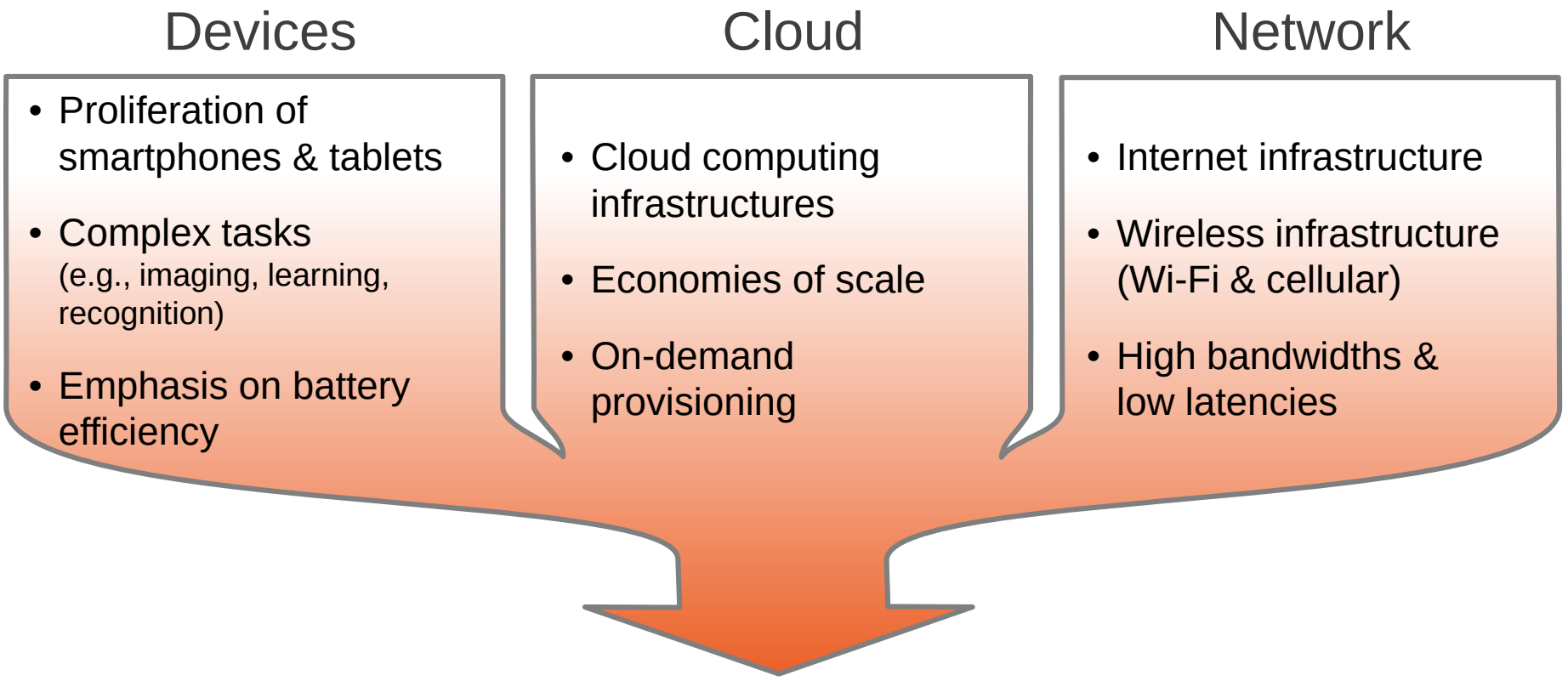
Mobile App Acceleration via Fine-Grain Offloading to the Cloud

