiled to native code. This has the advantage of including efficient signal processing functions that are often readily available as MATLAB toolboxes.

The remainder of truntimehis section describes the programming model and associated compiler analyses and optimizations. The runtime environment is described in next section.

3.2.1 Programming Model

Components are the building block of CSense applications. They encapsulate functionality common to MSAs including support for data collection, feature extraction, file I/O, and network-

```
1:  public class RMSClassifierM<T extends Vector>
     extends Module {
2:    InputPort<T> in = newInputPort(this, "in");
3:    OutputPort<T> above = newOutputPort(this, "above");
4:    OutputPort<T> below = newOutputPort(this, "below");
5:    double threshold;

6:    public RMSClassifierM(double threshold) {
7:      this.threshold = threshold;
8:    }

9:    public void onInput() {
10:     T v = in.getFrame();
11:     double rms = computeRMS(v);
12:     if (rms ≥ threshold) above.push(v);
13:     else below.push(v);
14:   }
15: };
```

Figure 3.1. The `RMSClassifierM` module.

```
1:  public class RMSClassifierC
     extends SimpleConfiguration {
2:    public RMSClassifierC(double threshold) {
3:      // === specify Java implementation ===
4:      super(RMSClassifierM.class);

5:      // === type definitions ===
6:      VectorC type = TypeC.newFloatVector()
7:      type.addConstraint(Constraint.GT(8000));
8:      type.addConstraint(Constraint.LT(24000));

9:      // === ports definitions ===
10:     InputPortC in = addInputPort(type, "in");
11:     OutputPortC above = addOutputPort(type, "above");
12:     OutputPortC below = addOutputPort(type, "below");

13:     // === specify internal links ===
14:     link(in, above); link(in, below)

15:     // === add component arguments ===
16:     addArgument(new Argument(threshold));
17:   }
18: };
```

Figure 3.2. Configuration of `RMSClassifierM`.

ing operations. Applications are written by connecting components into a directed acyclic graph called the *Stream Flow Graph* (SFG). We distinguish two types of components: *modules* and *configurations*. Modules provide the underlying Java implementation of a component. Existing Java libraries may be reused as part of module implementations. Configurations may be used to either (1) configure a single module (called simple configurations) or to (2) connect and configure groups of modules and configurations to create reusable components (called group configurations). A

module must have at least one configuration to be used in an application. Each application has a main group configuration that connects and configures all the modules of an application. We opted to implement both modules and configurations in Java. Using Java has many advantages including programmer familiarity, ease of integration with Android, and availability of compiler and analysis tools.

An example of a module is shown in Figure 3.1. The `RMSClassifierM` classifies frames based on their root means square (rms) value. The core of the module is the `onInput()` function that is called when there are frames to be processed on all the module's input ports. Within the `onInput` function, the `RMSClassifierM` retrieves a frame from the `in` port and, depending on the computed rms value, the frame is pushed on either the `above` or `below` port. More complicated components may maintain private state and schedule/handle events. CSense, also supports "pull" semantics: a component may request data from upstream components by calling `pull()` on any of its input ports. Pulls are implemented as polling requests and the upstream components may respond asynchronously by scheduling events. We will return to the details of event handling in the context of the concurrency model later in this section.

A simple configuration defines the ports, internal connections, and initialization parameters of the module it configures. An example of a simple configuration is shown in Figure 3.2. The public interface of a component is defined by its input and output ports (lines 10 – 12). The types of ports are specified according to the type system described in Section 3.2.4. A typical component execution involves retrieving frames from input ports, modifying these frames, and pushing them over output ports. The flow of frames *within* a component – from one input port to one or more output ports – is captured by its internal connections (line 14). A connection indicates the *potential* of a frame exchange rather than a requirement. Accordingly, a component that connects an input

port (I) to two output ports (O_1 and O_2) may, at run-time, output a frame on O_1, O_2, or on both O_1 and O_2. For example, the configuration `RMSClassifierC` connects the input port `in` to both `above` and `below` ports. At run-time, the module `RMSClassifierM` will output a frame on either `above` or `below` depending on the rms value of the frame. We prohibit components from creating new frames as precise information regarding the flow of frames is necessary to perform optimizations and error checking.

Figure 3.3 provides an example of a group configuration that specifies a speaker identification application. The group configuration allows for components to be instantiated, configured (lines 5 – 10), and linked (lines 11–17). While groups constitute "syntactic sugar", they facilitate code reuse. For example, `MFCCFeaturesG` is a group that includes components that implement several signal processing operations. Internally, the group automatically configures the filter bank required to compute the Mel-Frequency Cepstral Coefficients (MFCCs) based on the size of the feature type. These details are hidden from external components.

3.2.2 Memory Management

The overhead of memory operations, including object creation, copy, and garbage collection, can dwarf computation times as shown in Section 3.4. As a consequence, CSense adopts pass-by-reference semantics and incorporates memory management operations into the SFG. For memory management purposes, we distinguish three types of components: *sources*, *user components*, and *taps*. Sources are the only components that produce new frames. Frames are modified by user components and passed to a `Tap` when they are no longer used. As previously mentioned, user components are prohibited from creating or copying frames. These operations are supported by including `Copy` and `Ref` components in SFGs. We expect to raise the programmer's awareness of memory operations by incorporating explicit stream operators. More importantly, this approach

```
1:  public class SpeakerIdentifierG extends GroupConfiguration {
2:   public SpeakerIdentifierG(int rateInHz,
                               URL server, double rms) {

3:    // === type definitions ===
3:    VectorC speechT = TypeC.newFloatVector(1024);
4:    VectorC featureT = TypeC.newFloatVector(128);

5:    // === add components to group ===
6:    addComponent("audio", new AudioComponentC(rateInHz, 16));
7:    addComponent("rmsClassifier", new RMSClassifierC(rms));
8:    addComponent("mfcc", new MFCCFeaturesG(speechT, featureT));
9:    addComponent("toDisk", new ToDiskComponentC(featureT));
10:   addComponent("httpPost", new HttpPostC(server, "fileType"));
```

```
11:  // === connect components ===
12:  link("audio", "rmsClassifier");
13:  toTap("rmsClassifier::below");
14:  link("rmsClassifier::above", "mfcc::sin");
15:  fromMemory("mfcc::fin");
16:  toTap("mfcc::sout");
17:  link("mfcc::fout", "toDisk");
18:  toTap("toDisk");

19:  // === specify concurrency constraints ===
20:  getComponent("audio").setThreading(Threading.NEW_DOMAIN);
21:  getComponent("httpPost").setThreading(Threading.NEW_DOMAIN);
22:  getComponent("mfcc").setThreading(Threading.SAME_DOMAIN);
23: }
```

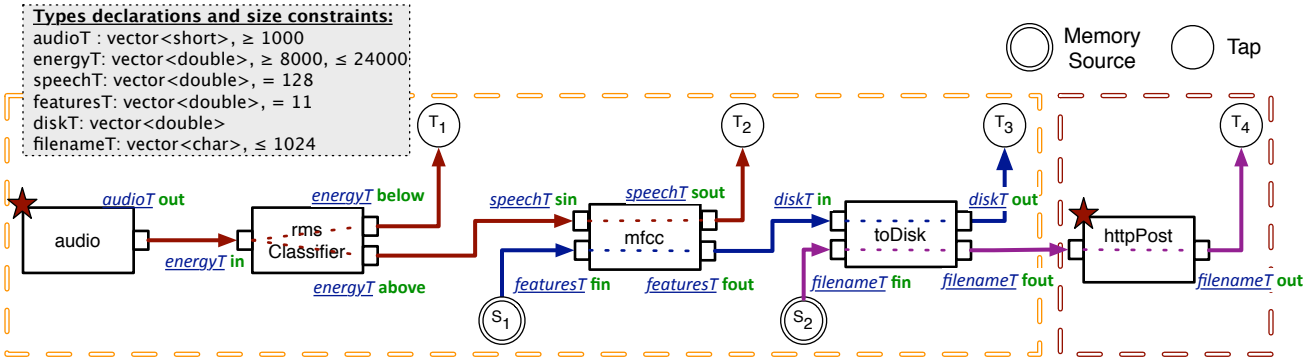


Figure 3.3. The main configuration of a speaker identification system adapted from [1]. `audio` records audio at a configurable frequency. `RMSClassifier` filters out silent frames. `mfcc` computes MFCCs which are saved to disk by `toDisk` and uploaded to a server by `httpPost` for identification. The types of ports are denoted by underlined text and their constraints are shown in the grayed box. `audio` and `httpPost` require to be executed in different domains shown in respective bounded boxes.

allows the entire flow of frames in an application to be known at compile time allowing for optimizations and static analysis.

3.2.3 Concurrency

Concurrency is prevalent in MSAs: sensors are sampled, data is uploaded to servers, and the user interacts with the system. This results in a mix of events and SP operations, which must be processed concurrently.

CSense provides four concurrency mechanisms: *domains*, *events*, *selectors*, and a global *workspace*. A *domain* includes a subgraph of components that are executed in the same thread. Components pertaining to the same domain exchange frames through function calls without re-

quiring synchronization. Data exchanges across domains are mediated by synchronization queues. Synchronization queues buffer frames to handle variations in the execution rate of different domains. A key advantage of the domain abstraction is its simplicity: the developer can reason about the behavior of components within a domain using sequential semantics.

Each domain has a *scheduler* that is responsible for managing events and selectors. Both mechanisms allow components to defer their execution to allow other components to run. CSense supports high concurrency by integrating with Java New I/O (NIO). A component may register NIO selectors with the scheduler. The scheduler calls the component when the selector has data available to read or write. This mechanism allows the scheduler to multiplex I/O requests. We restrict components to schedule events or register selectors for themselves, i.e., providing independent event streams per component. Moreover, to preserve the integrity of the domain abstraction, events and selector handlers are executed in the domain of the component that scheduled their execution. Components may share state through a global *workspace*. The workspace is organized as a dictionary in which shared variables are read and written through using agreed-upon keys.

CSense applications are multithreaded and may include shared state. As a consequence, there is a potential for race conditions to occur when variables are accessed from multiple domains. The race analysis ensures that individual accesses to shared variables occur in synchronized blocks. This invariant ensures that individual accesses to shared variables are race-free. While this approach avoids a majority of data races, there still is the potential for data races when implementing more complex synchronization protocols that involve multiple accesses to shared variables. Two factors make enforcing the above invariant in Java programs difficult: (1) determining the target of each update and (2) determining the complete set of potential execution inter-leavings. Our race analysis takes advantage of the restricted semantics of the CSense programming model to address

these challenges. First, the problem of identifying the target of an update is straightforward since the programming model requires shared variables to be accessed via the workspace. Second, unlike for general Java programs whose call graph is not fully known at compile time, the call graph in CSense is encoded by the SFG. The race analysis checks that all execution paths that access a shared variable are from within synchronized blocks. This is accomplished by accounting for the fact that the only entry points for a component are the `onInput` (called to exchange frames) and `onEvent` (called to handle events) method calls. Our analysis may also have false positives i.e., the compiler may issue a warning when a race does not exist. For instance, this may occur when a frame is accessed from two domains, but the two domains never execute concurrently.

The programmer can specify concurrency by defining constraints on components. First, the programmer may specify that a component should be executed in a new domain using a `NEW_DOMAIN` constraint. The constraint is associated with sources and components that include long/blocking operations. For example, the `NEW_DOMAIN` constraint may be added to the `audio` and `httpPost` to record and upload data concurrently (lines 20 – 22 in Figure 3.3). Second, the programmer may enforce that components are executed within the same domain using a `SAME_DOMAIN` constraint. For example, the components implementing the `mfcc` component should not be split across domains. Otherwise, a large number of frames would have to be exchanged via synchronization queues adding significant overhead.

The compiler uses a simple heuristic to partition the SFG into domains subject to the specified concurrency constraints. The algorithm operates on the SFG in which all groups are flattened except for those that include a `SAME_DOMAIN` constraint. The algorithm iterates through each source in the SFG assigning multiple components to a domain. Initially, the domain is set to zero and incremented in each iteration of the algorithm. Let c and d be the source and domain currently

under consideration. The algorithm assigns c to run in d. Additionally, it computes the predecessor subgraph of c that includes all components x such that there is a path from x to c. If no component in the predecessor subgraph requires a `NEW_DOMAIN`, all components of the subgraph will be executed in d. Otherwise, they will be assigned to a domain in a later iteration of the algorithm. Next, the algorithm computes the successor subgraph of c that includes all components x such that there is a path from c to x. Component x will be executed in domain d if no component on the path from c to x has a `NEW_DOMAIN` constraint. In a post-processing step, the groups with `SAME_DOMAIN` constraints are flattened and the members assigned to the group's domain. The proposed heuristic typically assigns subgraphs of components that share a path to the same domain, which reduces overhead.

3.2.4 Type System

Our type system is designed to provide the programmer with flexibility in specifying frame types. A frame can be either a vector or a multi-dimensional matrix of primitive Java types. While Java does not support matrices as types, CSense supports them to simplify the integration with MATLAB. The main extension to the type system is that we allow the programmer to specify simple constraints ($\leq$, $<$, $=$, $>$, and $\geq$) over the size of each dimension of an array. These constraints may be added cleanly as part of configurations (see lines 5 – 8 in Figure 3.2 for an example). Obviously, the size of a frame must be eventually determined. We define *type materialization* to be the procedure that determines the frame sizes subject to the defined constraints.

The support for parameterized and constrained frame types benefits error checking, component reuse, and optimization. Let us consider the `audio` and `mfcc` components. The `audio` component records sound in an underlying frame that is returned to the user when it is full. The Android OS enforces a minimum size for the recording buffer to reduce CPU utilization when

recording audio. In contrast, the input `mfcc` component outputs features that always contain 11 floats. CSense identifies configuration errors due to connections between ports of incompatible rates at compile time. In other SP models, such errors would go unnoticed until run-time.

Rich types also foster component reuse. For example, the `mfcc` component may be parameterized to use vectors whose element type is either float or double. This allows the developer to trade-off computational accuracy of MFCCs and computational overhead with a minor change to the frame type. Similarly, depending on its configuration, the `audio` component may record samples as bytes (8-bit samples) or shorts (16-bit samples). This allows `audio` to be configured based on the type of its output ports.

Defining components that operate over ranges of sample sizes is a powerful construct. For example, `audio` may use frames whose size exceeds 1024. Therefore, the same `audio` component may be used in two applications with different size configurations. This makes it feasible to select frame sizes such that components operate optimally from an application-wide perspective. CSense may accomplish this through the flow analysis presented next.

3.2.5 Flow Analysis

Type materialization requires that the compiler determine the size of frames subject to the constraints specified by developers. However, not all feasible solutions to this problem can be implemented efficiently. A source of inefficiency is *frame conversions* that can occur when ports, which require frames of different sizes, are connected. Consider the connection between `rmsClassifier` and `mfcc` in Figure 3.3. A feasible solution is for the `rmsClassifier` to output frames of 10,000 samples that `mfcc` must convert to frames of 128 samples. To handle this mismatch, the compiler must introduce a `Converter` that receives frames of 10,000 samples and outputs frames of 128 samples. Since 10,000 is not a multiple of 128, the `Converter` cannot

be implemented efficiently as it requires at least some samples to be copied. In contrast, if the `rmsClassifier` were to output a frame of 10,240 samples, then the samples can be divided into 80 vectors that contain 128 samples as required by `mfcc`. This may be implemented without copying by defining 80 non-overlapping views over the same memory buffer containing the 10,240 samples.

The goal of the flow analysis is to find a solution to the type materialization problem that may be implemented efficiently. The flow analysis is performed at compile time and, therefore, it does not introduce any run-time overhead. The analysis is performed on the application SFG (see Figure 3.3 for an example) by considering each path sequentially. A path in the SFG captures the flow of frames from a source to a tap following internal and external links. For example, the frames from `audio` follow two paths: `audio` $\rightsquigarrow$ $\mathtt{T}_1$ and `audio` $\rightsquigarrow$ $\mathtt{T}_2$. The behavior of a conversion is determined by three variables: super-frames (S), frames (F), and multipliers (M). A super-frame is a contiguous block of memory that can be divided into an integer number of frames. Components along all paths that originate at a source s use super-frames of the same size (S_s). Each port p of a component A may require different frame sizes $F_{A,p}$. This requirement is fulfilled by having A execute M_A times to ensure $S_s = F_{A,p} \cdot M_A$. When these constraints are satisfied, all conversions on the paths from s may be implemented efficiently.

The compiler casts the problem of determining the super-frames, frames, and multipliers as an Integer Linear Program (ILP). Integer linear constraints are generated based on the type constraints supplied by the programmer according to the pseudocode shown in Figure 3.4. For clarity, we consider the case of determining the appropriate conversions for a single source that has a super-frame of size S. The algorithm considers each port p of a component A on the path. Let $C_{A,p}$ be the set of type constraints associated with port p of component A. A constraint has

the form $(\bowtie, v)$, where $\bowtie$ is an operator ($\bowtie \in \{<, \leq, =, \geq, >\}$) and v is an integer. The algorithm iterates through each type constraint, adding new constraints to the ILP problem. If $\bowtie \in \{=\}$, then the size of the frame ($F_{A,p}$) is set to equal v (as specified by the type constraint) and we ensure that the super-frame (S) is a multiple of $F_{A,p}$ (lines 6 – 9). The multiplier M_A is optimized based the constraints of entire path. If the user does not supply a "=" constraint ($\bowtie \notin \{=\}$) (lines 13 – 14), then we set $M_A = 1$ indicating that the component can process the entire super-frame in a single call. In this case, the value of the frame $F_{A,p}$ will be optimized based on the constraints of the entire path. If $\bowtie \in \{\leq, <, >, \geq\}$ (lines 10 – 11), then the size of the frame ($F_{A,p}$) is constrained by v. Additionally, the frames sizes $F_{A,p}$ are constrained to be smaller or equal to then super-frame sizes S (line 3) and multipliers (M_A) to be at least 1 (line 4).

Solving the created ILP will determine the value of super-frames, frames, and multipliers subject to the type constraints specified by the programmer and those required to perform efficient frame conversions. A typical MSA has an associated ILP with multiple feasible solutions. Choosing an appropriate solution involves a trade-off between memory utilization and run-time overhead. CSense currently uses the solution that has the least memory utilization. This is because Android imposes strict limits on the memory utilization of an application, which limits an effective evaluation of trade-offs in selecting different solutions.

The ILP does not have a feasible solution in two cases: there is no solution to type materialization and there is no efficient implementation. In the former case, the compiler generates an error; in the latter case the compiler generates a warning indicating that inefficient conversions are used and then reruns the ILP without the efficient frame conversion constraints. In practice, the developers select frame sizes to be multiples of each other, in which case, a feasible solution to the ILP problem exists.

```
1:for (A, p) a component-port pair path
2:    for (⋈, v) ∈ C_{A,p}:
3:        ILP: 0 < F_{A,p} ≤ S
4:        ILP: M_A ≥ 1
5:        hasEquals = False
6:        if ⋈ ∈ {=}:
7:             hasEquals = True
8:             ILP:  F_{A,p} = v
9:             ILP:  S = F_{A,p} × M_A
10:       elif ⋈ ∈ {≤, <, >, ≥}:
11:            ILP:  F_{A,p} ⋈ v
12:   if hasEquals = False:
13:        ILP:  M_A = 1
14:        ILP:  S = F_{A,p} × M_A
```

Figure 3.4. Frame analysis algorithm and notation.

3.2.6 Compiler

The compiler has the following workflow. The main configuration instantiates components, and configures, and connect them. After flattening groups, the compiler checks that the SFG is structurally correct: no ports are unconnected and no fan-ins, fan-outs, or cycles exist. Additionally, we ensure that the all paths start with a source and end with a tap. The compiler runs the flow analysis to materialize types and includes `Converters`, as appropriate. The SFG is partitioned into domains and then checked for race conditions. The final step is to generate code for the target platform.

We use the MATLAB compiler to generate C code for components that use MATLAB functions. The C code is compiled as a static library. The general strategy for including a MATLAB function as a CSense component is to create a mapping between input/output ports of the component and the input arguments/return values of the MATLAB function. The compiler generates custom wrapper classes that call the generated static library. Data is exchanged using NIO buffers for efficiency. The compiler also generates a "main" application that configures components, connects them, and creates threads for their execution. The code generation completes by compiling the generated code. Even though in this chapterwe focus on Android, owing to Java's portability,

we have been able to run CSense applications on both Linux and OS X.

