



# Diffusing Your Mobile Apps: Extending In-Network Function Virtualisation to Mobile Function Offloading

Mario Almeida, Liang Wang*, Jeremy Blackburn, Konstantina Papagiannaki, Jon Crowcroft*

Telefonica Research, ES University of Cambridge, UK

# Observation: Pervasive Mobile Apps

Pervasive mobile clients have given birth to complex mobile apps. These apps are continuously generating, disseminating, consuming, and processing all kinds of information, in order to provide us convenient daily services.

# Motivation: Battery Is The Bottleneck

Unfortunately, given current battery technology, these demanding apps impose a huge burden on energy constrained devices.

While power hogging apps are responsible for 41% degradation of battery life on average.

Even popular ones such as social networks and instant messaging apps (e.g., Facebook and Skype) can drain a device's battery up to 9X faster due only to maintaining an on-line presence.

# Solution: Mobile Computation Offloading

Instead of doing the computation locally, let someone else do the job for you. A seemingly simple solution, but technically challenging, we need to answer:

| | |
|---|---|
| What to offload? | E.g., A function, a class, or a whole app? |
| When to offload? | E.g., statically or dynamically? |
| How to offload? | E.g., monitor and intercept function call? |
| Where to offload? | E.g., other user devices, cloud, or edge? |
| How to execute? | E.g., JVM, container, unikernel? |

How to sync, how to return results, how to discover functions, so on and so on ...

# More Observations

Quite different from a decade ago, network middle boxes are no longer simple devices which only forward packets.

ISPs' own network services have been shifting from specialized servers to generic hardware with the adoption of the NFV paradigm.

E.g., Telefonica is shifting 30% of their infrastructure to NFV by 2016. Other providers such as AT&T, Vodafone, NTT Docomo, and China Mobile.

There are many in-network resources we can exploit. Besides, many of them are underutilised.

# Our Contribution - INFv

INFv is an offloading system able to cache, migrate and dynamically execute on demand functionality from mobile devices in ISP networks.

Features:

- Offloading is at Java class granularity.
- Non-intrusive deployment, i.e., no need to repackage the existing apps.
- Offloaded functions are cached in the network for future use.
- The computation load is well well-balanced in the network by exploiting neighbourhood resources.

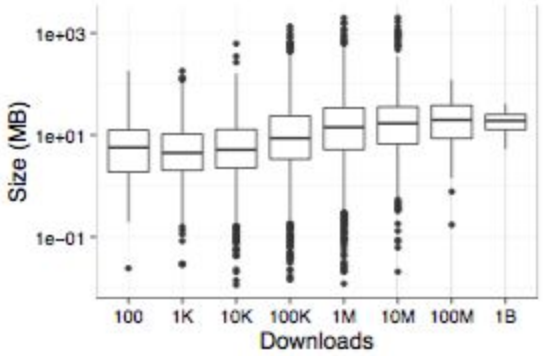
# Comparisons To The Existing Solutions

| MCO | Partitions | Dynamic | No Repackage | Stock OS | Off–Cloud | Off–Network | Deployment |
|---|---|---|---|---|---|---|---|
| MAUI [3] | Manual / Method | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| ThinkAir [5] | Manual / Method | ✗ | ✗ | ✓ | ✓ | ✗ | ✓(EC2,cost) |
| CloneCloud [9] | Auto / Thread | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ |
| Comet [8] | Auto / Thread | – | ✓ | ✗ | ✓ | ✗ | ✗ |
| Zhang et al. [7] | Auto / Class | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| INFv | Auto / Class | ✓ | ✓ | ✓ | ✓ | ✓ | ✓(cache,load) |

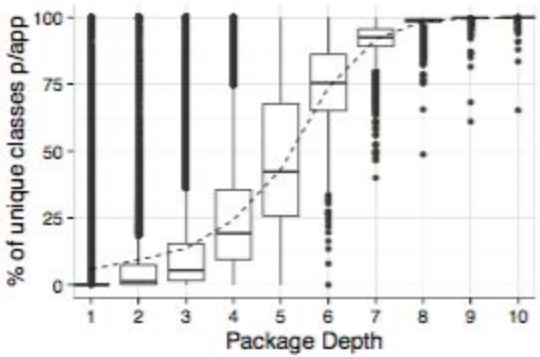
INFv is now a fully functional, working system after one-year development.

UNIVERSITY OF CAMBRIDGE

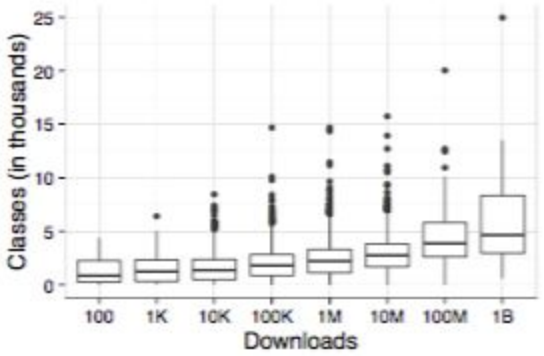
# Feasibility Study: Can We Cache Everything




(a) Installation size of apps as a function of app popularity.




