Exploring the Use of Purely Functional Programming Languages for Offloading of Mobile Computations

By Christian Kjær Laustsen (s124324)

Outline

- Offloading
- Approaches
- Evaluation
- Discussion & Conclusion

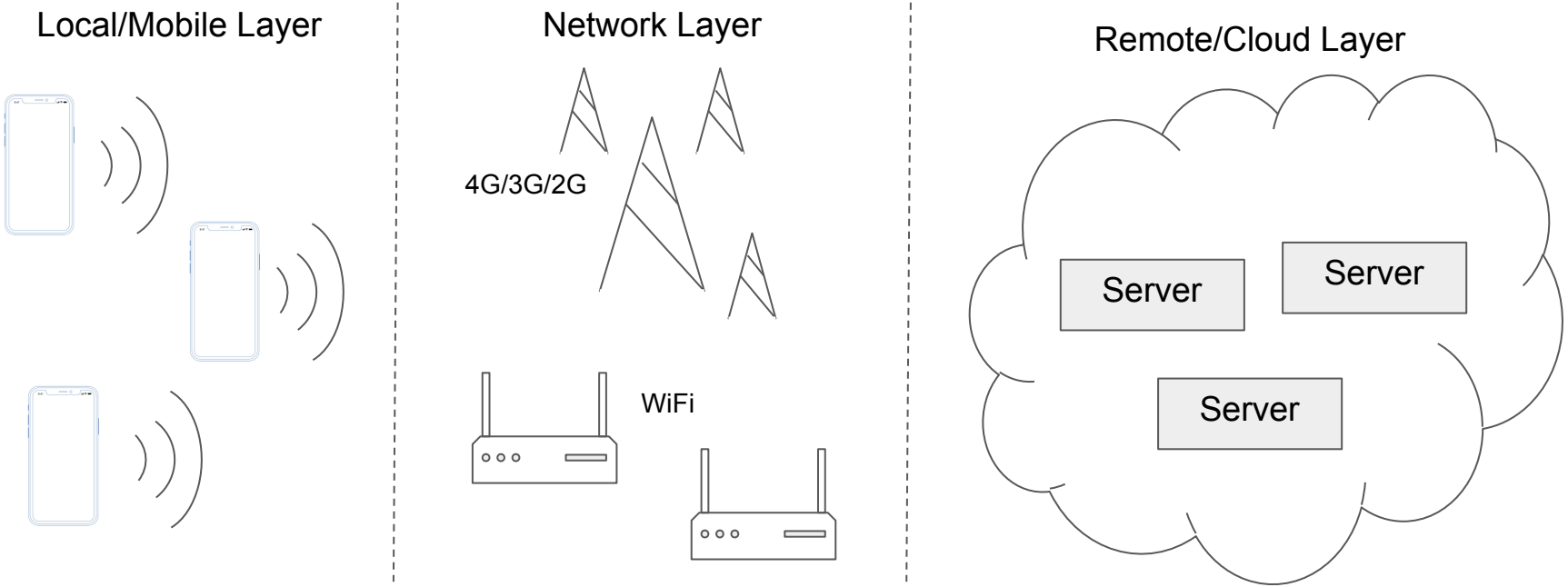
Offloading

Motivation

- Consumers expect
  - more and more performance from their devices,
  - a small/thin device size,
  - the battery capacity to improve (or at least not regress),
- Why is this a problem?
  - Performance improvements have gone far faster than battery improvements.
  - Form factor (small device) limits size of components, especially battery
- What if we could save energy and increase performance by running our code elsewhere?

Offloading

- What are we working with?



Offloading

- Network Layer means we care about
  - Latency, throughput/bandwidth, packet loss
  - Often tied to network type (2G, 3G, 4G, WiFi)
- Mobile Layer means we care about
  - Execution time, battery drain
- Cloud Layer means we care about
  - Round-trip times to remote location

Previous Solutions

- Most of the previous solutions **wrestle with handling state**
  - MAUI and CloneCloud traverse and transfer the state upon offloading
  - UpShift keeps the state in sync, unidirectionally
  - MobiCOP requires the programmer to supply the state
- All feature **profilers/decision-engines**
  - Most are dynamic at runtime and learn, but possible to do it statically (CloneCloud)
- **Adoption is still very low**