Chit-Kwan Lin

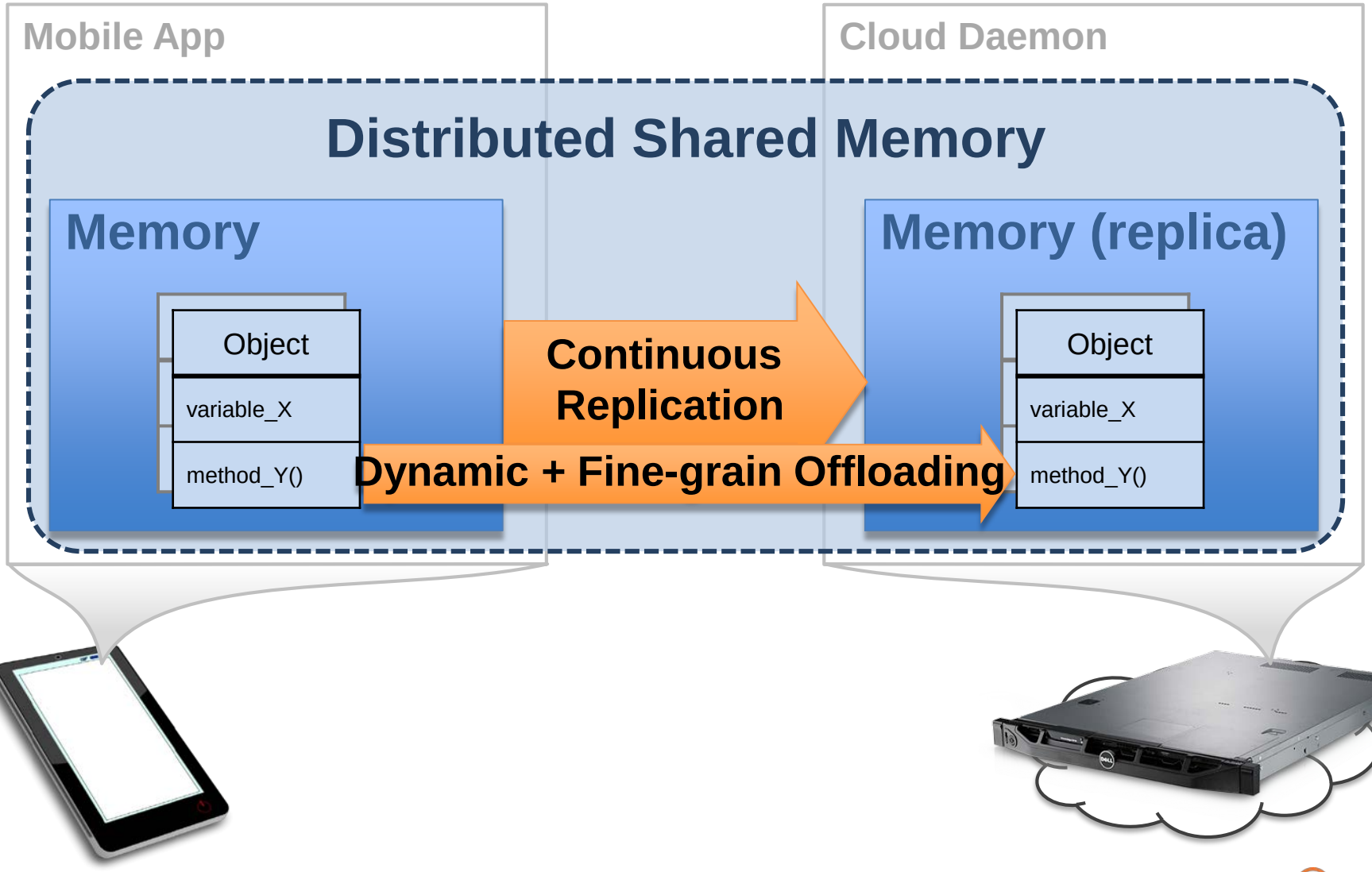
H. T. Kung

Confluence of Forces Points to Offloading to the Cloud



Compute Offloading

A Simple DSM Supports Offloading



* See last slide for image source references

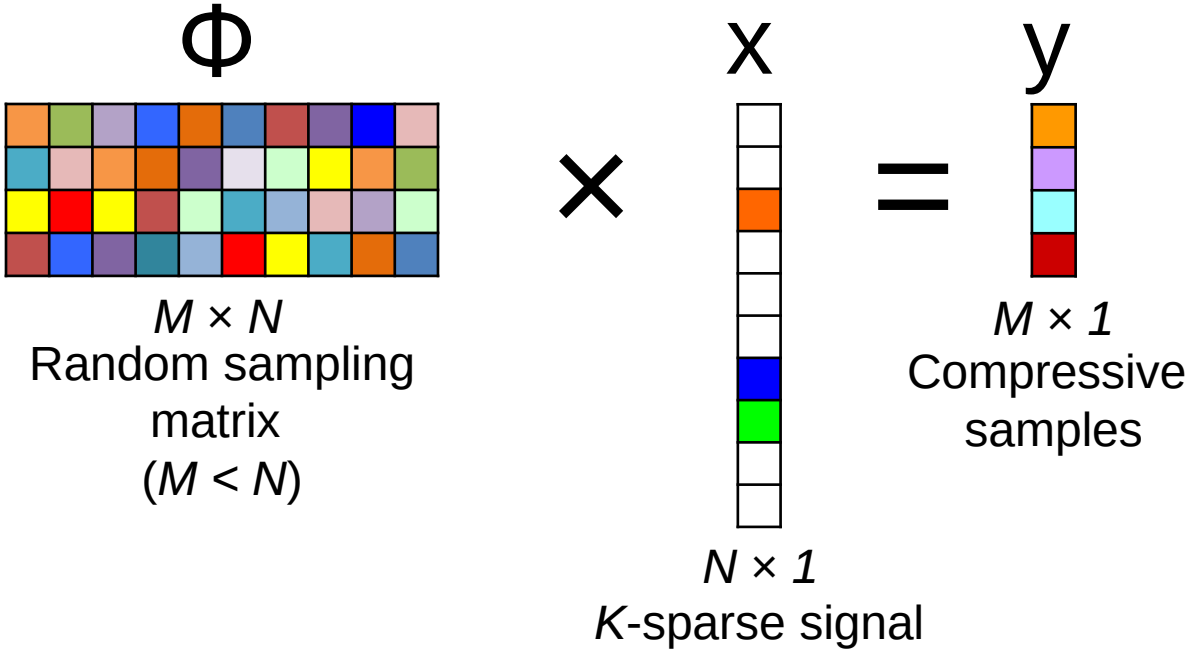
Advantages of Dynamic + Fine-Grain

- **Dynamic**
  - Can offload arbitrary work at runtime
  - Can optimize resource utilization (e.g., battery) at runtime
- **Fine-grain**
  - At the level of a method invocation
  - Feels more responsive when failure requires local restart to fix
  - New class of small workloads for cloud providers
    - Useful for leveling out utilization while providing low-latency services for end-users

