Special Topics: Trends in edge computing

Jon B. Weissman (jon@cs.umn.edu)

Department of Computer Science
University of Minnesota

Cloudlets

- Developed at CMU by Mahadev Satyanarayan "Satya" (http://elijah.cs.cmu.edu/)

- Three edge scenarios
  - Mobile -> edge
  - Cloud -> edge
  - Edge native

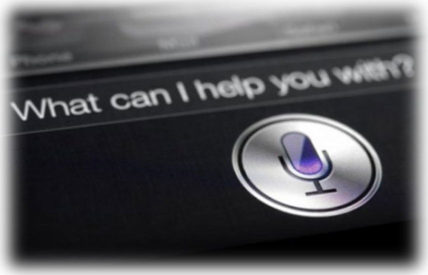
Two papers

Cloudlets: at the Leading Edge of Mobile-Cloud Convergence

Just-in-Time Provisioning for Cyber Foraging

Cloud Offloading

Rich, interactive applications are emerging in mobile context



- Apple’s Siri, AR apps
- Wearable devices

Cloud offloading

- These applications are too expensive to run on clients alone!
- Offload computation to a back-end server at cloud

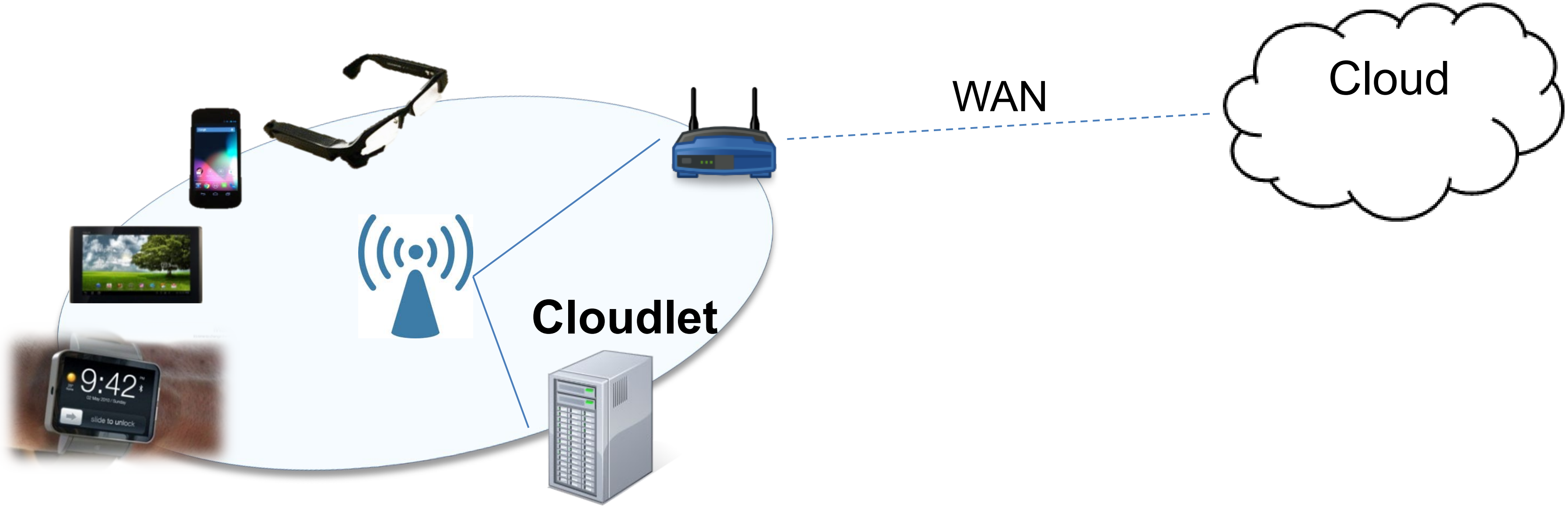
Today’s remote cloud is a suboptimal place; high latency and limited bandwidth

Optimize for user’s attention

Cloudlet as a Nearby Offload Site

Cloudlet: a nearby offloading site dispersed at the edges of the Internet

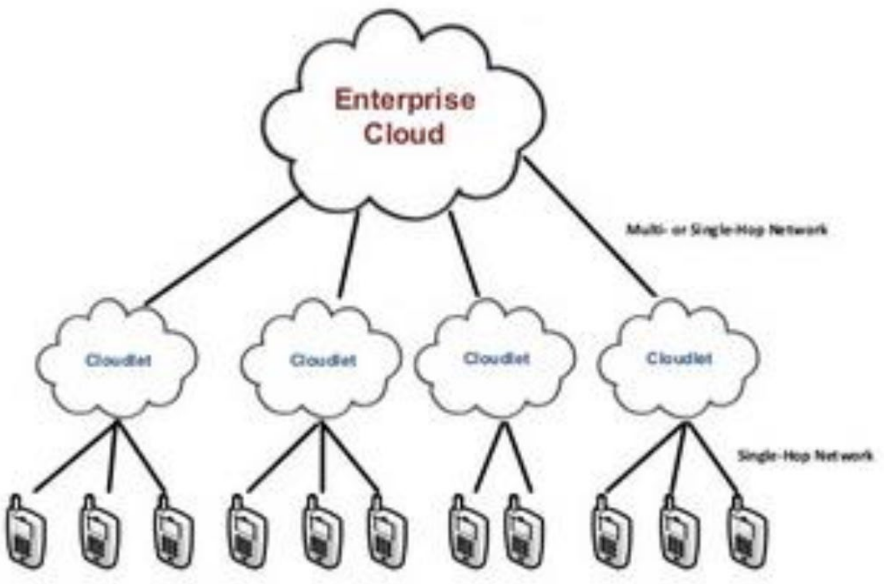
→ **Let's bring the cloud closer!**



How to launch a custom back-end server at an **arbitrary** edge?

Cloudlet

Focus on deployment and infrastructure



Challenge

- To make this viable and scalable, we need an edge infrastructure (maybe 3rd party)
  - Wide-area: think mobiles and travel
  - Shared: multiple apps running on the edge
  - Enable any apps in any language in any OS + software libraries, etc.
  - Robust
    - Secure
    - Disconnected fallback

- Need to encapsulate apps in VMs
- Granularity?

Options

- Static provisioning
  - Store all possible VMs on the edge nodes
  - Feasible?
  - Advantages?