Pure FP Languages

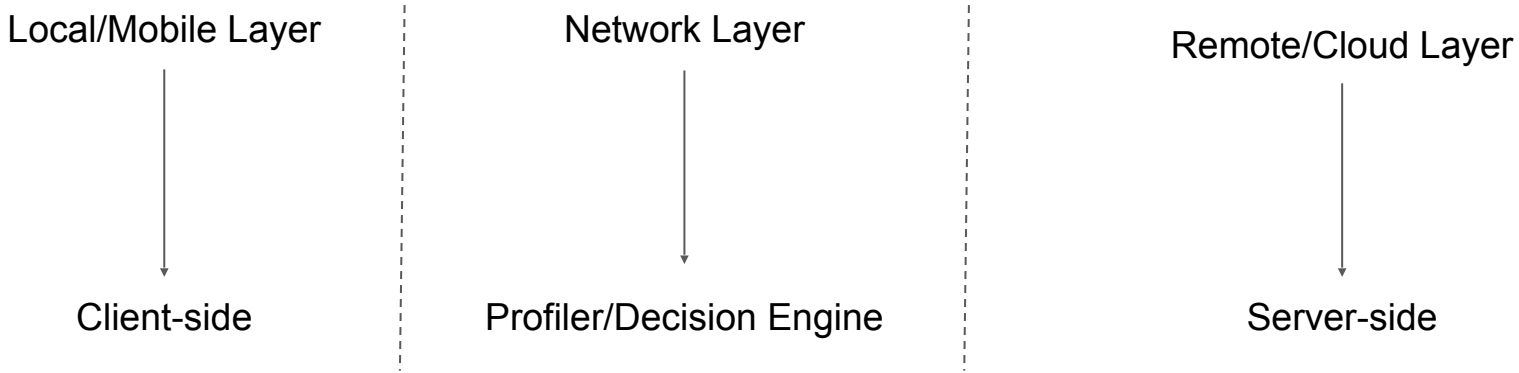
- Purity and pure functional programming languages can help us alleviate the problem with state, and maybe adoptability too

Let us explore how we could go about this!

Approaches

Layers → Program

- We need to consider all three layers
- Each layer corresponds to a part of the program



Approaches

Approaches

- Explore several solutions to the problem space
- Considerations (learned from previous literature and solutions)
  - **Input/output size** matters
  - **Buy-in**, **granularity** and **portability** matters to **adoptability**
  - Traversing and handling **state is a complex issue**
  - **Network conditions** can change frequently
    - You probably don’t want to offload on **2G**
    - On **3G** you strongly need to consider input/output size
    - **4G** and **WiFi** have favourable conditions for offloading

Evaluating Approaches

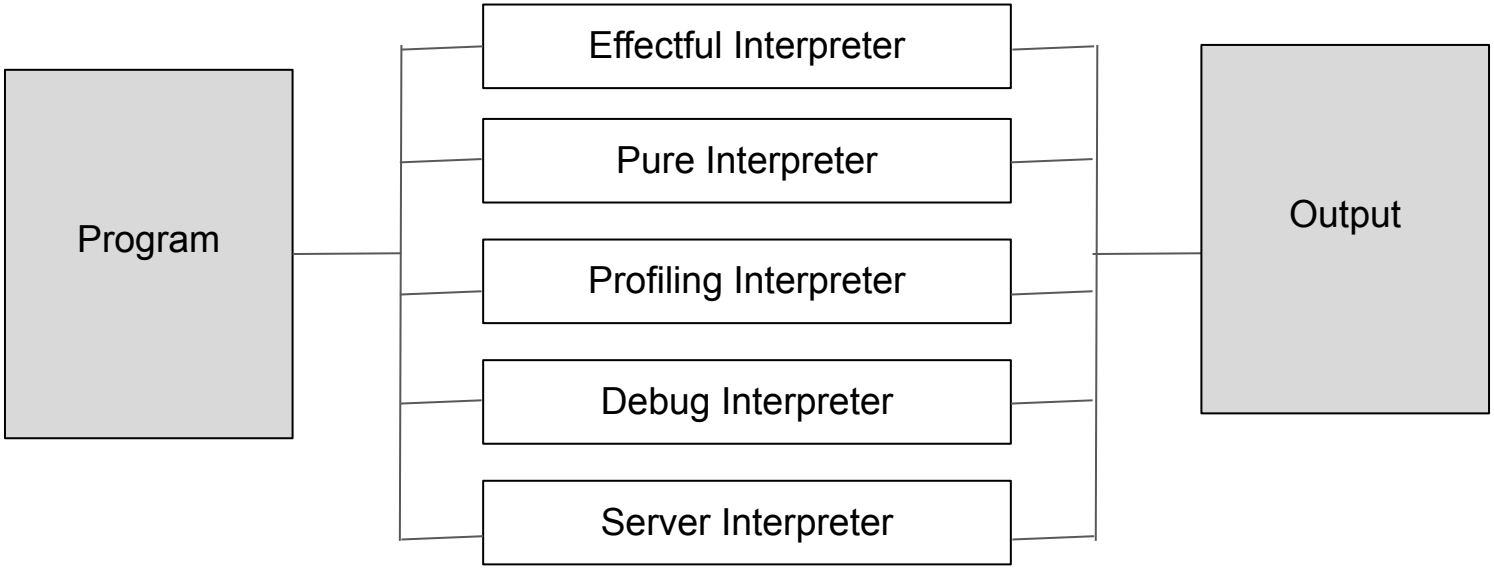
Approach	Complexity	Adoptability	Buy-in	Granularity	Server-side story	Portability
Extending the runtime	Very High	Low	High	Very Coarse	No	No
unsafePerformIO	Low	Mid	Low	Very Fine	No	Yes
Rewrite Rules	Low	Low	Low	Very Fine	No	No
Monadic Framework	Mid	High	Mid	Flexible	Yes	Yes
Template Haskell	High	Mid	Low	Very Fine	Yes	No