Challenges

- **Latency**
  - Dictates granularity: if update takes 5s, then workloads that run <5s don't benefit from offloading
- **Bandwidth**
  - We should only send deltas, but determining delta encoding has non-trivial costs
    - e.g., rsync can take 3 round trips to generate a delta encoding, on top of time to calculate hashes
- **Compute**
  - We should compress to save bandwidth, but compression can be computationally expensive
- **Battery**
  - Shouldn't end up consuming more battery budget

Compressive Sensing

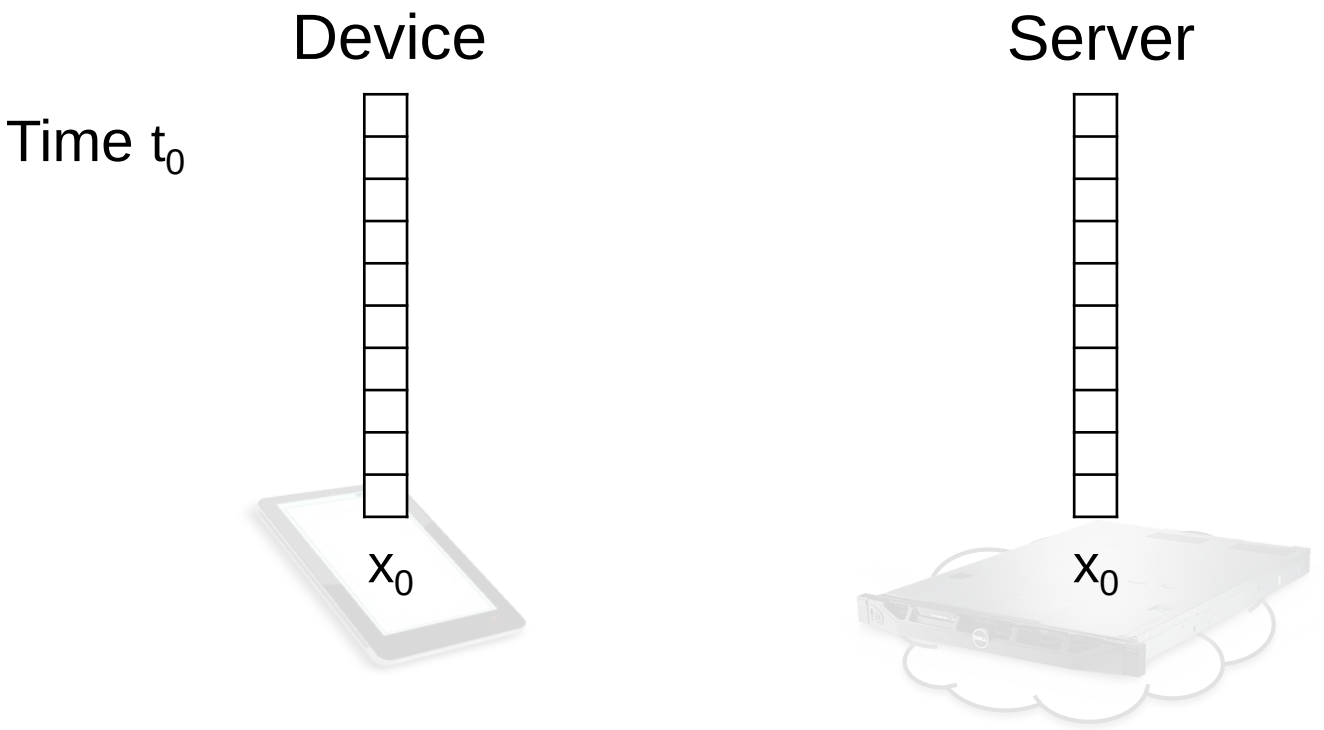


- Randomly mix signal elements by random projection onto lower-dimensional space
- Random Φ preserves Euclidean length/distance of sparse vectors with high probability when $M \geq O(K \log N/K)$
- Decode x from y by solving $y = \Phi x$ via optimization (linear programming)

Key Insight

Writes (deltas) to memory typically constitute a sparse signal that can be compressively sampled

Compressive Replication (1)