3.3 Runtime Environment

3.3.1 Scheduler

An application is partitioned into domains, each domain having its own scheduler. The scheduler is responsible for managing memory, events, and selectors.

The goal of memory management is to minimize the impact of object creation, copying, and garbage collection. We implement memory management as follows. Each source maintains a memory pool that contains a number of super-frames. A source retrieves a super-frame from the pool when it has data to write. Flow analysis (performed at compile time) ensures that frames are exchanged efficiently until they reach a tap. Upon reaching the tap, the scheduler must determine if it should put the super-frame back in the memory pool. We associate a reference counter with each super-frame. The reference counter is incremented each time a new reference is created. Conversely, the counter is decremented when taps are reached and, when the counter becomes zero, the super-frame is put back in the pool for reuse. During the initialization of an application, a configurable number of frames (currently set to 8) are preallocated in each memory pool. Additional frames may be allocated at runtime when new frames are request but none are available in the pool. These mechanisms limit the creation of new frames and their garbage collection.

A component may schedule events to run after a delay. The scheduler maintains two execution queues. The immediate execution queue is a FIFO queue that stores zero delay events. Components use zero delay events to yield their turn and allow other components to be executed. Non-zero delay events are inserted in a priority queue sorted by time when they are scheduled to fire. The scheduler operates in rounds. In each round, the scheduler drains the immediate queue and processes all the events in the priority queue scheduled to execute no later than the current

time. A component may also register selectors with the scheduler. Selectors are checked at the end of a round and components that have pending data are notified.

Memory pools and events may be accessed from different threads, so a concurrency mechanism is necessary. Java includes support for concurrent collections including blocking queues and synchronized arrays that may be used to implement the event queues of schedulers and memory pools of sources. However, the underlying implementation of these data structures uses reentrant locks. Locks are designed to handle high levels of contention. Under low or medium contention, locks introduce a high overhead since a thread must be suspended when it attempts to acquire a lock that is already held by a different thread. Atomic variables provide a lightweight synchronization mechanism that is implemented efficiently using hardware-supported compare-and-swap. The challenge with atomic variables is that the developer has to implement appropriate mechanism to handle concurrent access. To improve SP rates, we have implemented customized synchronization primitives. Our synchronization primitives use a two-level locking scheme. Atomic variables are used for concurrency in the low contention case. If the lock implemented using atomic variables is not acquired after several attempts, we switch to using reentrant locks.

3.3.2 Android Integration

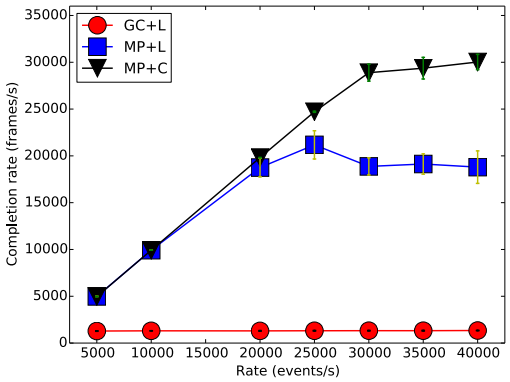
CSense is designed to take advantage of the underlying Android services. Consistent with the Android architecture, a CSense application uses activities for user interfaces and a service to host its runtime environment. The user interface and service run in the same process, but in different threads. CSense components have specialized implementations for Android. For example, components that use sensors leverage on the Android APIs to capture motion, GPS, and audio data.

CSense integrates with Android's power management to allow phones to sleep. Android uses power locks to prevent the CPU and display from entering a sleep state. When no power locks

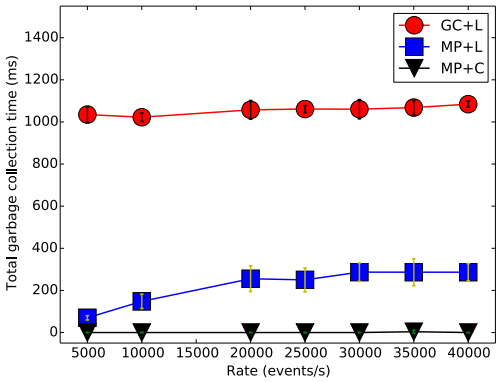
are acquired, Android will aggressively turn them off. Releasing power locks prematurely may result in an application being suspended for an indeterminate amount of time. Other resources, such as network or GPS, are not managed though power locks. Instead, the programmer must explicitly turn them on and off. These resources are typically accessed from a single CSense component that is also responsible for managing their power consumption.

There are two challenges to integrating our scheduler and Android's power management: (1) we must determine when it is safe to sleep, and (2) we must develop an efficient mechanism to enter and leave sleep states. To determine if it is safe to sleep, the scheduler consults the pending events and registered I/O handlers. Each scheduler maintains an independent power lock that is acquired during its initialization. In the following, we describe the behavior of each scheduler independently. The CPU will sleep only when *all* schedulers release their power locks. Let t_{now} be the current system time and t_{first} be the time when the next event in either one of the scheduler's queues is scheduled to run. If $t_{first} < t_{now}$, then the scheduler is running behind, effectively having to catch up with the sequence of events. Thus, the power lock cannot be released to allow the scheduler to catch up. Otherwise, if $t_{first} \geq t_{now}$, the scheduler can sleep for $d = t_{first} - t_{now}$ seconds. In this case, the scheduler registers an alarm to wake up the system after d seconds. Android guarantees that alarms wake up the system from sleep, at which point, the scheduler reacquires the power lock. In the case when no events are scheduled, the scheduler will go to sleep and may be woken by receiving data from other domains or by external events.

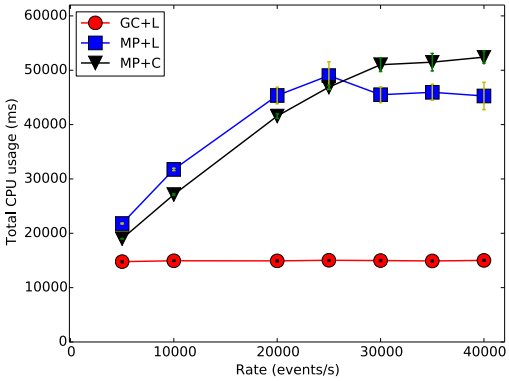
Initial testing indicated that the above algorithm has an important limitation: it does not account for the time necessary to transition to sleep and then to wakeup. Let t_{wakeup} be the time from the time when the power lock is released until the wakeup alarm is delivered. If the time the scheduler may sleep $d < t_{wakeup}$, then some events will be delivered late. This is particularly



(a) Scheduler implementations.



(b) Total garbage collection time.



(c) CPU usage.

Figure 3.5. Producer-consumer benchmark to assess memory pooling and concurrency techniques.

problematic when there are numerous events to be processed due to highly concurrent workloads. To address this limitation, we devised a two-level sleep strategy that only release the power lock when $d > t_{th}$, where t_{th} is user-specified constant. If $d < t_{th}$, then the scheduler will use Java's wait/notify mechanism to sleep for d seconds without releasing the power locks. Otherwise, we release the power locks and allow the CPU to sleep. This algorithm is safe in that it does not introduce additional delay penalties of pending events due to sleep.

3.4 Evaluation

In this section, we provide an empirical evaluation of CSense on Galaxy Nexus phones running Android Jelly Bean. Galaxy Nexus uses a Texas Instruments OMAP 4460 SoC that includes a 1.2 GHz dual-core ARM Contex-A9. The phone has 1GB of memory and 32 GB of storage. C code is generated from MATLAB functions using MATLAB R2012b and MATLAB Coder 2.3. The resulting code is cross-compiled into a static library using Android NDK (r8d). We evaluate CSense using both micro- and macro-benchmarks.

3.4.1 Micro-benchmarks

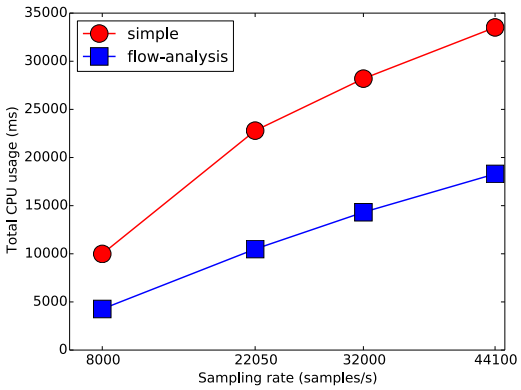
Scheduler Scalability: We evaluated the scalability of the scheduler using a Producer-Consumer benchmark. The producer generates frames at specified rates. The produced frames are passed to the consumer and then to a tap. The producer and consumer operate in different domains to capture the impact of inter-domain connections. Memory was managed using either Java's memory management (`GC`) or using memory pools (`MP`). In the former case, new objects are created for each frame and garbage collection is used to free them. Flow analysis is not used in this benchmark. Concurrency in the scheduler was implemented using locking primitives (`L`) and CSense's synchronization primitives (`C`). A scheduler implementation combines a memory management and a locking mechanism. The results are averages over five runs; each run lasting for a minute. 95%-confidence intervals are also plotted.

Figure 3.5a shows the performance of the three schedulers. We increase the offered rate linearly and measure the rate at which the consumer receives frames. A scheduler should match the offered rate until it reaches its peak rate. To understand the differences in performance, we also measure the total garbage collection time and CPU usage. The CPU usage is measured as the total

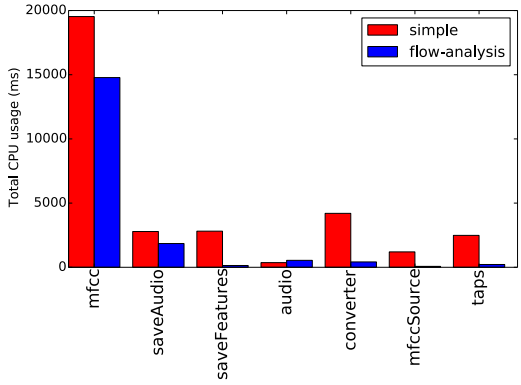
time the benchmark runs on either CPU core.

A naive implementation of the scheduler would use Java's memory management and locking concurrency primitives (`GC+L`). `GC+L` performs poorly; it supports a peak rate of only 1,534 events/s. `MP+L` incorporates memory pools and relies on locking concurrency primitives. Memory pools eliminate the creation of frames and reduce garbage collection. As a result, the peak event rate is increased to 21,176 events/s – a 13.8 times increase. Figure 3.5b plots the garbage collection time for each implementation as reported by Dalvik. As expected, the naive implementation has the highest garbage collection time. `MP+L` reduces garbage collection significantly, but does not eliminate it. In fact, the garbage collection time increases slowly with the offered rate. This increase may be attributed to `ReentrantLock` objects being created in Java concurrent collections. These objects are created when a thread attempts to access a lock that is already held by a different thread.

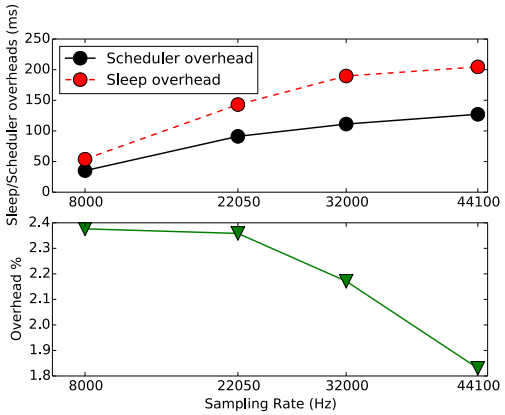
Using our concurrency primitives, the scheduler (`MP+C`) may support a peak rate of 30,029 events/s. This represents an additional 30% improvement over `MP+L`. Overall, the proposed optimizations provide a 19 times improvement over the naive implementation. Two factors contribute to these improvements. The garbage collection time is reduced to zero when our customized synchronization primitives are used. Additionally, as shown in Figure 3.5c, `MP+C` runs for a longer time as indicated by the higher CPU time. This is because our synchronization primitives reduce the number of thread suspensions and resumptions. This allows for multiple frames to be inserted by the consumer or removed by the producer from the synchronization queue without requiring context switches.



(a) Benefits of flow analysis.



(b) Detailed performance at 44100 Hz.



(c) Scheduler overhead.

Figure 3.6. MFCC benchmark to assess the benefits of the flow analysis.

3.4.2 Macro-benchmarks

We have implemented three applications to showcase the versatility of CSense: SpeakerIdentifier, ActiSense, and AudioSense. CSense facilitated incorporating MATLAB code in applications and its integration with additional Java components for data collection, file I/O, networking operations, and UI. Each application highlights an aspect of the toolkit: the benefits of flow analysis, the scheduler scalability, and its overhead. Statistics regarding the size of the ILP problem for each application and compilation times are also provided.

SpeakerIdentifier: SpeakerIdentifier (see Figure 3.3) determines the identity of speak-

Application	Number of constraints
SpeakerIdentifier	53
AudioSense	164
ActiSense (phone only)	172
ActiSense (phone + 3 Shimmer motes)	564

Figure 3.7. Number of ILP constraints for each application.

ers based on their voice fingerprint. The application is based on SpeakerSense [1]; however, in contrast to SpeakerSense, SpeakerIdentifier performs speaker identification remotely. The `mfcc` component is implemented in MATLAB. To evaluate the effectiveness of flow analysis on a realistic application, we will compile and profile the application with (`flow-analysis`) and without (`simple`) flow analysis.

Figure 3.6a plots the CPU usage when the audio sampling rate was 8000, 22050, 32000, and 44100 Hz. The figure clearly indicates the benefits of using the efficient frame conversions enabled by flow analysis. Moreover, these benefits increase with the audio sampling rate. At 44100 Hz, using flow analysis, the CPU usage is reduced by 45% compared to the baseline. To better understand the benefits of framing, Figure 3.6b plots the time spent in each component of the MFCC pipeline. Aside from minimizing the number of object copies, the use of super-frames has three additional advantages: (1) It reduces overhead since super-frames contain more samples than frames but require the same number of function calls to push. This results in lower overhead on `mfccSource` and `tap` components that are responsible for memory management. (2) Super-frames allow components to execute at different rates. The super-frame is 4096 samples, but the `mfcc` source and `saveFeature` are executed 32 and 1 time, respectively, to process a super-frame. This feature explains the reductions in the CPU time of `mfcc`, `saveFeature`, and `saveAudio`. (3) Finally, the `Converter` component is used to convert shorts to doubles. For

efficiency, this component is implemented in native code. A benefit of using CSense is that the compiler can automate such transformations thus reducing the development burden.

We instrumented the scheduler to measure the time each domain thread spends executing the user code and the total time the thread was executed. The difference between the total and user time is considered overhead. We divide the overhead into sleep overhead and scheduler overhead. The sleep overhead measure the time to access the underlying power locks associated with each domain. Figure 3.6c plots detailed scheduler overhead when the flow analysis is used. The overhead percentage is computed relative to the total CPU usage reported in Figure 3.6a). The overhead ranges from 2.37% to 1.83%, decreasing slightly as the sampling rate increases. The slight decrease is the result of the CPU usage increasing faster than the scheduler overhead. These results show that the scheduler introduces small overhead.

Figure 3.7 includes the number of ILP constraints that were generated by the flow analysis for each application. All applications include less than a thousand linear constraints. ILP solvers can solve problems of this size very efficiently. On a laptop with a 2.6 GHz Intel Core i7 with 16 GB of RAM all ILPs were solved in less than 10 ms.

ActiSense: ActiSense is an activity recognition application that uses accelerometers to recognize running, sitting, walking, standing, and climbing stairs. The system is based on [22]. ActiSense includes a mobile phone and three Shimmer motes. Data from the Shimmer motes is streamed to the phone over Bluetooth. Feature extraction is performed on the phone. The extracted features are mean, time-domain and frequency-domain entropy, and correlation features as described in [22]. A Support Vector Machine (SVM) classifier determines user activities in real-time.

We have conducted a small user study involving 3 volunteers to evaluate the accuracy of

the system. While instrumented, the volunteers performed the target activities as part of a circuit of activities. The collected data was annotated with the start and end times of each activity. The annotated data set was divided into training and testing data sets. We have evaluated four classifiers (Naive Bayes, ensembles of Naive Bayes classifiers, SVM, and ensembles of SVM classifiers) on the collected data set. The classification accuracy computed using 10-fold cross-validation is shown in Figure 3.8.

The components of ActiSense are partitioned into six domains. Four domains are responsible for collecting acceleration readings, saving them to flash, and making predictions using an SVM classifier. Three of the four domains are allocated for processing acceleration data from motes (one per mote). An additional domain is allocated to process acceleration data from the phone. The remaining two domains are responsible for recording data from the gyroscope and magnetometer sensors to disk.

Figure 3.9 plots the user time and associated overhead for each domain. The domains processing data from the Shimmer motes spent a similar amount of thread time – about 400 ms. The bulk of the time is spent on feature extraction (275 ms) and classification (68 ms). In contrast, the time to process the data collected from the phone's accelerometer is twice as long. The increase is reflected in a longer feature extraction (530 ms) and prediction (95 ms) times. An explanation for this difference is that the sensors use different sampling rates: the Shimmer motes are sampled at 50 Hz while the phones are sampled at 60 Hz. The overhead across all domains is about 13%. Half the overhead can be attributed to operations on Android's power locks. The higher overhead of ActiSense (compared to that of SpeakerIdentifier) is due to smaller super-frames. ActiSense was configured to produce activity predictions every second, which prevented the creation of large superframes. Relaxing this constraint will reduce overhead, as the flow analysis will use larger

Classifier	Configuration	Accuracy
Naive Bayes	Phone + Shimmer	69.54%
Naive Bayes Ensemble	Phone + Shimmer	76.55 %
SVM	Shimmer	88.43%
SVM	Phone	94.64%
SVM Ensemble	Shimmer + Phone	96%

Figure 3.8. ActiSense: Accuracy for different configuration and learning algorithms.

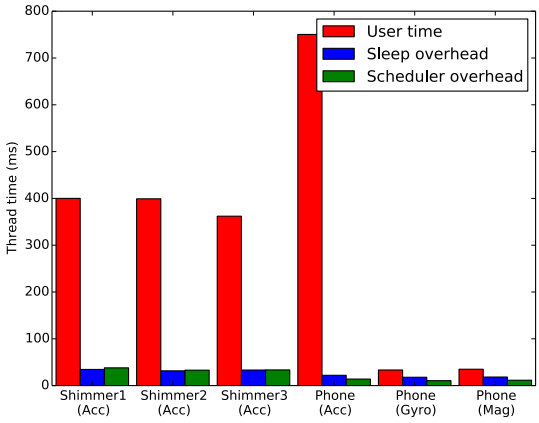
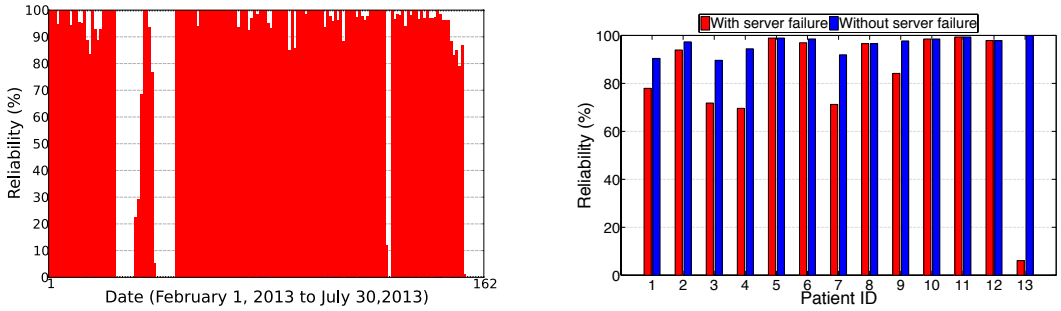


Figure 3.9. ActiSense: Execution time and overheads.

super-frames.

AudioSense: AudioSense [6] evaluates the performance of hearing aids using electronic surveys. Surveys may be user initiated or triggered at random intervals on average every 1.5 hours. Concurrent with the delivery of surveys, AudioSense collects audio samples and GPS locations to provide a context for the surveys. AudioSense caches data on the mobile phone and uploads it when it may establish a 3G connection to our remote server. AudioSense has been deployed for six months as part of a clinical study. The challenge of developing such a system is to ensure reliability during weeklong deployments.

Figure 3.10 plots the reliability for each day of the trial. The reliability is computed as



(a) Reliability per day.