- Dynamic provisioning
  - Issues?

Just-in-Time Provisioning

1. Support widest range of user customization including OS, language, and library
2. Strong isolation between untrusted computations
3. Access control, metering, dynamic resource management, …

A traveler wants to use natural language translation with speaker-trained voice recognition

→ VM (virtual machine) cleanly encapsulates this complexity, but delays provisioning : why?

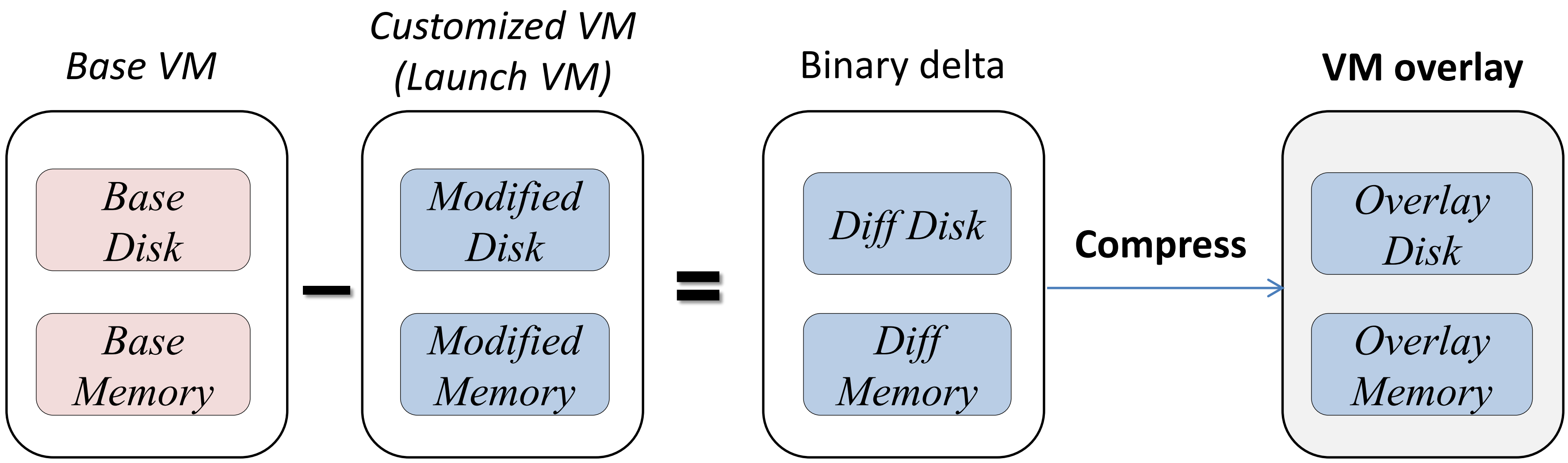
too expensive to send/boot a complete VM!

GOAL : Just-in-time provisioning of a custom VM for offloading. Ideally 10s latency

VM Synthesis

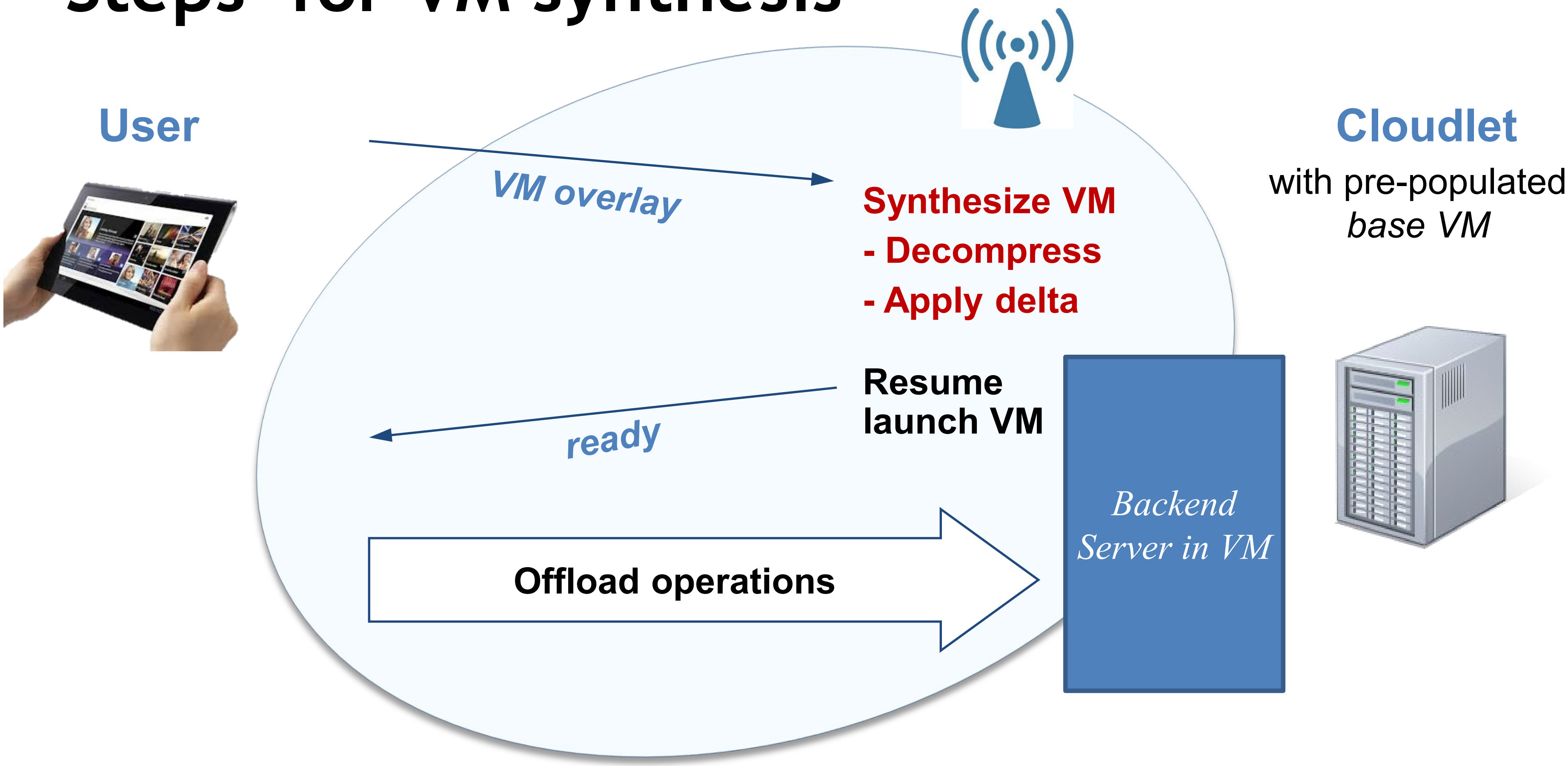
VM Synthesis: dividing a custom VM into two pieces