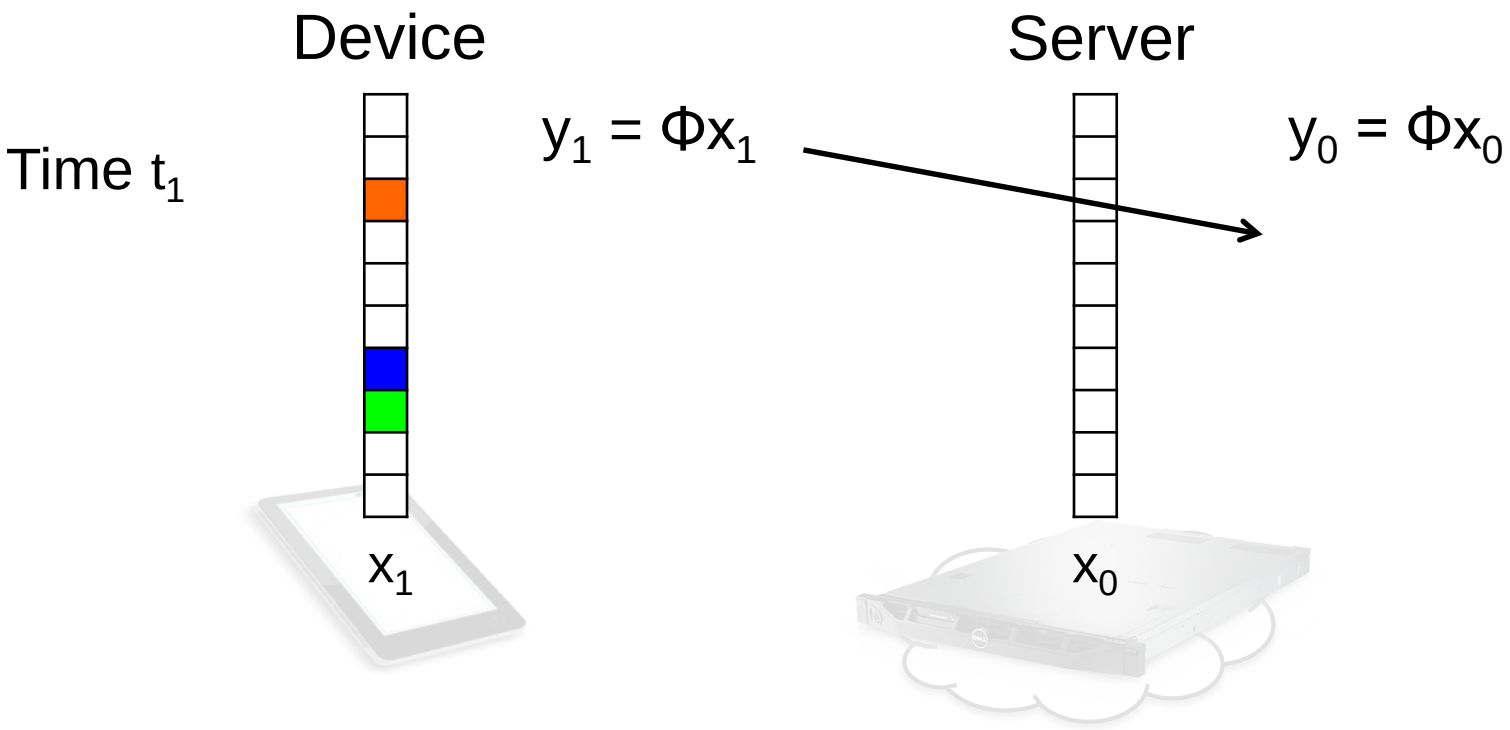


Memory starts out synchronized, with byte values x_0

Both device and server know Φ

Server calculates $y_0 = \Phi x_0$

Compressive Replication (2)