(b) Percentage of unique classes per app as a function of name depth used for comparison.




(c) Number of classes per app as a function of app popularity.

Figure 1: Study of over 20K Google Play apps regarding their size, structure and uniqueness.

UNIVERSITY OF CAMBRIDGE

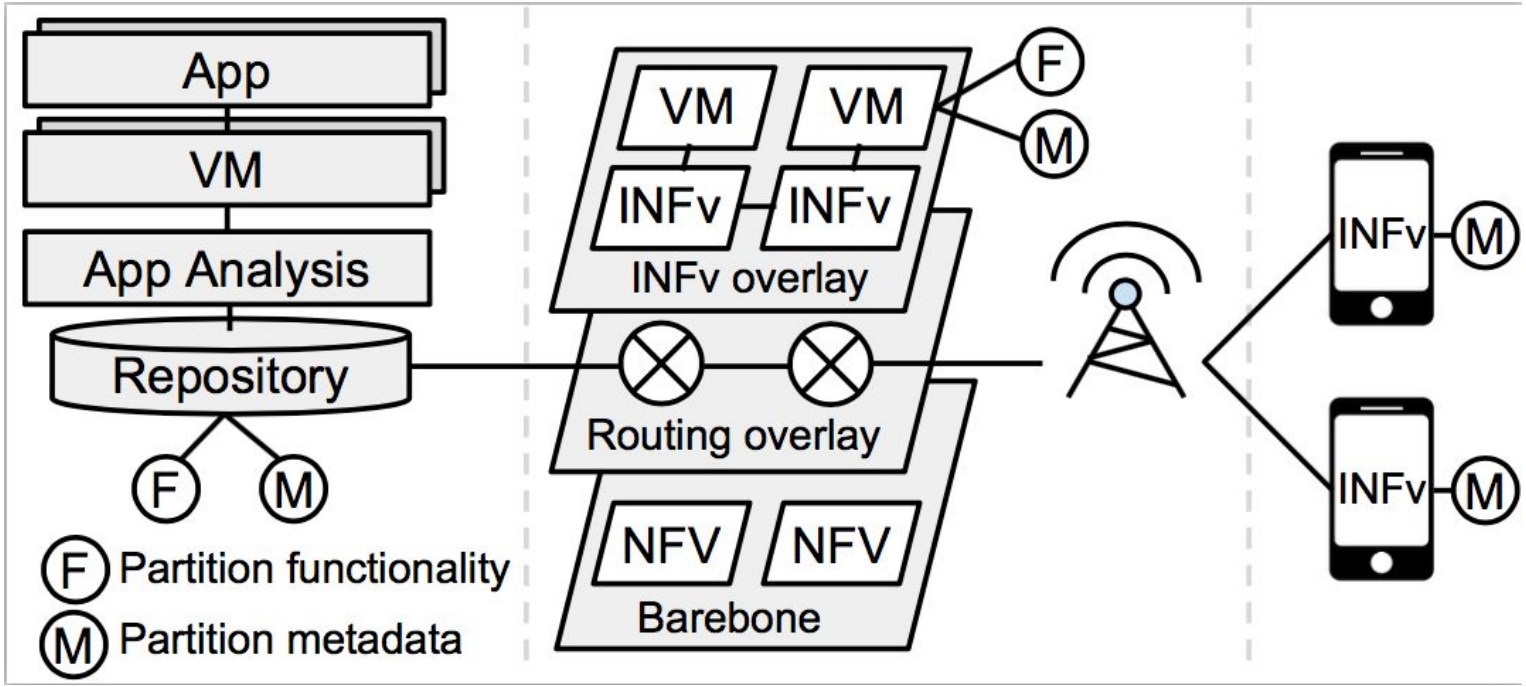
# Feasibility Study: Key Takeaways

A study of over 20K of the most popular apps on the Google Play Store in February 2016. In summary, our investigation reveals two important facts:

- a very small amount of apps account for most of the downloads;
- there is a significant code overlap due to commonly used libraries.

Two facts together indicate a medium storage requirement on INFv. We confirm that ≈ 15, 000 apps are responsible for 81% of all Google Play downloads and in total they require an aggregated storage of 409 GB. This is apparently a manageable size even without excluding a few outliers which are up to 2.1 GB.

# Architecture And Subsystems




Profiling Subsystem

Network Subsystem

Offloading Subsystem

UNIVERSITY OF CAMBRIDGE

# Profiling Subsystem: Cloud-based Profiler

We do not monitor or profile on mobile devices. There is a trade-off between accuracy and cost. In general, performance profiling is an expensive operation. Then how can we make the offloading decision?

Both profiling and partitions are performed by INFv's profiling subsystem which consists of an app analysis cluster and a repository containing apps' partition functionality as well offloading metadata (e.g., energy estimates).