1) *Base VM*: Vanilla OS that contains kernel and basic libraries
2) *VM overlay*: A binary patch that contains customized parts



VM Synthesis

Steps for VM synthesis



VM Synthesis – Baseline Performance

- Base VM: Windows 7 and Ubuntu 12.04
  - **8GB** *base disk* and **1GB** *base memory*

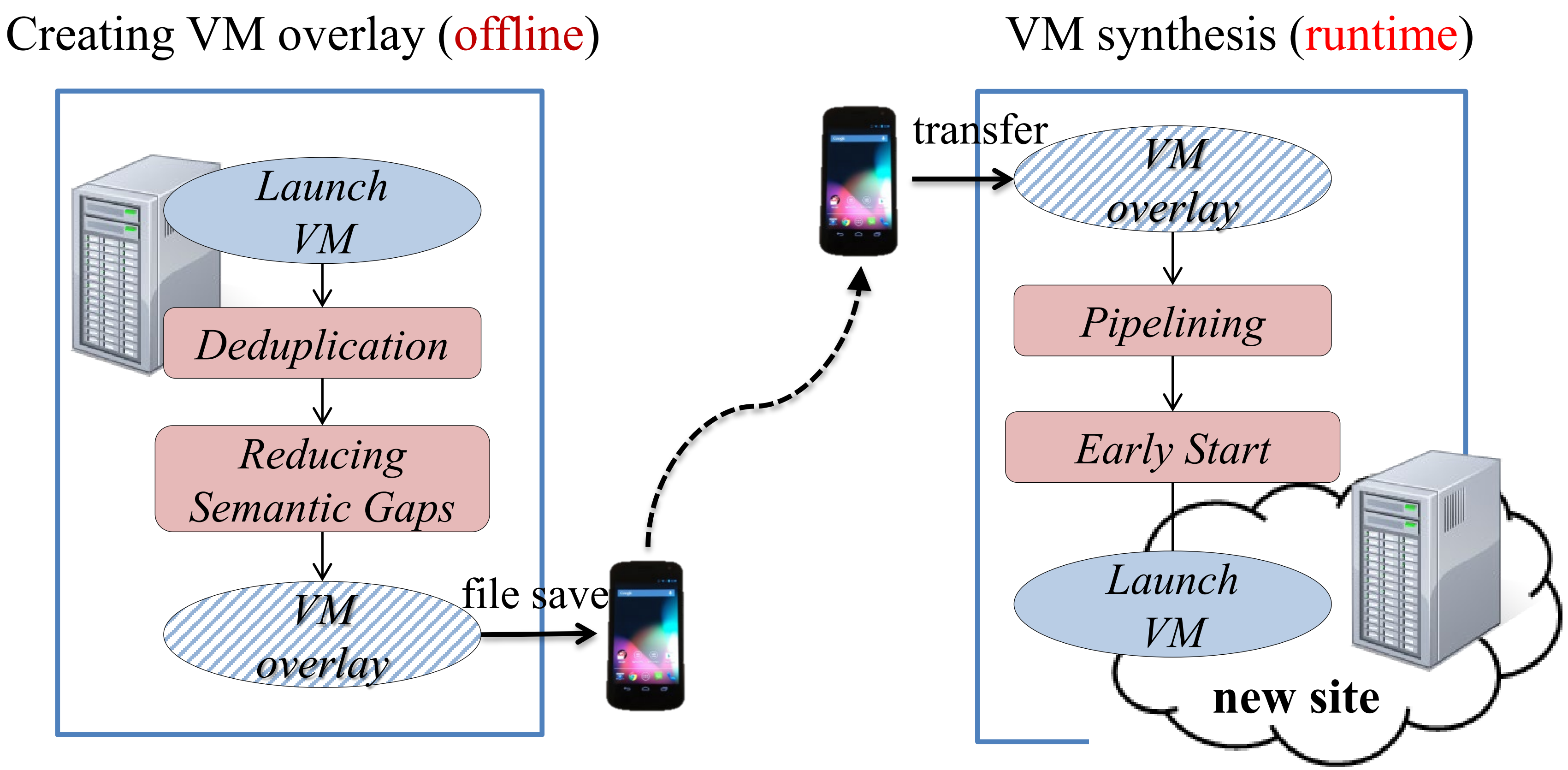
Application	Install size (MB)	Overlay Size		Synthesis time (s)
		Disk (MB)	Memory (MB)	
OBJECT	39.5	92.8	113.3	**62.8**
FACE	8.3	21.8	99.2	**37.0**
SPEECH	64.8	106.2	111.5	**63.0**
AR	97.5	192.3	287.9	**140.2**
FLUID	0.5	1.8	14.1	**7.3**

Overlay size reduced by order of magnitude

What does this table tell us?

Overview of Optimizations

1. Minimize *VM overlay* size
2. Accelerate VM synthesis



Deduplication

Approach

- Remove redundancy in the VM overlay
  – problem: same bits in *base VM* and *VM overlay* but in different locations in the respective images => delta fails
- Sources of redundancy

  Within *base VM*
  - Shared library copied from base disk
  - Loaded executable binary from base disk

  Between *VM overlay's memory* and *disk*
  - Page cache, disk I/O buffer

Deduplication

1. Get the list of modified (disk, memory) chunks at the *customized VM* *(delta)*
2. Perform deduplication to reduce this list to a minimum

Compare to *1)* ***base disk***, 2) ***base memory***, 3) other chunks within **itself**

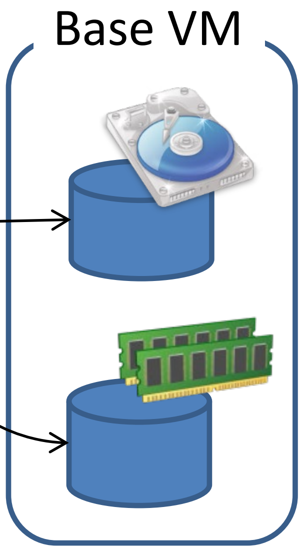
Compare between ***modified memory*** *and* ***modified disk***

type	offset	Data
mem	4096	data
mem	16384	data
mem	20480	data
mem	28672	data
mem	36864	data
…		
disk	0	data
disk	16384	data

<Modified chunks>

type	offset	reference	Data or pointer
mem	4096	unique	data
mem	16384	self	
mem	20480	Base disk	
mem	28672	Base memory	
mem	36864	unique	data
…			
disk	0	unique	data
disk	16384	overlay mem	

<Overlay chunks>



Offline operation!