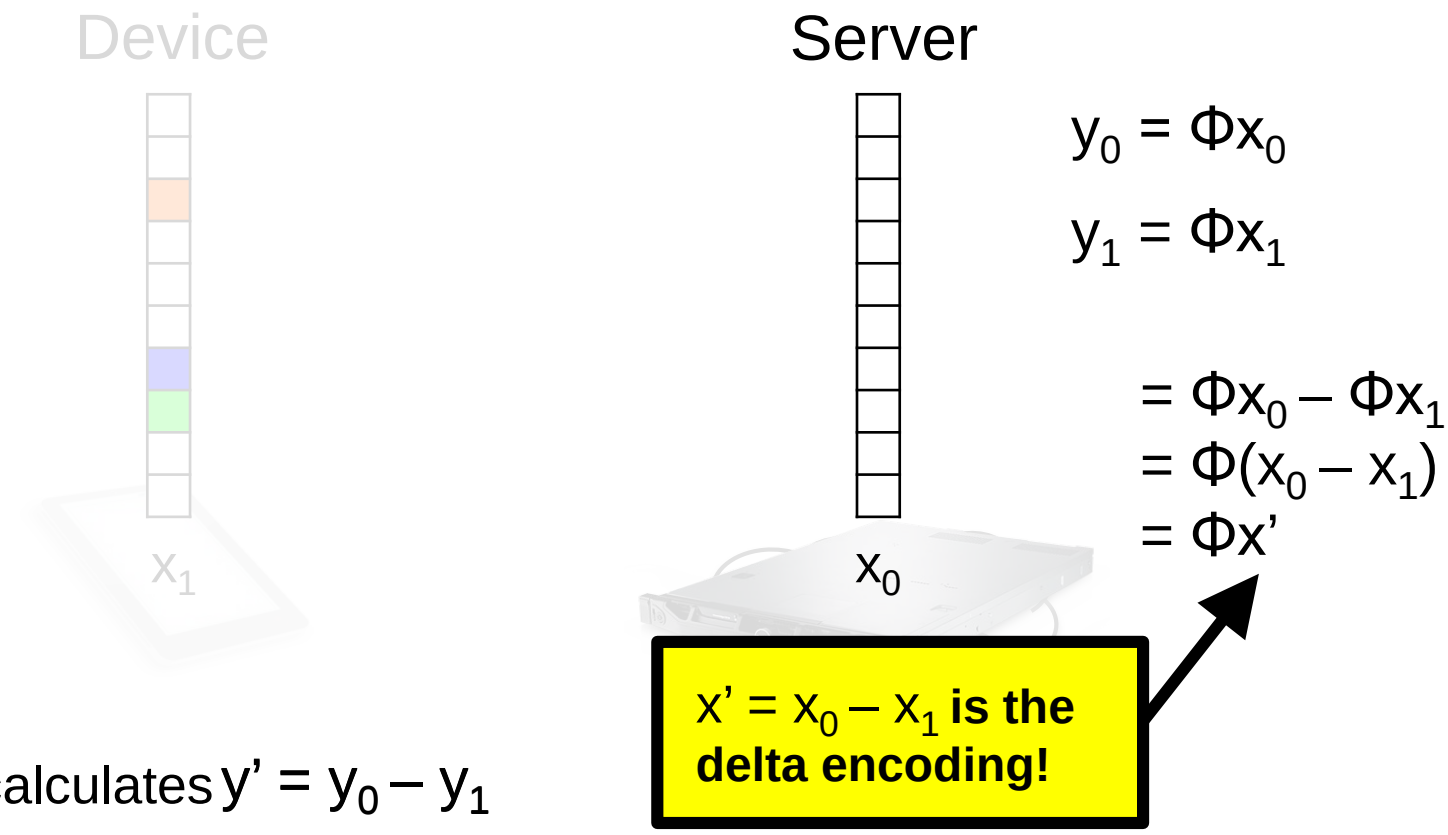


Some values in memory change, resulting in x_1

Device calculates $y_1 = \Phi x_1$

Device transmits $y_1 = \Phi x_1$ to server

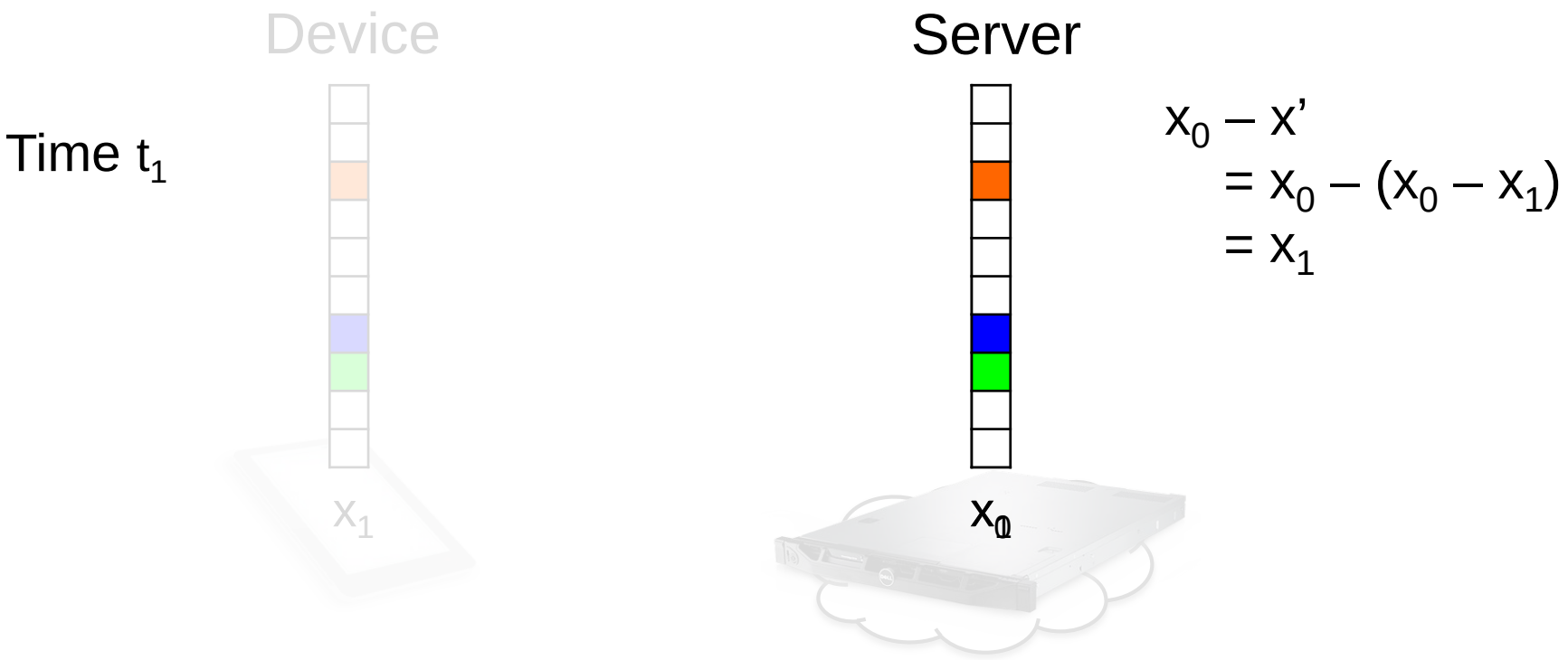
Compressive Replication (3)



Server calculates $y' = y_0 - y_1$

$y' = \Phi x'$ has same form as compressive sensing decoding problem, so server solves for x'

Compressive Replication (4)



Server calculates $x_0 - x' = x_1$, which is the new memory state

Server is now updated

Novel Characteristics

- **All-in-one**
  - Delta encoding + compression
- **Delta encoding figured out by server, not device**
  - Automatically recovered during decoding
  - Just send compressive samples; no add'l network costs
- **Codec is resource-commensurate**
  - Device: low-complexity encoder
  - Server: higher-complexity decoder
  - Unlike traditional compressors

What do we compare against?