However, we can have some “free variables” in the equation of making offloading decisions, which can be easily filled by mobile devices.

# Profiling Subsystem: App Partitioning

Functionality partitions can be devised on method and thread granularity. The latter incurs an extra cost of synchronization (e.g., thread state, virtual state, program counters, registers, stack). In order to better integrate with the NFV abstraction, our solution is based on method offloading.

However, app partition happens at class granularity. Most mobile architectures apps are developed in class-based object-oriented languages, when possible we use a class offloading granularity as methods can invoke other methods of the same class and share class state (e.g., class fields).

# Offloading System

Step 1: Offloading subsystem runs on user devices. It first contacts profiling subsystem to fetch the profiling and partition metadata. Given a partition entry and exit points, it enables the interception of its respective members (methods & constructors).

Step 2: Each app process transparently loads and executes an INFv monitor. Once a target invocation is intercepted, a message is created and sent to a network stub that transparently interacts with the closer network node to execute it.

# Offloading System

Step 3: Network nodes abstract the network topology by providing a message queue (MQ) between the stub and the execution backend.

Step 4: Within the MQ abstraction, routing is done using the user, device, app and version IDs, along with the fully qualified member name and its arguments.

Step 5: Offloaded functionality threads are suspended until the functionality finishes or invokes a local functionality in the same thread.

# Network Subsystem

The main goal is to balance the load in the network. Note the difference between load balancing at edge and scheduling in the cloud.

We considered CPU, memory, and bandwidth. But it seems that CPU always becomes the first bottleneck, because most offloaded tasks are computation intensive.

Two strategies are studied: Passive and Proactive, by using a simple M/M/1-PS queuing model to analyse the performance and derive the balancing algorithm.

# Proactive Strategy: C3PO

**Algorithm 1** C3PO - Proactive Computation Control

```
1:  void on_arrival (request r):
2:      buf_λ[i] ← timestamp (r)
3:      λ ← mean_rate (buf_λ)
4:      Δλ ← max(0, λ − λ′)
5:      λ ← λ + Δλ
6:      q ← eq.4 (λ, μ, c′, c″, m′, m″)
7:      if draw_uniform ([0,1]) < q then execute (r)
8:      else forward_to_lightest_load_node (r)
9:      i ← (i+1) mod k
10:     if i == 0 then λ′ ← 0.5×(λ′ + λ − Δλ)
11:
12: void on_complete (function s):
13:     buf_μ[i] ← execution_time (s)
14:     buf_c″[i] ← CPU_consumption (s)
15:     buf_m″[i] ← memory_consumption (s)
16:     i ← (i+1) mod k
17:     if i == 0 then
18:         μ ← 0.5×(μ + mean(buf_μ)^−1)
19:         c″ ← 0.5×(c″ + mean (buf_c″))
20:         m″ ← 0.5×(m″ + mean (buf_m″))
21:     forward_result (s)
```

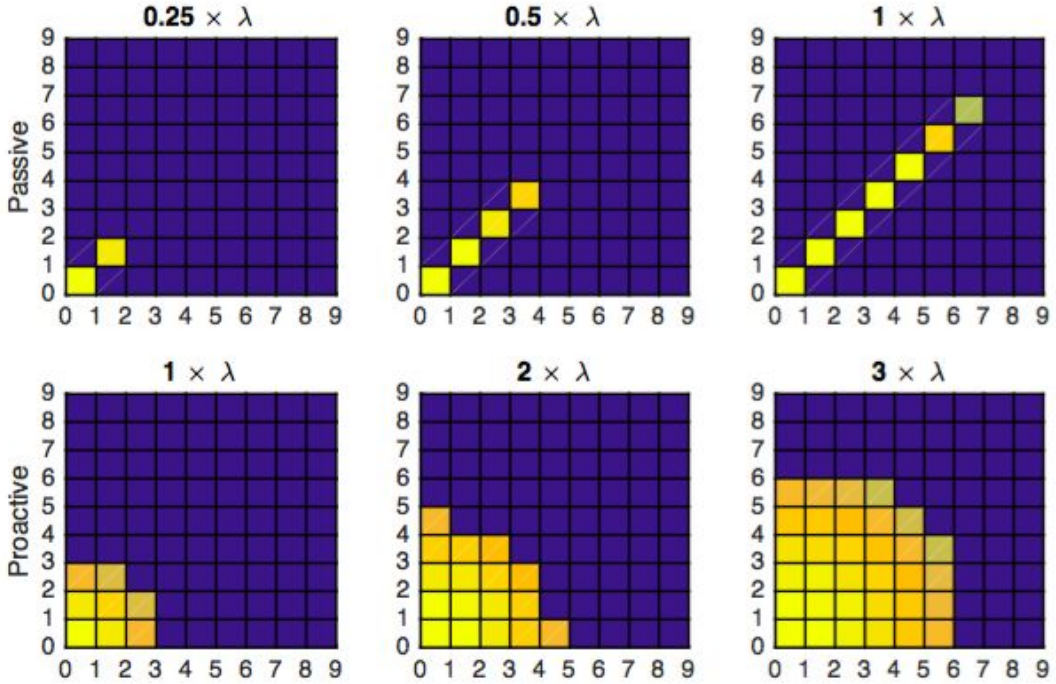
The algorithm is simple yet effective. We need to make sure the load balancing itself will not cause too much overhead.

