

Resource-Centric Serverless Computing

Yiying Zhang

Outline

- PL for disaggregation: Mira [SOSP'23]
- Serverless on disaggregation: Zenix
- Quiz 2 this Wed (covers lectures from cloud computing to today's)
- Project progress report due this Thur

Existing Far Memory Systems

- How to **run applications with** far memory?
- Existing approaches in two broad categories:

1. New memory programming model:

AIFM [OSDI22], Clio [ASPLOS22]

Application

2. Transparent far memory:

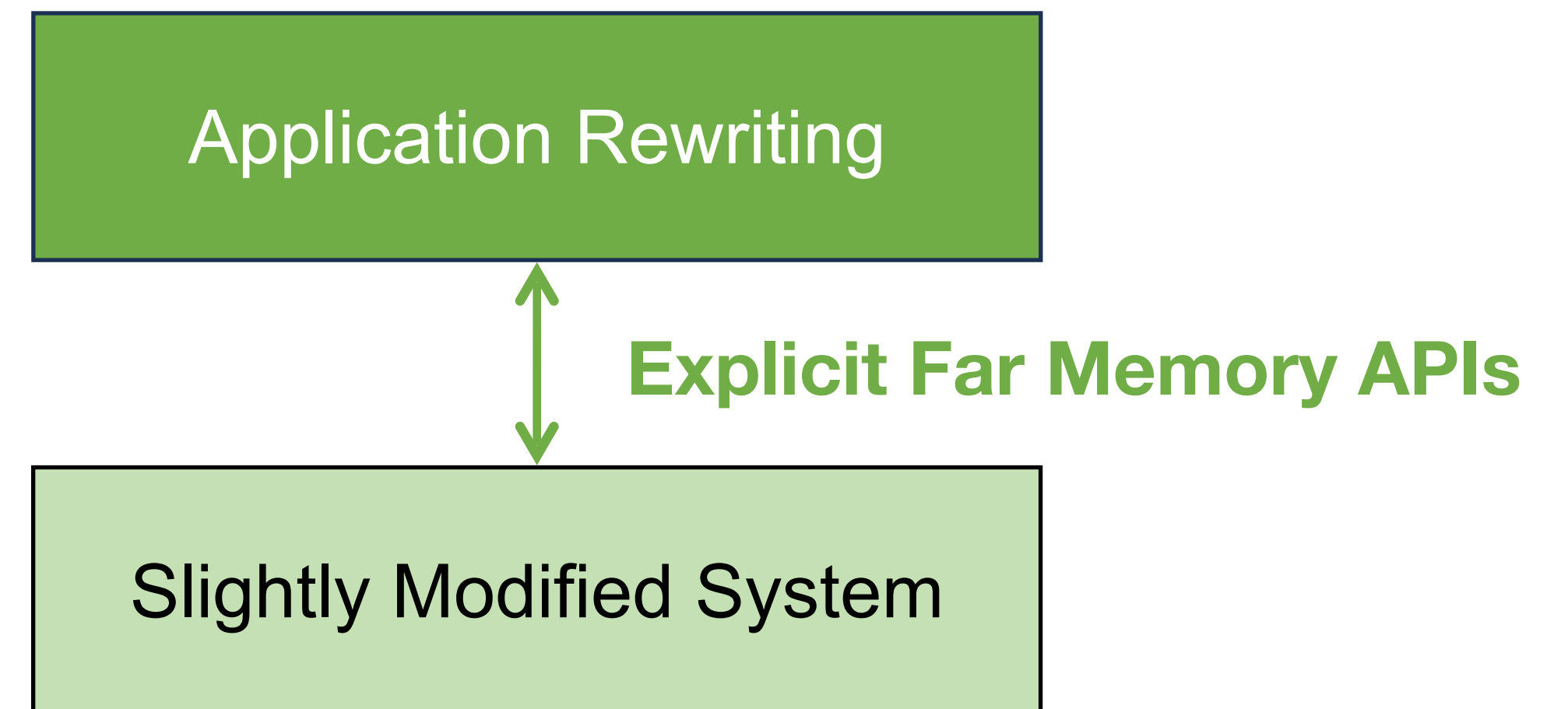
LegoOS [OSDI18], FastSwap [EuroSys21], MemLiner [OSDI22]

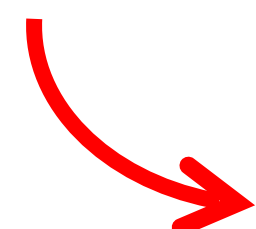
System

New Far Mem Programmng Model

- Programs rewritten to access far memory with new APIs

- + Explicitly express access pattern
 - Require programmers' effort
 - Cannot adapt to dynamic behavior



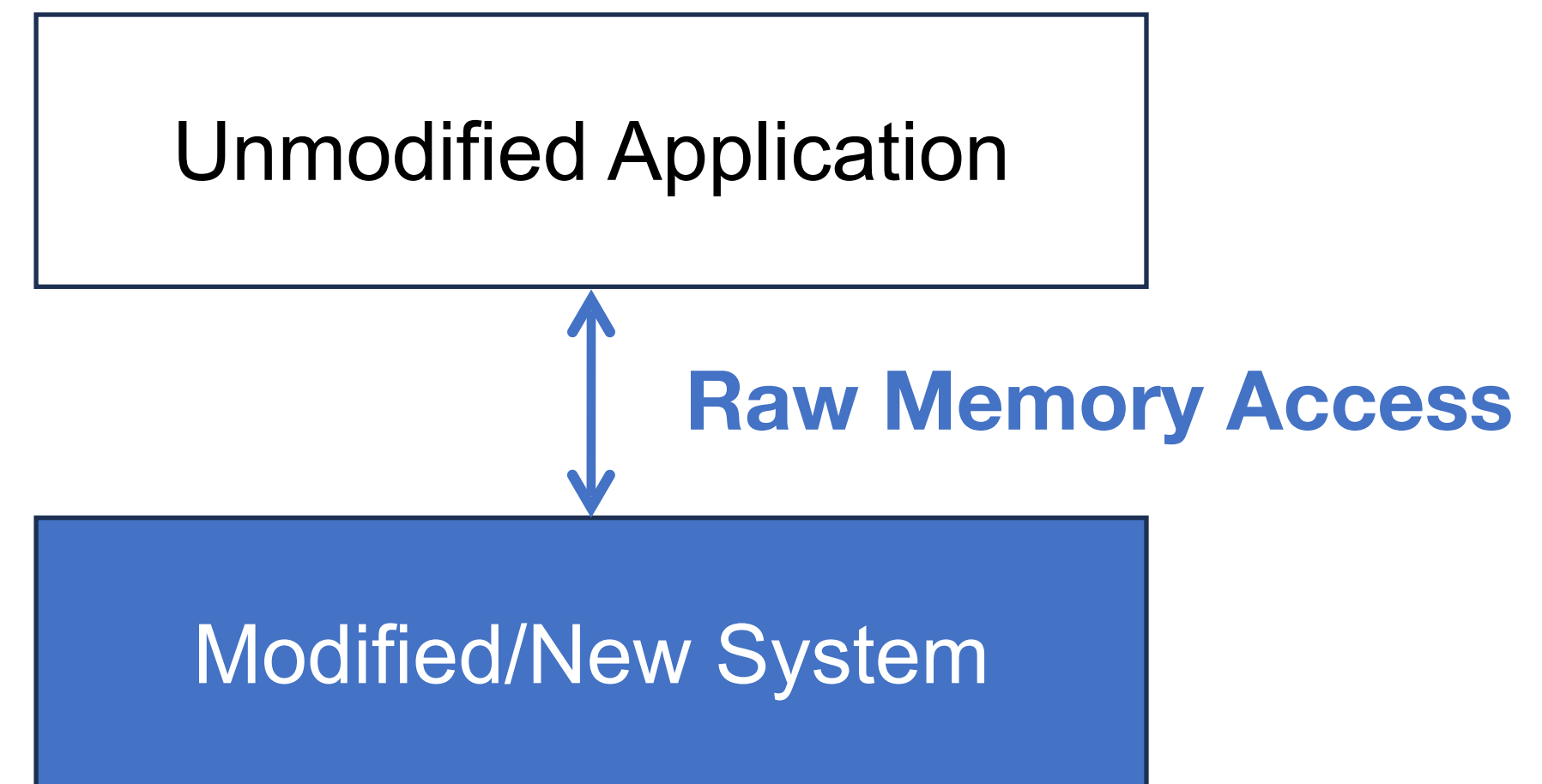
 **Overhead on both human and runtime**

Transparent Far Memory

- Transparently swap to far memory at runtime or OS layer

+ No program changes

- Unaware of application semantics
- No adaptation to different applications



Inefficient far memory operation
(read/write amplification, no or wrong prefetch, ...)

How to build a transparent and efficient far memory system?

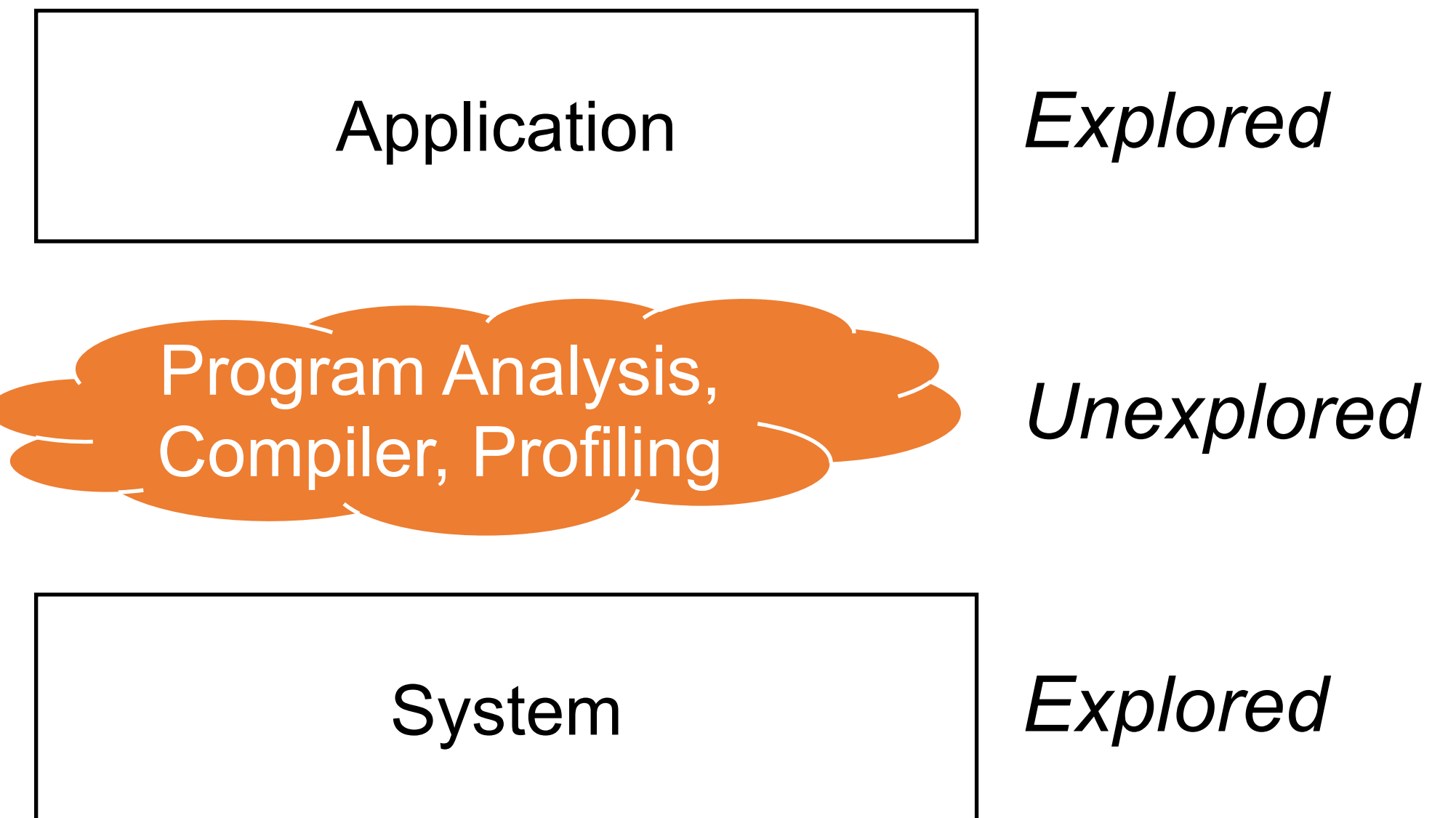
Application

*Explored,
Non-transparent*

System

*Explored,
Inefficient*

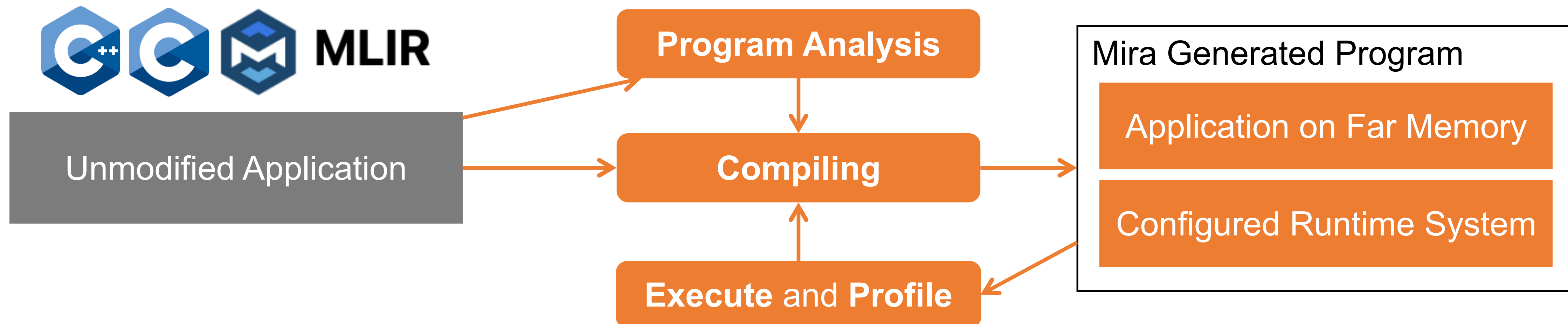
Solution: Behavior-Guided Far Memory via Program Analysis, Compiling, and Profiling