(b) Reliability per patient.

Figure 3.10. AudioSense: Reliability during a 6-month trial.

the fraction of data uploaded out of the data that was collected. As shown in the graph, there were three instances when the reliability was zero during the trial. The cause of these outages was the server being offline for several days due to power outages. The remainder of the outages may be attributed to coverage gaps in the study area. Figure 3.10b shows the reliability for the 13 patients in the study included in the first six months of the study. Excluding the server outages, the reliability of AudioSense exceeds 90%. This shows that our toolkit is sufficiently mature to support long-term deployments.

3.5 Related Work

CSense uses a stream processing (SP) model to support the development of high-rate and robust MSAs. This section places CSense in the context of prior work on SP systems and static analysis of Java programs.

Stream Programming: SP models have been studied for decades (see [33] for a review). SP systems can be broadly divided into synchronous and asynchronous systems. Synchronous systems operate on a shared clock (or clocks) that dictates when components are executed. The rigid timing of synchronous systems is suitable for compiler optimizations. Compilers can determine

execution rates, buffering requirements, and implement efficient scheduling [16, 33, 34]. Asynchronous systems provide a more flexible concurrency model but sacrifice performance, as many of the optimizations developed for synchronous systems do not translate these systems. CSense adopts an asynchronous model to support workloads that include both concurrent operations and asynchronous events.

The problem of efficiently supporting asynchronous SP has been previously considered in systems such as Click [24], XStream [35], and WaveScript [34]. Click executes components in a single thread but avoids creating inflexible fixed schedules. Click maintains a task queue to which sources and queue components are added when they have data to process. A scheduler determines the execution order of the tasks in the queue. The execution of the other type of components is triggered by function calls that traverse the component graph in a depth-first manner. A similar approach to component scheduling is used in XStream and WaveScript. As described in Section 3.2.3, CSense extends this mechanism by including support for multiple execution domains and event handling. CSense further reduces overhead through a flow analysis that allows components to be executed multiple times without involving the scheduler.

Memory management can have a significant performance impact on SP. XStream and WaveScript use an abstract data structure called SigSegs to efficiently exchange frames between components. SigSegs have some similarity to our flow analysis that optimizes the memory allocation of frames. However, in contrast to SigSegs that operate solely at run-time, we optimize memory management by leveraging type information and explicit knowledge of frame flows at compile time. This optimization is feasible in CSense due to the additional information supplied by developers.

Static analysis: There have been several efforts to detect concurrency problems in Java

programs. For example, ESC/Java2 [36] and Checker Framework [37] are static analysis tools that identify potential bugs such as data races. Unfortunately, these tools are neither sound nor complete, they cannot fully address the problems of aliasing and limited visibility into the Java/Android run-time environment. More promising results are obtained in domain specific languages. For example, NesC [38] limits programmers to using only static memory and a restricted concurrency model to facilitate static analysis. CSense adopts a similar strategy by explicitly capturing memory operations as part of component graphs and limiting its concurrency model.

3.6 Conclusions

In this chapter, we presented CSense – a stream programming toolkit for developing high rate and robust mobile sensing applications on Android. CSense provides developers a programming model, a compiler, and a run-time environment. The programming model extends existing SP models by incorporating a flexible concurrency model, a new type systems that fosters component reuse, and incorporates memory operations as part of the SFG. We leverage this additional information for both the compilation and static analysis. Our compiler incorporates a novel flow analysis that optimizes frames exchange across components from an application-wide perspective. Empirical results indicate that the flow analysis may reduce CPU utilization as much as 45%. Moreover, static analysis techniques can prevent a range of programming errors including the incorrect usage of the memory management system and data races. These techniques enabled us to deliver a mobile sensing application that uploads data to a server with over 90% reliability. We have identified that the memory management and concurrency limit the scalability of SP on Android. We incorporate memory pools, frame conversion optimizations, custom synchronization primitives, and careful integration with power locks to develop a scalable run-time environment. Micro-benchmarks indicate that these optimizations increase the peak stream rate by as much as

19 times over a baseline implementation that uses Java's concurrency management.

CHAPTER 4

STATIC MEMORY MANAGEMENT FOR MOBILE SENSING APPLICATIONS

4.1 Introduction

The advancing capabilities of smartphones have enabled the development of a new generation of mobile sensing applications such as those for context monitoring [39], user identification [40], personal health [41], and environmental monitoring [42]. At the heart of these applications, there are sophisticated stream engines that must process high-rate sensor data efficiently.

Memory management is a key challenge in the development of stream engines. Previous studies have shown that poor memory management leads to applications with large memory footprints and excessive memory accesses that reduce stream processing rates [26, 35, 43]. A common source of inefficiency is the fact that stream operations such as windowing, splitting, appending, and downsampling have been traditionally implemented using memory copying. Avoiding copying requires the memory management to support data sharing among the components of an application. Data sharing typically reduces both memory usage and the number of memory accesses. However, in order to improve the stream processing rate, we must ensure that data sharing does not reduce cache locality or increase code complexity.

The problem of effective memory management for stream processing may be addressed through dynamic or static memory management. Dynamic memory management relies on specialized data structures for manipulating streams. A representative example is the SigSeg [34, 35]. A SigSeg is organized as a list of buffers containing data samples; each buffer may be shared between components using reference counters. While the SigSeg is a clever data structure, dynamic memory management suffers from two intrinsic limitations. (1) Dynamic memory management

can exploit only a fraction of data reuse opportunities as some run-time overhead may be introduced for analyzing an application's behavior. (2) The data structure unavoidably adds a level of indirection in accessing streams of samples, which reduces the performance of stream engines.

Compile-time solutions may be used to determine memory allocations without introducing run-time overhead. The key is to develop a static analysis technique that may precisely identify *location* and *temporal sharing* opportunities. Unfortunately, static analysis of general purpose languages such as C or Java quickly becomes imprecise because of complex control structures and pointer aliasing. We avoid these difficulties by focusing on a domain-specific stream processing language called StreamIt [16]. We show that it is feasible to leverage the constrained semantics of stream programs to implement stream operations efficiently through static memory allocation. Since StreamIt is a representative example of a synchronous data-flow (SDF) language, we expect that the results presented in this chapterwill translate into other systems and languages based on SDF models.

We propose Efficient Static Memory management for Streaming (ESMS) that addresses the above challenges. In this chapter, we make the following contributions: (1) We develop a novel *static analysis* that characterizes the global memory behavior of a complete stream application. The static analysis can precisely identify the location and temporal reuse opportunities in most applications. The analysis is imprecise for a fraction of components whose control logic depends on the input data. In these cases, we provide a conservative but safe approximation of the application's memory behavior. (2) We propose a novel *layout algorithm* that leverages the identified location and temporal reuse opportunities along with the application's structure to optimize the memory layout. We incorporate code generation techniques that transform a stream program into efficient C code that effectively uses the generated memory layouts. (3) The memory optimizations are

implemented as a new compiler for the StreamIt language. We evaluate the memory optimization on both Intel and ARM platforms using 14 benchmarks including three realistic mobile sensing applications. We compare our compiler against the standard StreamIt compiler with and without the cache optimization enabled [44]. We find that our optimizations reduce the memory footprint up to 96% while matching or improving the performance of the StreamIt compiler with cache optimizations enabled.

4.2 StreamIt Overview

Language Overview. Our work builds on the StreamIt programming language and compiler infrastructure (see [16] for a detailed description). The basic computation unit of StreamIt is a `filter` that may interact with other components by consuming data from the input channel, performing computations that may affect the state of the `filter`, and producing data on the output channel. StreamIt defines three basic memory operations: `pop`, `peek`, and `push`. The `peek` reads a sample at a given index in the input channel without consuming it, `pop` consumes a sample from the input, and `push` appends an item to the output channel. The `pop` and `push` access a channel sequentially while `peek` provides (limited) random access. Consistent with the SDF model, the number of items `peek`-ed, `pop`-ed, or `push`-ed during an execution is fixed and known at compile time. A filter has a `work` function that is executed each time the component is invoked. The `work` function specifies the rates r_{peek}, r_{pop}, and r_{push} for the `peek`, `pop`, and `push` instructions. A component may include states that are initialized using an `init` function.

StreamIt programs are written by hierarchically composing `filters` using `pipeline`, `split-join`, and `feedback` constructs into a Stream Flow Graph (SFG). The `pipeline` construct composes `filters` in sequence by connecting their inputs and outputs. The `split-join` construct distributes a stream to a parallel set of streams that are joined later. A `split` has a sin-

gle input channel but multiple output channels. The `split` may either *duplicate* its input so that each parallel stream works on the same data or distribute it in a *round-robin* fashion. The `join` performs the opposite operation by taking data from multiple input channels and merging them into a single output channel in a *round-robin* fashion. The programmer may specify the number of elements to be produced by splitters and consumed by joiners during each invocation by providing a set of weights. The `feedback` construct is used for specifying feedback loops.

Execution Model. The problem of scheduling SDF components has been widely explored [18,45, 46]. It is well understood that scheduling can have a significant impact on the program performance and memory utilization. A key advantage of SDF models is that they may be executed according to *cyclo-static* schedules. A cyclo-static schedule includes an initialization phase that is executed once and a steady phase that is executed repeatedly forever. The scheduling ensures that a `filter` is executed only if there is enough data on its input channel. In this chapter, we assume a fixed schedule and do not consider the interaction between scheduling and memory optimization, which we will investigate in future work. We adopt the single appearance schedules (SASs) in [46], where a filter's work function appears only once in the schedule to reduce the code size.

The StreamIt language is tailored for stream programming. StreamIt follows SDF and adopts copy-by-value semantics and does not support pointers. These restrictions simplify the static analysis of stream programs and facilitate reasoning about their memory behavior statically.

4.3 Design

ESMS builds on the properties of stream programs to optimize memory management. A key property of SDF systems is that their components may be executed using a cyclo-static schedule. Accordingly, the complete memory behavior of a program can be observed during an execution of the initialization phase followed by a *single* execution of the steady phase. This property

is essential for providing a complete description of the memory behavior of the program. Furthermore, we propose a novel static analysis that leverages the semantics of stream programs to identify location and temporal sharing opportunities.

The layout algorithm uses the location and temporal sharing opportunities to allocate memory efficiently. The layout algorithm is based on two empirical insights regarding the memory operations of stream programs: (1) data sharing is often captured explicitly and may be exploited to reduce the memory footprint and number of memory accesses and (2) owing to the data flow structure, a filter may typically reuse the memory freed by its predecessor in the SFG. These insights coupled with three heuristics for handling memory conflicts form the basis of our layout algorithm. Code generation techniques are then employed to efficiently implement the derived layouts.

The subsequent sections detail the static analysis and layout algorithm. Empirical evidence regarding the effectiveness of our techniques is included in Section 4.4.

4.3.1 Static Analysis

The goal of our static analysis is to provide a sound approximation of the memory operations of a stream program under all possible executions. The memory behavior of the program is summarized as a hierarchical *Memory Graph* (MG)[1]. The top level of the graph is represented by components. Each component has one or more ordered input and output elements that form a *fragment*. Each *element* represents a sample that may be consumed or produced during the component execution. The grouped component inputs and outputs form the middle hierarchy while the elements constitute the bottom.

We distinguish two types of data reuse opportunities: *location sharing* and *temporal shar-*

[1] An example memory graph is shown in Figure 4.4.

ing. Location sharing is captured by adding edges between elements to indicate when samples are passed without modification either within a component or between components. A property of MG is that the elements on any path can be stored at the same memory location, exposing location sharing. Temporal sharing is modeled separately by associating a live range L with each element that captures the time interval when an element is "live" given the fixed schedule. Obviously, elements that have non-overlapping live ranges may be stored at the same memory location.

The construction of MG proceeds in two steps:

- *Component Analysis:* Abstract interpretation (AI) is used to analyze the code of each component. The analysis constructs a fragment that captures the location and temporal sharing of a single component during an invocation of its `work` function.

- *Whole-program Simulation:* The schedule is simulated to stitch the previously constructed fragments and compose a MG that characterizes the entire application. The stitching process involves scaling component fragments to account for multiple invocations during the schedule, adjusting the live ranges of elements, and adding edges to capture data sharing between components.

The separation of the static analysis into two parts is motivated by the need to minimize the compile time. AI is significantly more expensive than the stitching process. Limiting the number of invocations of AI to one per filter allows us to handle multiple invocations of components.

4.3.1.1 Component Analysis

In this subsection, we present a static analysis that identifies location and temporal sharing during the component execution. Our analysis builds on AI initially developed by Cousot et al. [47]. The abstract domain of program variables is the intervals which approximate their concrete

values. The functions α and γ map concrete values to intervals and vice-versa. The symbols $\perp$ and $\top$ represent the bottom (i.e., the empty set) and top (i.e., $[-\infty, +\infty]$ interval) of the real interval lattice. Aside from arithmetic operations, interval approximations may be defined over a broad range of functions [48] including those supported by StreamIt. Additionally, we use operators $\sqcup$ and $\sqcap$ to represent union and intersection of intervals respectively.

The `work` function of the component is represented as a control flow graph (CFG). The CFG has distinguishable entry and exit nodes, junction nodes with exactly two predecessors, branch nodes with a true successor and an optional false successor, and block nodes with one successor and predecessor. The junction nodes may be either simple or loop junctions. A block may contain multiple instructions, but we constrain each block to include a single `peek`, `pop`, or `push`. We will use I_i and O_k to denote the i^{th} input and k^{th} output elements of the considered component.

The state of a component may include local variables, global variables, and constant parameters known at compile time. The value of a global variable is maintained across component invocations. Since the derived results must hold for any component execution, states and the values of input elements are initialized to $\top$. In contrast, local variables are set to $\perp$ prior to their first assignment. Before analyzing the `work` function, constants including the component parameters are propagated.

The pseudocode for the analysis is included in figure 4.1. The line numbers included in this section refer to this algorithm. The analysis extends the basic worklist algorithm, which updates a mapping from CFG edges to data flow facts until no new facts are derived. The notation $IN_v[n]$ and $OUT_v[n]$ refers to the data flow facts available immediately before and after a node n regarding variable v. Variables are interpreted over abstract intervals similar to the approach in [49].

However, our approach differs in two important aspects. First, to reduce pessimism, loop junctions are handled by unrolling loops rather than applying interval widening. We ensure termination by imposing an upper bound on the number of unrollings and applying widening when this bound is exceeded. The upper bound is set to the number of iterations if loop bounds are available or a large constant otherwise. Second, we handle function calls in a context-sensitive manner. The remainder of the discussion focuses on the unique aspects necessary for analyzing memory operations.

Temporal sharing. The analysis creates a fragment that includes r_{peek} input and r_{push} output elements. We determine the live range of each element to identify temporal sharing opportunities. At a high level, this requires determining when and which elements are referenced by a `peek`, `pop`, and `push`.

To keep track of the order of memory accesses, we add memory counters (mc) to the propagated data flow facts. The mc provides a time frame that captures the time when a memory operation is performed relative to the beginning of the component's execution. The mc of the entry node is initialized to zero. A memory operation increments the mc of the previous block (lines 11, 17, and 22). MCs are combined using the maximum at junctions (line 36).

To determine which elements are referenced by the memory instructions we define two sets – pop and $push$ – that include the elements referenced by `pop`, `peek`, and `push`. The domain of the two counters is the concrete intervals. The pop and $push$ are initialized to zero at the entry node and incremented after each block node that includes a `pop` (line 12) and `push` (line 23), respectively. The values of the pop and $push$ counters are merged using *interval union* (lines 37 – 38) to track all possible referenced elements over all execution paths.

The derived live range facts of an element e are summarized in an interval $L[e]$. $L[e]$ is a global variable maintained during the analysis. We initialize $L[I_i] = [0, 0]$ since an input element

I_i is live in the beginning of the component execution. In contrast, we set $L[O_k] = \emptyset$, since an output element O_k becomes live when it is first referenced by a `push`. For `pop`, `peek`, and `push`, the set $elem$ contains the elements that are accessed by each instruction (lines 13, 18, and 24). The concretization function γ is used in the computation of $elem$ to map an abstract interval representing these access indexes (i.e., IN_{pop} and OUT_{push}) to concrete values. The live range $L[e]$ grows monotonically by extending its interval to the current mc (lines 14, 19, and 25).

Location sharing. The analysis can also identify location sharing opportunities. Location sharing happens when a component reads an element I_i and passes it unmodified as an output element O_k. Two constraints must be satisfied for location sharing: ($L1$) the element I_i must be passed as O_k in all executions, ($L2$) no other element I_j ($i \neq j$) is passed to O_k in any execution. Next, we will discuss how to determine whether these conditions hold.

Determining location sharing opportunities requires tracking how elements are passed from the input to the output. Let x be a variable whose value is set as a result of a `pop` or `peek`. Besides propagating facts regarding the value of x, our analysis also propagates the input element referenced by the `pop` or `peek` as $\Gamma[x]$. The values of $\Gamma[x]$ are propagated in assignments as long as x is not modified. Assignments such as `y = x` are handled by setting $\Gamma[y] = \Gamma[x]$ (line 31).

The analysis determines if the two location sharing constraints are satisfied during the interpretation of `push` (x). The analysis determines whether the argument x is passed ($\Gamma[x] \neq \emptyset$) or updated ($\Gamma[x] = \emptyset$) by inspecting $\Gamma[x]$. The analysis ensures that this property holds across all paths by associating a variable $A[O_k]$ with each output element O_k. $A[O_k]$ is initialized to $true$ and set to $false$ if there exists an execution where O_k is updated rather than passed. Condition ($L1$) is satisfied for element O_k when $A[O_k]$ is $true$. Additionally, we keep track of all potential passes from input element I_i to output element O_k during all executions in variable P. Condition ($L2$) is

Input: CFG cfg
Output: Fragment f(V, E, L)
Data: worklist : queue
L – live ranges
A – pass or update
Γ – elements referenced in `pop` or `push`
P – potential edges

worklist = {cfg.entry()}
$\text{OUT}_{\{mc,pop,push\}}$[cfg.entry()] = 0
foreach *e* ∈ *input* **do** L[e] = [0, 0]
foreach *e* ∈ *output* **do** L[e] = ∅; A[e] = $true$;
P =∅
while *worklist* ≠ ∅ **do**
  remove n from worklist
  $\text{OUT}_v[n]$ = $\text{IN}_v[n]$ $\forall v$
  switch *n* **do**
    case [`x = pop() — pop()`] **do**
      if *n* is *x* = *pop()* **then** OUT_{mc}[n] = IN_{mc}[n] + 1
      OUT_{pop}[n] = IN_{pop}[n] + 1
      // Live range computation
      Let elem={ input[e] — $e \in \gamma(\text{IN}_{pop}[n])$ }
      L[e] = L[e] ⊔ [IN_{mc}[n], IN_{mc}[n]] ∀ e ∈ elem
      // Track location sharing
      if *n* is *x* = *pop()* **then** $\text{OUT}_{\Gamma[x]}$ = elem
    case `x = peek(y)` **do**
      OUT_{mc}[n] = IN_{mc}[n] + 1
      // Live range computation
      Let elem={ input[e] — $e \in \gamma(\text{IN}_{pop}[n] + y)$ }
      L[e] = L[e] ⊔ [IN_{mc}[n], IN_{mc}[n]] ∀ e ∈ elem
      // Track location sharing
      $\text{OUT}_{\Gamma[x]}$ = elem
    case `push (x)` **do**
      OUT_{mc}[n] = IN_{mc}[n] + 1
      OUT_{push}[n] = IN_{push}[n] + 1
      // Live range computation
      Let elem={ output[e] — $e \in \gamma(\text{IN}_{push}[n])$ }
      L[e] = L[e] ⊔ [IN_{mc}[n], IN_{mc}[n]] ∀ e ∈ elem
      // Track location sharing
      A[e] = A[e] ∧($\text{IN}_{\Gamma[x]}[n] \neq \emptyset$) ∀$e$ ∈ elem
      P = P ∪{$(e, f) \mid e \in \text{IN}_{\Gamma[x]}[n] \wedge f \in$ elem }
    case $z = x \oplus y$ **do**
      $\text{OUT}_{\Gamma[z]}$[n] = ∅
    case `x = y` **do**
      $\text{OUT}_{\Gamma[x]}$[n] = $\text{OUT}_{\Gamma[y]}$[n]
    case *branch* **do**
      $\text{OUT}_v^T[n]$ = $\text{IN}_v[n]$ $\forall v$, if condition is true/undetermined
      $\text{OUT}_v^F[n]$ = $\text{IN}_v[n]$ $\forall v$, if condition is false/undetermined
    case *simple junction* **do**
      $\text{OUT}_{mc} = \max_{p \in pred(n)} \text{OUT}_{mc}[p]$
      $\text{OUT}_{pop}[n] = \sqcup_{p \in pred(n)} \text{OUT}_{pop}[p]$
      $\text{OUT}_{push}[n] = \sqcup_{p \in pred(n)} \text{OUT}_{push}[p]$
      $\text{OUT}_{\Gamma[x]}[n] = \sqcup_{p \in pred(n)} \text{OUT}_{\Gamma[x]}[p]$ ∀ variables x
    case *loop junction* **do**
      unroll the loop using widening if necessary
  end
  add the descendants whose facts have changed to the $worklist$
end