Dedup Results

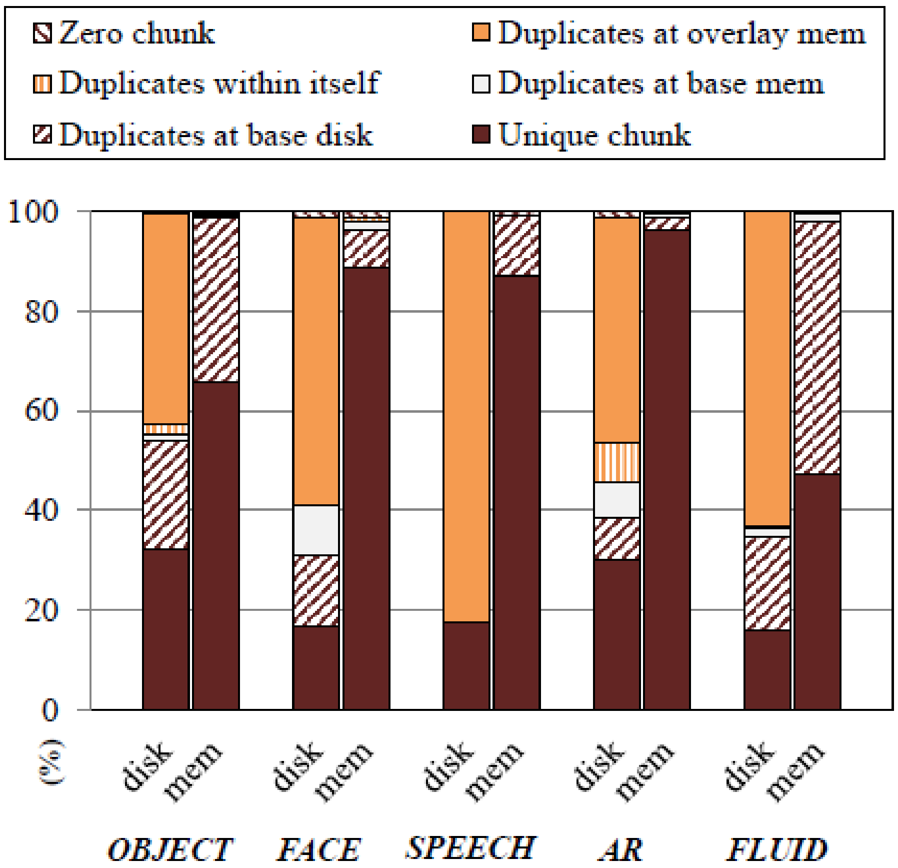


Figure 4: Benefit of Deduplication

Reducing Semantic Gaps

VM is a black box

- VMM cannot interpret high-level information of memory and disk

E.g: Download 100 MB file over network and delete it

- Ideally, it should result in no increase in VM overlay size
- However, VMM will see **200 MB of modifications:**
  - 100 MB of changed disk state
  - 100 MB of changed memory state (in-memory I/O buffer cache)

→ Include only the state that actually matters to the guest OS

Reducing Semantic Gaps – Disk

Disk semantic gap bet. VMM and Guest OS

- File deletion operations only mark blocks as deleted, without discarding the contents
- VMM can't distinguish between deleted and valid contents

Approach

- **Exploit TRIM commands**
  - Allows an OS to inform a disk device which blocks of data are no longer in use
  - **Captured the TRIM commands so host knows about deleted data**

- **File system introspection**
  - **Exploit knowledge of FS disk layout to find free-map, etc.**

Reducing Semantic Gaps – Memory

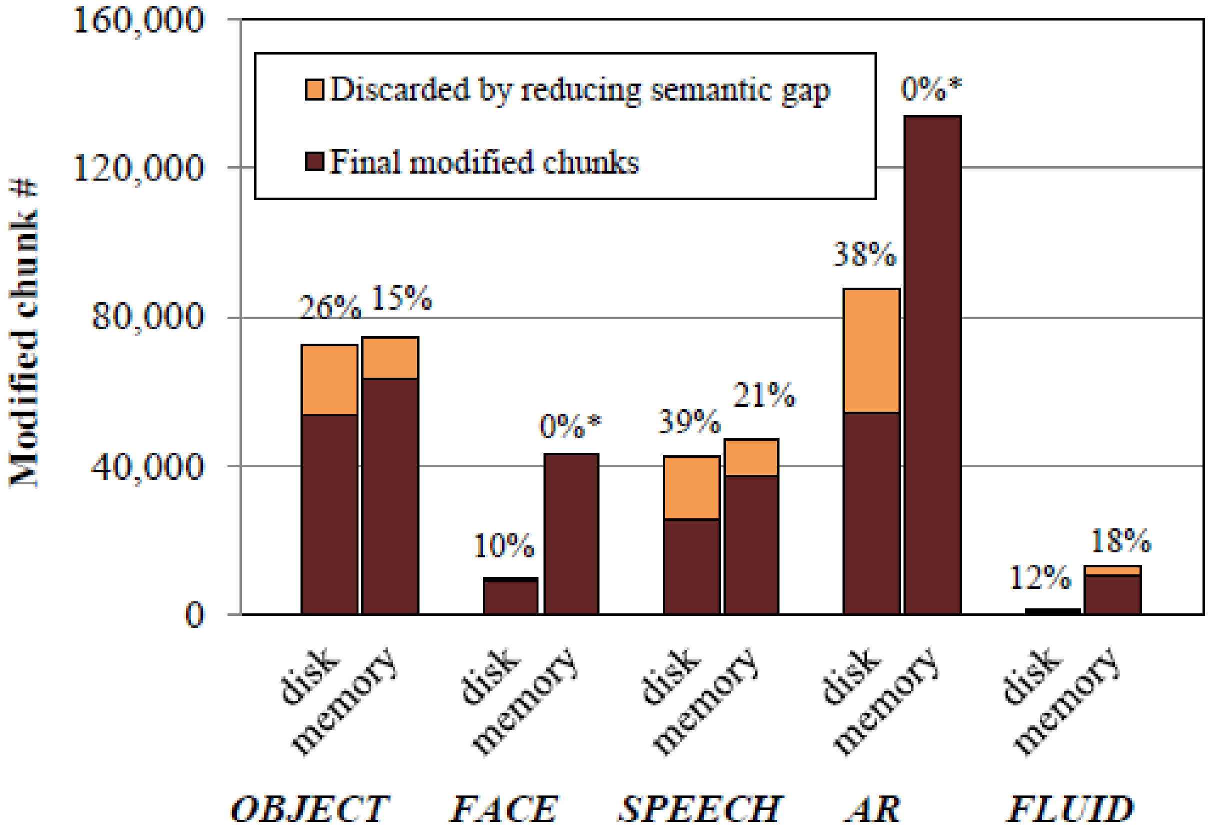
Memory semantic gap between VMM and Guest OS