The algorithm maintains four circular buffer with fixed size. on_arrvial and on_complete two functions need to be performed whenever a request arrives or finished.

The “proactiveness” is achieved by “being conservative”. Technically, by smoothing the load curve, or getting the derivative of the load increasing rate.

UNIVERSITY OF CAMBRIDGE

# C3PO Exploits Its Neighbourhood




An illustration of different behaviors of Passive and Proactive control on grid topology. A client connects to the router at (0,0) while a server connects to the router at (9,9). Proactive is more capable at utilizing the nearby resources within its neighborhood, leading to better load balancing and smaller latency. (Yellow indicates high load.)

UNIVERSITY OF CAMBRIDGE

# Preliminary Evaluation: Setup

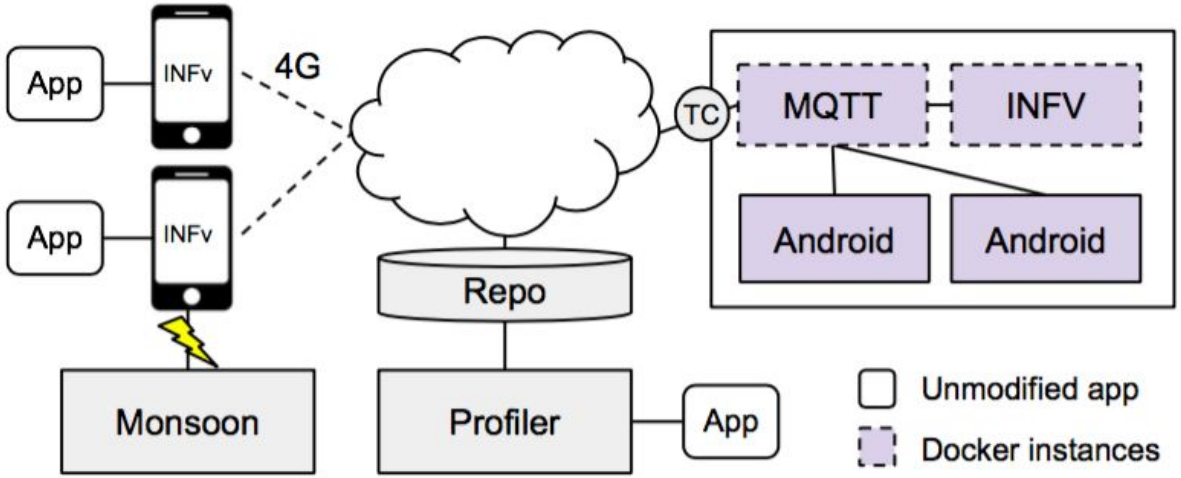



Figure 3: Experimental setup.

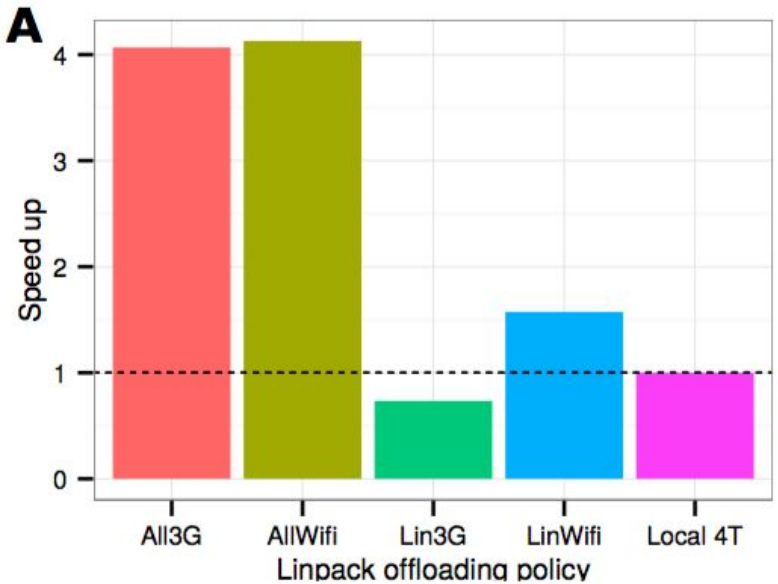
Two apps are tested:

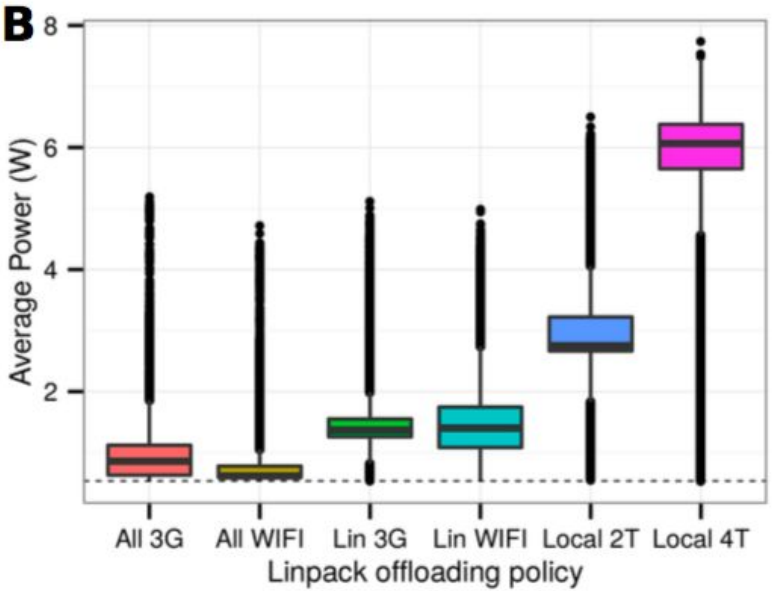
**Linpack**: represents pure computation intensive apps.