- **No similar replication methods**
  - Compressive replication gives all-in-one delta encoding + compression
  - e.g., rsync
    - Compression is an add'l step
    - Needs multiple round-trips to determine delta encoding

- **Compressed snapshots probably fairest**
  - No add'l round-trip overheads
  - Just compress the whole memory page (snapshot)
  - snappy, zlib

- **Metrics**
  - Replicate 64KB memory block
  - Total Latency = encoding + network + decoding
  - Compression ratio (bandwidth cost)

Latency/Compression Trade-off

Near ~100ms threshold of user-perceptible app delay

	Encoding (ms)	Network (ms)	Decoding (ms)	Total Latency (ms)	Compression Ratio	Update Size (KB)
snappy	4	15	--	19	3.8 : 1	17.2
zlib	487	13	--	500	6.0 : 1	10.9
compressive replication	53	12	70	135	7.3 : 1	9.0

Highest compression ratio

Latency/Compression Trade-off

Remains good w/ high Kolmogorov complexity, which would confound snappy/zlib

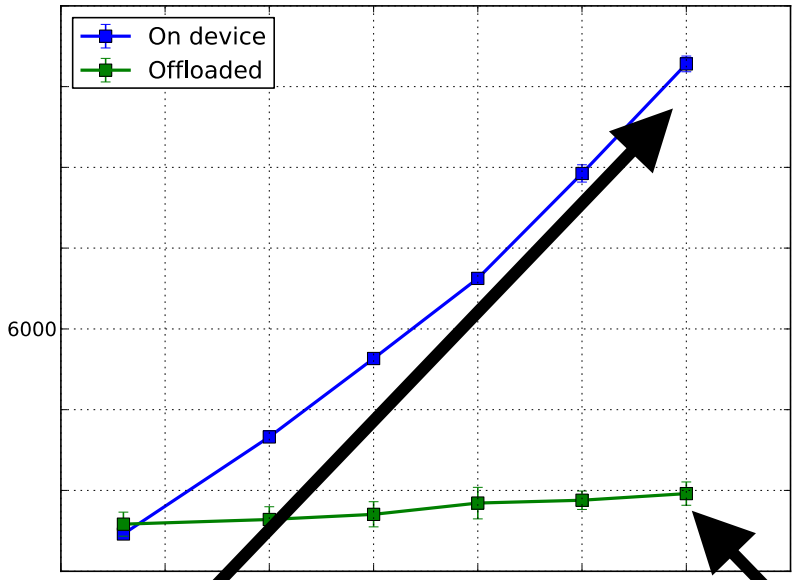
	Encoding (ms)	Network (ms)	Decoding (ms)	Total Latency (ms)	Compression Ratio	Update Size (KB)
snappy	4	15	--	19	3.8 : 1	17.2
zlib	487	13	--	500	6.0 : 1	10.9
compressive replication	53	12	70	135	7.3 : 1	9.0

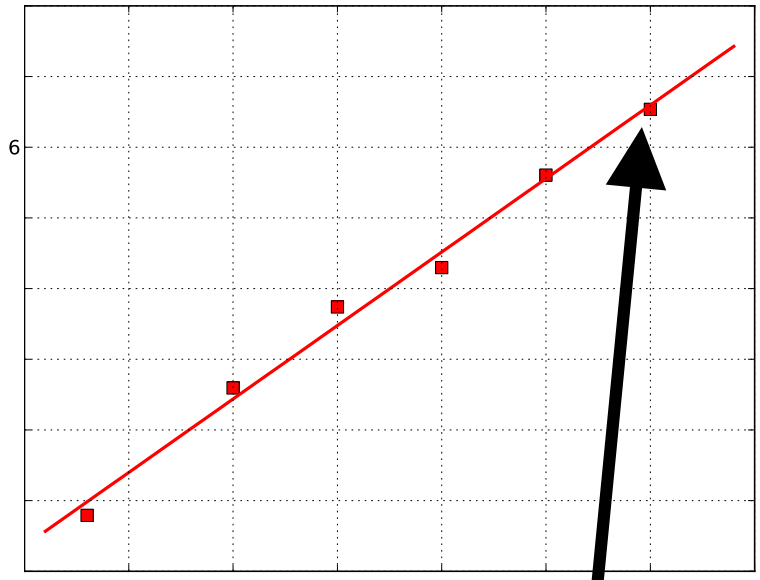
Unlike snappy/zlib, comp ratio fixed & b/w cost predictable because $y = \Phi x$ is independent of input's Kolmogorov complexity

Example Handwriting Recognition App

- UpShift platform for **compressive offloading** of iOS apps
- SVM for Chinese handwriting recognition
  - Stroke count is measure of task complexity
- Device: iPad (3rd gen)
- Server: Amazon g2.xlarge
  - GPU for compressive replication decoding
  - CPU for SVM evaluation
- Network: 802.11g
  - Office setting
  - 19ms RTT to us-east-1a server