- Released memory is moved to the OS's free page list, but is still filled with garbage
- VMM can't distinguish between valid memory and garbage data
- No way to communicate free page information between the guest and VMM

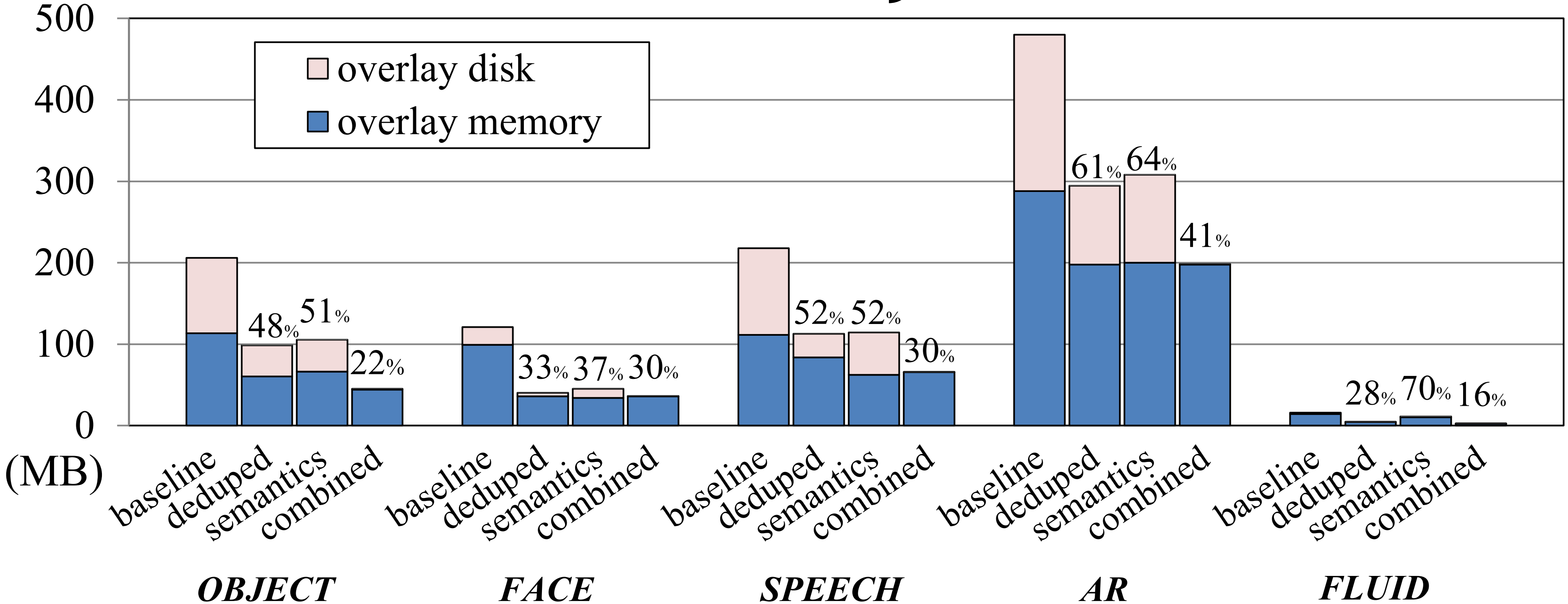
Approach

- Scan memory snapshot: locate frame free list data structure in kernel memory
- Requires kernel mods in guest OS (Linux only for now)