**FaceDetect**: represents both computation heavy and synchronisation heavy tasks.

UNIVERSITY OF CAMBRIDGE

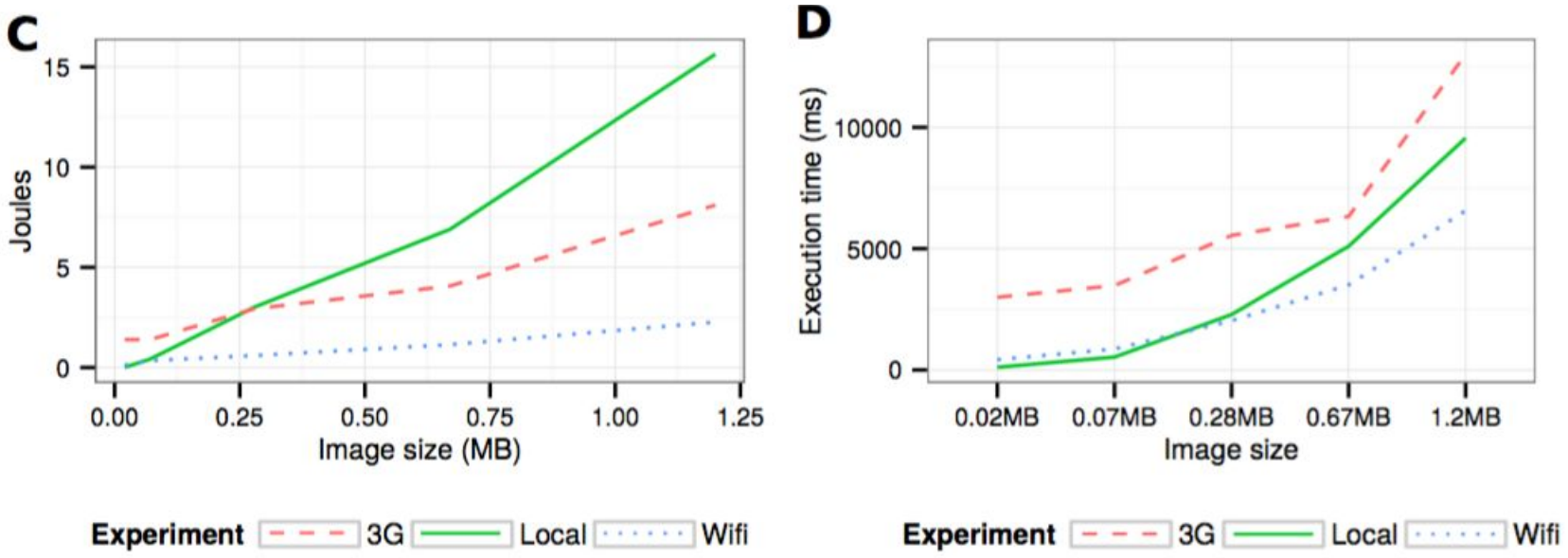
# Impact of Code Partitions




UNIVERSITY OF CAMBRIDGE

# Cost Of State Synchronisation




UNIVERSITY OF CAMBRIDGE

# Scalable to Workload

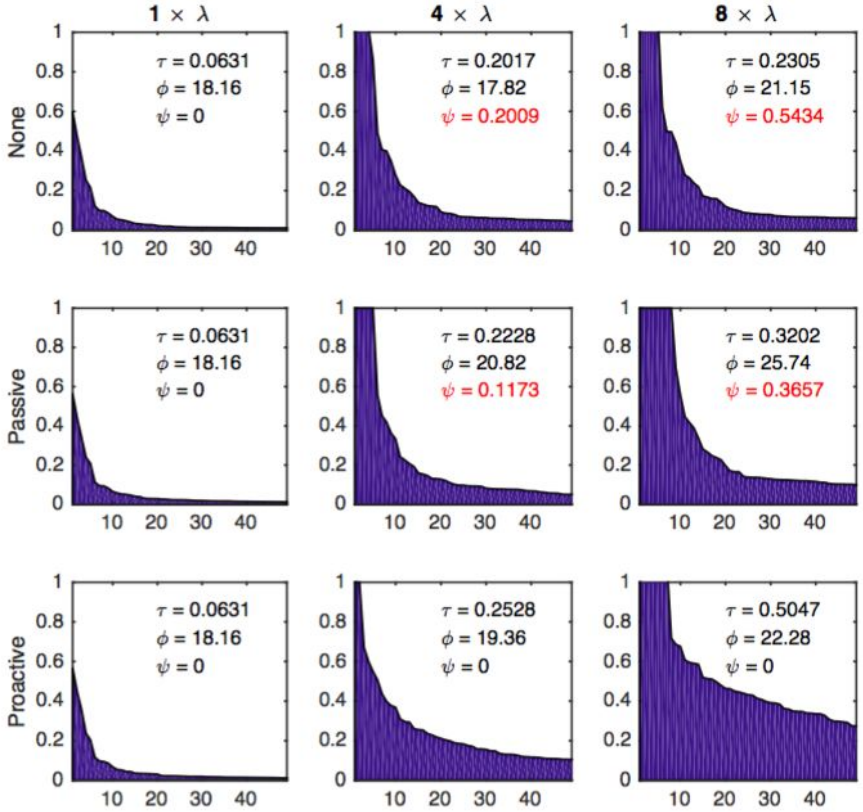



Figure 6: Comparison of three control strategies (in each row) on Exodus ISP network, the load is increased step by step in each column. *x*-axis is node index and *y*-axis is load. Top 50 nodes of the heaviest load are sorted in decreasing order and presented. Notations in the figure: τ : average load; φ : average latency (in *ms*); ψ : ratio of dropped requests.