Figure 4.1. Component analysis.

	pop	*push*	*mc*
$a_{1,2}$	0	0	0
n_2	$L[I_0] = [0,0] \sqcup [0,0] = [0,0]$ $x = \top$ $\Gamma[x] = \{I_0\}$		
$a_{2,3}$	1	0	1
$a_{3,4}$	1	0	1
n_4			
$a_{4,5}$	2	0	1
n_5	$L[O_0] = \emptyset \sqcup [1,1] = [1,1]$ $A[O_0] =$ `false`		
$a_{5,8}$	2	1	2
$a_{3,6}$	1	0	1
n_6	$L[I_1] = [0,1]$ $t_1 = \top$ $\Gamma[t_1] = \{I_1\}$		
$a_{6,7}$	2	0	2
n_7	$L[O_0] = \emptyset \sqcup [2,2] = [2,2]$ $A[O_0] =$ `true` $P = \{I_1, O_0\}$		
$a_{7,8}$	2	1	3
n_8	$L[I_0] = [0,0]$ $L[I_1] = [0,1]$ $L[O_0] = [1,1] \sqcup [2,2] = [1,2]$ $A[O_0]$=`false` $P = \{I_1, O_0\}$		
$a_{8,9}$	2	1	3
n_9	$L[I_2] = [0,0] \sqcup [3,3] = [0,3]$ $t_2 = \top$ $\Gamma[t_2] = \{I_2\}$		
$a_{9,10}$	3	1	4
n_{10}	$L[O_1] = \emptyset \sqcup [4,4] = [4,4]$ $A[O_1]$=`true` $P = \{(I_1, O_0), (I_2, O_1)\}$		
$a_{10,11}$	3	2	5

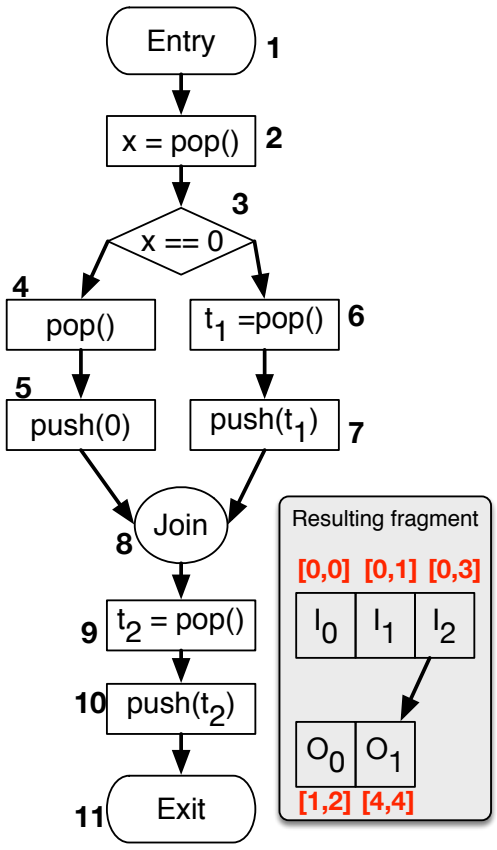


Figure 4.2. Abstract interpretation of a code fragment.

satisfied for an pair (I_i, O_k) if $(I_i, O_k) \in P$ is the only edge. The edges that satisfy both conditions are added as the final edges (E) of the fragment.

An example of AI on a CFG fragment is shown in Figure 4.2. The table shows the data flow facts available on each arc before and after the interpretation of a node. Only the updated or newly derived facts are shown in the table. Initially, the values of the mc, pop, and $push$ are set to zero. Interpreting the `pop` on node 2 updates the fact that I_0 is live in the interval $L[I_0] = [0,0]$. Since the value of I_0 is unknown during the analysis, x is set to $\top$ and $\Gamma[x]$ is set to I_0. The truth-value of the condition at branch 3 cannot be determined, so the analysis will execute both branches. The `pop` in node 4 simply indicates that I_1 is not in use anymore and, as a result, no new facts are derived. The interpretation of the `push` in node 5 results in increasing the $push$ and mc

counters. Additionally, $L[O_0]$ that was initially $\emptyset$ is updated to $[1, 1]$. The algorithm records that O_0 was updated rather than passed by setting $A[O_0] = false$. I_1 is saved in t_1 in node 6 and passed as O_0 in node 7. Thus during the execution of node 7, the analysis checks for location sharing. Since $t_1 = I_1$, the analysis sets $A[O_0] = true$ and adds (I_1, O_0) as a potential location share. The `pop` and `push` in nodes 6 and 7 are handled similarly. The analysis handles join nodes by using either the maximum, interval union, or "and" operator. There are two interesting cases. The live ranges $L[O_0]$ are merged to be $[1, 2]$ using the interval union. Similarly, $A[O_0]$ has different values indicating that O_0 was passed on one path but updated on the other. Thus, the combined value of $A[O_0]$ is set to $false$ using the "and" operator. The analysis continues producing the results shown in the figure.

4.3.1.2 Whole-program analysis

The component analysis constructs fragments that describe the memory optimization opportunities during a single invocation of a component. The whole-program analysis stitches these fragments to compose a MG that characterizes the memory operations of the entire program. We remind the reader that a stream schedule is composed of an initialization phase executed once and a steady phase that is executed repeatedly forever. To characterize the entire program it is sufficient to simulate the initialization phase and a single execution of the steady phase.

The stitching algorithm considers the execution of components in the schedule order. Consider the execution of a component that is invoked n times during the schedule. For each component, we add $|I| = r_{peek} + r_{pop} \times (n - 1)$ input elements and $|O| = r_{push} \times n$ elements in the MG.

The stitching algorithm must relate the indexes of input/output elements in MG to those in the memory fragment (F). The mapping must recognize that a component may access its input over

overlapping windows and produce samples in overlapping windows. Accordingly, the memory operations of I_i is the union of memory operations of input elements I_j^F such that $I_i \in I_{MG \to F}(I_i)$:

$$I_{MG \to F}(I_i) = \{I_j^F \geq 0 \mid I_j^F = I_i - k \times r_{pop} \quad k \in \mathbb{N}\}$$

Similarly, the memory operations of O_k are the same as the operations of O_l^F such that:

$$O_{MG \to F}(O_k) = \{O_l^F | O_k = \mod (O_l^F, r_{push})\}$$

Location sharing opportunities are computed by iteratively considering each possible link (I_i, O_k). A link (I_i, O_k) is added to MG if there exists a link (I_j^F, O_l^F) in the fragment such that $I_j^F \in I_{MG \to F}(I_i)$ and $O_l^F \in O_{MG \to F}(O_k)$. The construction does not introduce location sharing conflicts regardless of potentially overlapping output windows.

The next step in the construction of MG is to capture the exchange of data between components. This process takes advantage of the hierarchical nature of the stream graph. The components connected in a pipeline are adjacent in topological order and linked through their inputs and outputs. The split-join constructs are handled by adding split and join components that internally implement either duplicate data or round-robin policies.

The live ranges included in the fragment are the time when each memory operation was performed relative to the beginning of the component execution. A live range in MG is represented as a triple $(phase, step, mc)$ where the $phase$ is the phase of the schedule (0 for init, 1 for steady), $step$ is a counter that is incremented after each component invocation, and mc is the memory counter in the fragment. The interval union operator can be easily extended to operate over triples. To account for window overlaps, the live ranges of an element I_i is set to equal $\bigsqcup_{I_j^F \in I_{MG \to F}(I_i)} L[I_j^F]$. Additionally, we also need to account that some elements may be shared.

An element e shares the same location with f if there is a path between them. The live range of e is expanded to $\bigsqcup_{f \in shared_{MG}(e)} L[f]$ to account for location sharing.

4.3.2 Memory Layout

The layout algorithm operates on a single-appearance schedule [46]. Generating a good layout is necessary for reducing memory usage, improving performance by reducing memory accesses, ensuring good cache locality, and generating efficient code. We propose three heuristics and our empirical evaluation shows they effectively balance these requirements. Our approach is driven by two empirical insights into stream programs. (1) StreamIt and other data flow languages include constructs such as split-joins that share and reorder samples. Traditionally, memory copying across channels are used to implement these constructs. We opt for the alternative of changing the logical layout of samples without performing any copy operations. This leads to significant reductions in both the memory size and number of accesses. Samples are reordered using round-robin split-joins can often be accessed efficiently using linear iterators of the form $base + step \times i$. Typically, $step$ is a small constant leading to reasonable cache locality. (2) A filter operates on the input provided by the previous component in the SFG. It is often possible for a filter to reuse the memory allocated for the previous filter. This is because as a filter `pops` samples from the input, the memory locations where the samples were located become available for reuse.

Prior work has considered different buffer management strategies for storing samples due to sliding windows. StreamIt filters process their input in sliding windows by peeking more samples than r_{pop}. The proposed techniques include modulation [44, 50] and copy-shift [44]. The modulation strategy stores samples in a circular buffer and requires modulo operations for indexing. Copy-shift avoids potentially expensive modulo operations by shifting the remaining samples of sliding windows in the buffer from the previous filter execution. When coupled with execution

scaling, the copy-shift approach can significantly outperform modulation [44]. Execution scaling works by scaling the number of times each component in a schedule is executed. This has the effect of reducing the number of copy operations relative to the size of the input. Our memory management approach uses the copy-shift buffer management.

We organize the memory M as cells, each cell having an address and a size equal to the machine word (64 bits on considered platforms). The memory supports two operations: `memappend` and `meminsert`. The `memappend(`w`)` operation allocates w words at the end of current memory allocation. The `meminsert(`$addr$`, `w`)` operation inserts w words at location $addr$, reindexing the memory addresses to account for the insertion. Additionally, `meminsert` ensures that the mapping between elements and cells is maintained after the insertion. A layout is defined as a mapping $\Pi : V \rightarrow M$ that maps each element in MG to a memory location. We also maintain the reverse mapping $\Phi : M \rightarrow 2^{|V|}$ from an address to a set of elements stored at the address. A valid mapping ensures that for any memory location m, the intersection of the live ranges of the elements in $\Phi(m)$ is empty. This is accomplished by inspecting the appropriate live ranges in MG.

The layout algorithm generates the memory layout by simulating the execution of a component in scheduling order and constructing Π incrementally. The pseudocode is included in Algorithm 4.3 and the line numbers included in this section refer to this algorithm. The input to the algorithm is the MG and the SFG with appropriately defined successor and predecessor functions. Components operate over windows of input or output elements. We define two operations for manipulating windows: `flatten` and `window`. The `window`(w, $size$, $overlap$) transforms a single window into a group of windows that overlap by $overlap$ elements with each window having $size$ elements. Conversely, `flatten` produces a single window from overlapped windows. If c is a source, the algorithm creates a window of size $r_{push} \times n$ that is mapped to the next available mem-

```
Input: MG(V, E, L) – memory graph
       SFG - stream flow graph
       R[c] – elements that are remainders of component c
       store[c] – storage for the remainders of component c
Data:  in[prev] – input windows from the prev component(s)
       out[next] – output windows to the next component(s)
1  foreach (phase, n, c) in schedule.init do
2      if c is source then
3          out = newwindow(r_push[c] × n )
4          p = memappend (r_push[c] × n)
5          foreach i = 0 . . . n do Π[out(i)] = p(i)
6      else
7          switch c do
8              case duplicate-split do
9                  in = flatten (pred (c) .out[c])
10                 foreach next ∈ succ (c) do
11                     out[next].append(in)
12                 end
13             case round-robin-split do
14                 in = flatten (pred (c) .out[c])
15                 S = vector of weights of the splitter
16                 start = 0
17                 for 0 . . . n do
18                     foreach next ∈ succ (c) do
19                         out[next].append(in[start:start+S[next])
20                         start = start + S[next]
21                     end
22                 end
23             case round-robin-join do
24                 J = vector of weights of the joiner
25                 next = succ (c)
26                 foreach prev ∈ pred (c) do start[prev] = 0
27                 for 0 . . . n do
28                     foreach prev ∈ pred (c) do
29                         w = prev.out[c])
30                         out[next].append(w[start[prev]:start[prev]+J[prev]])
31                         start[prev] = start[prev] + J[prev]
32                     end
33                 end
34             case filter do
35                 windows = window(in, r_peek[c], r_pop[c])
36                 for e ∈ R[c] do
37                     Φ[e] = store[c][e]
38                 end
39                 next = succ (c)
40                 out = newwindow (r_push × n)
41                 j = 0 // index of the output sample;
42                 for w = 0. . . n do
43                     U = sharedElements (windows[w])
44                     for i ∈ U \ R[c] do
45                         shared = Φ[Π[i]]
46                         hasConflict = ⋂_{e∈shared} L(e)
47                         if hasConflict = ∅ then
48                             Π[j] = Π[i];
49                             j++;
50                         else
51                             ... handle conflicts using described heuristics ...
52                         end
53                     end
54                 end
55             end
56     end
57     Procedure sharedElements (window)
58         U = {I_i — I_i ∈ window}
59         for I_i in window do
60             L = {I_i|(I_i, O_k) ∈ E}
61             for e_o in L do
62                 Π[e_o] = Π[I_i]
63                 U = U \{e_o}
64             end
65         end
66         return U
67 end
```

Figure 4.3. Layout algorithm.

ory location (line 2). All other components will generate their layouts based on the windows of the previous components in the SFG, the details depending on whether the component is a `split`,

`join`, or a `filter`.

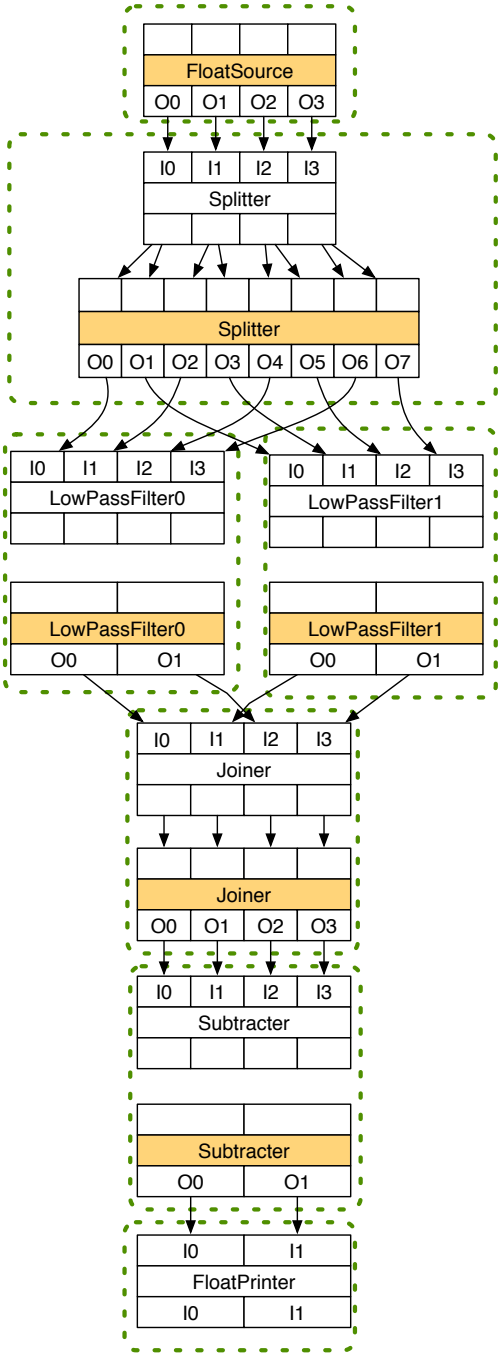
Split-joins are used to share and to reorder samples and their behavior is captured as edges in MG. The first step in handling both splits and joins is to flatten the output window(s) from the previous component(s) into a single window. If c is a duplicate splitter, the layout algorithm will pass the input window to the each of its successors (lines 10 – 11). If c is a round-robin splitter, the layout algorithm passes a subset of the elements in the input window to the output window of each successor (lines 15 – 20). The number of elements passed to each successor is part of the splitter specification in the program (stored in S). Round-robin joiners are handled in an equivalent manner. The layout generates an output window in which elements from each one of the predecessor components are appended (lines 24 – 31). Samples are inserted by considering the predecessors in order and adding the programmer specified number of samples (stored in J). Note that `split` and `join` operate in the logical space and do not require changing the mapping between logical space and memory.

The layout process for a filter starts by flattening the output window of the previous component and windowing the result according to the component's peek and pop rates. This generates n windows each containing r_{peek} samples that the algorithm will manipulate. The algorithm will first consider each element I_i in the input window, determining if there is an edge (I_i, O_k) in MG, where O_k is an output element. The existence of the edge indicates that I_i and O_k may share the same memory location. Accordingly, we map O_k to I_i's memory cell (i.e., $\Pi[O_k] = \Pi[I_i]$). These operations are performed as part of `sharedElements` procedure in the pseudocode. Let U include the set of elements whose mapping has not been determined. The elements in U are updated by the component during its execution. We consider three heuristics for laying out these conflicts: *always-append*(AA), *append-on-conflict*(AoC), and *insert-in-place*(IP). (1) The AA heuris-

tic maps elements in U to a group of contiguous memory cells at the end of the current allocation. This approach has the advantage of ensuring cache locality and simplifies the generated code. (2) The AoC heuristic will first try to layout windows within the memory region allocated for the previous component. When this is not possible due to conflicts in the live ranges of variables, then the window will be mapped to a contiguous portion of memory at the end of current allocation. We expect that this heuristic will reduce the size of memory allocation over the previous heuristic albeit at the cost of increased code complexity and execution time. (3) The IP heuristic inserts a number of memory cells at the location where a conflict is determined. This has the effect of allowing the subsequent components to operate on a layout that maps their input elements to proximate memory locations.

The layout algorithm must account for the fact that after the initialization phase of the schedule and at the completion of each steady phase, components with $r_{peek} > r_{pop}$ will have input elements that are used in subsequent executions. The remainders of a component c are stored in $R[c]$. Consistent with the copy-shift strategy, such components are responsible for saving these remainders at the completion of the initialization and steady phases. The remainders are loaded in the beginning of the input window of a component prior to the beginning its execution. Remainders are treated as a special case since their live ranges cover the entire phase, creating few opportunities for temporal reuse. Accordingly, remainders are saved and loaded from special remainder stores. Components that operate on shared buffers also have shared remainders. We optimize the loading and storing of data from shared stores to avoid duplicate memory operations.