Mira: Behavior-Guided Far Memory

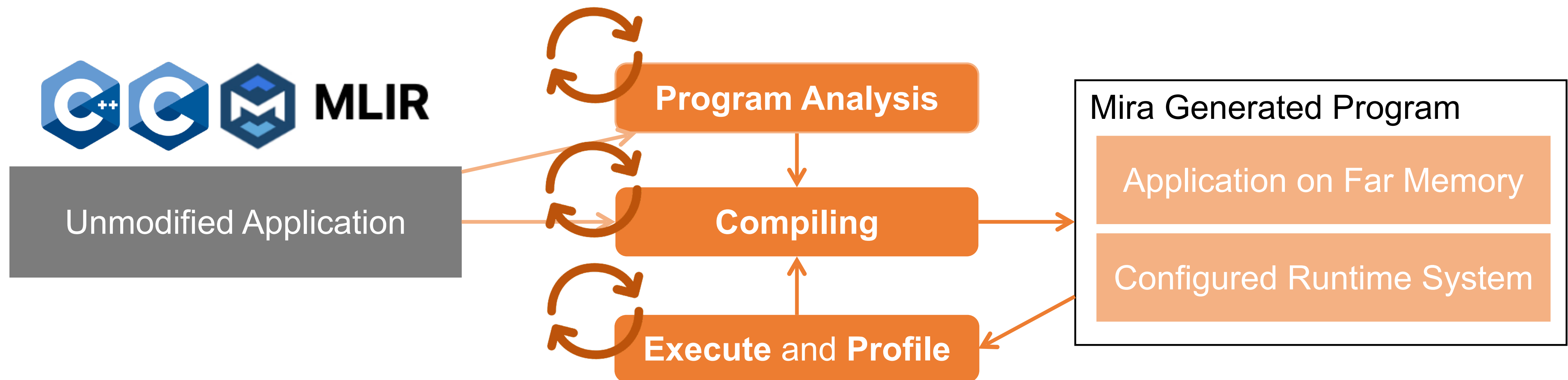


- **Program analysis** for capturing static behavior
access sequence, locality ...
- **Profiling** for capturing dynamic behavior
Object size, execution overhead ...
- **Compiler** for behavior-guided transparent far memory generation



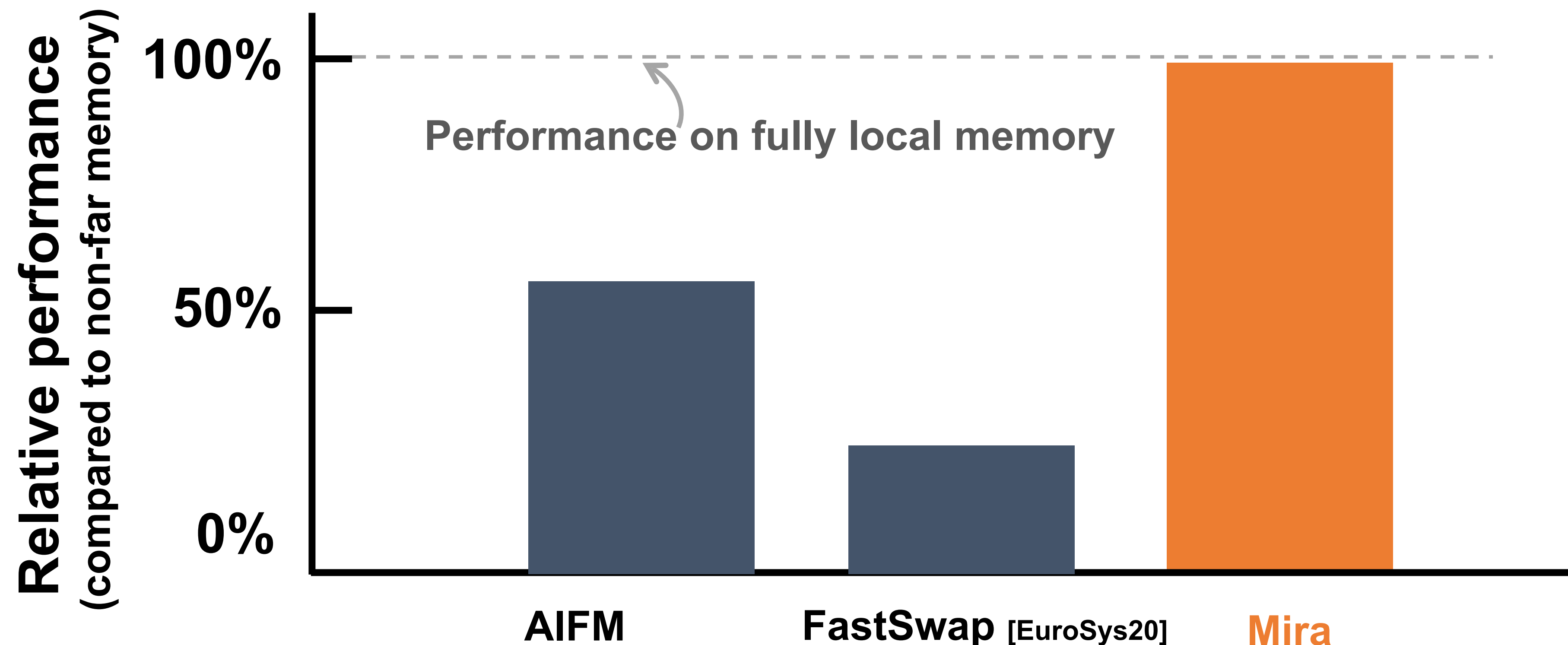
Mira: Behavior-Guided Far Memory

- **Program analysis** for capturing **static behavior**
- **Profiling** for capturing **dynamic behavior**
- **Compiler** for behavior-guided **transparent** far memory **generation**



Mira: Behavior-Guided Far Memory

- Captures and utilizes a full view of **program behavior**
- Outperforms state-of-the-art far memory systems by 43% to 70%



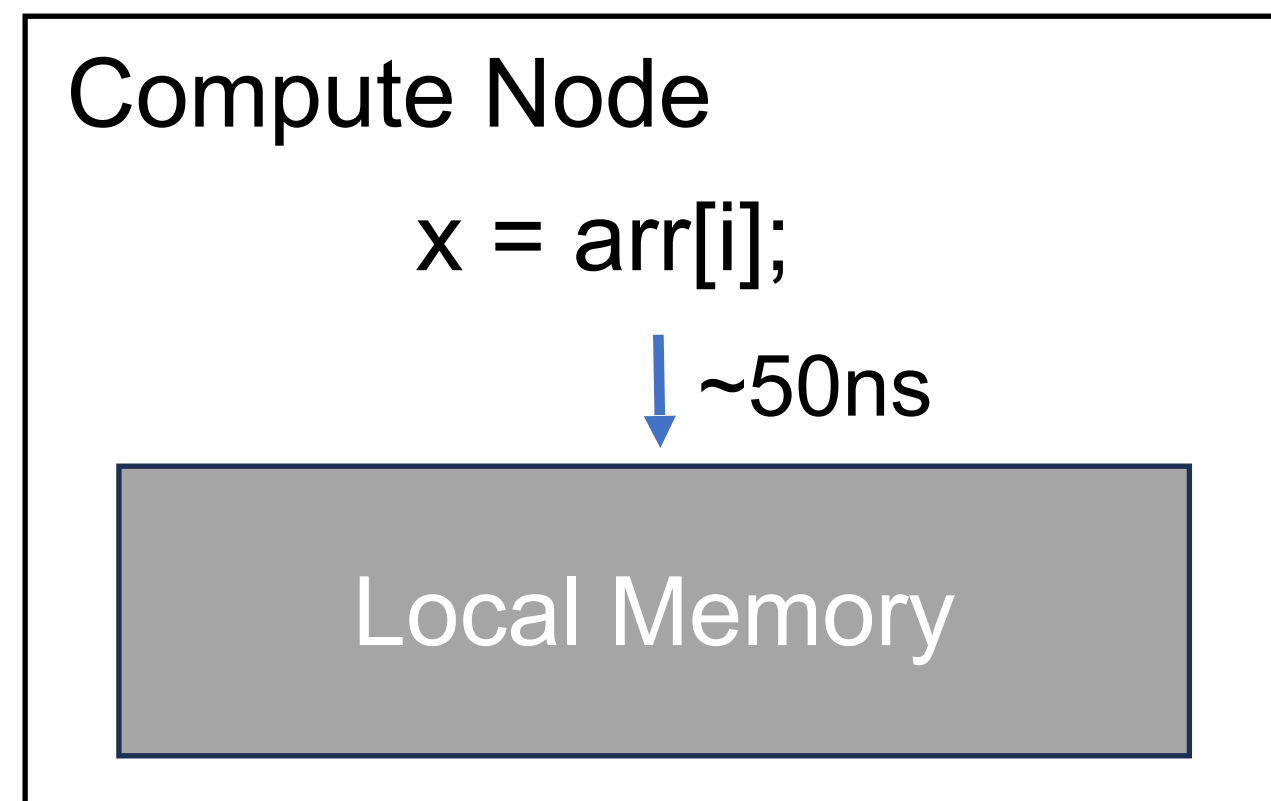
* Data collected on the DataFrame application, with local memory size 20%

Challenge:

**How to generate efficient far memory program
based on program behaviors?**

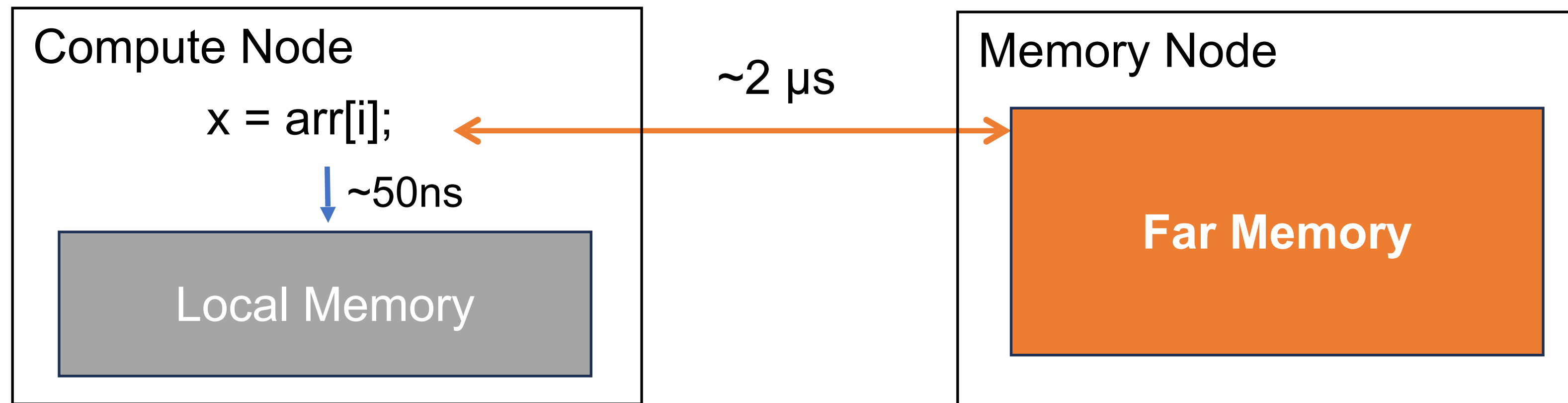
The Key to Far Memory Performance

- Far memory has **non-trivial overhead**



The Key to Far Memory Performance

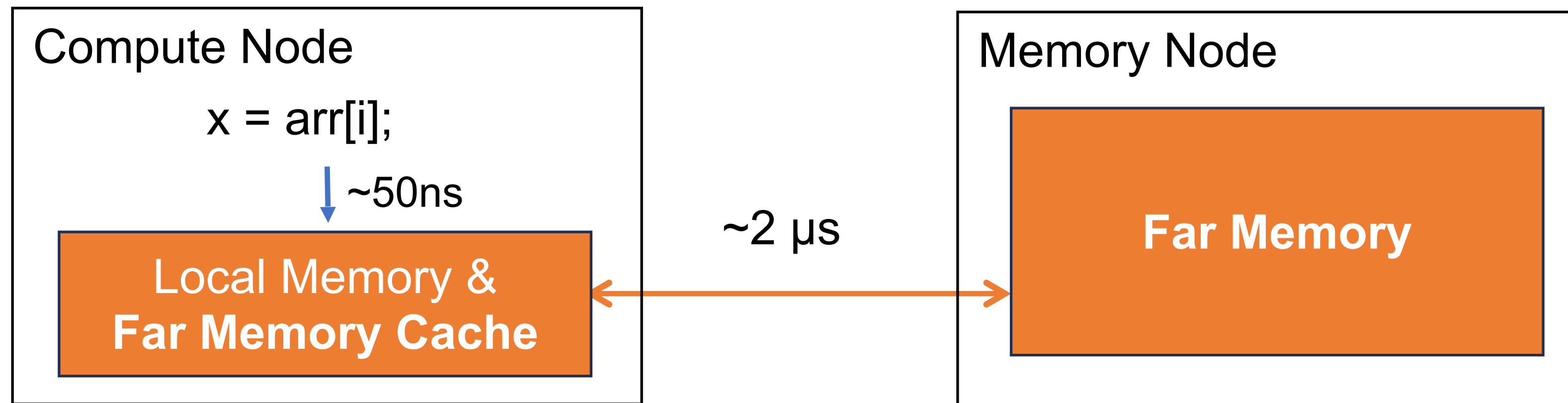
- Far memory has **non-trivial overhead**