Semantic Gap Results



VM Overlay Size

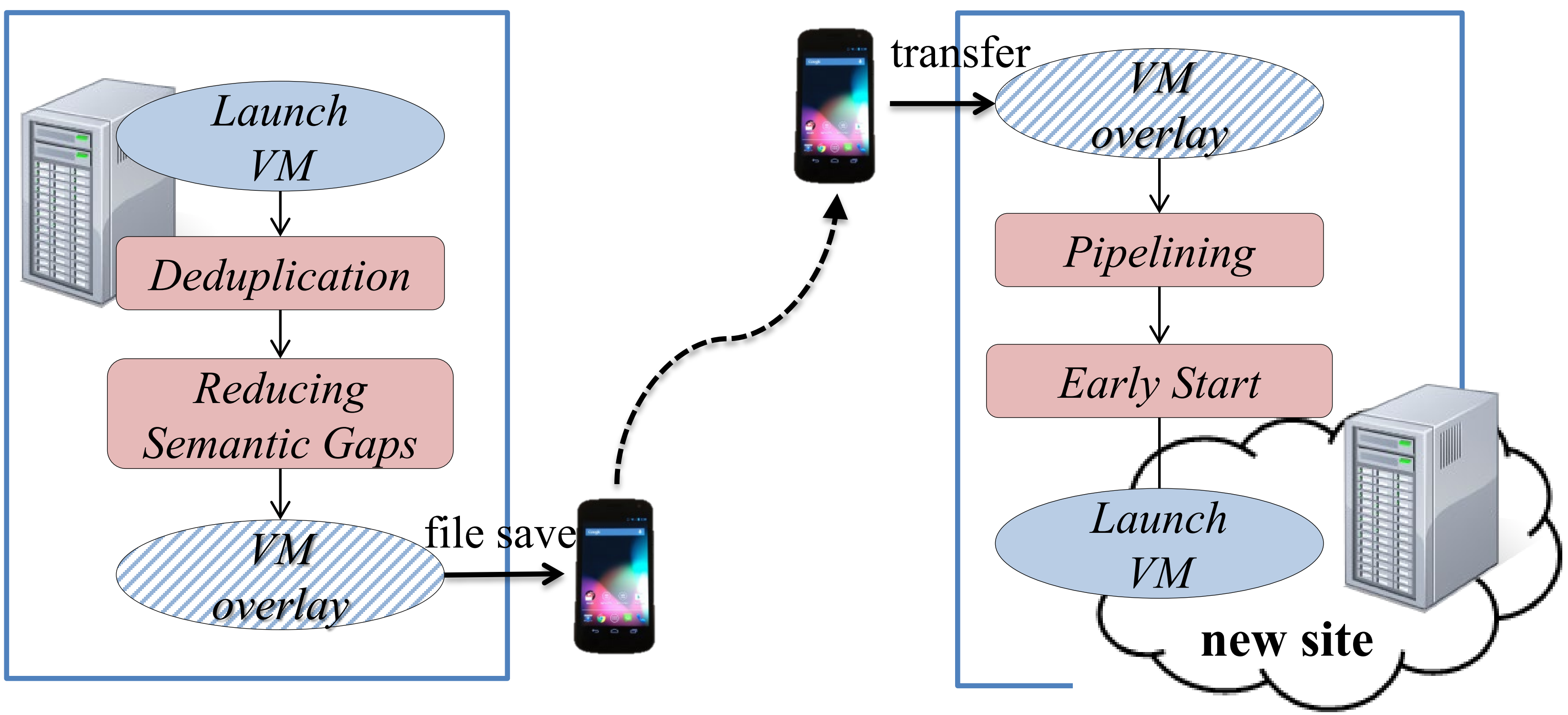


- Deduplication optimization reduces the VM overlay size to 44%
- Using semantic knowledge reduces the VM overlay size to 55%
- Both applied together, overlay size is reduced to **28% of *baseline***

Overview of Optimizations

1. Minimize *VM overlay* size ✓ Creating VM overlay (offline)
2. **Accelerate VM synthesis**

VM synthesis (runtime)

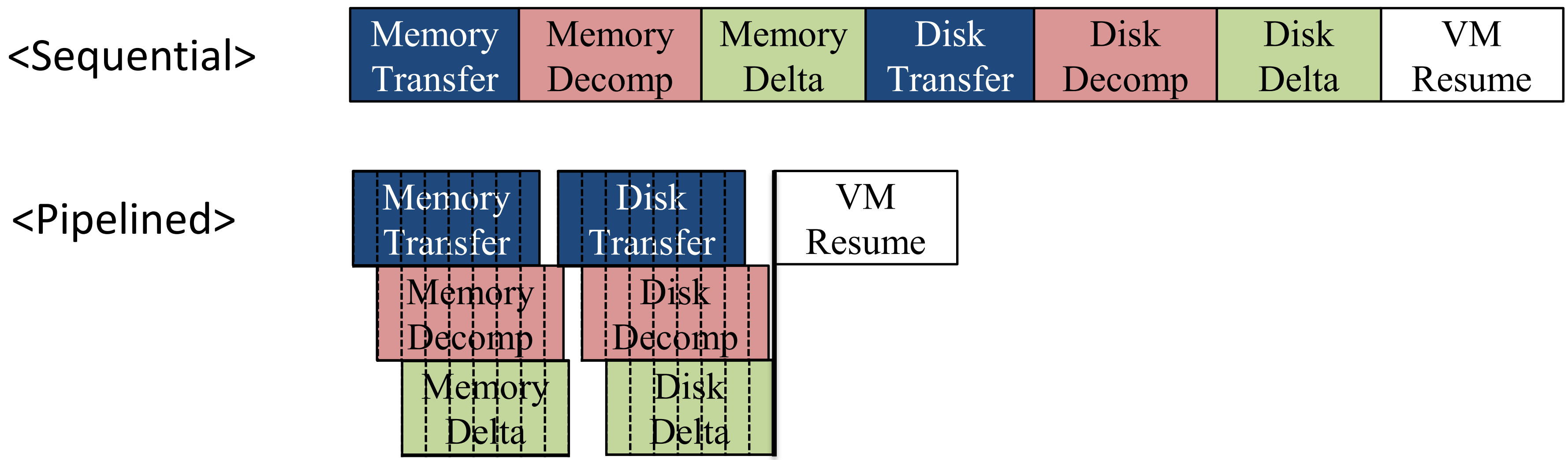


VM synthesis time is still too large

Pipelining

- Steps for VM synthesis

  ① Transfer VM overlay ② Decompress ③ Apply delta



- Unit of transfer: segment. How big?
  - Bigger more efficient; finer better on latency