Why go with Monadic Framework?

- The Free/Freer-style is already being used for general code structure
  - MTL has the drawback of allowing functions defined outside of the set of operations
- It has a fairly low complexity and retains portability across other pure FP languages
- One of the few that presents a server-side story
- The granularity is very flexible
- Provides the cleanest separation between program semantics and implementation

Why go with Monadic Framework?

- We can easily write a effectful interpreter, pure interpreter, testing interpreter, server interpreter, etc



Evaluation

KAIST DTU

Evaluation

Criteria For Success

Three main criteria we will evaluate our approach on

1. Lower Energy Consumption
    a. Less battery drained
2. More performance
    a. Faster execution
3. Complexity of testing 1. and 2.
    a. Who much work is involved in evaluation the effectiveness

Testing Setup

- Construct two interpreters on the client
  - One that offloads
  - One that runs locally
- Run each interpreter while gathering diagnostics (over WiFi)
- Compare results

Testing Setup

- Reuse as much code for the interpreters as possible

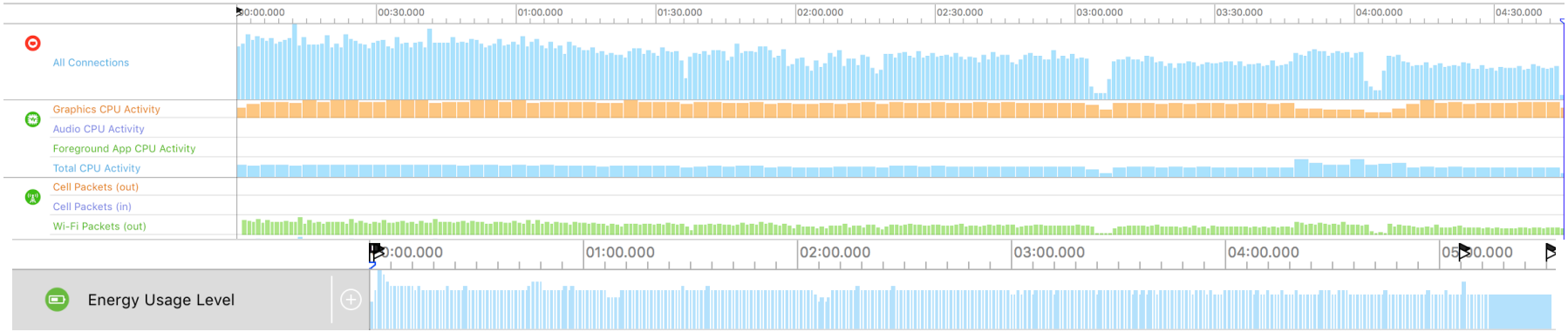
```haskell
runLocalComputationM :: forall effs a. LastMember IO effs
                     => Eff (Computation ': effs) a -> Eff effs a
runLocalComputationM = interpretM $ \case
  (IsPrime i) -> do
    let computation = computeIsPrime i
    setiOSLabel1 $ show computation
    setiOSIncLocalCounter
    pure computation
  (FactorialLength i) -> do
    let computation = computeFactorialLength i
    setiOSLabel2 $ show computation
    setiOSIncLocalCounter
    pure computation
```

```haskell
runComputationM :: forall effs a. LastMember IO effs
                => Eff (Computation ': effs) a -> Eff effs a
runComputationM = interpretM $ \case
  p@(IsPrime i) -> do
    computation <- offloadComputation (Proxy :: Proxy Bool) computeIsPrime p i
    setiOSLabel1 $ show computation
    pure computation
  p@(FactorialLength i) -> do
    computation <- offloadComputation (Proxy :: Proxy Int) computeFactorialLength p i
    setiOSLabel2 $ show computation
    pure computation
```