The Key to Far Memory Performance

- Far memory has **non-trivial overhead**
- **Accessing and caching** policies are key to far memory performance

Problem: Existing system use **single unified global cache**



Problem of a Unified, Global Cache

- Objects with different behaviors **co-exists** in a single program

```
arr = vector<int>;  
table = hash<int, int>;  
for (i <- 0 to len(arr)){  
    idx = arr[i];  
    acc += table[idx];  
}
```

Sequential, Accessed once, ...

Random, No spatial locality,
Non-predictable temporal locality ...

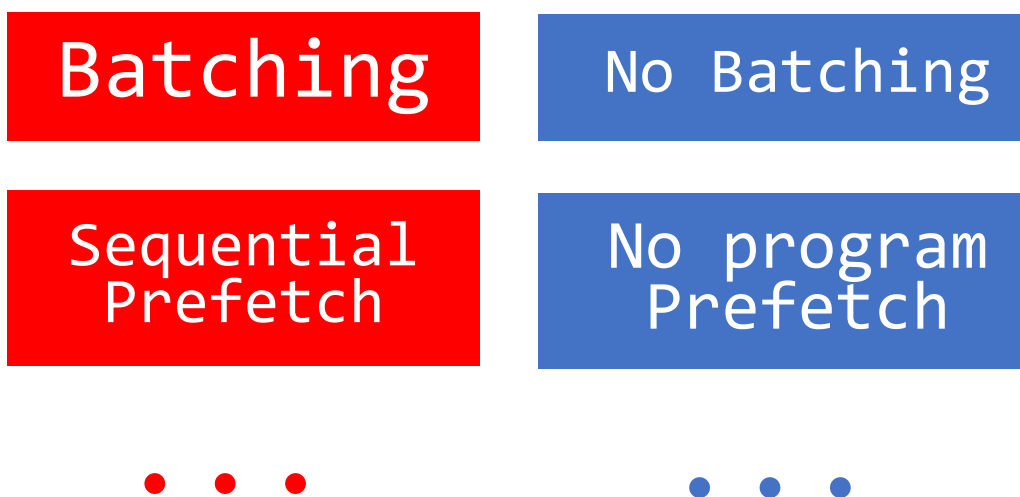
Need to capture both behaviors

Problem of a Unified, Global Cache

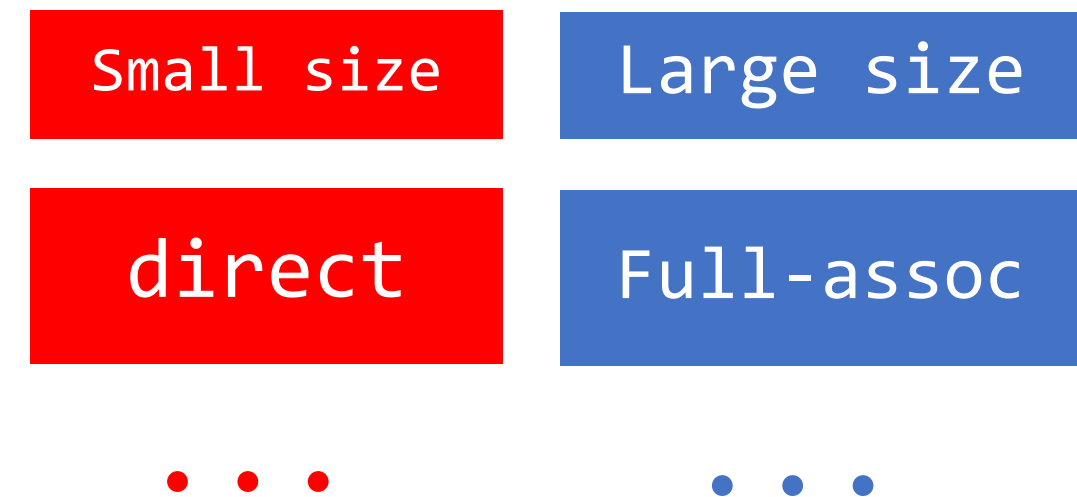
- Objects with different behaviors **co-exists** in a single program
- Different behaviors require different **access and cache** configurations

```
arr = vector<int>;  
table = hash<int, int>;  
for (i <- 0 to len(arr)){  
    idx = arr[i];  
    acc += table[idx];  
}
```

Optimal Access Policies



Optimal Cache Config



Unified cache cannot optimize for all

**Solution: Separate local cache into sections
based on program behaviors**

Solution: Separate Cache Sections

- Leveraging software-managed DRAM cache
- Each section customized for **one single** type of behavior

With different {cache size, cache structure, prefetch/eviction policy, ...}

```
arr = vector<int>
```

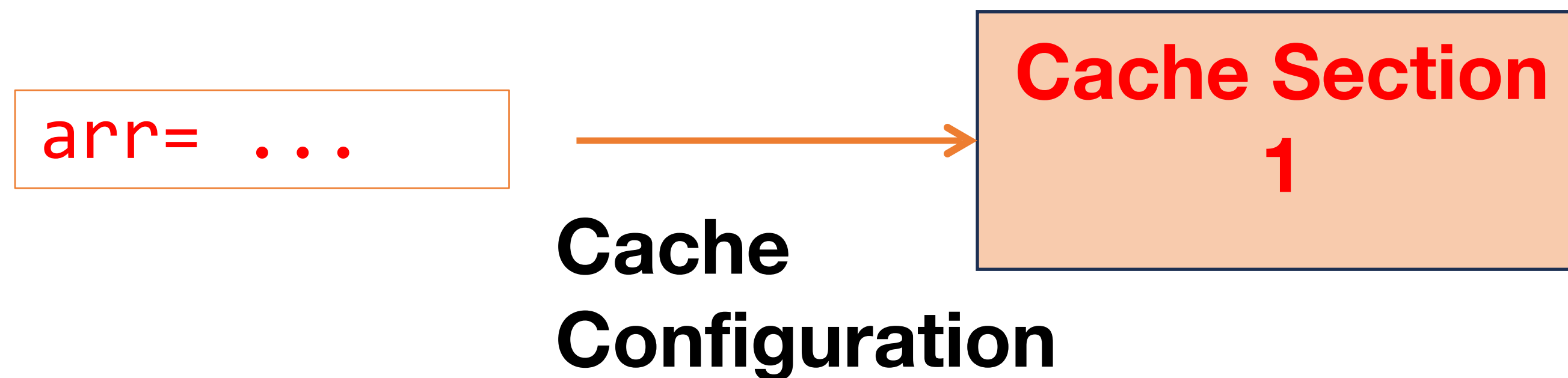
Cache Section 1

```
tab = hash<int, int>;
```

Cache Section 2

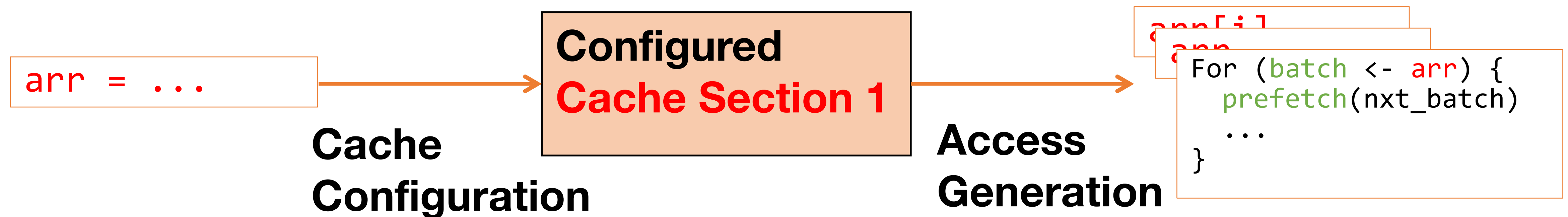
Using Separated Cache Sections

1. **Configure** each separated section according to its behavior



Using Separated Cache Sections

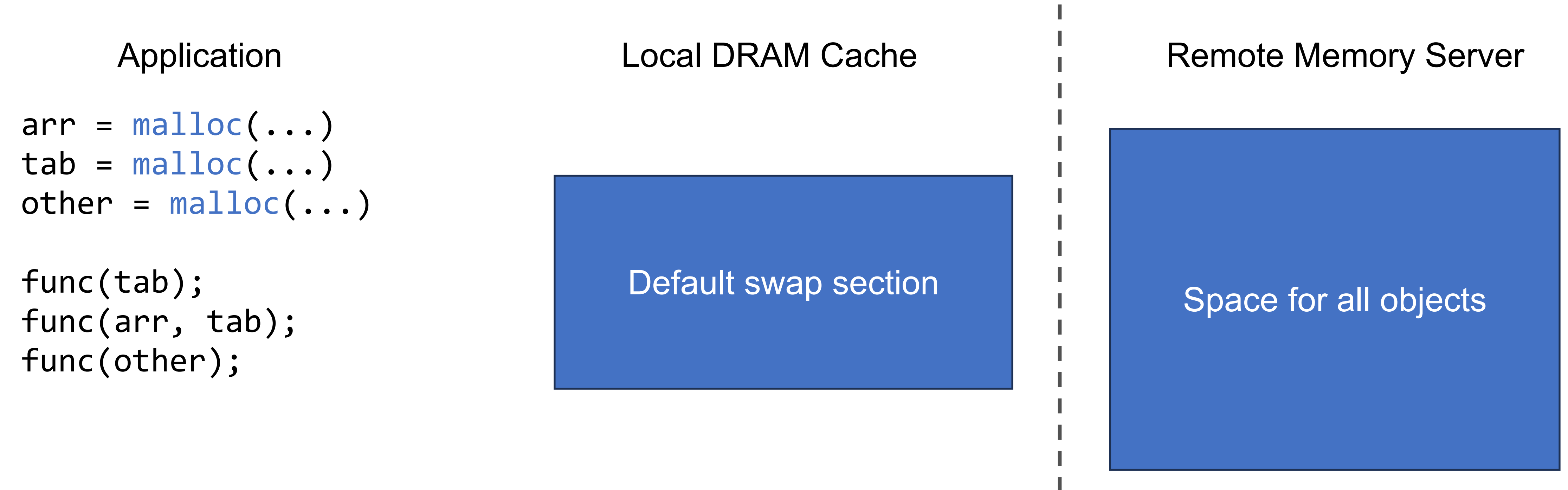
1. Configure each separated section according to its behavior
- 2. Generate** accesses based on section configuration and behavior



Iterative Cache Section Configuration

1st Iteration:

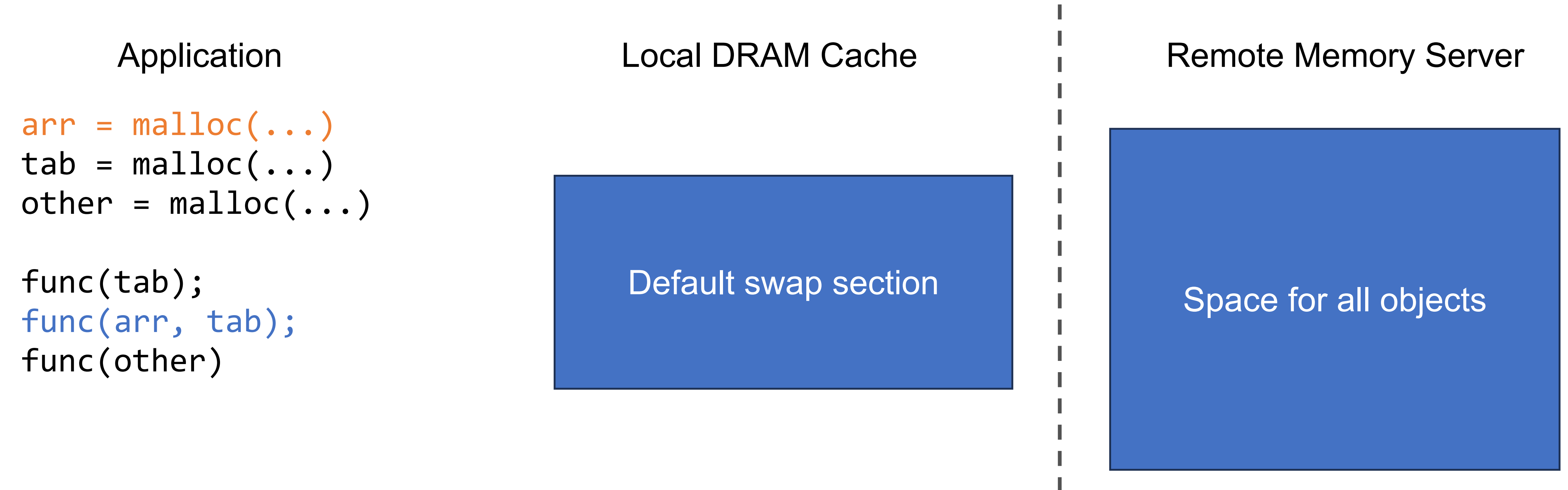
All data objects are placed into the generic swap section with profiling



Iterative Cache Section Configuration

2nd Iteration:

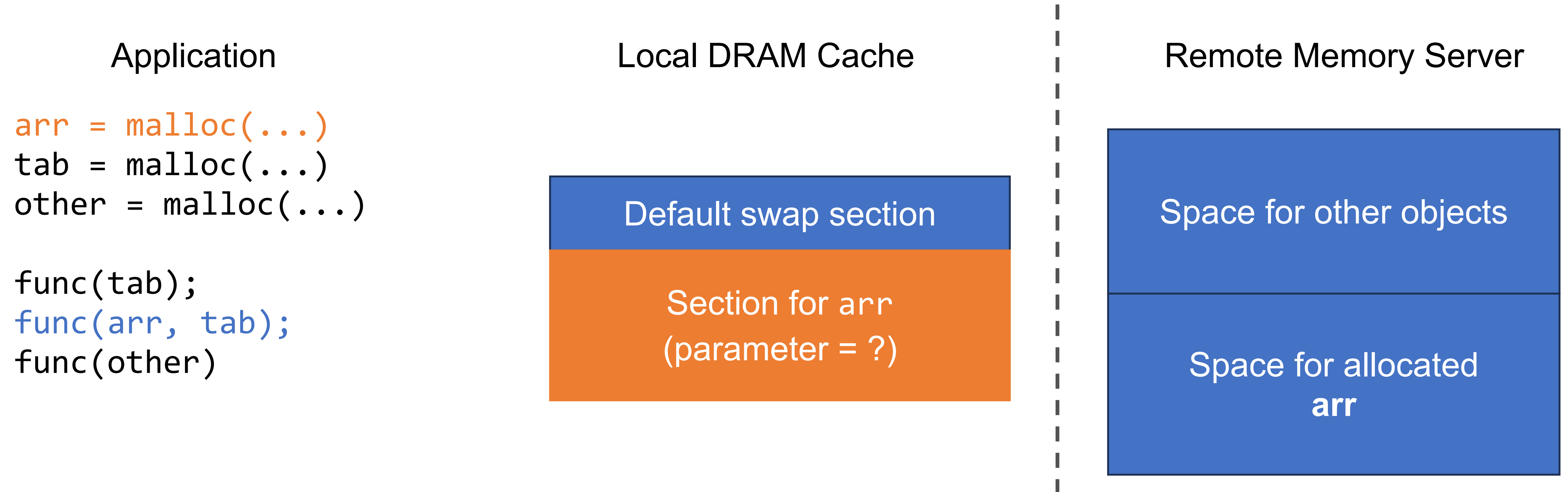
1. Profiling finds optimization targets: highest overhead **objects** and **functions**



Iterative Cache Section Configuration

2nd Iteration:

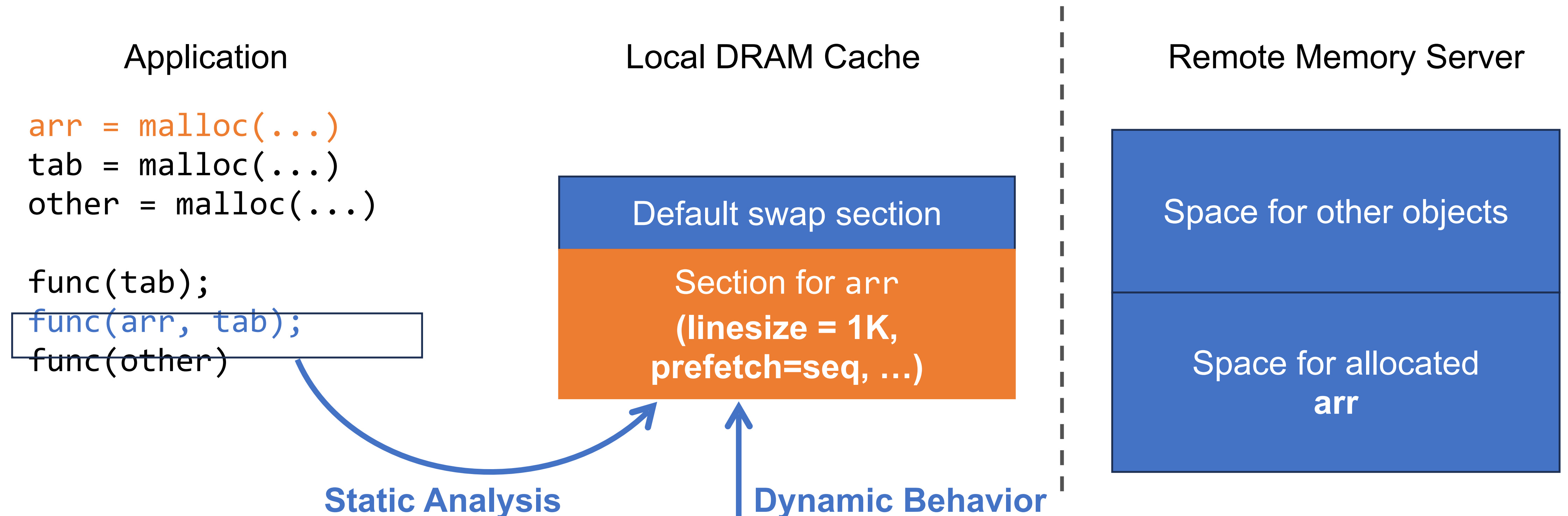
2. Separate cache sections for selected objects



Iterative Cache Section Configuration

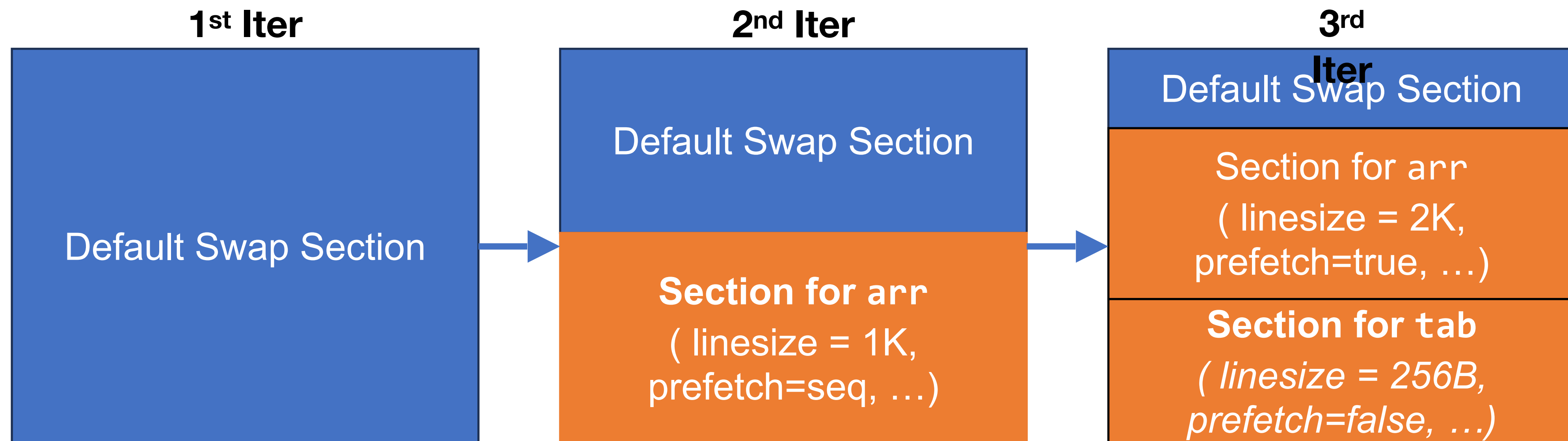
2nd Iteration:

3. Configure cache section based on static and dynamic behavior



Iterative Cache Section Configuration

- In later iterations, Mira continues the optimization loop
- Until no optimization target is found or maximum iteration is reached
- Evaluated applications converge **within 4 iterations**



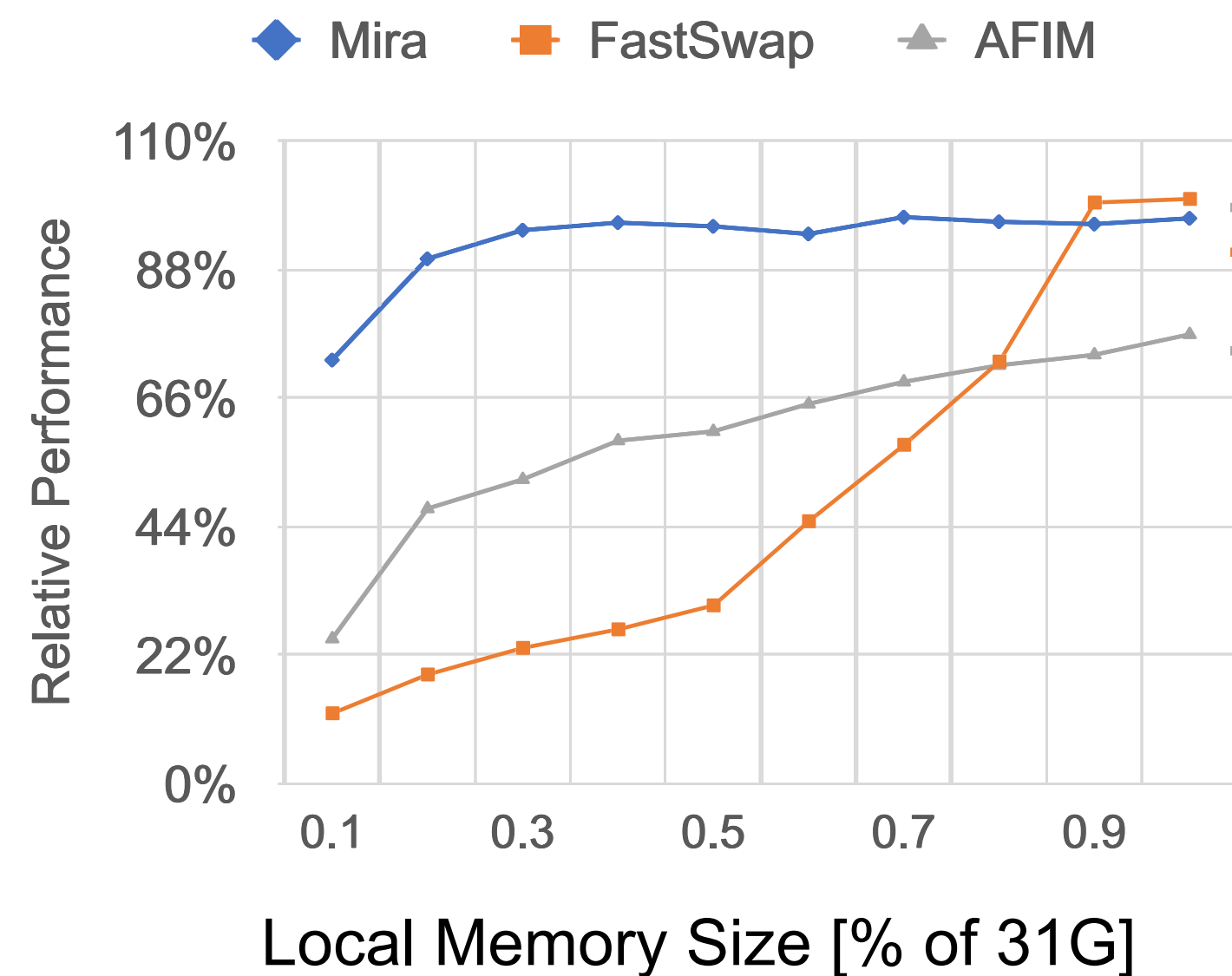
Implementation

- **Compiler** implemented on top of MLIR*
 - Implemented as two new dialects: **remotable** and **rmref**
- **Configurable runtime library** implemented in user space
 - With kernel-bypassing RDMA and `userfaultfd`
- LoC: 7.7K C++ (compiler), 12.1K C++ (runtime)