Pipelining Results

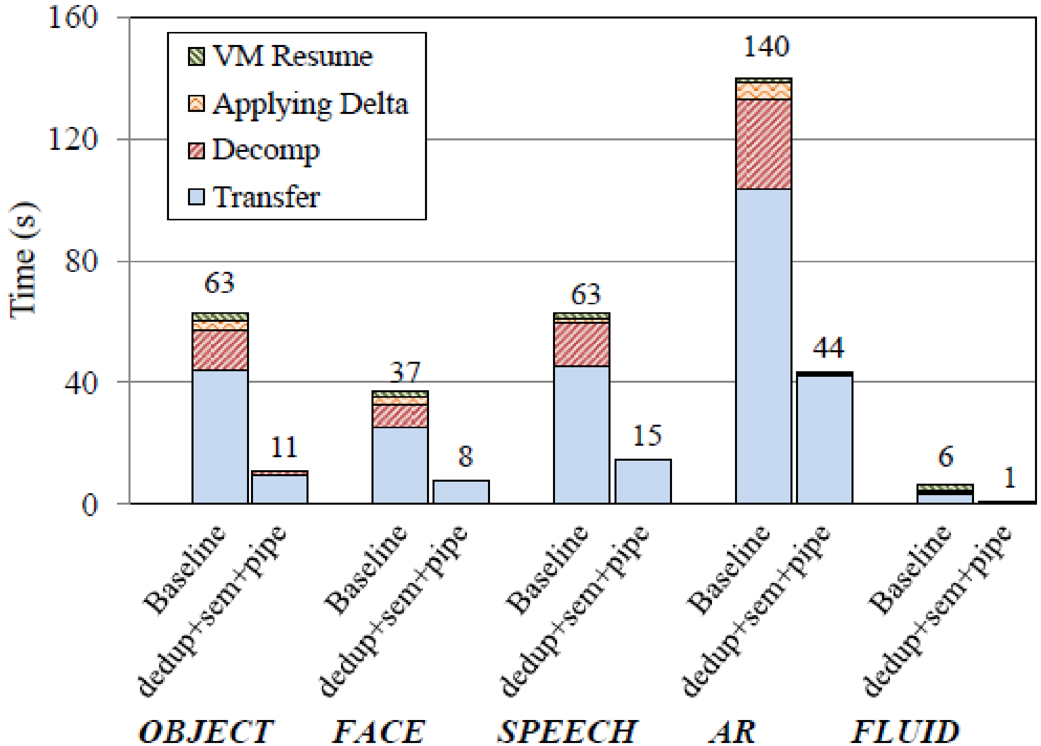


Figure 9: Effect of Pipelining + Earlier Optimizations

Early Start

Idea

- From user's perspective, first response time of offloading is most important
- Starting VM even before finishing VM synthesis

→Do not wait until VM synthesis finishes, but start offloading immediately and process the request while synthesis is ongoing

Early Start

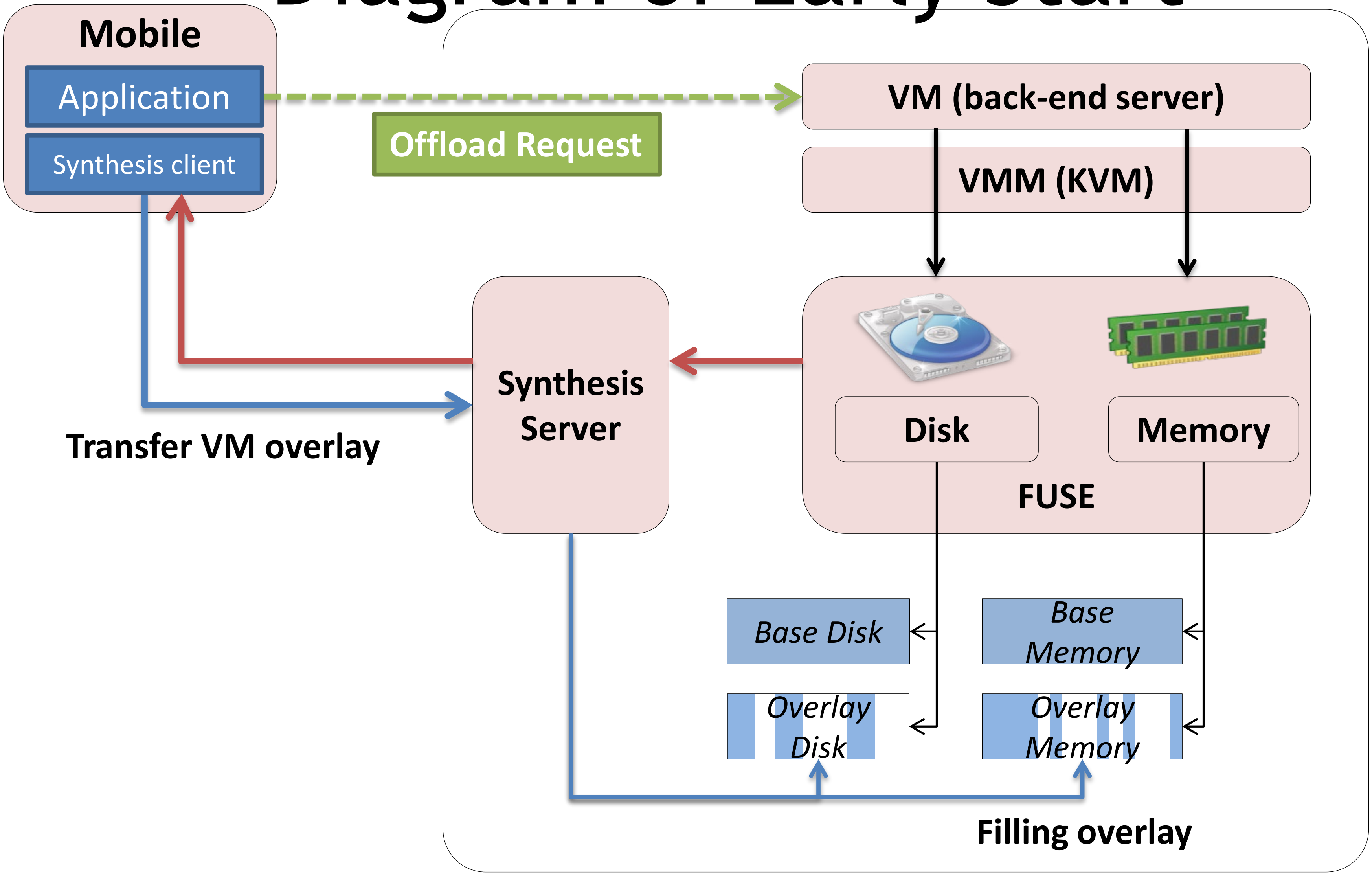
Approach

1) **Reorder the chunks in estimated access-order**
2) Break the ordered overlay into **smaller segments for demand fetching**