Figure 4.4 shows the memory graph generated and stream schedule for a bandpass filter. The figure also includes the physical layout generated using the AoC heuristic. In the initialization phase, the algorithm starts by creating a window containing 3 elements that start



Schedule
Init: (FloatSource,3) (Splitter,3)
(LowPassFilter0,1) (LowPassFilter1,1)
(Joiner,1) (Subtractor,1) (FloatPrinter,1)
Steady: (FloatSource,1) (Splitter,1)
(LowPassFilter0,1) (LowPassFilter1,1)
(Joiner,1) (Subtractor,1) (FloatPrinter,1)

Live ranges:

Element	Start	End
FloatSource:O_0	(0, 0, 0)	(0, 7, 3)
FloatSource:O_1	(0, 1, 0)	(1, 3, 3)
FloatSource:O_2	(0, 2, 0)	(1, 6, 1)
FloatSource:O_3	(1, 0, 0)	(1, 6, 2)
LowPassFilter0:O_0	(0, 6, 4)	(0, 9, 3)
LowPassFilter0:O_1	(1, 2, 4)	(1, 5, 3)
LowPassFilter1:O_0	(0, 7, 4)	(0, 9, 4)
LowPassFilter1:O_1	(1, 3, 4)	(1, 5, 4)
Subtractor:O_0	(0, 9, 2)	(0, 10, 0)
Subtractor:O_1	(1, 5, 2)	(1, 6, 0)

Physical layout:

MEM	Initialization	Steady
0	FloatSource:O_0 LowPass1:O_0	FloatSource:O_1 LowPass1:O_1
1	FloatSource:O_1	FloatSource:O_2
2	FloatSource:O_2	FloatSource:O_3
3	LowPass0:O_0	LowPass0:O_1
4	Subtractor:O_0	Subtractor:O_1

Remainder store: 2 cells

Figure 4.4. Bandpass filter: memory graph, stream schedule, and physical layout.

at MEM[0]. The duplicate splitter replicates this window to the input of the low pass filters. `LowPassFilter0` checks if it may reuse the cell allocated for `LowPassFilter0`:I_0 to store its `LowPassFilter0`:O_0. Since the intersection of the live ranges $[(0,0,0),(0,7,3)]$ and

$[(0,6,3),(0,9,3)]$ is not empty, this is not possible. This is expected since this sample is also an input to `LowPassFilter1`. Accordingly, it is mapped to MEM[3]. `LowPassFilter1` performs a similar check to determine if the cell allocated for `LowPassFilter1`$:I_0$ may be reused for `LowPassFilter1`$:O_0$. In this case where the intersection of the live ranges $[(0,0,0),(0,7,3)]$ and $[(0,7,4),(0,9,4)]$ is empty, the cell may be reused. Accordingly, `LowPassFilter1`$:O_0$ will reuse MEM[0]. In the steady phase, `FloatSource`$:O_3$ is pushed and shifted to MEM[2] because `FloatSource`$:O_1$ and `FloatSource`$:O_2$ will be loaded to MEM[0:1]. The following filters produce samples at the locations as in the initialization phase, proving the mapping allows to execute the steady schedule infinite often. The resulting layout is shown in Figure 4.4. We note that a naive memory management approach that uses respective buffers for splits and joins (as it is the case for the default StreamIt compiler) would require a total of 13 cells, one for each output element. In contrast, our layout algorithm requires only (5+2) cells by taking advantage of location and temporal sharing.

Code generation. The last step in ESMS is the generation of C code from a StreamIt program. Most of the details of code generation are unsurprising. The only aspect that requires careful handling is the generation of code for `peek`, `pop`, and `push`. When the memory is accessed contiguously, generating code for memory instructions is straightforward. However, handling a fragmented memory layout is challenging when memory operations are nested in loops. Our compiler implements two methods for handling this case. In most cases, we opt for splitting the loop at the boundary of contiguous memory locations. Obviously, this trade-off increased code complexity for execution time improvements. The alternative is to apply indirect addressing where the memory instructions operate in the logical space and a static mapping between the logical and physical space is concretized as a lookup table used at run-time. A tunable constant is used to

Benchmark	Description
AutoCor	autocorrelation computation
Bitonic Sort	bitonic sort
MergeSort	merge sort
Repeater	repeats odd samples M times
FIR/FIRCourse	finite impose response filters
FMRadio	FM radio with 10-way equalizer
FFT2/FFT3	FFT computation
MatrixMult/MatrixMultBlock	matrix multiplication
BeepBeep [3]	acoustic localization
MFCC	computation of MFCC coefficients
Crowd [39]	estimation of number of co-located speakers

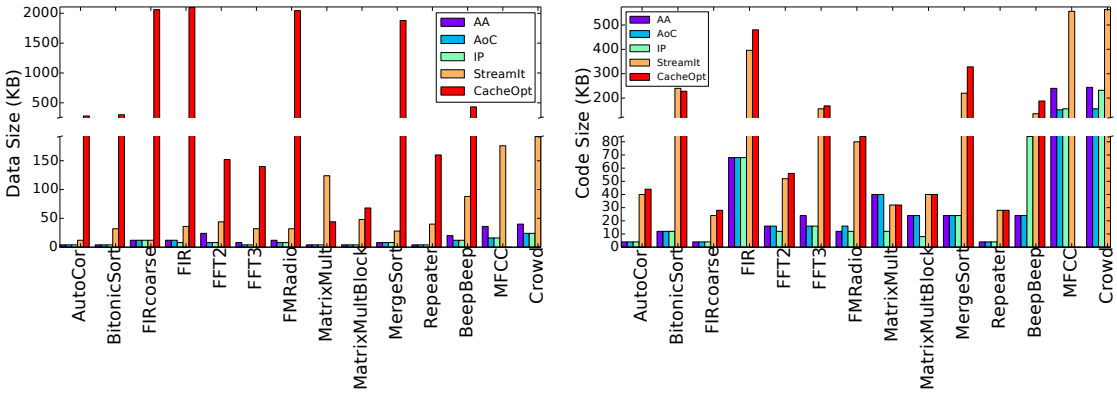
Table 4.1. Benchmark suite.

control between the two options.

4.4 Experiments

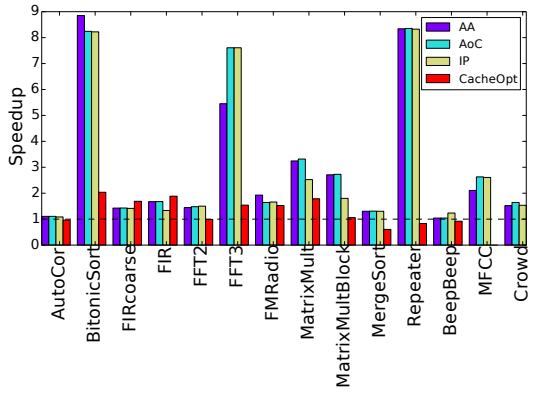
In the following, we show the benefits of the proposed static analysis and memory layout algorithm. Benchmarks are made on a desktop machine that has a 3GHz Intel(R) Xeon(R) CPU E5-1680 v2, and a Nexus 10 tablet that has a Samsung Exynos 5250 SoC with 1.7 GHz Dual-core Cortex-A15. The Xeon has 32KB L1 instruction and data caches, 256K L2, and shared 25MB L3 caches. The ARM has 32KB L1 instruction and data caches, and a shared 1 MB L2 cache. The system runs Android 5.1. Programs are compiled using the native development kit (NDK) r10d that uses `gcc 4.8`. A wrapper Java application is generated to invoke the generated code. All programs are compiled using the highest optimization level (i.e., `-O3`).

The suite of benchmarks consists of 14 stream applications (see Table 4.1). The majority of the benchmarks were developed as part of the StreamIt project. In addition, we implemented three mobile sensing applications using StreamIt: the `BeepBeep` app [3] performs sound-based localization, the `MFCC` app implements the core of a speaker identification app (e.g, [40]), and the `Crowd` app [39] estimates the number of co-located speakers.



(a) Data size.

(b) Code size.



(c) Speedup.

Figure 4.5. Data, code, and speedup improvements on Intel Core i5-3550.

Static Analysis. We have evaluated the pessimism of the static analysis proposed in Section 4.3.1. The static analysis was able to precisely characterize the memory behaviors of all benchmarks with the exception of the MergeSort benchmark. It includes nondeterminism since the control flow depends on the input data. This shows that for a wide range of programs our analysis can precisely characterize the complete memory behavior of a stream program. For the MergeSort benchmark, the analysis derived a safe approximation of location and temporal sharing opportunities. We note that even for these benchmarks, the nondeterminism is confined to a single component and does not affect the others in the application.

Intel Measurements. We have measured the impact of memory optimizations on three dimensions: data size, code size, and speedup. We report both absolute and relative improvements in these dimensions. The relative improvements are computed based on the performance of the default StreamIt compiler (labeled `StreamIt`). We have also run the StreamIt in conjunction with the cache optimization described in [44]. These results are reported as `CacheOpt`. The compiler with cache optimizations failed to generate code for `MFCC` and `Crowd` applications because it ran out of memory. The performance of the *always-append*, *append-on-conflict*, and *insert-in-place* heuristics are denoted by labels `AA`, `AoC`, and `IP` respectively. The code and data size results were obtained only on the Intel perform using the `size` utility. The tool reports both the code and data sizes in multiples of a page. The speedup results are based on the CPU user time reported by the `time` utility.

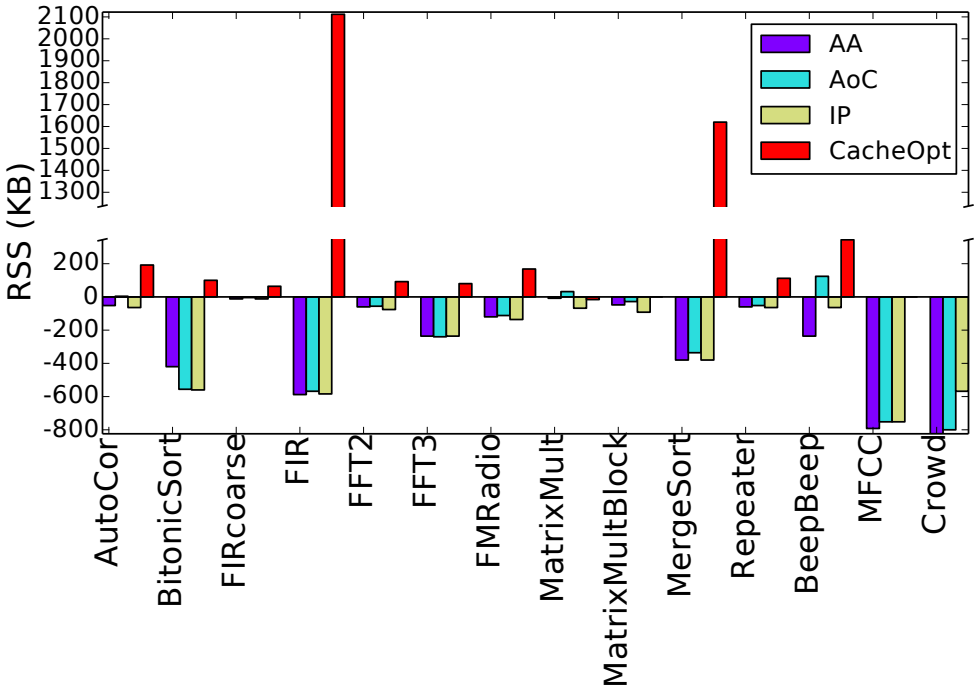
Figure 4.5a shows the data size with `StreamIt` as the baseline. The average reductions on memory footprint of `AA`, `AoC`, and `IP` are 50KB, 55KB, and 98KB. These represent reductions on data size between 45–96%. The `AA` heuristic provides the smallest reductions on data size since it always appends rather than attempts to resolve memory conflicts. The `AoC` and `IP` heuristics achieved comparable performance in terms of memory usage. In contrast, enabling the `CacheOpt` increased memory consumption by an average of 627KB. This increase can be as large as 98% for MergeSort or FFT2.

Figure 4.5b shows the code size with `StreamIt` as the baseline. The average code reductions for `AA`, `AoC`, and `IP` are 130KB, 143KB, 136KB. In relative terms, the average reductions are 69%, 72%, and 77% respectively. The ESMS optimizations reduce the code size by not generating code for `split-join` constructs and other components that reorder (without modifying) the input data. Nevertheless, even with these savings, there are cases when ESMS has larger code

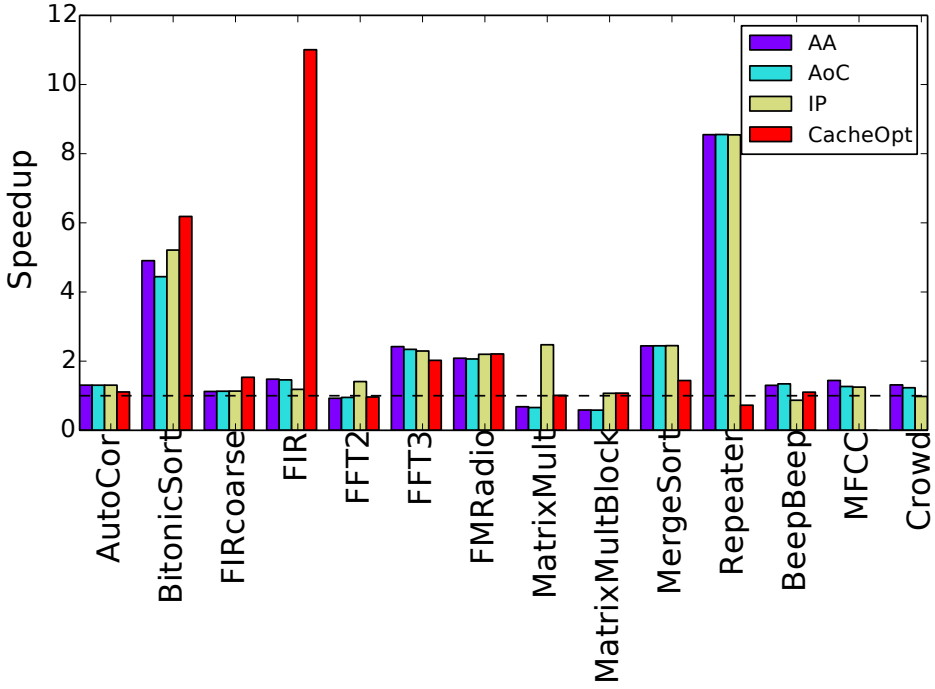
size than StreamIt. CacheOpt typically has a minimal impact on the code size. On average, it adds 14KB to the code size of the considered benchmarks.

Figure 4.5c shows the speedup relative to StreamIt. The average speedups of AA, AoC, and IP are 3, 3.1, and 3. In contrast the average speedup of CacheOpt is merely 1.07. All of ESMS heuristics outperformed CacheOpt with the exception of the two FIR benchmarks on both platforms because the FIR pipeline is long enough to cause instruction cache misses in one schedule iteration and the CacheOpt performs better by reducing the cache miss rate with execution scaling which is constrained by the data cache size. ESMS with reduced data size is expected to support more execution scaling in the future. Otherwise, the reason for these significant performance improvements is the fact that ESMS uses less memory access by effectively sharing data across components. We validated that this was the case by using cachegrind to track the number of memory accesses (data not included due to limited space). Moreover, the smaller footprint leads to a smaller working set that fits within the large cache of this platform. On the Intel platform, ESMS managed to improve the stream processing throughput while significantly reducing the memory consumption. The heuristics that handle conflicts either by inserting or appending achieve more memory savings than AA. Resolving conflicts through insertion tends to achieve more reductions on data size but slightly lower performance than appending.

ARM Measurements. We generate Android applications that include the compiled code as a shared library. To quantify the impact of memory optimizations on Android applications, we measure the maximum resident working set size (RSS), which includes the total memory allocation at run-time for both the Android application and the shared library. Figure 4.6a shows the RSS relative to the baseline StreamIt on ARM. The average reductions for AA, AoC, and IP are 274KB, 247KB, and 261KB respectively. In contrast, CacheOpt increased the buffer size by 347KB in



(a) RSS relative to baseline.



(b) Speedup.

Figure 4.6. RSS and speedup improvements for Samsung Exynos 5250.

order to operate on larger buffers. This shows ESMS facilitates effective memory savings. Figure 4.6b shows the speedup on ARM. The average speedups for `AA`, `AoC`, and `IP` are 2.18, 2.03, and 2.3, respectively. `CacheOpt` achieved a comparable speedup of 2.18. The standard deviation for the speedup improvements for `AA`, `AoC`, `IP`, and `CacheOpt` were 2.12, 2.16, 2.3, and 2.95. The lower standard deviation of `AA` indicates that it performed the best. ESMS achieves similar performance but significantly reduces the memory footprints.

4.5 Related Work

The memory optimization problem is similar to the classical aggregate update [51] problem in functional programming languages. The numerous solutions proposed to address this problem can be broadly classified as either run-time or static approaches. Run-time approaches typically rely on either garbage collection or reference counting. In contrast, static approaches require compiler analyses to determine live ranges of variables in order to ensure safe data sharing. Live range information may be extracted either through heuristics [52] or abstract interpretation [53]. A distinguishing aspect of our analysis is that it takes advantage of stream properties to characterize *complete* applications.

In [54], a greedy in-place reuse of memory allocations is proposed for the data flow model of LabView. The heuristic approach chooses variables using a cost metric to merge and store them in the same location. In contrast, our layout algorithm takes advantage of the structure of the data flow for memory optimization. More recently, an annotation-based approach has been proposed to address memory management in the context of data flow languages [55]. We note that StreamIt includes explicit data sharing information as part of the `split-join` construct. Moreover, we show that significant improvements in stream processing rates and reductions on memory footprints may be achieved without requiring annotations.

The problem of scheduling SDF graphs to optimize different metrics has been studied extensively [45,50]. The previous work of phased scheduling [46] shows a steady state SDF schedule can be rearranged in phases to shorten the output latency compared with SASs which have only one phase at the expense of increasing code size. Since our goal is to optimize memory usage and performance, ESMS may support phased scheduling in the future.

On the other hand, several cache performance improvements were proposed including the copy-shift buffering and execution scaling for StreamIt in [44], and cache-aware optimizations for synchronous data flows [56] as part of the Ptolemy project [57]. In contrast with [44] that trades space for performance, our approach improves cache locality by saving space while eliminating unnecessary memory operations to improve the performance. In [50], the memory reuse is based on overlaying channel buffers in terms of their live ranges while maintaining periodical modulo access. Compiler optimizations were also considered to generate instructions to avoid the modulo overhead selectively given the static schedule. From this perspective, our compiler optimizations allow for more aggressively reuse and even remap non-contiguous memory accesses across filter invocations at some cost of increasing code complexity.

In addition, the linear analysis in [58] is an effective alternative to improve the performance by reducing the number of linear filters while saving memory usage accordingly. Instead, we consider filters only in terms of general memory operations and are able to eliminate non-productive filters such as split-joins or filters that reorder their input without modifications. In this sense, our optimizations are not limited to linear filters but more general.

Memory management can have a significant performance impact on stream processing engines. XStream [35] and WaveScript [34] use an abstract data structure called SigSeg to merge and segment data streams efficiently. CSense [43] proposes several memory management tech-

niques to optimize the exchange of frames between components. StreamFlex [59] is yet another stream processing toolkit written in Java, aiming at avoiding the garbage collection overhead while satisfying real-time constraints.

4.6 Conclusions

In this chapter, we presented – ESMS – a novel approach for optimizing the memory management of stream programs. Our approach leverages the unique properties of stream programs for both static analysis and memory layout. We developed a novel static analysis technique that characterizes the *behavior* of complete stream programs by identifying location and temporal sharing opportunities. Our analysis scales to handle large stream programs by separating the components analysis from the creation of memory graphs through stitching. An evaluation conducted on 14 benchmarks including three for mobile sensing applications reveals that the analysis is precise for a majority of stream applications. Besides, sound approximations of the memory behavior are provided for the other applications.

We developed a novel memory layout algorithm. The algorithm recognizes that stream programs have significant opportunities for location sharing. In StreamIt, these opportunities are often the result of constructing programs using `pipeline` and `split-join` constructs. Additionally, we observe that connected filters in a SFG may often operate on the same memory since the live ranges of their buffers usually do not overlap. Obviously, this is not always possible. We introduced three simple heuristics to handle conflicts when they arise during the memory layout process. Our empirical evaluation indicates that ESMS may achieve significant memory savings. On the Intel platform, these memory savings are coupled with improvements in stream processing rates over StreamIt with cache optimizations. On the ARM platform, the stream processing improvements are comparable to those achieved by StreamIt with cache optimizations. These results

show that ESMS is effective in developing efficient memory management for stream programs. In the future, we plan to continue exploring compiler optimization techniques to further improve the performance of stream processing engines for mobile sensing applications.

CHAPTER 5

WORKLOAD SHAPING ENERGY OPTIMIZATIONS FOR MOBILE SENSING

5.1 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 [60]. Similarly, Sociophone uses microphones to track face-to-face interactions and identify close social relations [61]. 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 power management (PM) techniques that coordinate the use of hardware resources to minimize energy consumption.