* MLIR = Multi-Level Intermediate Representation, <https://mlir.llvm.org/>

Evaluation

- Compared **Mira** with **FastSwap**[EuroSys 20] and AIFM[OSDI 20]
- On Dataframe, Mira achieves near-optimal performance



DataFrame Performance

Compared to **AIFM**,

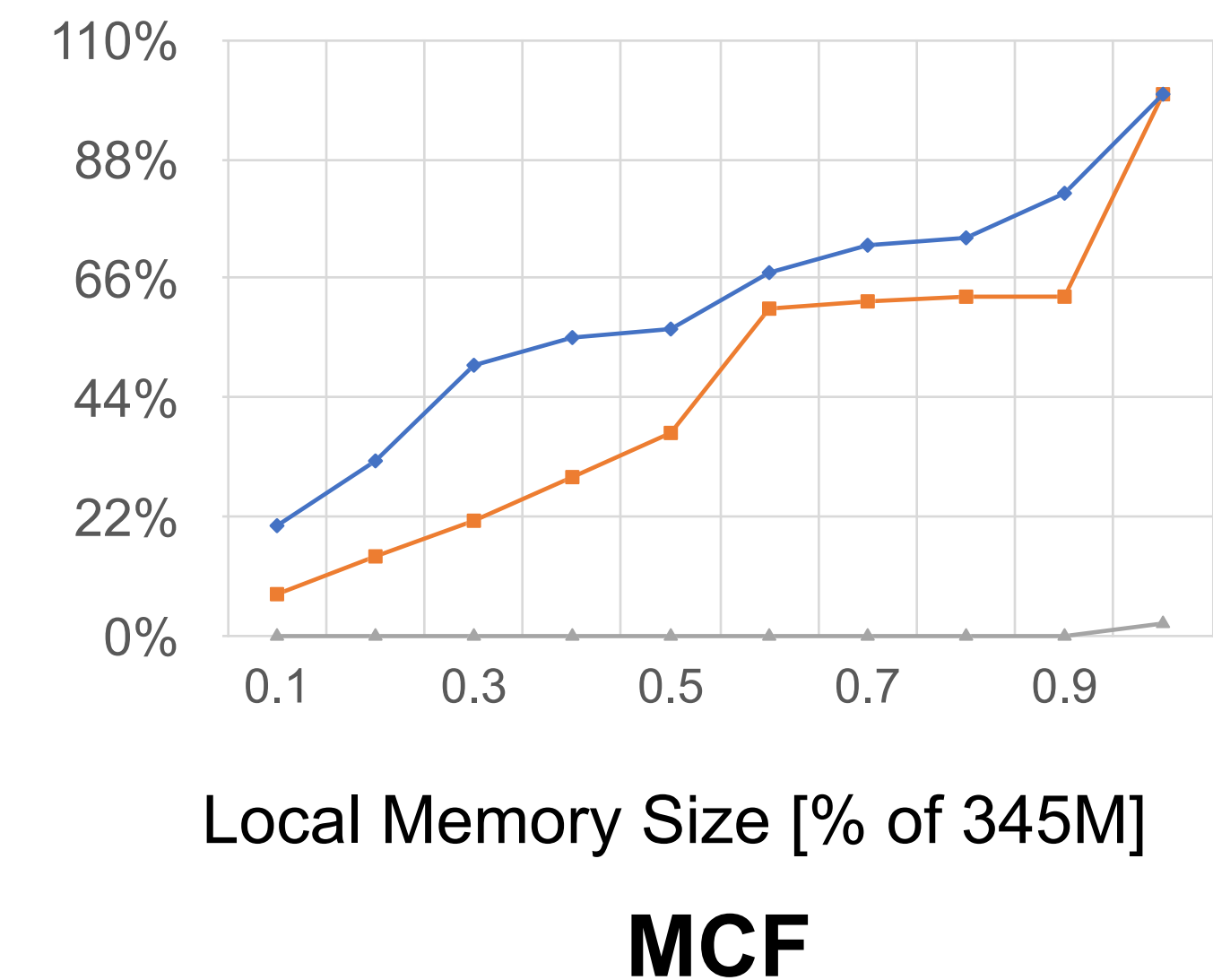
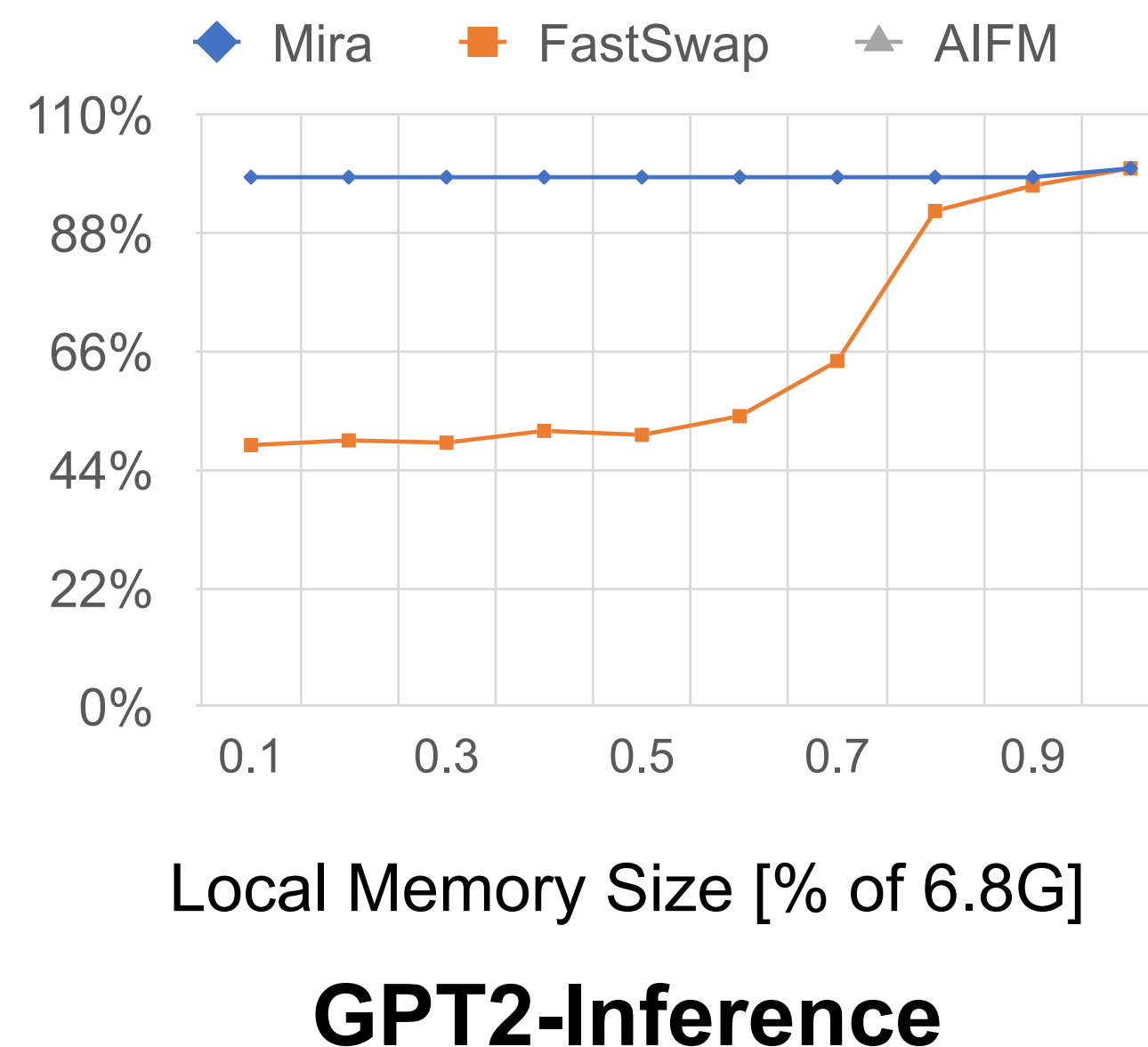
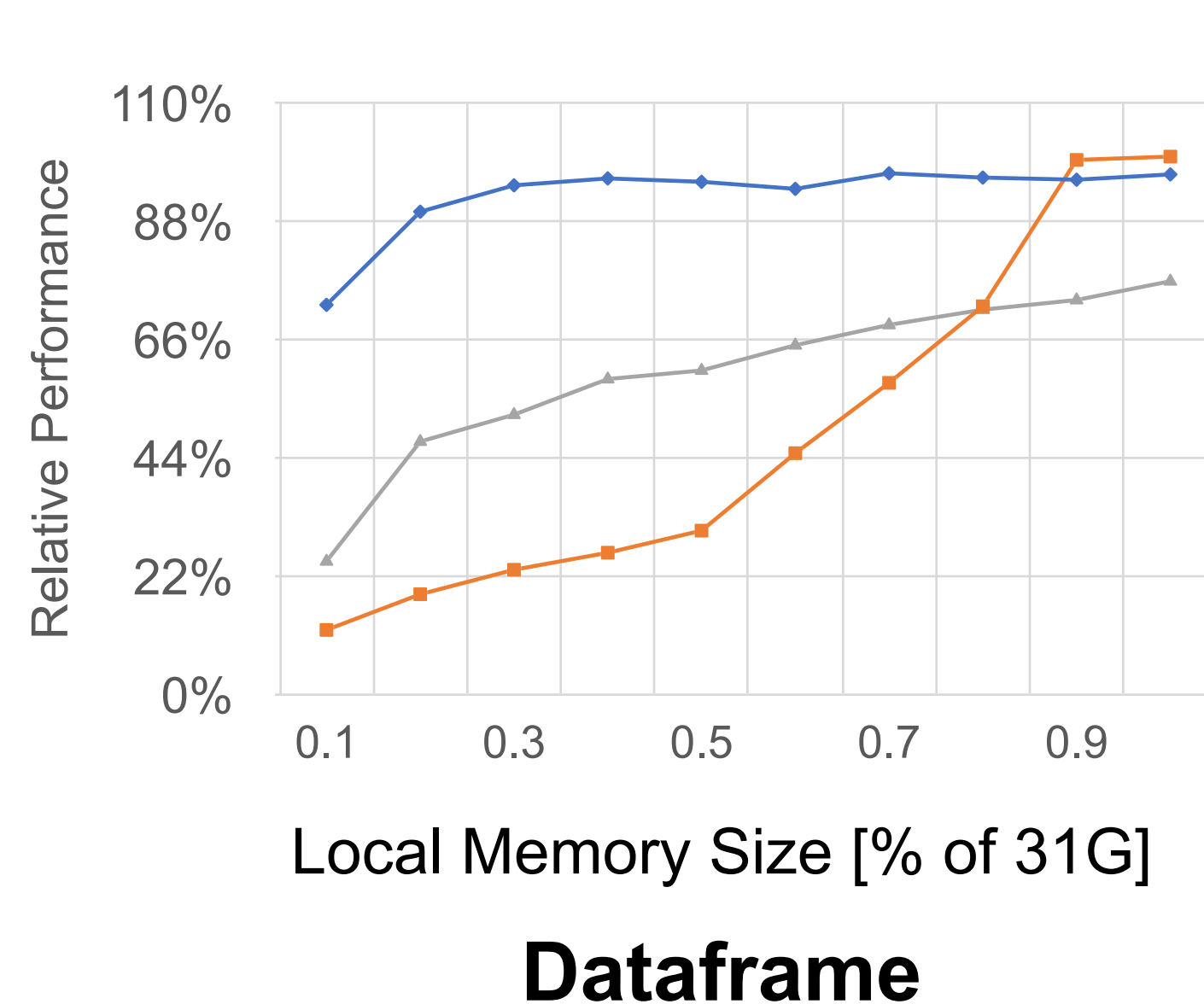
- **Adaptive code generation** reduces runtime overhead

Compared to **FastSwap**,

- **Section separation** reduce the interference between accesses
- **Analysis based prefetch** hide more latency

Comparing Typical Applications

- Dataframe reduces interference and could benefit from function offload
- GPT2-Inference has fixed access sequence and statically scheduled
- MCF section isolation and prefetch still benefits indirect access pattern



Summary of Mira

Mira: the first attempt to behavior-guided far-memory system

Achieves **application-aware** and **transparent** at the same time!