UNIVERSITY OF CAMBRIDGE

# Responsiveness to Jitters

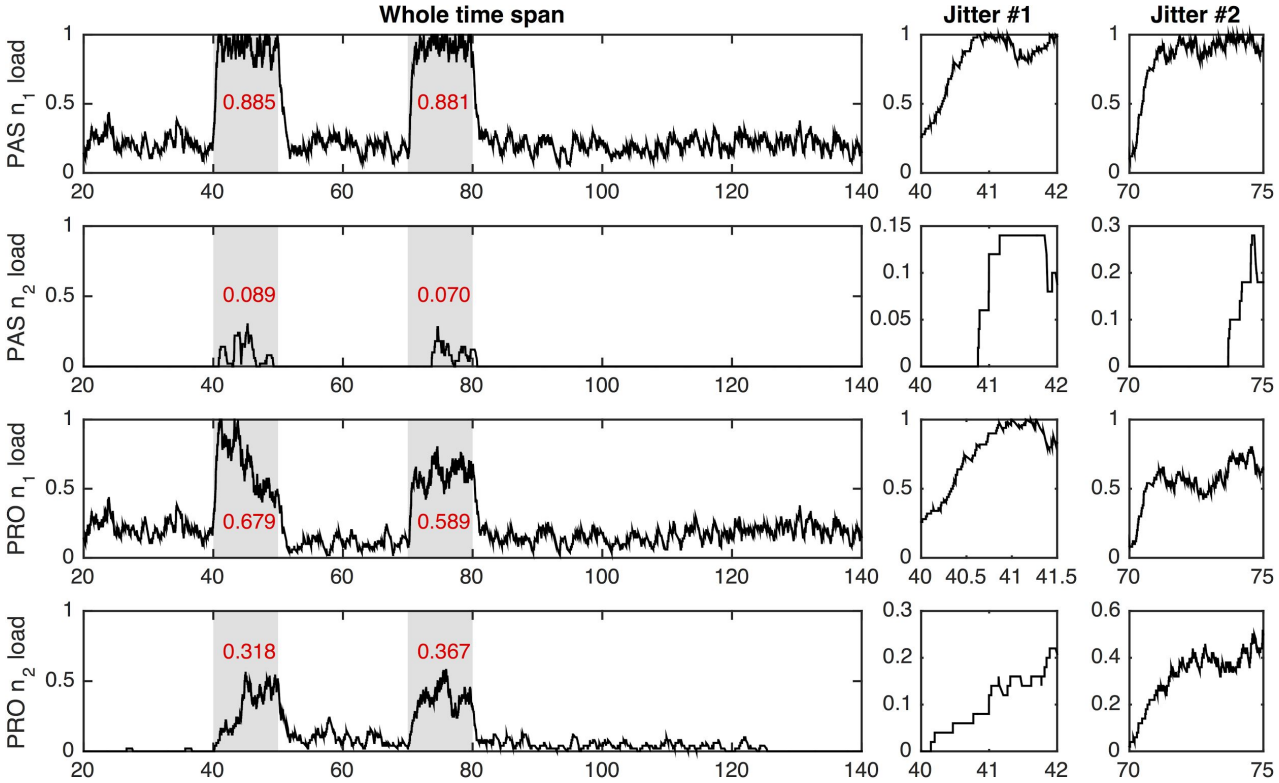



Figure 7: Comparison of two control strategies using a simple line topology: client → router *n*1 → router *n*2 → server. Two jitters are injected at time 40 ms and 70 ms. *x*-axis is time (ms) and *y*-axis is normalized load. Red numbers represent the average load during a jitter period.

UNIVERSITY OF CAMBRIDGE

# Power Consumption Distribution

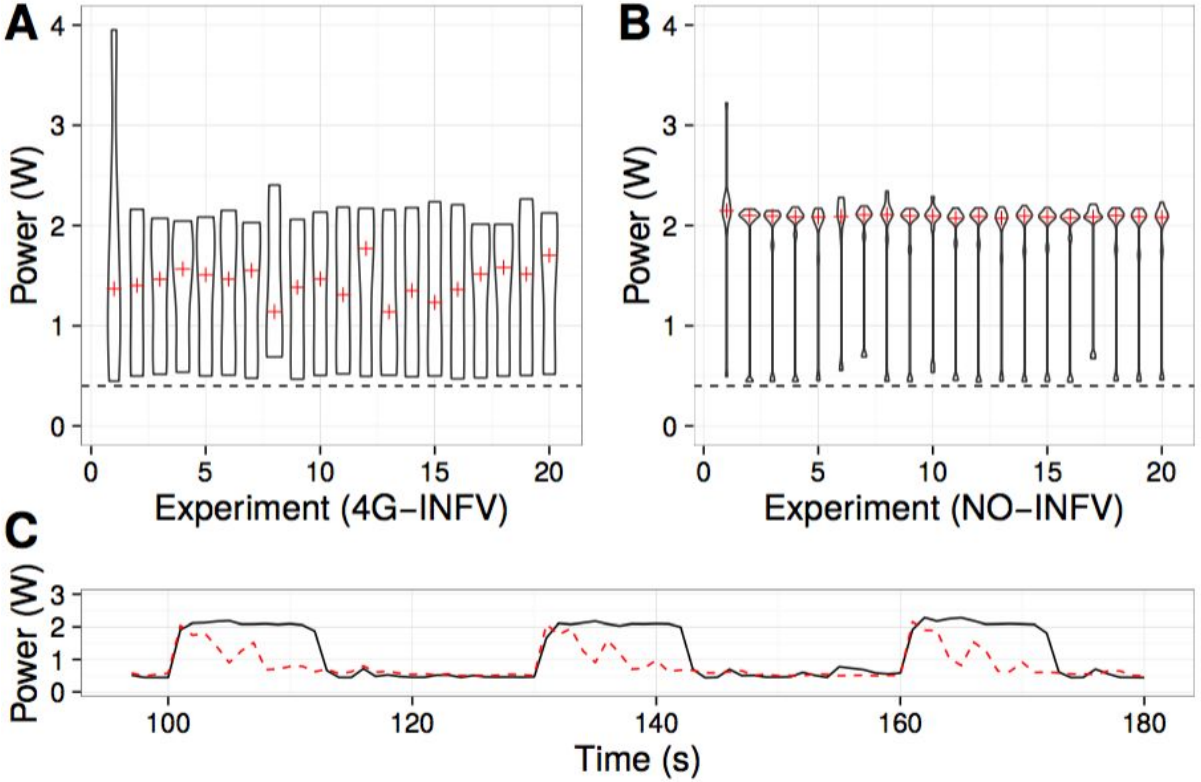



Figure 8: Power consumption distribution (A & B) for 20 executions over 4G, with and without INFv. In C the dashed line represents INFvs' consumed power versus the local execution (continuous).