In this chapter, we focus on *workload shaping energy optimizations with predictable performance* — a class of optimizations that save energy by controlling the time when hardware resources are used. Workload shaping energy optimizations build on two insights regarding the workload of MSAs. First, today's mobile platforms have powerful processors and I/O interfaces that provide high peak performance at a significant energy cost. Unfortunately, MSAs typically generate light workloads that use a phone's hardware inefficiently [62, 63]. Second, most of the operations performed by a 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 hinder the user experience. Thus, it is essential for the

developer to use PM policies that *coordinate when hardware resources are used to reduce energy consumption under soft end-to-end deadlines.*

A unique property of the considered class of energy optimizations is that they provide *predictable performance*. A PM policy provides predictable performance if its energy and the delay can be estimated in a computationally efficient manner, from a few measurements, at compile time. Energy optimizations that have predictable performance have several significant practical advantages. A developer can explore a broad range of PM policies to understand the trade-off between energy consumption and delay. This information allows a developer determine the best energy-delay tradeoff for his app. Equally important, energy optimizations with predictable performance foster an agile software development approach in which energy considerations are addressed throughout the development cycles. In fact, we advocate that the performance of a PM policy should be estimated as part of the build process. Finally, we should provide developers with templates of energy optimizations that can be instantiated using automated tools. The tools are geared towards developers who have a limited knowledge of PM.

Manually implementing workload shaping energy optimizations is difficult. It amounts to exposing the locations in the code where an app uses a hardware resource as synchronization points. Then, a PM policy would dynamically determine when each thread of the app will block or resume at each synchronization point. Reasoning about the safety of such policies is inherently difficult, let alone estimating the delay or energy consumption of the app. Our insight is that these challenges can be addressed by adopting stream programs as a high-level abstraction for power management: a stream program describes a MSA as a graph of components whose execution is coordinated by an explicit PM policy at run-time. We will show that MSAs written using this approach 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 property provides a foundation for writing PM policies that have predictable performance.

In this chapter, we introduce Gratis – a new paradigm for specifying, configuring, and synthesizing workload shaping energy optimizations. Specifically, we make the following contributions:

- We present a novel paradigm and coordination language for specifying PM policies that implement workload shaping energy optimizations. The policies control the execution of a component based on the data frames in its input/output queues. A component can be triggered to execute in a data-driven manner when the number of frames in the queues exceeds a threshold. Alternatively, the execution can be triggered in a time-driven manner based on the time remaining until the deadline of a frame expires. At run-time, a scheduler coordinates the execution of the app according to a PM policy.

- We developed an app simulator that can estimate the energy and delays of an app accurately from a few measurements of its constituent components. The proposed technique is computationally efficient and provides accurate performance estimates even for apps with dynamic workloads. We present techniques for synthesizing policy templates and for configuring their parameters. The combination of these tools provides an automated solution for PM in MSAs.

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 the energy consumption. For example, the battery of a phone running the `SI` app lasts for only 7 hours when audio data is

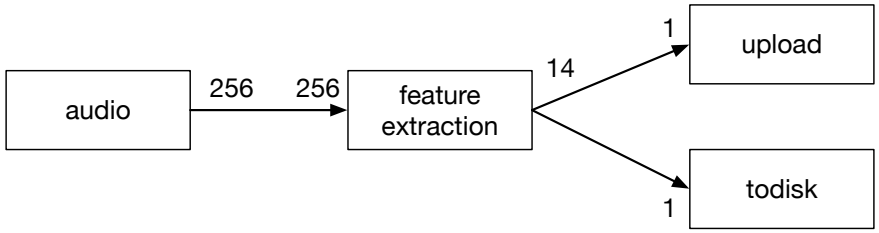
processed as soon as possible. In contrast, if the user is willing to tolerate a 60-second latency, the battery life is extended to 19 hours, which is a 2.7 times improvement. Adaptive sensing, which collects audio only when speech is detected, extends the battery life to 45 hours. The `AR` app shows 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.

The remainder of the chapteris organized as follows. The problem formulation is presented in Section 5.2. The design and prototype implementation of Gratis are detailed in Section 5.3. A detailed experimental study of using Gratis to implement two realistic MSAs is included in Section 5.4. Conclusions are provided in Section 5.6.

5.2 Problem Formulation

Workload shaping energy optimizations save significant energy by adding delays to shape the workload of a 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 5.3.

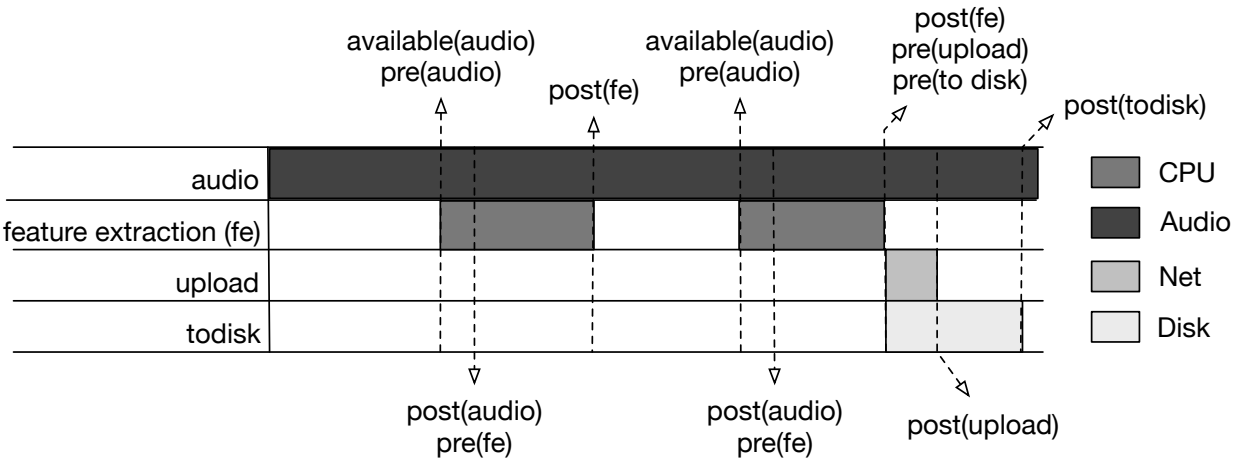
Consider a MSA that collects information about a user's social interactions using speaker identification techniques. A basic version of the Speaker Identification (`SI`) app would continuously collect audio data, extract features from the collected data, and upload them to a cloud service to determine the identity of the speakers (see Figure 5.1a). The app has a lax end-to-end deadline (e.g., on the order of minutes) since it operates in the background.



(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` $\geq th_1$:
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 5.1b with per component resource usage.

Figure 5.1. The basic version of the `SI` app that implements batching and scheduled concurrency.

Timing Semantics: An important consideration is how to capture timing in our system. A natural approach is to incorporate timing by associating a timestamp with each data sample. When a component executes, it consumes some samples 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. A policy is associated with an environment data structure that stores the values of its parameters. The end-to-end deadline is specified by setting the value of the $deadline$ variable in the environment (see line 1 in Figure 5.1b).

A limitation of maintaining a timestamp for each sample is that it has significant overhead. A more efficient approach, and the one used in Gratis , is to organize samples into frames that contain multiple samples and maintain a single timestamp per frame. This approach is similar to the SigSeg data structure used in XStream [35] and WaveScript [34].

Workload Shaping Energy Optimizations: Workload shaping energy optimizations are a broad class of optimizations that include batching, scheduled concurrency, and adaptive sensing. Batching saves energy by having power-hungry resources process multiple frames in a single activation to minimize the usage time and offset startup and shutdown costs. In our example, the network interface dominates the energy consumption of the app. Thus, significant energy savings may be achieved by buffering the frames containing the extracted audio features and uploading them in a single activation of the network interface. We may save additional energy by controlling the concurrency of the app. For example, it is possible to overlap the feature extraction and network upload. The net effect of this optimization is that it consolidates periods of activity and sleep allowing the device to exploit deeper sleep states. Another approach to saving energy is adaptive sensing (see Figure 5.3a). The app can use voice activity detection (vad) to determine whether someone is speaking. If someone is speaking, the app should remain active to monitor the ongoing

conversation. Otherwise, the app should sleep for a while to save energy. An app may use any combination of the above optimizations.

In the following, we introduce informally a PM policy that combines batching and scheduled concurrency (see Figure 5.1b). Gratis executes the PM policy in an event-driven manner. Figure 5.1c shows a timeline of the events that are generated when components are executed, and the hardware resources used by each component. 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 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 features. 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` component 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 `todisk` component). 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. Our experiments show that this approach is sufficient to control the energy-delay tradeoff in MSAs.

Predictable Performance: The unique property of the considered energy optimizations is that

they provide predictable performance. It is very difficult to reason about the impact a PM policy has on an Android app in general since neither the dependencies between components nor the concurrency of the app can be easily analyzed 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. We can ensure this property by building on three key insights: (1) The dependencies between the components and the amount of data produced and consumed by a component are captured explicitly in the stream program. (2) The PM policy coordinates when each component starts its execution explicitly. (3) MSAs execute continuously in the background suffering minimal interference from other apps [64].

Gratis takes advantage of these properties to estimate the energy and delay of an app offline with minimal computational overhead. We build on the following intuition. Since a PM policy controls concurrency explicitly, we can determine for any time the set of components that execute concurrently. We enforce at compile-time that a component uses a single hardware resource. Therefore, we can determine the set of components that use each hardware resource. If some components use the same hardware resource, they should use that resource in a fair manner. The Linux scheduler enforces fair resource usage among competing threads. If components use different hardware resources, they can be executed concurrently. These rules can be employed to build an event-driven simulator that 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 5.1c).

5.3 Design

Gratis provides an intuitive and practical approach to specifying, evaluating, configuring, and synthesizing workload shaping energy optimizations. Our approach uses stream programs as a high-level abstraction for PM. Stream programs have the benefit of explicitly capturing production/consumption rates of each component as well as their dependencies. We have developed a novel coordination language to specify a broad range of workload shaping energy optimizations as detailed in Section 5.3.1. Gratis apps are written in the StreamIt programming language [16] that we extended to support I/O operations such as reading sensor data and send/receive packets. A source-to-source compiler transforms the StreamIt code into an Android service. The execution of the app is managed by a scheduler that enforces the PM policy at run-time.

Gratis is designed to support the agile software development methodology where energy concerns are addressed throughout development cycles (see Figure 5.2). As part of the build process, the energy and delay of the applied PM policy are evaluated using an app simulator. The app simulator estimates the energy and end-to-end delay of an app based on the performance profile of each component. Gratis supports an incremental development process. A change to a component is localized and requires regenerating the energy and delay profile of only that component. The app simulator is described in Section 5.3.2. One of the key challenges of writing PM policies is configuring its parameters to reduce energy consumption while meeting the user-specified deadlines. We have developed tools for configuring and synthesizing PM templates[1] as described in Section 5.3.3.

[1] A PM template is a policy that does not include concrete values for its parameters.

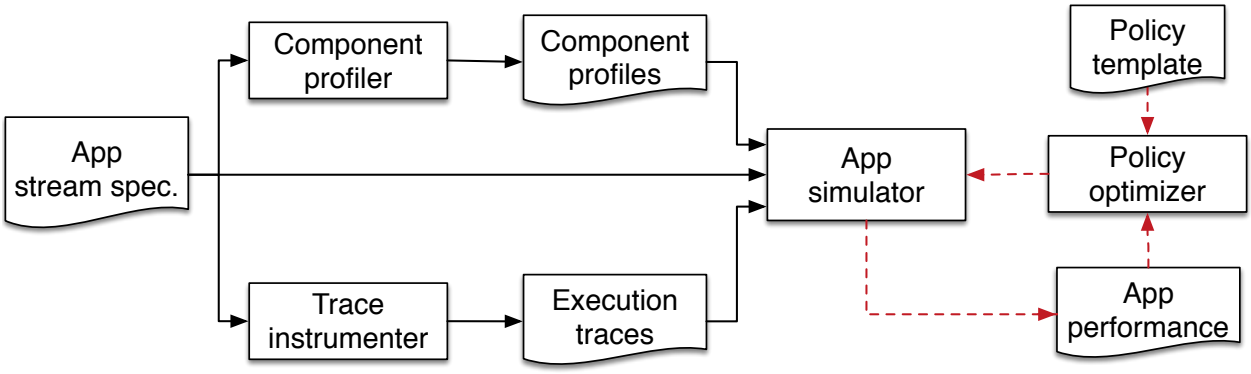


Figure 5.2. Gratis policy evaluation (black solid lines) and configuration (dotted red lines). The *App simulator* determines the performance of an app based on the stream specification, the energy and delay profiles of its components, and a set of execution traces even with dynamic workloads. The *Policy optimizer* determines the values of the policy parameters.

5.3.1 Gratis Programming Model

The Gratis programming model 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.

5.3.1.1 Mobile Sensing Apps as 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 data flow model (SDF) requires that a component produces and consumes the same number of frames in all its executions [12]. Gratis allows components produce and consume a variable amount of data.

The key novelty of Gratis and the aspect where our work departs from SDF is its model of

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

computation (MoC). 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 introduces delays to create workloads that may be processed more efficiently. Additionally, Gratis coordinates multiple hardware resources to achieve the desired energy-delay tradeoff. The execution of a Gratis app is controlled by an explicit PM policy that determines when each component runs.

5.3.1.2 Policy Specification

The policy specification language is built on four constructs: events, event handlers, guarded commands, and an execution 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.

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

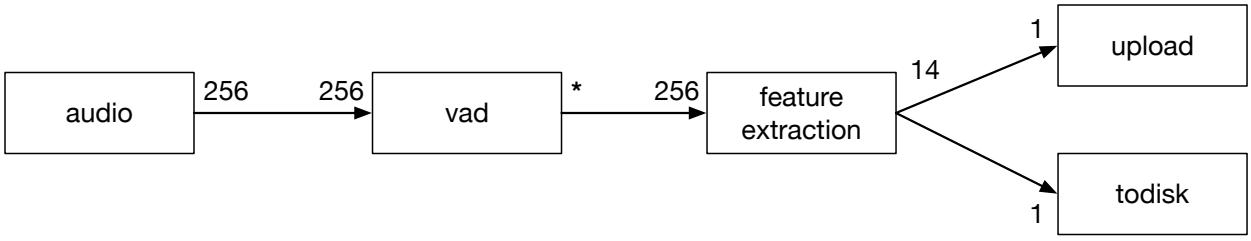
these events by `A` for `d` seconds. This allows the device to sleep. The `execute(A)` command triggers the execution of A when there are sufficient frames in `A`'s input queue to execute at least once. After `A` starts executing, it may execute multiple times until the number of frames in its input 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.

Guards: The guards are boolean 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 and output queues as `input_slack` and `output_slack`. The value of the four properties can be computed efficiently by updating them as frames are inserted or removed from a queue. We limit the complexity of the guard conditions since the PM policy must be sufficiently lightweight to execute efficiently at run-time.

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 `input_timeout(A, t)` when the value of `input_slack` of `A` falls below `t` seconds. The event handler `output_timeout(A, t)` can be used in a similar fashion.

Environment: The app interacts with the PM policy through its execution environment. The 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.

Optimization: A naive approach to implementing PM policies is to have each component



(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`). A "*" indicates that the rates are variable.

```
1:  E = { deadline : 10 min, th_1 : 2, th_2 : 30 sec,
                th_3 : 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 ≥ th_1:
9:          execute(feature_extraction)
10: input_timeout(feature_extraction, th_2):
11:     execute(feature_extraction)
12: post(feature_extraction):
13:     if feature_extraction.num_output ≥ th_3:
14:         execute(upload)
15:         execute(todisk)
```

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

Figure 5.3. The advanced version of the `SI` app that implements batching, scheduled concurrency, and adaptive sensing.

operate in a different thread. This method would incur significant overhead since the app would include many threads whose execution must be synchronized. Additionally, configuring the parameters of these policies would be remarkably time-consuming because of a large parameter space to explore and configure. 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. This constraint ensures that we can control

the hardware resources by starting/stopping the domains. In practice, this approach reduces the complexity of writing and configuring PM policies significantly. For example, the `SI` app evaluated in Section 5.4 has 16 types of component that are instantiated in a stream program that has 1804 components. However, the app has only four domains whose behavior must be coordinated using a PM policy and significantly few parameters that must be configured.

Example: Lets us consider the advanced version of `SI` that uses adaptive sensing (see Figure 5.3a). 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 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.

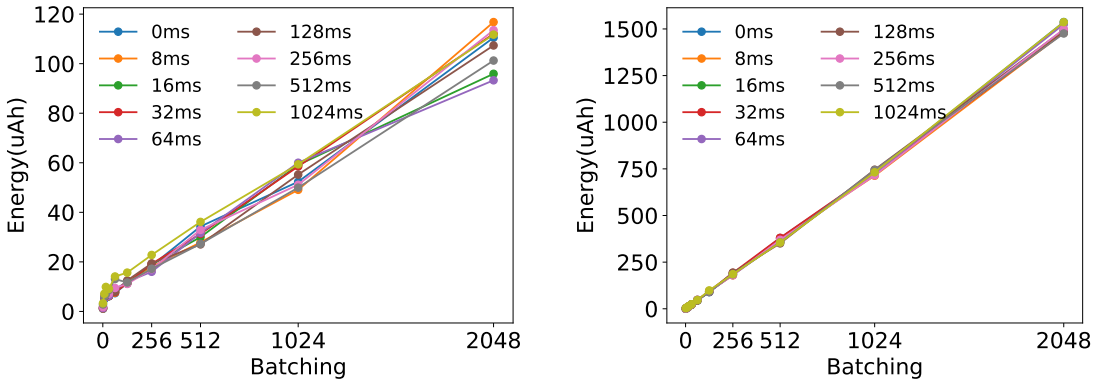
A better approach to handling this situation is to use a `timeout` handler. The timeout handler can be used to trigger the execution of the app based when the slack falls below a threshold. In our example, the execution is triggered when the minimum slack of the frames in the input queue of `feature_extraction` 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.

5.3.2 Evaluating Power Management Policies

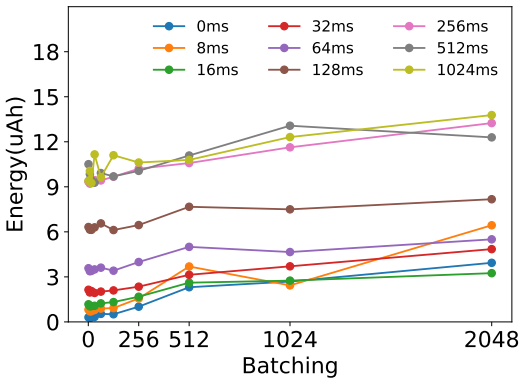
An important feature of the considered energy optimizations is that they have predictable performance. We are interested in developing techniques that assess the performance of a policy in a computationally efficient manner using a small number of measurements. Our solution involves two steps: (1) the domain profiler constructs a performance profile for each domain and (2) the app simulator estimates the overall performance of the app based on the performance profiles and a set of traces. The set of traces is used to simulate dynamic apps and capture the timing of `activity` events. This information is sufficient to replay the behavior of an app deterministically. Additionally, we assume that the MSA works in the background with minimal interference from other apps. Large-scale user studies support this observation [64].

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 for a domain to execute one time. The interim time controls the time between two consecutive batch executions. The values of each input must be selected carefully when profiling dynamic domains. The compiler generates code that tracks the execution time for a domain. We measure the energy consumed using a power meter, though software-based solutions are also possible (e.g., [8, 65, 66]).

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



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



(c) `upload`

Figure 5.4. Energy consumption for a subset of domains of the `SI` app evaluated in Section 5.4.

Figure 5.4b). For domains that use linear-scaling resources, it is sufficient to profile them with different batch sizes using a fixed interim value. Figure 5.4b shows that changes in the interim have little impact on the energy consumed by the `feature_extraction` component. Network interfaces (e.g., Wi-Fi or cellular) are examples of hardware resources whose energy consumption scales non-linearly with the workload. For domains that use non-linear-scaling resources, we profile them with different batching as well as different 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 [65]. For example, the energy consumed by `upload` varies significantly with the interim

time as shown in Figure 5.4c. 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, 67]. 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. A straight-forward approach is to generate the set of potential inputs based on the available execution traces. While this is possible in principle, the set of possible inputs is too large to meet our requirement of using a small number of measurements to generate a profile. An approach to solving this problem is to compute the resource utilization of the domain for each element in the input space. Since each domain uses a single resource, the resource utilization is a scalar. Its value can be estimated by simulating the StreamIt code of that domain with the considered input. The advantage of this approach is that it projects the input space onto a space of manageable size. For example, the only dynamic domain in the `SI` app is the domain that includes the `audio` and the `vad` components. In this case, the input space collapses into two clusters: a cluster where for the case when `vad` detects speech and the other for when it does not detect speech[3]. In the general

[3]In the case of the `SI` app, the domain has similar performance regardless of whether speech is detected or not. This is because the `vad` is lightweight, introducing little processing overhead. The benefit of using adaptive sensing is that for a sample that does not contains speech, `SI` does not extract features or upload them.

case, clustering algorithms may be used to further partition the input space when its size is large. Accordingly, when profiling a dynamic domain, we perform a set of measurements for each input cluster.