Program semantic should be captured and utilized in system layers

Discussion of Disaggregation

- Pros and cons of resource disaggregation?
- Do you believe this can be the future?
- New developments in the space: CXL

Outline

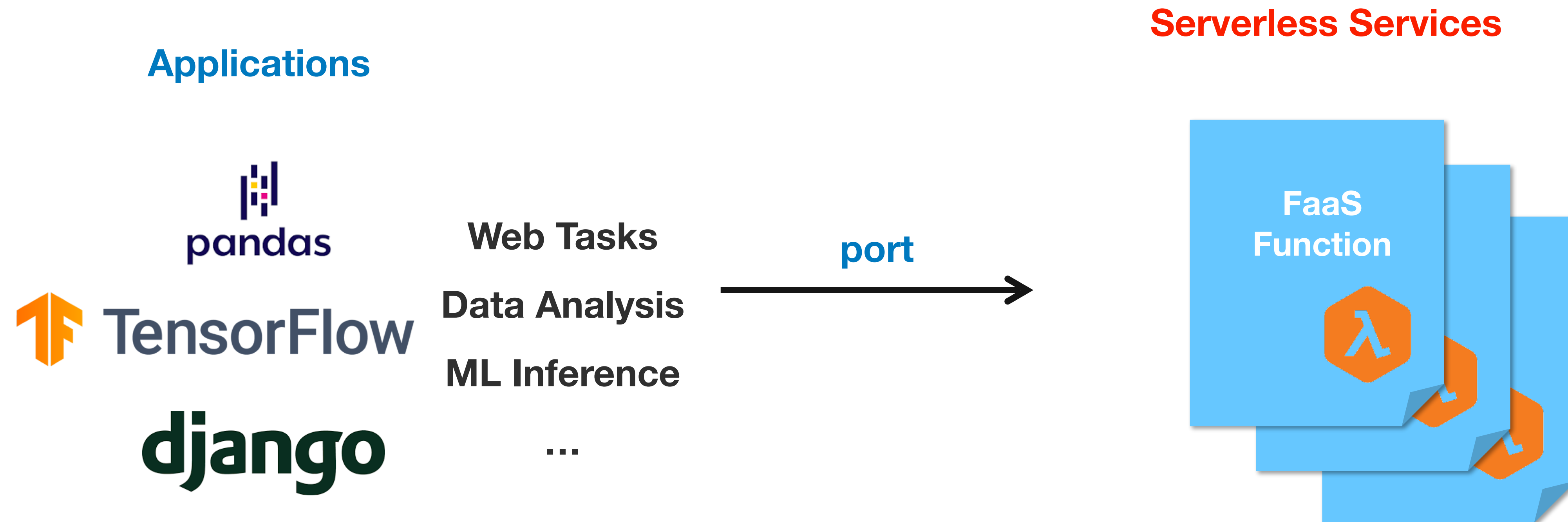
- PL for disaggregation: Mira [SOSP'23]
- Serverless on disaggregation: Zenix

Serverless Computing

- **No Management:** Users do not manage server or provision resources
- **Auto-Scale:** User computation is event triggered and auto scaled
- **Cost Efficient:** Users only pay for what they use



Serverless Computing

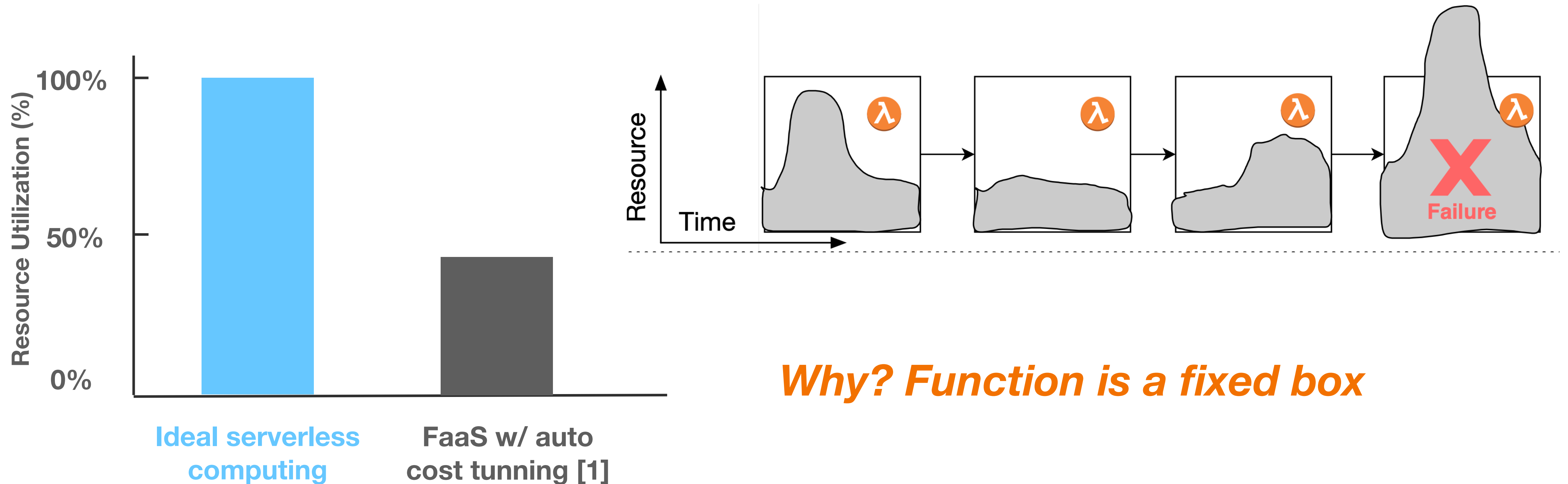


a **Function-as-a-Service (FaaS)** model:
Adapting user applications to **Fix-Sized Functions**

Does FaaS Meet All Serverless Goals?

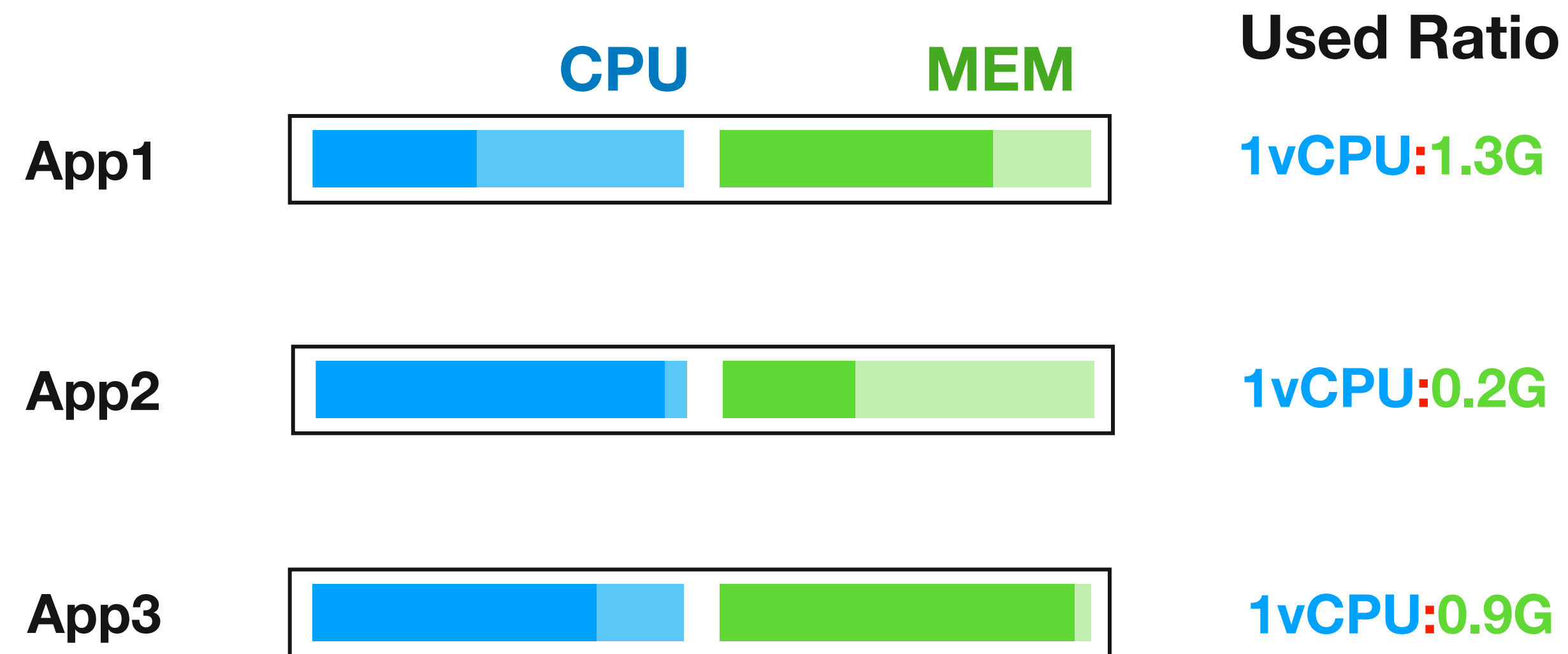
- **No Management:** Users do not manage server or provision resources
- **Auto-Scale:** User computation is event triggered and auto scaled
- **Cost Efficient:** any allocated but unused resources?

FaaS is Resource Inefficient



- Running Logistic Regression with 10 different input data samples

FaaS Issue 1: Fixed Resource Type Ratios



CPU & memory allocated in **fixed ratios**

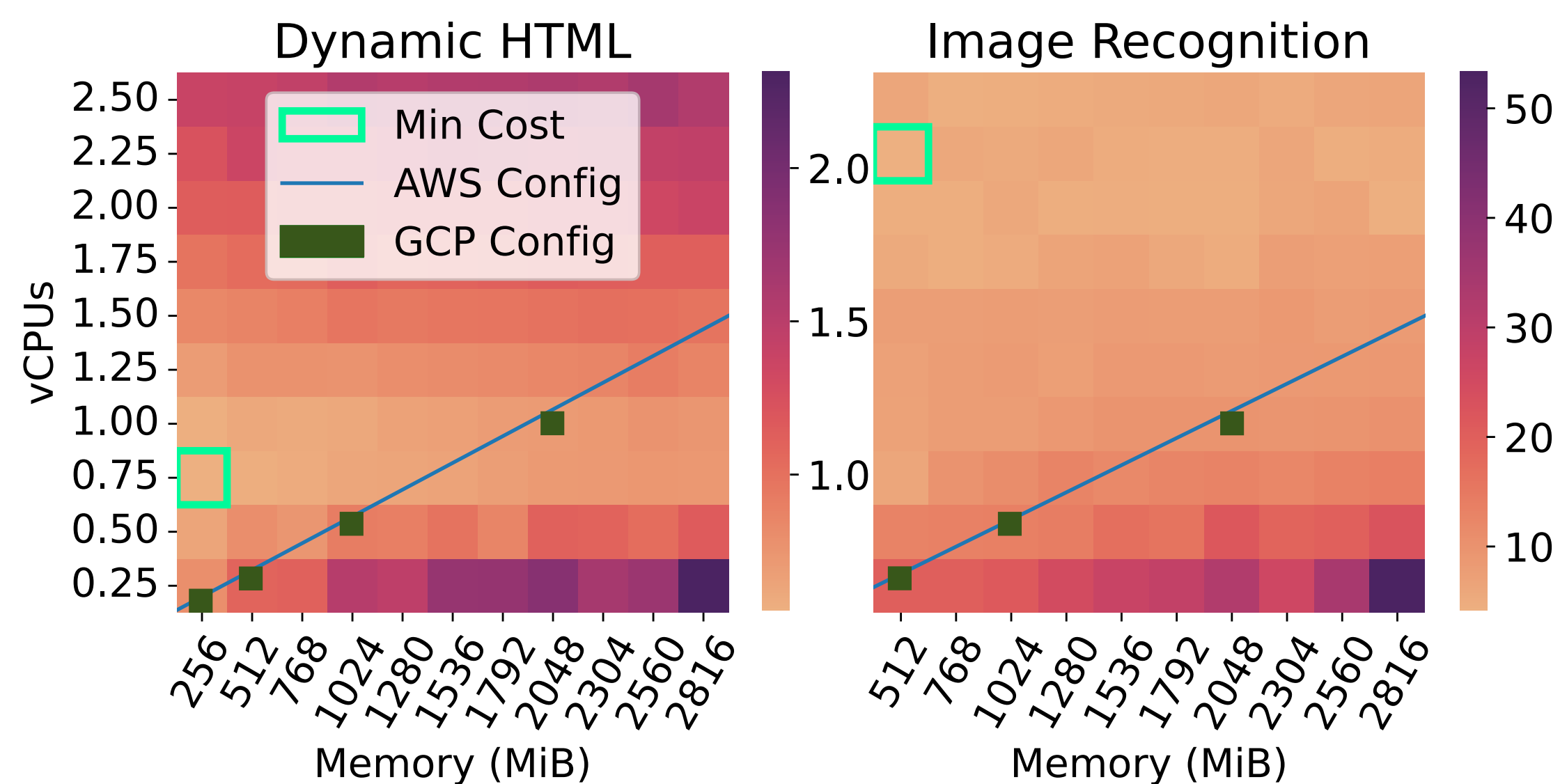
=> User applications' needs are not!