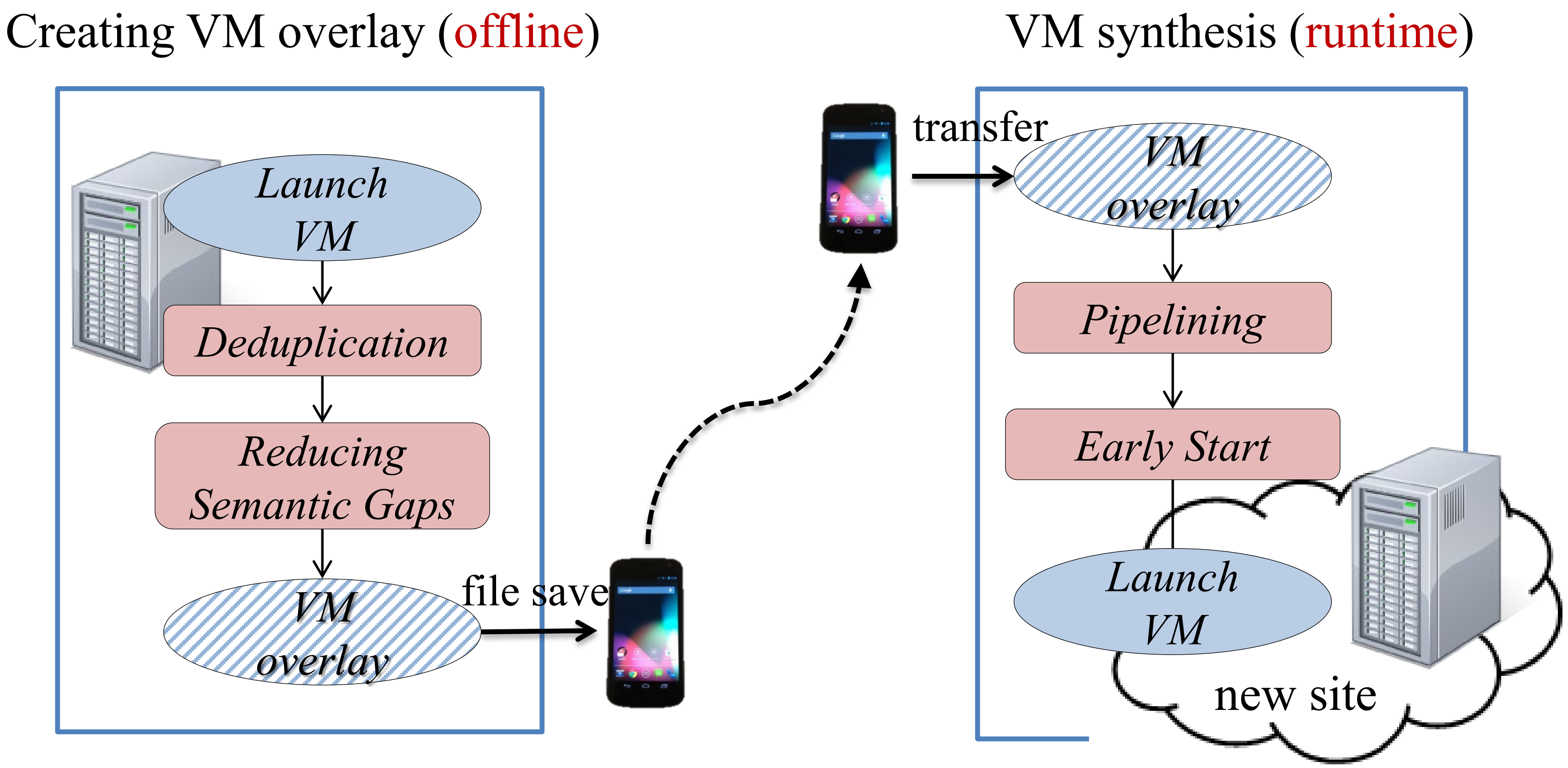
→ Start the VM and begin streaming the segments in order, but also allow out-of-order demand fetches to preempt the original ordering

Downside of demand fetching?

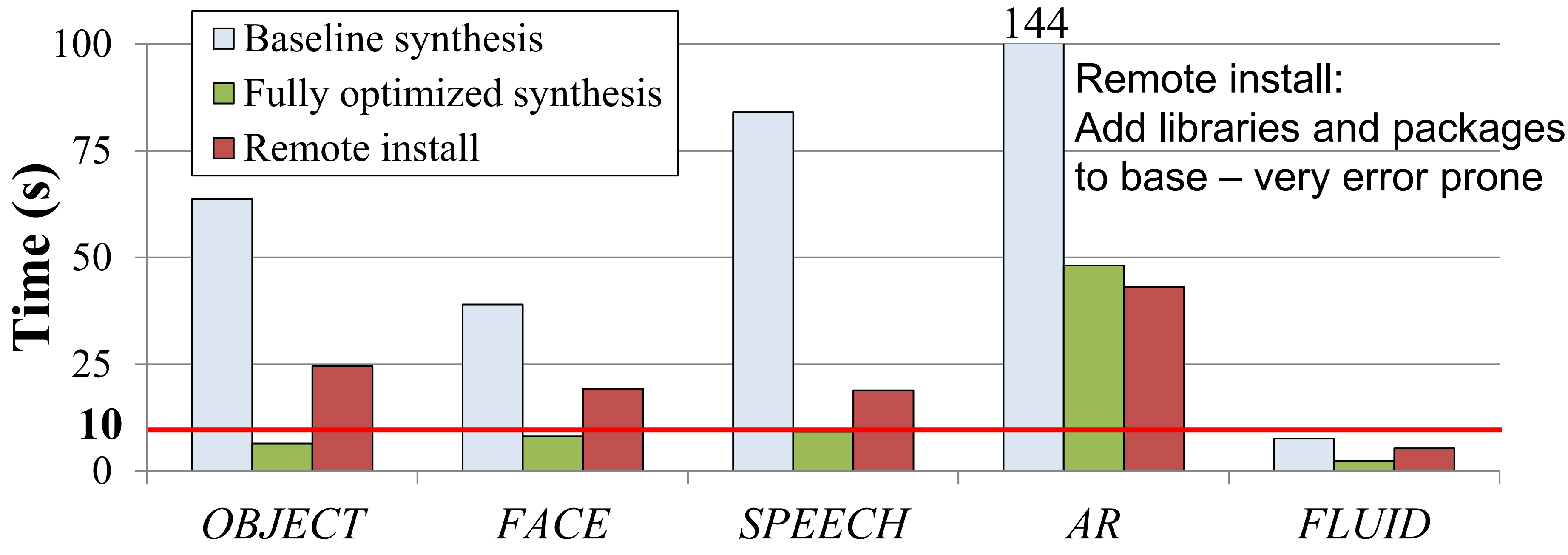
Diagram of Early Start



Review of Optimizations



First-response vs. *baseline*



* Chunks are ordered with segment size of 1 MB

Time between starting VM synthesis and receiving the first offload result

- It is faster than remote installation
- Except *AR*, we can get first-response within 10 seconds (up to 8x improvement)

Next week

Edge Fault Tolerance

Volunteers please?

Have a great weekend!