UNIVERSITY OF CAMBRIDGE

# Energy Consumption & Execution Time

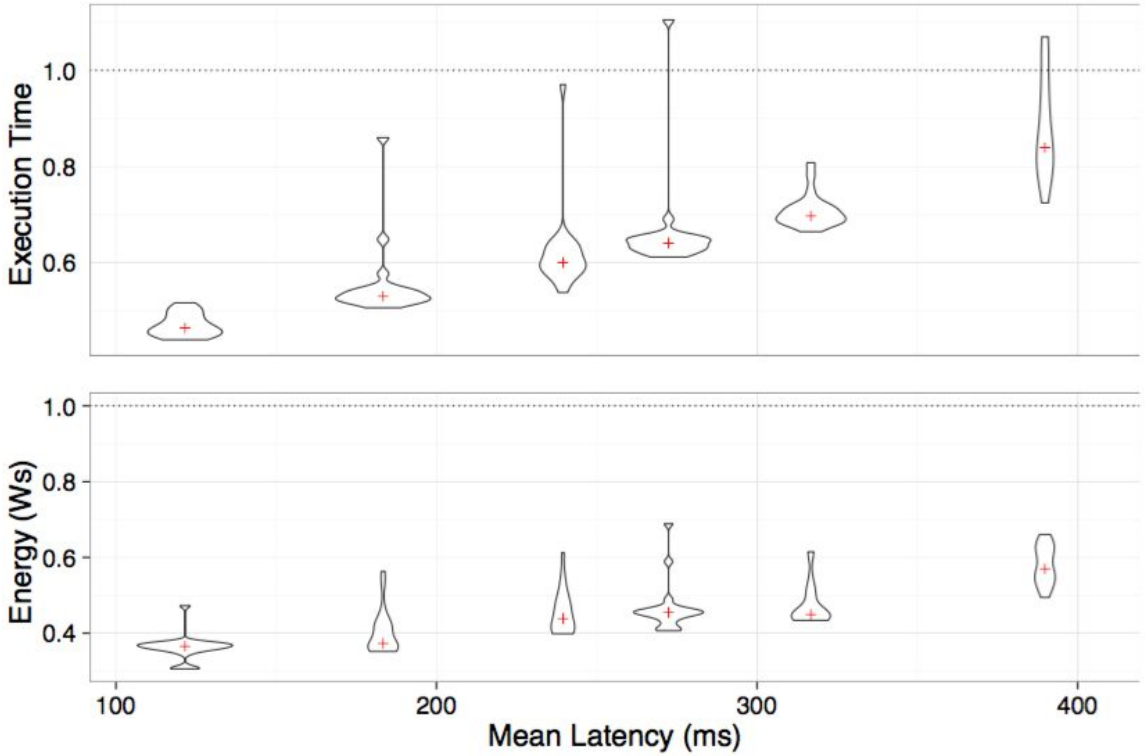



Figure 9: Energy and execution time of FaceDetect execution with INFv enabled. Crosses represent the median. Values are normalized by the mean values of local execution.

UNIVERSITY OF CAMBRIDGE

# Runtime Decision & Performance

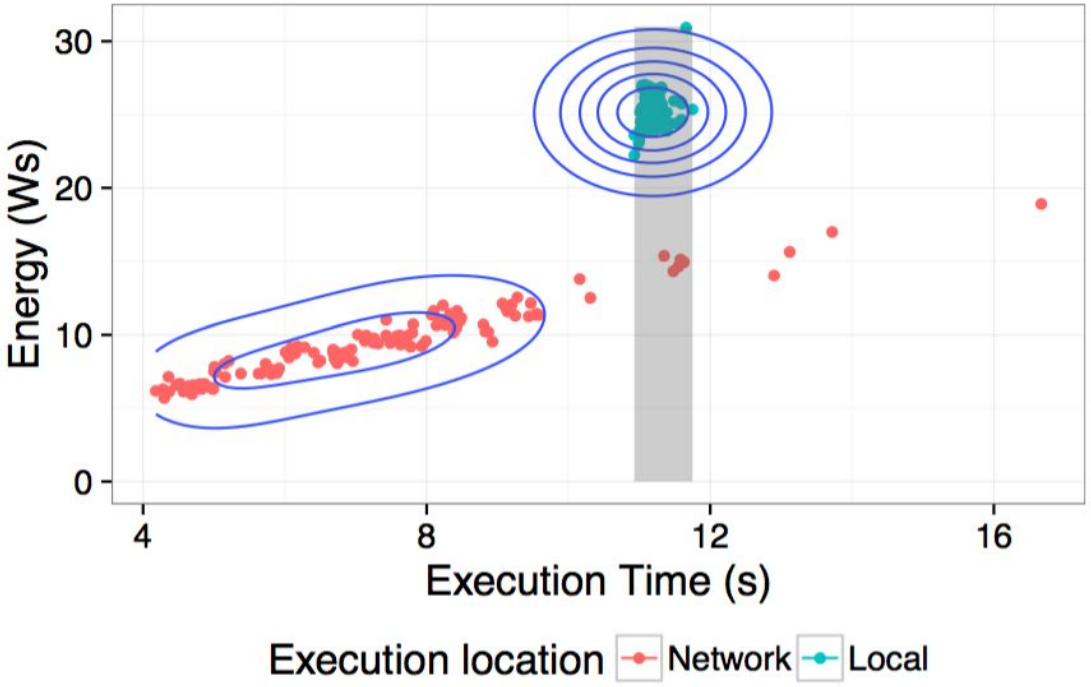



Figure 10: Energy consumption vs. execution time of FaceDetect using INFv under varying latency.

UNIVERSITY OF CAMBRIDGE

Thank you. Questions?

UNIVERSITY OF CAMBRIDGE