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 component given the state of its the input queue and the time from the previous invocation of the domains. 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 5.4c, 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 Figure 5.5). The input to the simulator is the trace of `activity` 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

```
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) = the latency profile 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)
        end
        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
            end
        end
        case post(d): do
            policy_handler(time, event, data, queue)
        end
    end
    sim_execution(ready, time, queue.peek().time, queue)
end
Procedure policy_handler(time, event, data, queue)
    // Execute the instructions of the handler associated with the event
1   for instr in handler(event) do
2       if instr.guard() then
3           queue.schedule(time, pre(instr.target), data)
4       end
5   end
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
6   tick = 5 ms
7   new_event = False
8   while (time ≤ next_event) and (new_event = False) do
9       for resource in ready do
10          d = ready[resource].next_domain()
11          d.duration -= tick
12          d.num_input -= d.input(d.data)
13          d.num_output += d.output(d.data)
14          if domain.duration ≤ 0 then
15              new_event = True
16              ready.remove(d)
17              queue.schedule(time, post(finished), None)
18          end
19      end
20      time = time + tick
21  end
```

Figure 5.5. Pseudo-code for the app simulator whose output is used to assess the energy consumption and end-to-end delay.

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

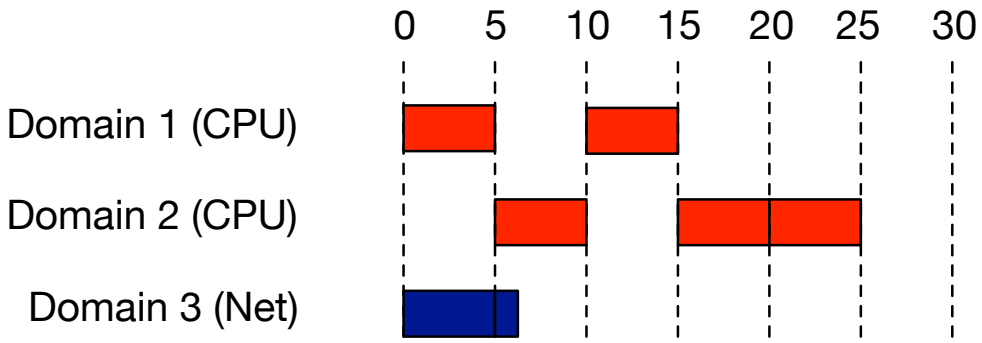


Figure 5.6. Simulation of three domains. Domain 1 and 2 use the CPU and have delays of 10 and 20, respectively. Domain 3 uses the network. Domains 1-2 and domain 3 execute in parallel since they use different hardware resources. Domains 1 and 2 use the CPU fairly.

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 [68]. 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 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 5.6). 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 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.

5.3.3 Configuring and Synthesizing PM Templates

The coordination 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 determines the amount of energy that may be saved. To overcome this challenge, we have developed a tool for automatically configuring a PM template. Moreover, we observed that PM templates have a common structure despite different scheduled concurrency. Based on this observation, we built a tool that automatically synthesizes PM templates. By combining the policy synthesis and configuration tools, we provide developers with an automated mechanism for PM.

Policy Configuration: The input to the configuration tool is a PM template implementing some scheduled concurrency with guards and timeouts as free variables to configure[4]. The configuration tool works as follows. We start by analyzing the template and classifying its parameters as controlling either domain batching or queue timeouts. A batching parameter controls the threshold in the number of frames for a domain to start execution. We can identify the batching parameters

[4]The PM policies in Figures 5.1b and 5.3b can be transformed into templates by removing the associated environment `E` that specifies the policy parameter values.

by inspecting the guards of the `execute` commands. A timeout parameter controls when a domain is executed in a time-driven manner. Timeout parameters are essential for handling the case when the workload is dynamic. We can identify the timeout parameters by inspecting the second argument of the `timeout` handler.

The overall strategy to configure the template parameters is to first configure the batching parameters and then the timeout values. 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 determine its energy consumption and the maximum end-to-end latency using the app simulator. If the maximum end-to-end latency of the configuration is within the end-to-end deadline and the energy consumption is better, 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 timeout parameters. We proceed with this step if the considered configuration has better energy consumption than the current best solution. We iteratively decrease the timeout parameters of the template until it either meets the deadline or its energy consumption becomes worse than the current best solution.

The grid search is usually computationally feasible for apps that have a few domains (as those described in Section 5.4). This is possible because the app simulator, which is invoked to evaluate the performance of each configuration, is highly scalable as shown in Section 5.4.5. 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 batching parameters to the minima with the fixed concurrency. Then, we evaluate the impact of increasing each batching 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 w.r.t. the end-to-end deadline. We repeat the process until there is no more improvement in energy consumption. After the batching and timeout parameters are determined, the other scheduled concurrencies are evaluated for the most energy saving one.

Policy Synthesis: After writing several PM policies, we have observed that the PM templates have a regular structure in spite of different scheduled concurrency. A domain in the stream program triggers the execution of one or more of its descendants when number of frames in its queue exceeds a threshold. The execution of these domains could either be sequential or in parallel. Generating the PM commands to execute domains sequentially or in parallel is straightforward. A timeout handler added to each component may trigger its execution when the slack falls below a threshold. Building on these heuristics, we have developed a tool that can automatically synthesize PM templates for a MSA. The parameters of the template are configured using the policy configuration tool.

5.3.4 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 in a greedy manner. 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 prior to executing the domain, and a `post` event after the execution completes [34, 35, 43].

5.4 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 a broad range of workload shaping policies be specified using Gratis ?
- 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 templates be configured and synthesized effectively and efficiently?

5.4.1 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.

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) [69] from the collected frames. MFCCs have been extensively used for speaker identification and speech recognition on mobile phones [26, 40, 61]. 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 an audio frame contains speech. The voice activity detector is based on the algorithm proposed by Moattar et al. [70]. 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

App	Component Types	Components	Domains	StreamIt Code Size	Java Code Size
SI	13	1,306	4	409	21,631
AR	16	1,804	4	551	19,984

Table 5.1. App size statistics.

several long audio recordings. The audio was collected in the homes of older adults as part of a previous study[5]. 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 [71]. 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 assuming a capacity of 3000 mAh.

5.4.2 Gratis Simplifies Power Management

The Gratis programming model provides mechanisms for specifying the structure of an app and its PM policy. Adopting stream programs as a high-level abstraction significantly simplified the implementation of MSAs: a developer can focus on the signal processing and machine learning algorithms without having to consider the details of using Android. The statistics shown in Table

[5]A reference to the study is omitted due to the double-blind requirement.

5.1 show that MSAs can be implemented as small stream programs. Our source-to-source compiler transforms them into optimized Android code. The majority of the Android code is implemented in the Gratis runtime library that is common to all apps.

The main benefit of Gratis is that it simplifies the integration of workload shaping optimizations in MSAs. For each app, we have developed several PM policies that combine batching, scheduled concurrency, and adaptive sensing. This demonstrates that our coordination language can express a broad range of workload shaping optimizations. Manually writing PM policies for `SI` and `AR` is relatively straightforward. This is primarily because the `SI` and `AR` apps have only four domains whose operations must be coordinated. The complexity of writing PM policies increases with the number of domains that must be coordinated. However, results detailed later in the section show that PM policies with varying scheduled concurrency and batching yield little energy improvement compared with those including just batching. Thus, a pragmatic approach is to always execute domains sequentially, significantly reducing the complexity of writing policies.

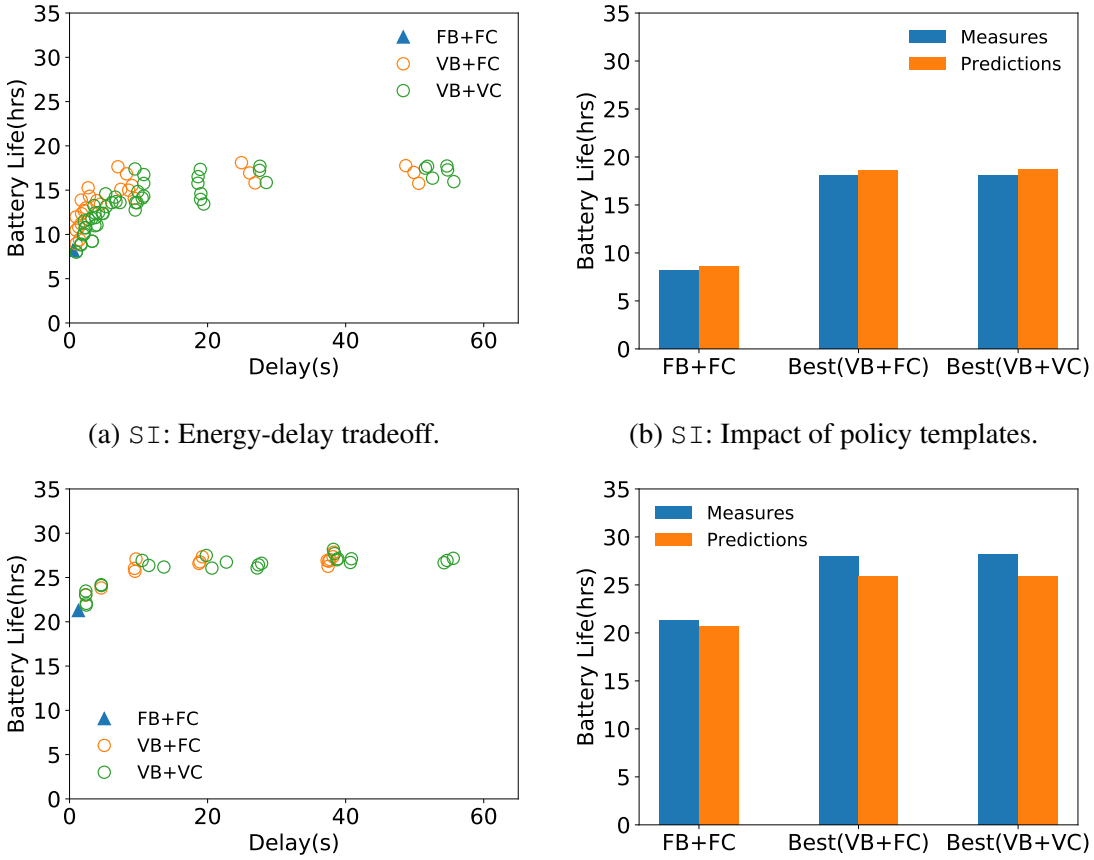
An alternative that is most suitable for developers with minimal PM knowledge is to use our automated PM template synthesis and configuration tools. The template synthesis tool generates PM templates that combine batching, scheduled concurrency, and adaptive sensing. The templates are generated in a brute-force manner by enumerating all the possible configurations where domains can be executed sequentially or in parallel. The policy configuration tool is then used to configure the parameters controlling the batching and timeouts. When the app uses adaptive sensing, we require the developer to specify how long the app should sleep after detecting no activity. Input from the developer is required because this parameter depends on the nature of the app and may have a significant impact on its sensing accuracy. The results presented in this section have been obtained using policies generated in this manner.

5.4.3 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 templates that we instantiated with different parameter values. The policy template `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 in a naive manner. Fixed concurrency indicates that the domains execute one after another as soon as possible without seeking to overlap hardware access.

This baseline shows the performance a naive execution of a stream program would have. The policy template `VB + FC` uses batching but fixes concurrency as described above. We have evaluated the template by configuring each domains 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 templates overlap sensing with feature extraction, or feature extraction with network upload. We use a total of 8 templates with different scheduled concurrency.

Figures 5.7a and 5.7c 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 a 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,



(a) SI: Energy-delay tradeoff.

(b) SI: Impact of policy templates.

(c) AR: Energy-delay tradeoff.

(d) AR: Impact of policy templates.

Figure 5.7. The energy-delay trade-off for SI and AR when using *static sensing*. Batching significantly improves energy efficiency. Combining batching with scheduled concurrency provides no additional benefit.

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 5.7b and 5.7d 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.

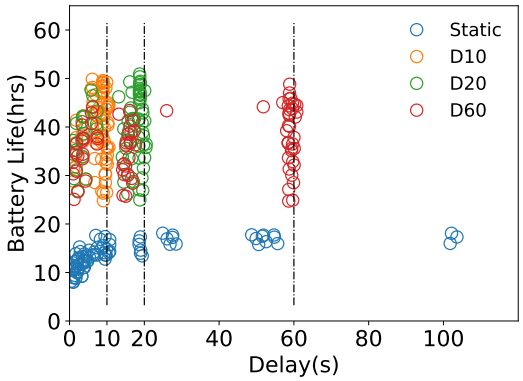
Figures 5.8a and 5.8c show the energy-delay trade-off when adaptive sensing is used. For comparison, we include the static sensing data from the previous experiment. In contrast to the static sensing case, most of the configurations are concentrated around the deadline given sufficient batching. This 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 5.8b and 5.8d 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, different scheduled concurrency may provides 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, combined scheduled concurrency can extend the battery life by additional 8 hours over the best batching. AR has a similar behavior to SI with smaller improvement of scheduled concurrency that extends the battery life by 2 hours given a deadline of 20 seconds.

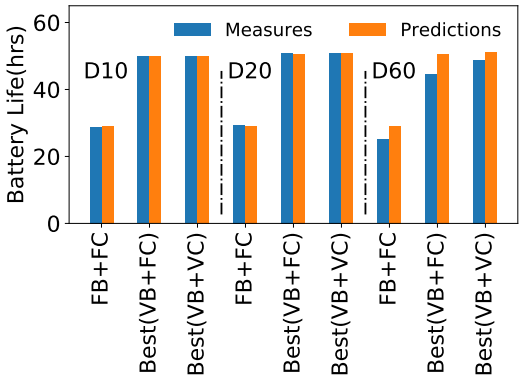
5.4.4 Gratis Apps Have Composable Performance

We have evaluated the accuracy of the app simulator by comparing the difference between the predicted and measured performance. Figure 5.9 plots the accuracy of the predictions for the considered apps. Overall, the average error for energy predictions is 7%. The error for latency is 15% on average. 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.

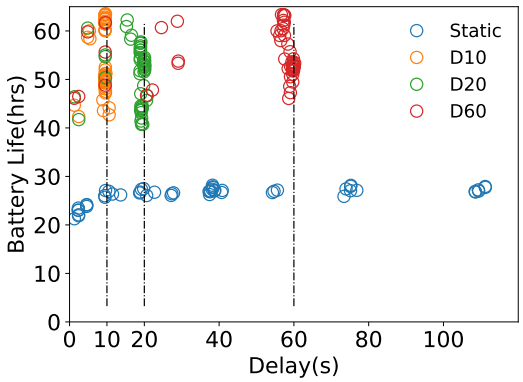
We remark that the overall performance accuracy is slightly better in the static sensing case than adaptive sensing. This is not surprising since in the adaptive case, we also have to cope with variations in user input. Perhaps more interestingly, there is significant dependency between the errors and the deadlines when adaptive sensing is used. The reason for this is because when



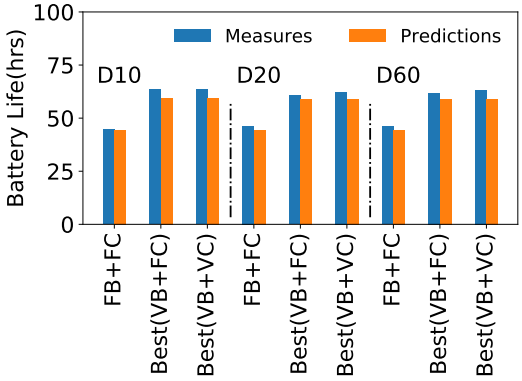
(a) SI: Energy-delay tradeoff.



(b) SI: Impact of policy templates.



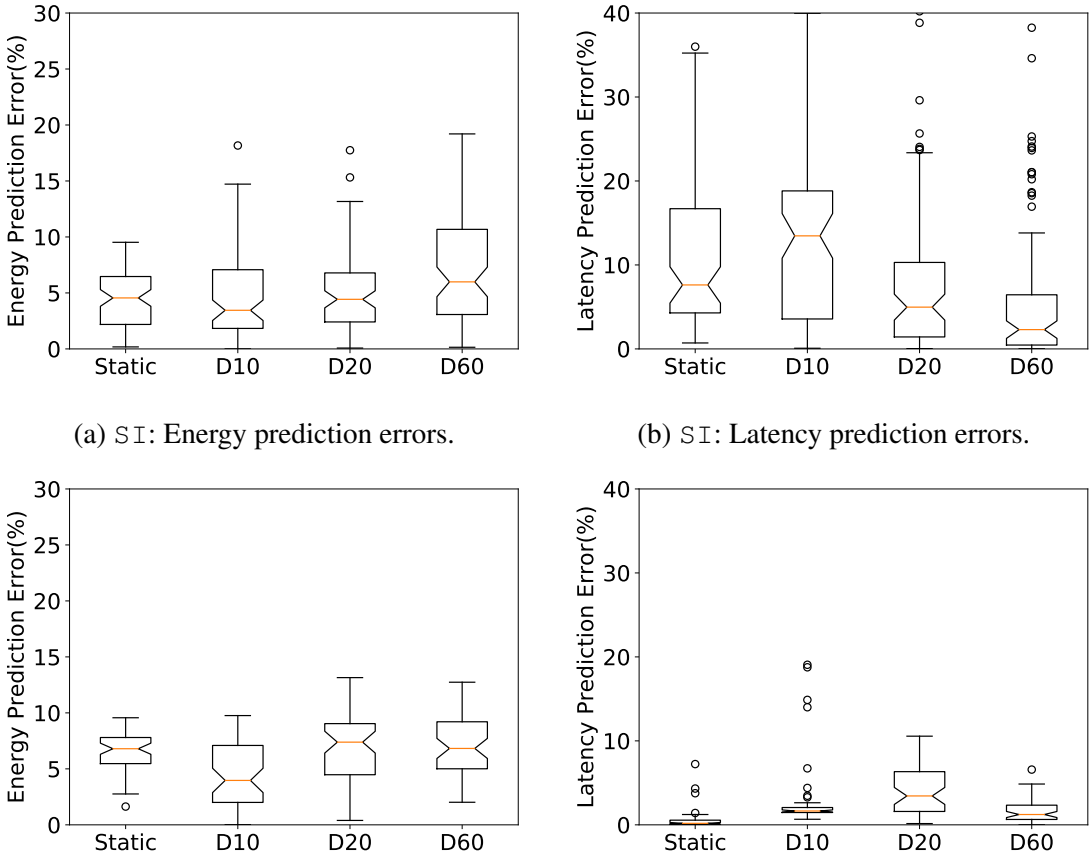
(c) AR: Energy-delay tradeoff.



(d) AR:Impact of policy templates.

Figure 5.8. 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.

deadlines are tight the fraction of 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.



(a) SI: Energy prediction errors.

(b) SI: Latency prediction errors.

(c) AR: Energy prediction errors.

(d) AR: Latency prediction errors.

Figure 5.9. Accuracy of energy and delay predictions for the app simulator.

5.4.5 Gratis App Simulator is Scalable

A key ingredient to the effectiveness of the template configuration and synthesis tools 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 5.10 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 trace length. The simulator can process traces that are several hour long in a few seconds.