App Speedup





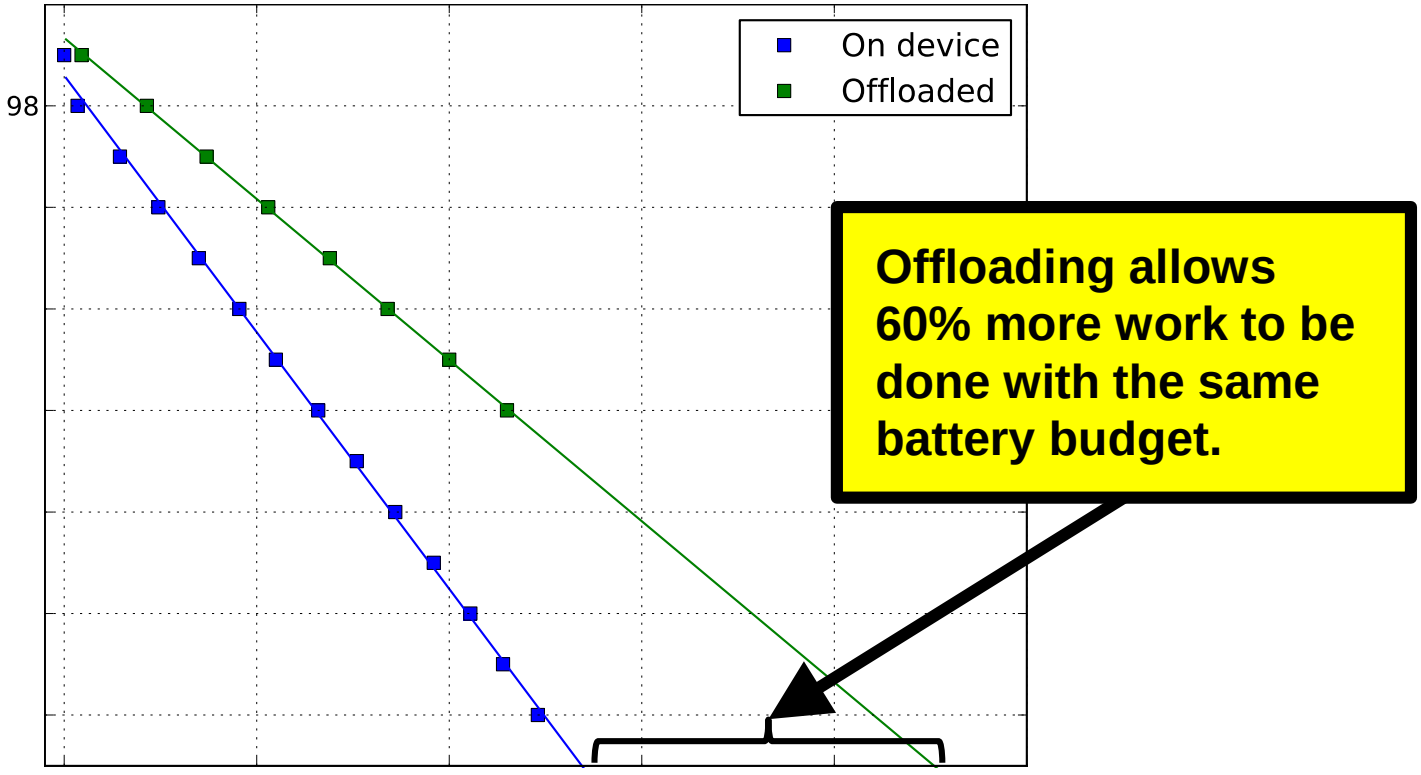
On-device execution time scales poorly with task complexity

Offloaded execution time stays short due to low overhead of compressive offloading. Users expect this.

More complex tasks have greater speedup

Battery Savings

High task complexity (25 strokes), 250 iterations



Summary

- Low-latency DSM updates enable fine-grain offloading
- UpShift supports offloading ~100ms workloads, while keeping resource utilization low
- Achieves significant speedups and battery savings
- Key insight: memory writes are a sparse signal that can be compressively sampled
- Implications for future ARM-based DC's or x86-based devices

Questions?

ck@upshiftlabs.com

Backup Slides

What about the reverse direction?

- Note that the system is asymmetric
  - Downlink is typically not as constrained as uplink
  - Servers can do zlib compression very fast
- So the snapshot method is OK for reverse direction
- Our results are obtained under a unidirectional replication constraint (replicas are read-only). Can we relax this?
  - If both ends can write to their memories independently, then we have a data consistency problem
  - However, latency is low enough that we can probably support synchronous offloading (i.e., block device until result is returned)
  - If app is already written under asynchronous model, then no consistency problem to begin with

How to trigger an update?

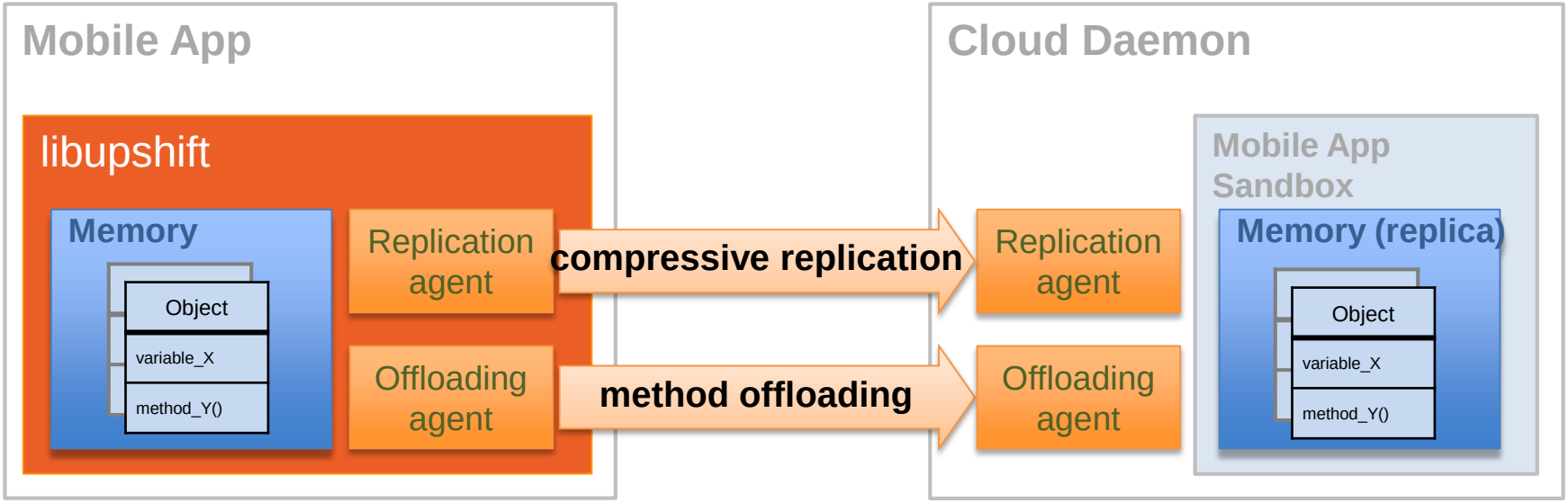
- How do we know when *k* writes have been made to memory?