AWS Lambda Allocation:

Fixed Ratio, 1 vCPU : 1.7G Memory

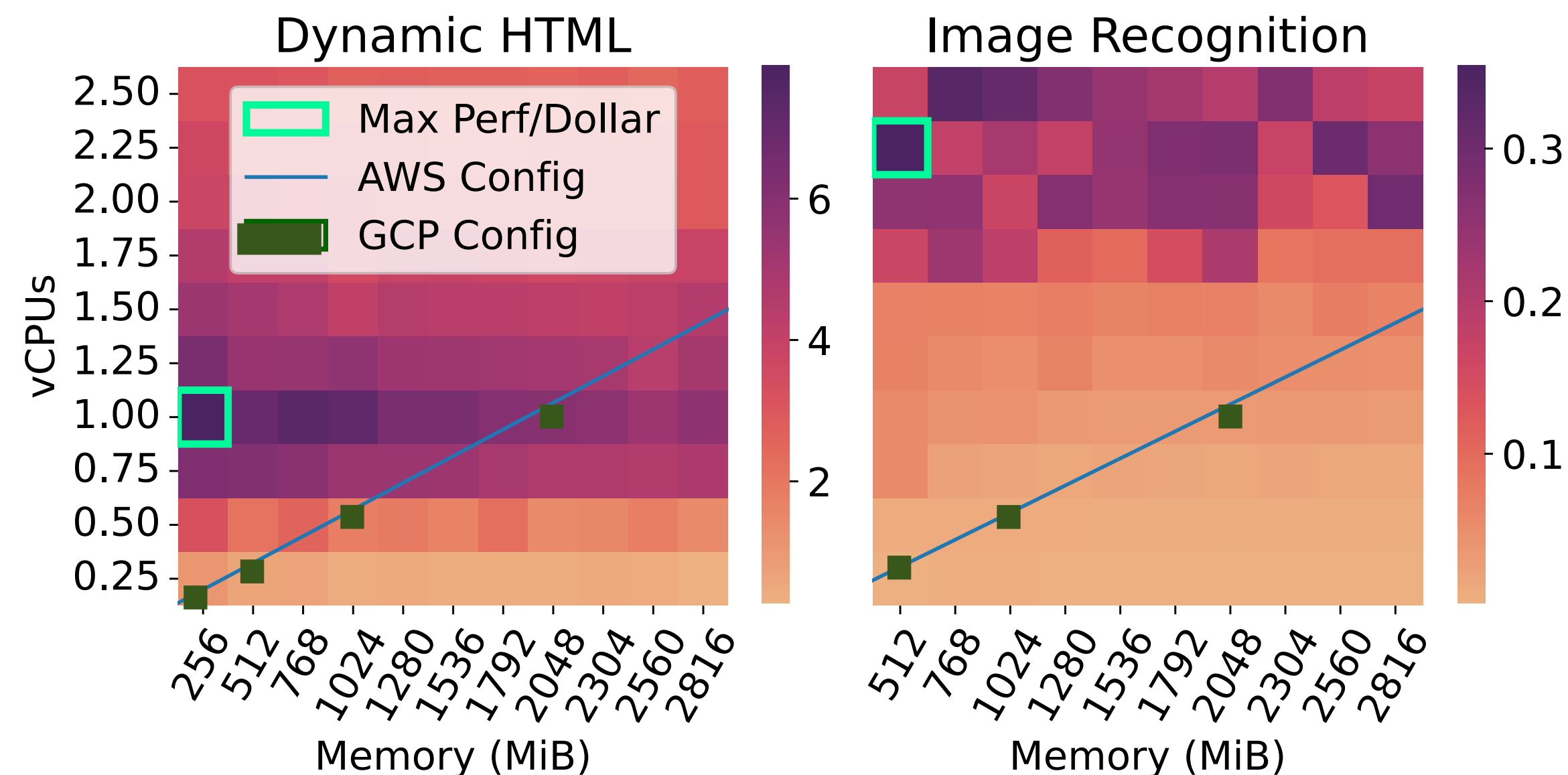
* Data Collected from Image Recognition, HTML and MapReduce examples, from SeBS benchmarks

FaaS Issue 1: Fixed Resource Type Ratios



Resource Consumption Heatmap

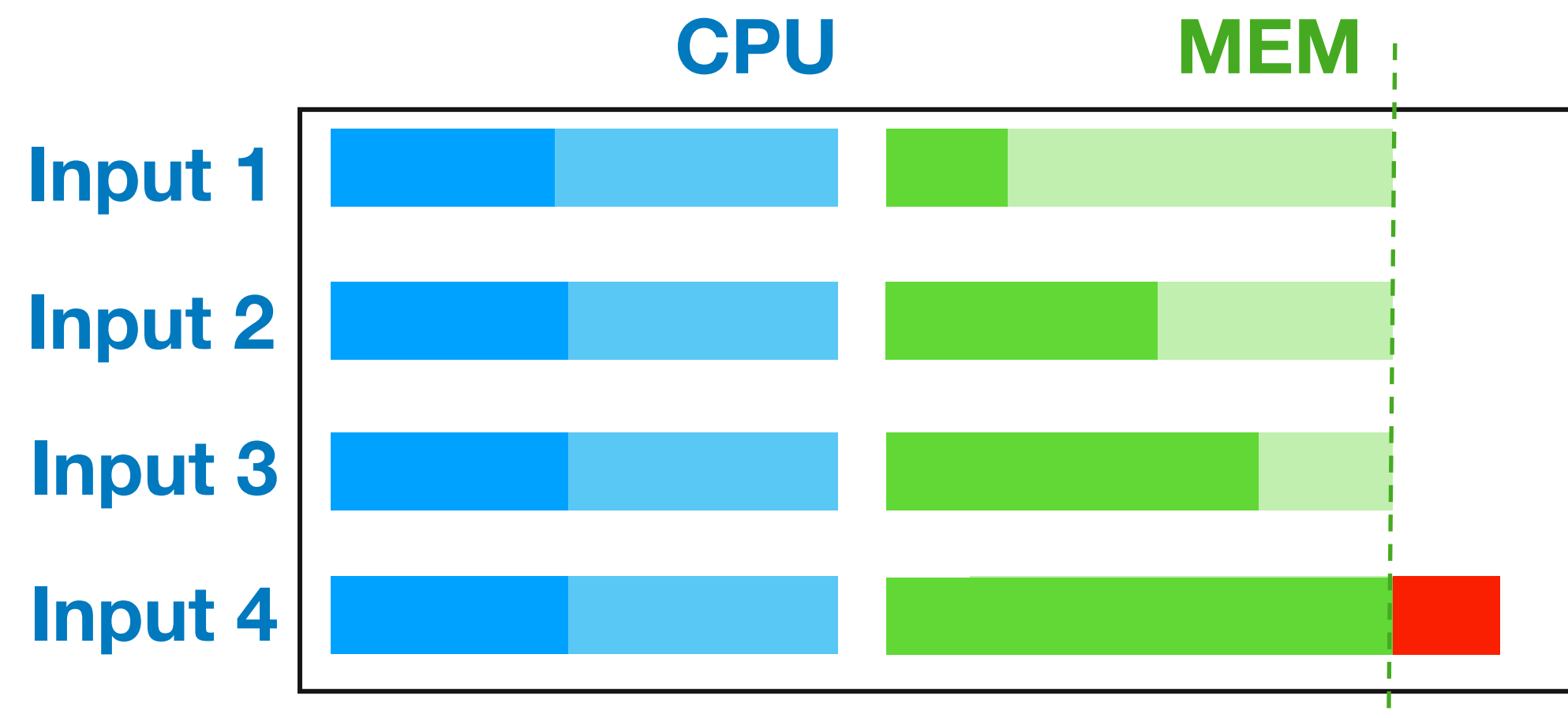
lighter color: lower cost



Perf-per-Dollar Heatmap

darker color: better perf/\$

FaaS Issue 2: Fixed Amount across Invocations



Uses set **one function size across invocations**

=> Different inputs cause different usages

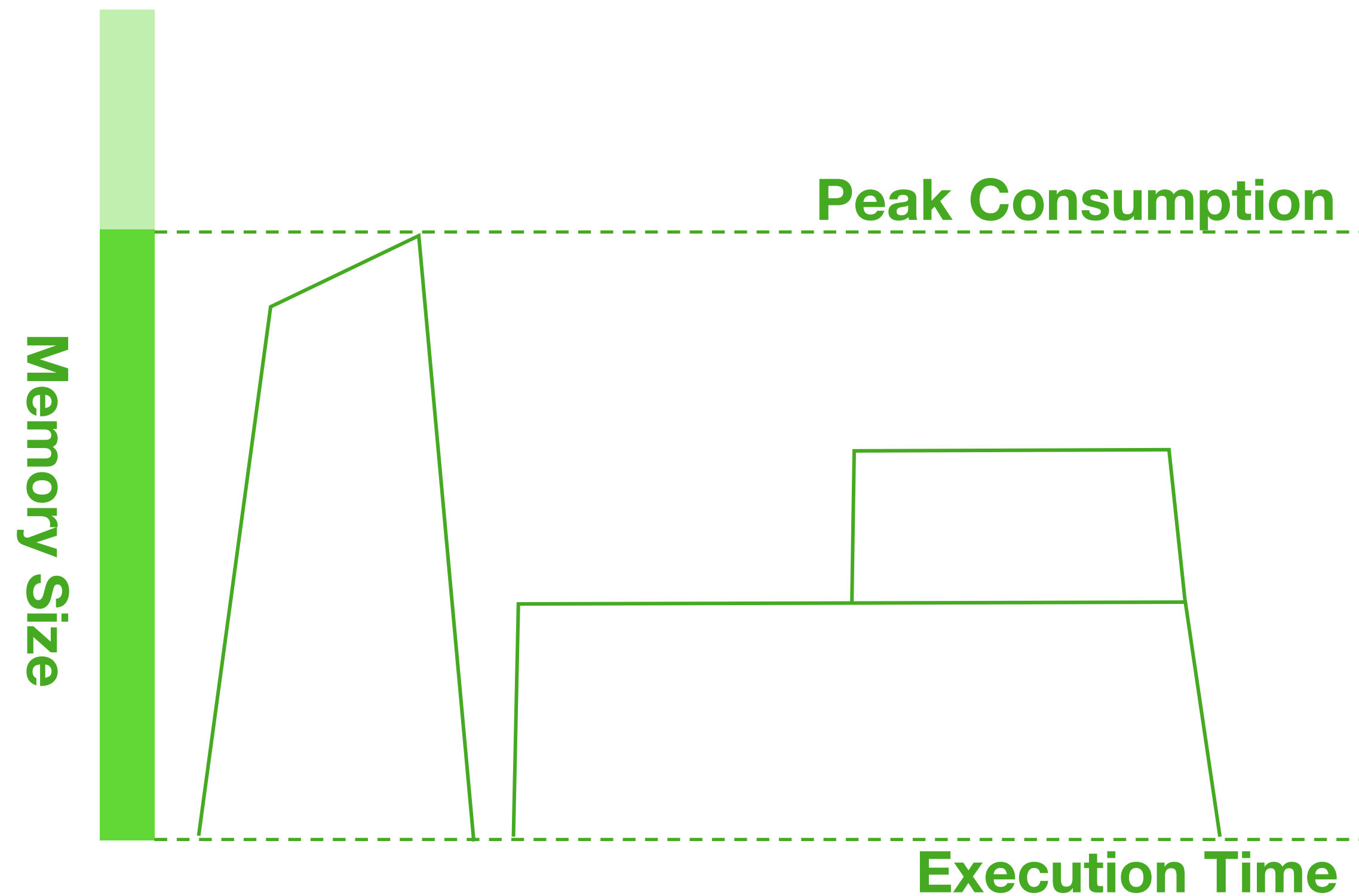
Provision for the peak?

Resource wastage for non peaks

Provision for avg?

Execution fails when larger than avg

FaaS Issue 3: Fixed Amount throughout Execution



One function size **throughout execution**
=> Resource usage changes in real apps

*Image Processing Example. On a large size image, each line showing one object.