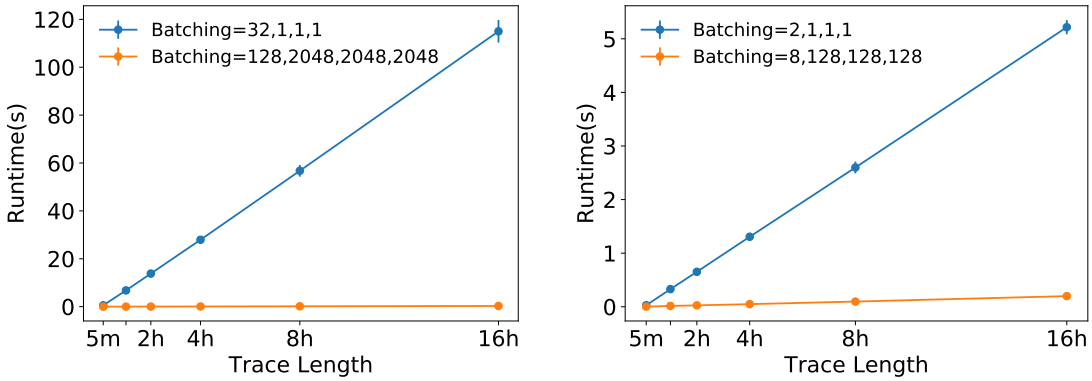
App	Search Method	Best(VB + FC)	Best(VB + VC)
SI	Grid	14717 μAh / 254 s	14717 μAh / 7166 s
	Gradient	14952 μAh / 56 s	14952 μAh / 66 s
AR	Grid	220 mAh / 183 s	220 mAh / 2022 s
	Gradient	221 mAh / 78 s	221 mAh / 88 s

Table 5.2. Policy synthesis and configuration for SI using a 10-minute trace and AR using a 5-hour trace. Policies combine adaptive sensing, batching, and controlled scheduling with a deadline of 60 seconds.

Table 5.2 shows the results of configuring the PM policies for SI and AR using adaptive sensing, batching, and controlled scheduling. The synthesis and simulation tools 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. Synthesizing and configuring all the templates 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, synthesizing and configuring the templates 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 automatically generate PM policies and that the gradient descent search method provides an effective approach to reducing synthesis and configuration time.

5.5 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



(a) SI simulation performance.

(b) AR simulation performance.

Figure 5.10. Simulation time for simulating SI and AR with adaptive sensing.

algorithms used to make inferences. For example, Kobe constructs offline an efficient sensing pipelines by optimizing the features and classifiers that are used [31]. Similarly, Orchestrator constructs multiple variants of a sensing pipeline and dynamically switches between these variants at run-time [29]. 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 [72]. In this chapter, 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 [73]. 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 [74]. LAB provides the APIs that allow developers to switch between multiple implementations of a sensing algorithm to save run-time energy [75]. Closer to our work, Tempus uses annotations that control when power-hungry operations are invoked [76]. 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 [77] and StreaMorph [78] 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 a MSA component produces only a small amount of data that cannot be processed efficiently. Gratis provides a flexible and general mechanism for specifying 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 [27] is a stream execution engine that focuses on handling overload conditions due to interfering applications. Gratis on the contrary 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 further reduce energy consumption.

The challenge to assessing the energy consumption and delay of computing programs has been studied extensively. Existing solutions span the trade-off between accuracy and computational/monitoring overhead. Emulators such as GEM5 [79] can provide accurate estimates of delays and energy consumption of mobile apps in an offline manner. Unfortunately, emulators typically have high processing demands requiring hours of emulation to estimate the performance of even simple apps. They are insufficient for our needs as we are interested in quickly determining the impact of a PM policy and optimizing its parameters. Several lightweight tools have been proposed to evaluate the energy consumption of apps at run-time. PowerTutor can generate power models for different components of a mobile device based on the battery drain behavior [8]. Eprof is another power estimation tool that captures and accounts for the power usage of an application through tracing system calls made by the program when it runs on a smartphone [65]. WattsOn is an extension to Eprof that attempts to provide finer grained details regarding where energy is consumed within an app [66]. In contrast to these works, BatteryExtender is an effort to allow users to extend the phone battery life for a specific duration to accomplish a particular task [80]. Gratis focuses on evaluating the energy consumption and delay from a small set of measurements. Our technique takes advantage of the structure of stream programs and the fact that the PM policy controls explicitly the scheduled concurrency to derive estimates of energy and delay in a constructive manner based on individual domain profiles to compose and predict the overall performance of the app. It is important to note that the above tools primarily focus on predicting energy consumption through run-time profiling. Instead, Gratis advocates fast compile-time synthesis and configuration of PM policies to save energy w.r.t. deadline constraints.

5.6 Conclusions

Gratis is a novel paradigm for incorporating workload shaping energy optimizations with predictable performance of MSAs. We adopt the stream programming model as a high-level abstraction for reasoning about the impact that energy optimizations have on the energy and the delay of an app. Gratis departs from the traditional SDF in which components are executed as soon as possible by using a PM policy that introduces artificial delay to generate workloads that may be processed in a more energy efficient way. We control the impact of the introduced delays on the user experience by specifying soft end-to-end deadlines. We presented a coordination language that can express a broad range of workload shaping energy optimizations including those for batching, scheduled concurrency, and adaptive sensing. We have developed an app simulator that can estimate the performance of a PM policy in a computationally efficient manner from a small number of measurements at compile time. The efficiency of the simulator is a consequence of the Gratis apps having composable performance, which is a unique feature of our programming model. The simulator is the basis for tools that configure and synthesize PM templates automatically. In combination, these tools provide a developer with an automated approach to incorporating workload shaping policies into MSAs and require minimal PM expertise.

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 app simulator. We have developed a simulator that can predict the energy and delay with average errors of 7% and 15% respectively even when applications have variable workloads. The simulator is highly scalable in simulating short traces in less than a second and long traces of sixteen hours under 2 minutes. These results demonstrate that it is feasible to accurately estimate the performance of MSA PM policies at compile time.

CHAPTER 6

CONCLUSIONS & FUTURE WORK

This dissertation illustrates the progress of stream processing optimizations for MSAs from the perspective of memory management and energy efficiency. Chapter 2 gives the theorectical background of SDF MoC that allows stream compilers to validate the composition of stream programs at compile time. An automated approach to ensuring long term robustness of high-rate MSAs is demonstrated in chapter 3. Common application composition and programming errors such as leaks and races can be checked by the stream compiler in ahead. Platform-specific PM and foreign code integration to ease the development of MSAs are also supported. Chapter 4 describes the static analysis to capture the whole program memory behavior based on abstract interpretation and the efficient memory layout generation. The stream memory operations may be rewritten by the compiler for efficient access to the generated layout. Energy efficiency improvement for MSAs in terms of batching and deadline constraints is discussed in Chapter 5. We characterized the composable energy behavior of MSAs by building separate domain profiles and an app simulator. The policy space exploration for the most energy saving configuration can be done offline and save MSA development cycles accordingly.

6.1 Future Work

I believe that the work developed as part of this thesis can serve as the basis of significant future research in relevant areas with MoC in dataflow. There are several directions to extend in the context of real user interactions, deep learning and cloud computing as follows:

- Our work so far is done analytically without reactions to real user activities and intentions at

runtime. There are use cases where the runtime user activity predictions may further save the energy consumption adaptively. For example of online photo viewing. Prefectching photos generally shortens the tail energy consumption of network components if the photos are of the user interest. Otherwise, excessive network transmissions waste energy when the user quickly browses through photos of less interest. In this case, stream programs may react to the user activities with dynamic batching between SFG partitions. While stream programs are typically optimized at compile time, appropriate repartitioning allows for making dynamic decisions to improve energy efficiency at runtime. Ultimately, a reinforcement learning algorithm may be developed to adaptively synthesizes PM policies for aggregation of energy intensive operations w.r.t. deadline constraints.

- We have shown the runtime energy delay trade-off can be optimized through offline parameter space exploration. In view of similar dataflow modeling of deep networks, a similar trade-off between the accuracy and deep learning hyperparameters deserves further investigation for data scientists to make effective design decisions.

- Though consistency and fault-tolerance in cloud computing seem to serve as separate concerns independent of existing stream optimizations, there might be inter-dependencies worth explorations. For instance, super fine-grained partitioning and placement probably induce increasing overhead to maintain consistency and high availability such that the service level agreement (SLA) might not hold. An analytical model incorporated into stream compilers would be useful to combine the best of both worlds.

REFERENCES

[1] H. Lu, A. J. B. Brush, B. Priyantha, A. K. Karlson, and J. Liu, "SpeakerSense : Energy Efficient Unobtrusive Speaker Identification on Mobile Phones," in *Pervasive Computing*, vol. 6696, 2011, pp. 188–205.

[2] N. Nikzad, N. Verma, C. Ziftci, E. Bales, Q. Nichole, Z. Piero, K. Patrick, S. Dasgupta, I. Krueger, T. S. Rosing, and W. G. Griswold, "CitiSense: Improving Geospatial Environmental Assessment of Air Quality Using a Wireless Personal Exposure Monitoring System," in *Wireless Health*, 2012, pp. 1–8.

[3] C. Peng, G. Shen, Y. Zhang, Y. Li, and K. Tan, "BeepBeep: A High Accuracy Acoustic Ranging System using COTS Mobile Devices," in *SenSys*, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1322265

[4] P. L. Schneider, S. E. Crouter, and D. R. Bassett, "Pedometer Measures of Free-Living Physical Activity: Comparison of 13 Models," *Medicine and Science in Sports and Exercise*, vol. 36, no. 2, pp. 331–335, 2004.

[5] D. A. Reynolds and R. C. Rose, "Robust Text-Independent Speaker Identification Using Gaussian Mixture Speaker Models," *IEEE Transactions on Speech and Audio Processing*, vol. 3, no. 1, pp. 72–83, 1995. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=365379

[6] S. Hasan, F. Lai, O. Chipara, and Y.-H. Wu, "AudioSense: Enabling real-time evaluation of hearing aid technology in-situ," in *Proceedings of CBMS 2013 - 26th IEEE International Symposium on Computer-Based Medical Systems*, 2013.

[7] Y. C. Hu and S. P. Midkiff, "What is keeping my phone awake? Characterizing and Detecting No-Sleep Energy Bugs in," *MobiSys*, pp. 267–280, 2012.

[8] L. Zhang, B. Tiwana, Z. Qian, and Z. Wang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *CODES+ISSS*, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1878982

[9] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *MobiSys*. New York, New York, USA: ACM Press, jun 2011, p. 335. [Online]. Available: http://dl.acm.org/citation.cfm?id=1999995.2000027

[10] F. Lai, D. Schmidt, and O. Chipara, "Static Memory Management for Efficient Mobile Sensing Applications," in *EMSOFT*, 2015.

[11] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, pp. 471–475, 1974. [Online]. Available: http://www.citeulike.org/group/872/article/349829

[12] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=1458143

[13] T. M. Parks, "Bounded Scheduling of Process Networks," Ph.D. dissertation, 1995.

[14] M. Geilen and T. Basten, "Requirements on the Execution of Kahn Process Networks," in *ESOP*, ser. ESOP'03. Springer, 2003, pp. 319–334. [Online]. Available: http://dl.acm.org/citation.cfm?id=1765712.1765736

[15] J. Buck and E. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," in *ICASSP*, vol. 1, 1993, pp. 429–432.

[16] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *NEPLS*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed., vol. LNCS 2304, no. Lecture Notes in Computer Science, Massachusetts Institute of Technology. Springer-Verlag, 2002, pp. 179–196. [Online]. Available: http://www.springerlink.com/index/LC5B77HWR8J2UBHK.pdf

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language Lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[18] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, jan 1987.

[19] R. L. Graham, "Optimization and Approximation in Deterministic Sequencing and Scheduling-A Survey," *Annals of Discrete Mathematics*, pp. 287–326, 1979.

[20] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," pp. 841–848, 1961.

[21] S. Consolvo, D. W. McDonald, T. Toscos, M. Y. Chen, J. Froehlich, B. Harrison, P. Klasnja, A. LaMarca, L. LeGrand, and R. Libby, "Activity sensing in the wild: a field trial of ubifit garden," in *SIGCHI*, 2008.

[22] L. Bao and S. S. Intille, "Activity Recognition from User-Annotated Acceleration Data," in *Pervasive Computing*, A. Ferscha and F. Mattern, Eds., vol. 3001. Springer, 2004, pp. 1–17. [Online]. Available: http://www.springerlink.com/content/9aqflyk4f47khyjd

[23] O. Chipara, C. Lu, T. C. Bailey, and G.-C. Roman, "Reliable clinical monitoring using wireless sensor networks: experiences in a step-down hospital unit," in *SenSys*, 2010.

[24] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[25] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *MobiSys*, 2011.

[26] 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.

[27] 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.

[28] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song, "SeeMon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments," *MobiSys*, 2008.

[29] 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 *SenSys*, ser. SenSys '10, J. L. Rosenfeld, Ed., vol. 18, no. 7, MIT. New York, NY, USA: ACM, jan 2008, pp. 267–280. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4497527http://www.citeulike.org/group/872/article/349829http://portal.acm.org/citation.cfm?id=781133http://portal.acm.org/citation.cfm?id=1869992http://portal.acm.org/citation.cfm?doid=3548

[30] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI : Making Smartphones Last Longer with Code Offload," in *MobiSys*, ser. MobiSys '10. ACM, 2010, pp. 49–62. [Online]. Available: http://research.microsoft.com/en-us/um/people/ssaroiu/publications/mobisys/2010/maui.pdf

[31] 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*. New York, New York, USA: ACM Press, 2011, p. 54. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2070942.2070949

[32] S. Nirjon, R. F. Dickerson, P. Asare, Q. Li, D. Hong, J. A. Stankovic, P. Hu, G. Shen, and X. Jiang, "Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones," in *MobiSys*, 2013.

[33] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541, 1997.

[34] 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 *LCTES*, ser. LCTES '08, vol. 43, no. 7. ACM Press, 2008, p. 131. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1379023.1375675

[35] L. Girod, Y. M. Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "XStream: a Signal-Oriented Data Stream Management System," in *ICDE*, ser. SenSys '06, vol. 00, Citeseer. Ieee, 2008, pp. 1180–1189. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4497527

[36] ESC/Java2, "http://goo.gl/1wBvvA."

[37] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for {Java}," in *ISSTA*, 2008.

[38] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *PLDI*, ser. PLDI '03, vol. 38, no. 5. ACM, 2003, pp. 1–11. [Online]. Available: http://portal.acm.org/citation.cfm?id=781133

[39] C. Xu, S. Li, G. Liu, Y. Zhang, E. Miluzzo, Y.-F. Chen, J. Li, and B. Firner, "Crowd++: Unsupervised Speaker Count with Smartphones," in *Ubicomp*, 2013.

[40] 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 *MobiSys*, ser. MobiSys '10, vol. 14, no. 2. ACM, 2010, pp. 5–20. [Online]. Available: http://portal.acm.org/citation.cfm?id=1814437

[41] M. Lin, N. D. Lane, M. Mohammod, X. Yang, H. Lu, G. Cardone, S. Ali, A. Doryab, E. Berke, A. T. Campbell, and Others, "BeWell+: multi-dimensional wellbeing monitoring with community-guided user feedback and energy optimization," in *Wireless Health*, 2012.

[42] N. Nikzad, N. Verma, C. Ziftci, E. Bales, N. Quick, P. Zappi, K. Patrick, S. Dasgupta, I. Krueger, T. Š. Rosing, and W. G. Griswold, "CitiSense: Improving Geospatial Environmental Assessment of Air Quality Using a Wireless Personal Exposure Monitoring System," in *Wireless Health*, 2012.

[43] F. Lai, S. S. Hasan, A. Laugesen, and O. Chipara, "CSense: A stream-processing toolkit for robust and high-rate mobile sensing applications," in *IPSN*, 2014.

[44] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe, "Cache Aware Optimization of Stream Programs," in *LCTES*, ser. LCTES '05, vol. 40, no. 7. ACM, 2005, p. 115. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1070891.1065927

[45] S. S. Bhattacharyya and E. a. Lee, "Scheduling Synchronous Dataflow Graphs for Efficient Looping," *Journal of VLSI Signal Processing*, vol. 6, no. 3, pp. 271–288, 1993.

[46] M. Karczmarek, W. Thies, and S. Amarasinghe, "Phased Scheduling of Stream Programs," in *LCTES*, 2003.

[47] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *POPL*, 1977.

[48] M. Daumas, D. Lester, and C. Muoz, "Verified Real Number Calculations: A Library for Interval Arithmetic," *Computers, IEEE Transactions on*, vol. 58, no. 2, pp. 226–237, feb 2009.

[49] P. Cousot and R. Cousot, "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *POPL*, 1977, pp. 238–252. [Online]. Available: http://dl.acm.org/citation.cfm?id=512973{%}5Cnhttp://portal.acm.org/citation.cfm?doid=512950.512973

[50] S. Bhattacharyya and E. Lee, "Memory Management for Dataflow Programming of Multirate Signal Processing Algorithms," *IEEE Transactions on Signal Processing*, vol. 42, no. 5, pp. 1190–1201, may 1994. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=295199

[51] P. Hudak and A. Bloss, "The aggregate update problem in functional programming systems," in *POPL '85*, 1985, pp. 300–314.

[52] P. Schnorf, M. Ganapathi, and J. L. Hennessy, "Compile-time copy elimination," *Software: Practice and Experience*, vol. 23, no. 11, pp. 1175–1200, 1993.

[53] M. Odersky, "How to Make Destructive Updates Less Destructive," in *POPL*, 1991.

[54] S. Abu-Mahmeed, C. Mccosh, Z. Budimli, K. Kennedy, K. Ravindran, K. Hogan, P. Austin, S. Rogers, and J. Kornerup, "Scheduling Tasks to Maximize Usage of Aggregate Variables in Place," in *CC*, 2009, pp. 204–219.

[55] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet, "A modular memory optimization for synchronous data-flow languages," *LCTES*, 2012.

[56] S. Kohli, "Cache aware scheduling for synchronous dataflow programs," UC Berkeley, Tech. Rep., 2004.

[57] E. A. Lee., "Overview of the Ptolemy Project," UC Berkeley, Tech. Rep., 2003.

[58] A. A. Lamb, W. Thies, and S. Amarasinghe, "Linear Analysis and Optimization of Stream Programs," in *PLDI*, 2003.

[59] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, "Streamflex: High-throughput Stream Programming in Java," *SIGPLAN Not.*, 2007.

[60] (2017) Moves Activity Diary. [Online]. Available: https://www.moves-app.com/

[61] 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 *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 2013, pp. 375–388.

[62] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong, "Reflex: using low-power processors in smartphones without knowing them," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 13–24, 2012.

[63] C.-C. Han, M. Goraczko, J. Helander, J. Liu, B. Priyantha, and F. Zhao, "CoMOS: An operating system for heterogeneous multi-processor sensor devices," *Res. Tech. Rep. MSR-TR-2006-177. Microsoft Research*, 2006.

[64] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "LiveLab: measuring wireless networks and smartphone users in the field," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, pp. 15–20, 2011.

[65] 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 *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 29–42.

[66] R. Mittal, A. Kansal, and R. Chandra, "Empowering developers to estimate app energy consumption," in *Proceedings of the 18th annual international conference on Mobile computing and networking*. ACM, 2012, pp. 317–328.

[67] W. Thies and S. Amarasinghe, "An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design," in *PACT*, ser. PACT '10, ACM. ACM Press, 2010, p. 365. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1854273.1854319

[68] I. Molnar, "Modular Scheduler Core and Completely Fair Scheduler," \url{https://lwn.net/Articles/230501/}, 2007.

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

[70] M. Moattar and M. Homayounpour, "A Simple but Efficient Real-Time Voice Activity Detection Algorithm," in *EURASIP*, no. Eusipco, 2009, pp. 2549–2553. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.176.6740{&}rep=rep1{&}type=pdf

[71] "Monsoon Solutions - Power Monitor." [Online]. Available: http://msoon.com/LabEquipment/PowerMonitor

[72] S. Nath, "Ace: exploiting correlation for energy-efficient and continuous context sensing," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. ACM, 2012, pp. 29–42.

[73] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu, "Energy Types," in *OOPSLA*, vol. 47, no. 10, nov 2012, p. 831. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2398857.2384676

[74] 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 *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 164–174.

[75] A. Kansal, S. Saponas, A. J. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola, "The latency, accuracy, and battery (LAB) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 661–676, 2013.

[76] N. Nikzad, M. Radi, O. Chipara, and W. G. Griswold, "Managing the energy-delay tradeoff in mobile applications with tempus," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 259–270.

[77] T. W. Bartenstein and Y. D. Liu, "Green streams for data-intensive software," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 532–541.

[78] D. Bui and E. a. Lee, "StreaMorph: A Case for Synthesizing Energy-Efficient Adaptive Programs Using High-Level Abstractions," in *EMSOFT*, 2013.

[79] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and Others, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[80] G. Metri, W. Shi, M. Brockmeyer, and A. Agrawal, "BatteryExtender : An Adaptive User-Guided Tool for Power Management of Mobile Devices," in *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2014, pp. 33–43.