- Just one line changed
- Effectively the same code (can even just set the profiler to False)
- Very simple to mock user input if needed

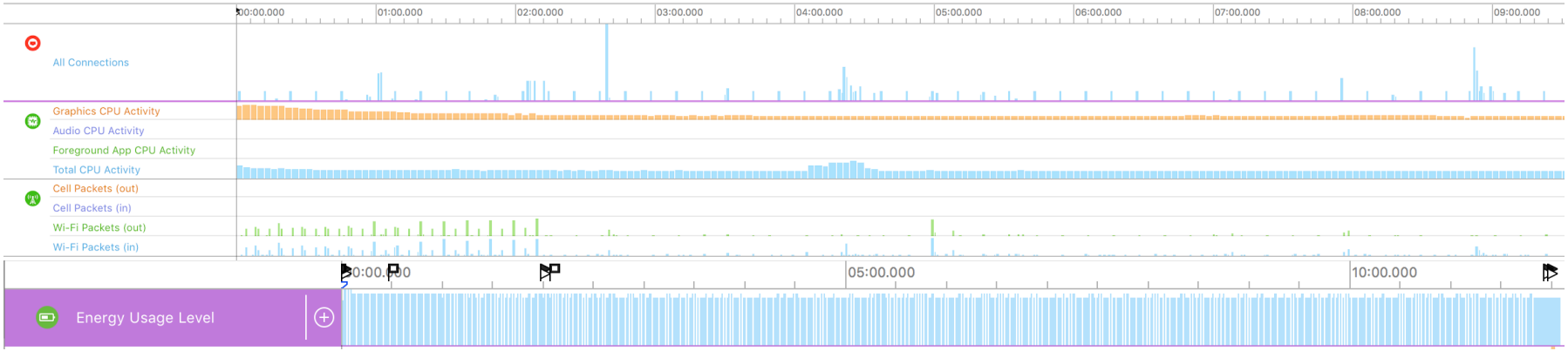
Measurements: Offloaded

- Runtime: 4:45 minutes
- Energy: 10/20 (average)
- CPU: 45% (average)
- GPU: 10% (average)
- Network: 4.58 MiB Transferred (total)



Measurements: Local

- Runtime: +11:00 minutes
- Energy: 15/20 (average)
- CPU: 38% (average)
- GPU: 5% (average)
- Network: 82.95 KiB Transferred (total)



KAIST DTU

Measurements: Comparison

Evaluation

Local

- Runtime: +11:00 minutes
- Energy: 15/20 (average)
- CPU: 38% (average)
- GPU: 5% (average)
- Network: 82.95 KiB Transferred (total)

Offloaded

- Runtime: 4:45 minutes
- Energy: 10/20 (average)
- CPU: 45% (average)
- GPU: 10% (average)
- Network: 4.58 MiB Transferred (total)

Comments

- Local is **+6:15 minutes slower**
- Local **energy consumption is 50% higher** than offloaded
- CPU and GPU utilization is **lower on local** (GPU because of slower UI repaints)
- Network transfer is of course **far higher when offloading** (no more than a high-res JPG image)
- Local slows down around ~3:00 minutes, and very much after ~7:00 minutes

Apple Guidelines on Mobile Battery Leaks

- Battery drain ← from keeping the device running longer (local) and from networking (offloading)
- Activity when you expect your app to be idle
- An unresponsive or slow user interface
- Large amounts of work on the main thread ← the need to run computationally intensive code on the device (local)
- High use of animations
- High use of view opacity
- Swapping
- Memory stalls and cache misses
- Memory warnings
- Lock contention
- Excessive context switches
- Excessive use of timers
- Excessive drawing to screen ← side effect of executions going faster (offloading), but can't say it's a problem
- Excessive or repeated small disk I/O
- High-overhead communication, such as network activity with small packets and buffers ← with the need to communicate (offload)
- Preventing device sleep ← because of longer execution durations (local)

Discussion & Conclusion

Discussion

- Even though our average CPU utilization is higher, we still see **clear gains by offloading**, particularly when execution time increases
  - Other factors like memory, CPU idle time, etc, might affect
- Evaluating/profiling our program **did not touch the program itself**, only the interpreters
- We have **full control** over our effects because of purity
- Haskell is still very much in the early stages for usage on mobile devices (via Cross-compilation)!

Conclusion

- By utilizing a pure functional programming language, **we can get rid of the need to worry about state**
- We are able to construct **just one program** that is **used for both** running our actual code, and evaluating its effectiveness
- Mobile devices impose **interesting considerations**:
  - Trade-off between which components to keep active (CPU, Screen, Network)
  - Code needs to run on ARM
  - Latency matters a lot in instances of UI interactions

Thank You!