- Inject statistics collection into Objective-C setter methods
  - Can do it at compile-time (preprocessing) or at run-time (method swizzling)

What if there are lots of memory writes?

- Examples
  - Large block initialization (calloc)
    - However, large block allocation is OK since most modern memory managers defer allocation until initialization
  - Reading large file into memory
- The signal is no longer sparse, so compressive sensing can no longer help
- Fall back to snapshotting
  - Can probably get away with it since users have some expectation of delay

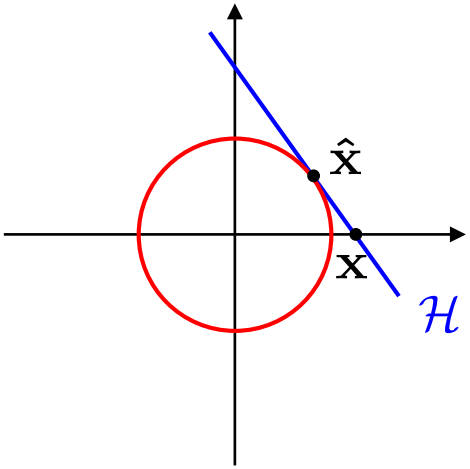
UpShift Platform Prototype



- Mobile app links against libupshift
- Objects are allocated from privately managed memory heap
- Replication agent manages replication of this memory
- Offloading is done via method swizzling (Objective-C is late binding)
- Offloadable objects are abstracted out into cross-compiled libraries, so server has class definitions
- Address spaces are fully managed by UpShift so we do address translation

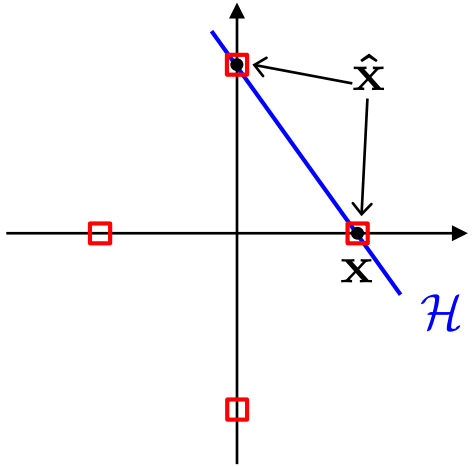
* See last slide for image source references

Compressive Sensing Decoding



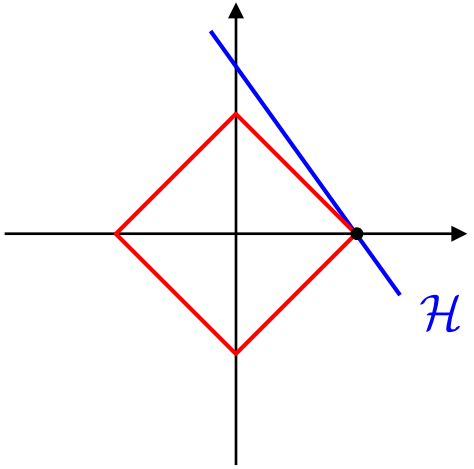
L2-norm minimization

Fast
Incorrect



L0-norm minimization

Intractable
Correct



L1-norm minimization

Tractable
Correct (usually)

- A linear program solves the L1-norm minimization:

$$\min_{\hat{\mathbf{x}} \in \mathbb{R}^N} \|\hat{\mathbf{x}}\|_1 \quad \text{s.t.} \quad \mathbf{y} = \mathbf{\Phi}\hat{\mathbf{x}}$$

Image Source References

- http://www.computerhistory.org/timeline/images/1944_harvard_markI_large.jpg
- http://history-computer.com/ModernComputer/Electronic/Images/PDP-1.jpg
- http://oldcomputers.net/pics/Altair_8800.jpg
- http://oldcomputers.net/pics/ibm5150.jpg
- http://media.t3.com/img/resized/ib/xl_IBM_ThinkPad_greatest_624.jpg
- http://www1.pcmag.com/media/images/383992-first-iphone.jpg?thumb=y
- http://zapp1.staticworld.net/reviews/graphics/products/uploaded/apple_ipad_family_710821_g2.jpg
- http://cdn.gottabemobile.com/wp-content/uploads/2014/01/Nest-Cooling-2-300x300.jpg
- http://i.dell.com/das/xa.ashx/global-site-design%20WEB/25b9106d-0ab1-0508-371f-f79a549236dc/1/OriginalPng?id=Dell/Product_Images/Dell_Enterprise_Products/Enterprise_Systems/PowerEdge/PowerEdge_R310/hero/server-poweredge-r310-left-hero-504x350.psd

scratch