Running “Bulky” Applications on FaaS

- Bulky applications like data analytics, ML, video processing, scientific computing
 - Exhibit resource usage variations across different phases of computation
 - Require different amounts of resources with different inputs
 - Run longer or consume more memory than function size limits
- Running bulky applications on today’s FaaS
 - Manually split or rewrite a program as a set of functions
 - Provider increasing function sizes (AWS: 3GB to 10GB and 5 min to 15 min in 2020)

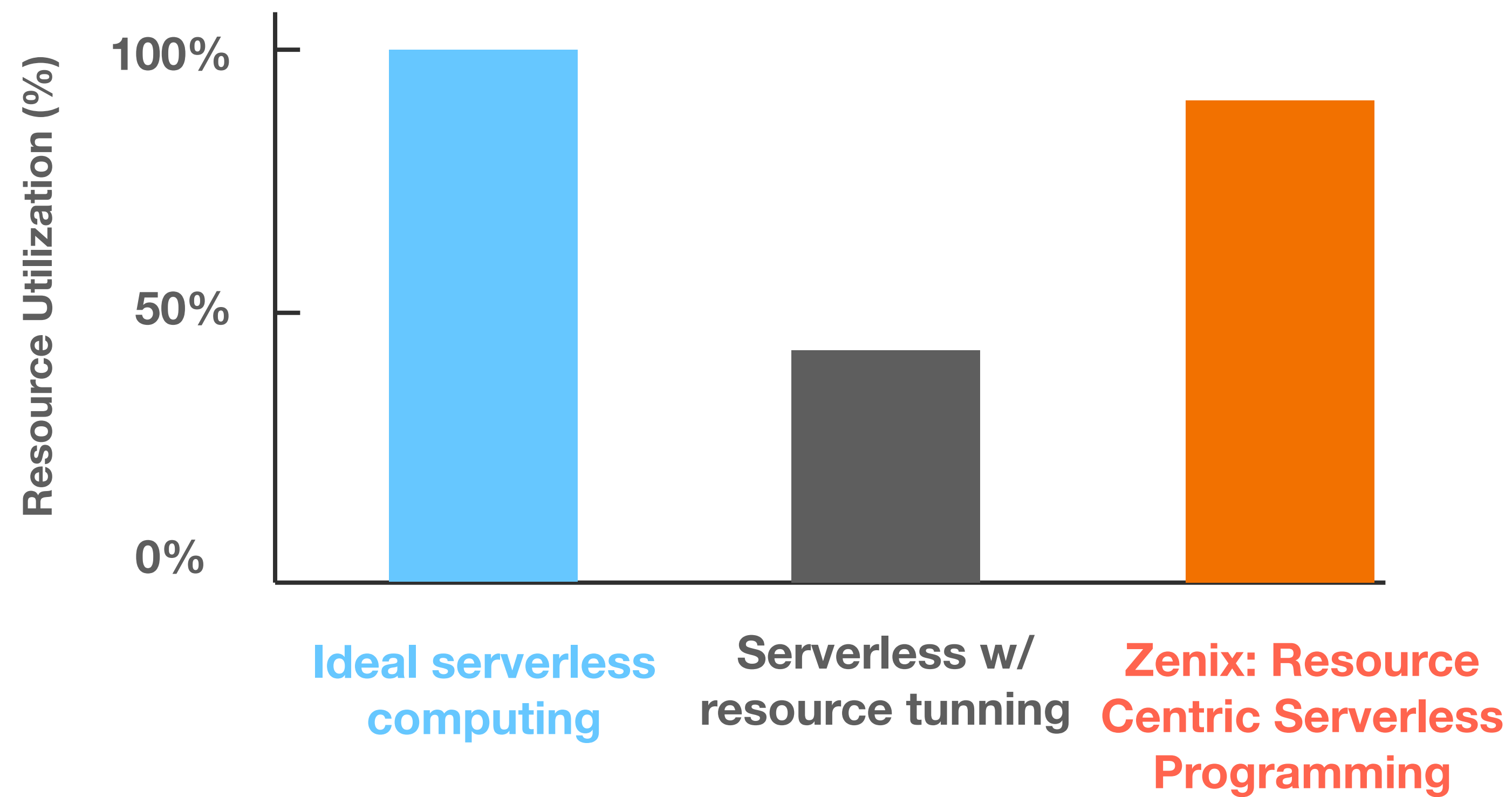
Root cause for FaaS issues:

***User Applications Adapt to
Provider-Defined Functions***

Resource-Centric Serverless Computing:

Adapt Execution Towards User Applications

Zenix: a Resource-Centric Serverless Computing Platform



*Logistic Regression Example with 10 different input data samples



Zenix: Resource-Centric Serverless Computing

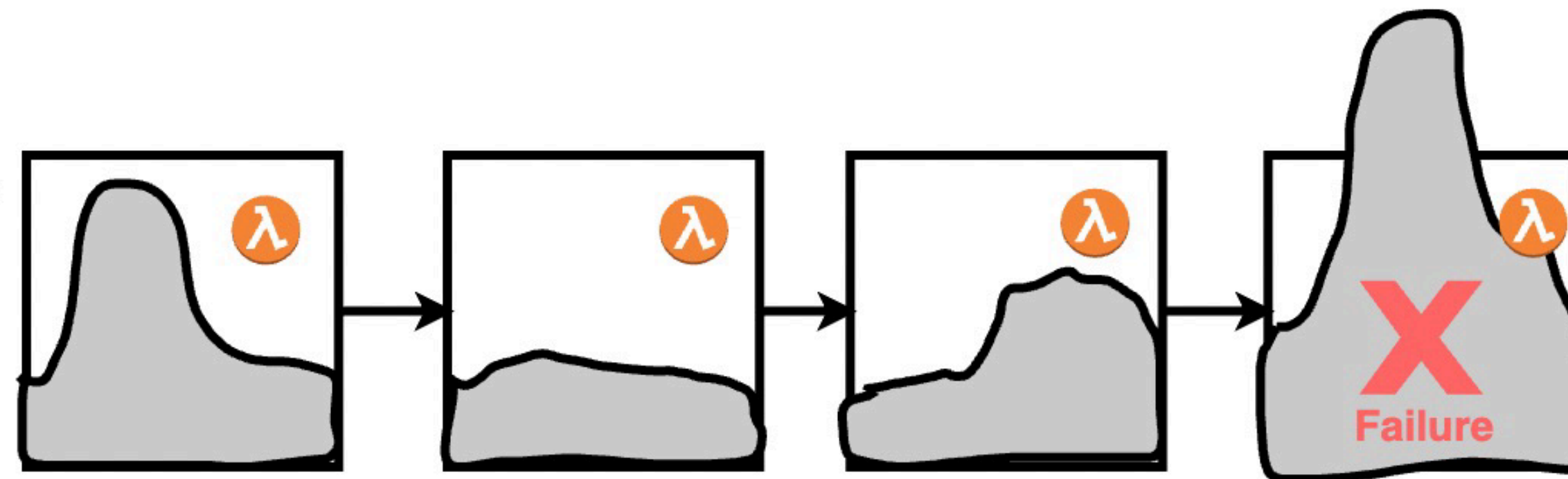
Center serverless computing around the resource features of applications by decomposing them on a disaggregated system

Resource-Centric Serverless Computing

- Decomposing and executing applications according to their *resource needs*

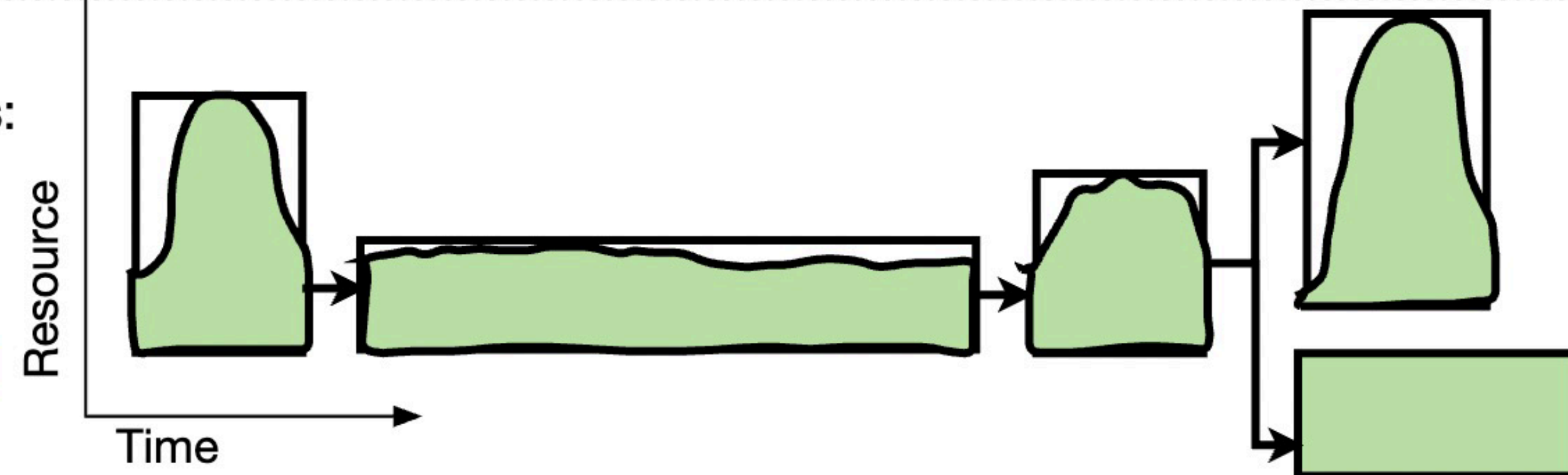
Function-Based Serverless:

Resource wastage
Resource limits
Potential failure
Manual application porting
Runtime overhead



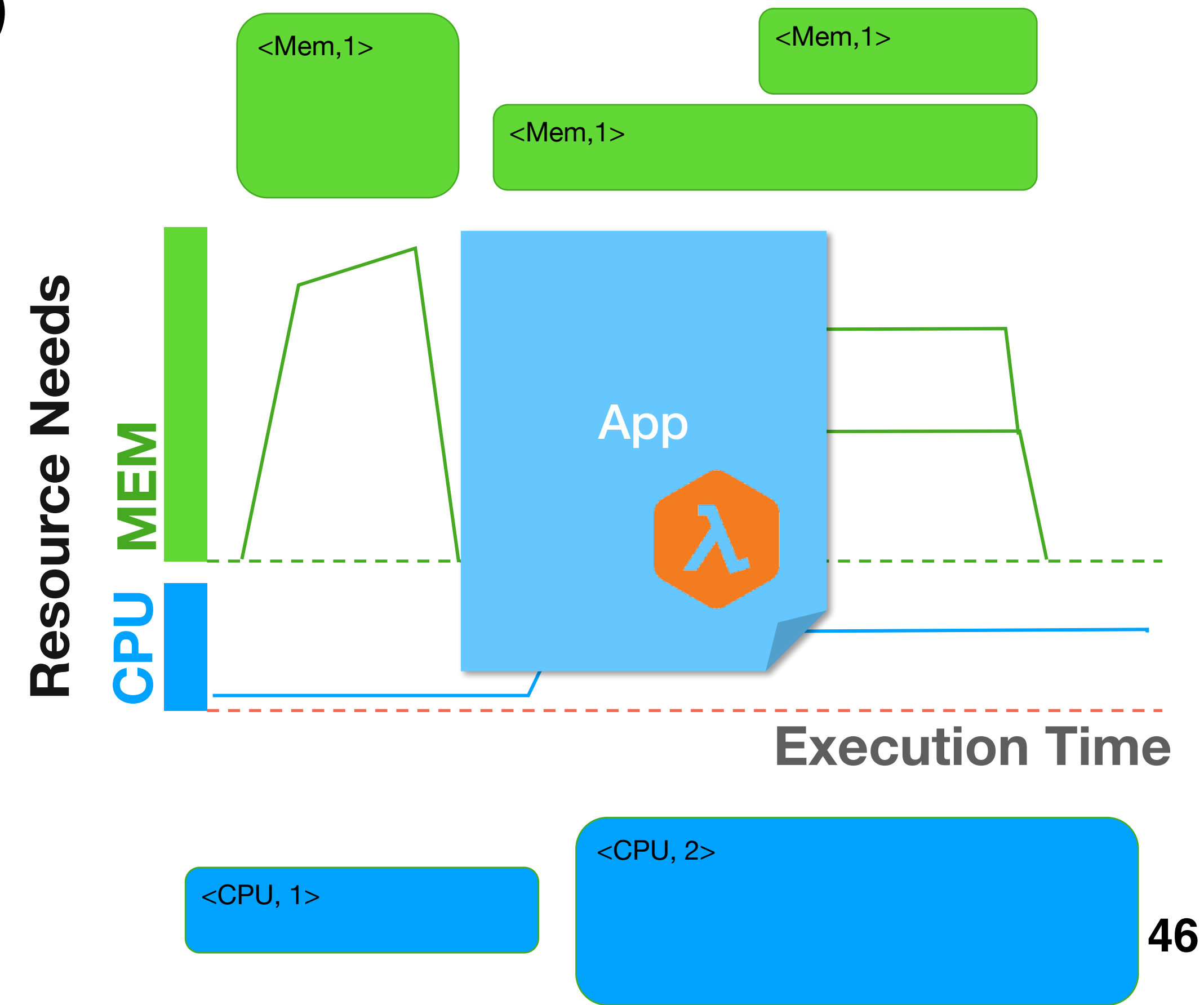
Resource-Based Serverless:

Tight resource packing
No resource limits
Fine-grained autoscaling
No or light manual porting
Efficient runtime perf



Resource Graph: A New Model for Capturing Application Resource Behavior

- Resource component (nodes in a resource graph)
 - A piece of code or data object that captures unique **<type, lifetime, amount>**
- Each component **autoscaled on its own**
- Scale *without limits*, and adapt to each input

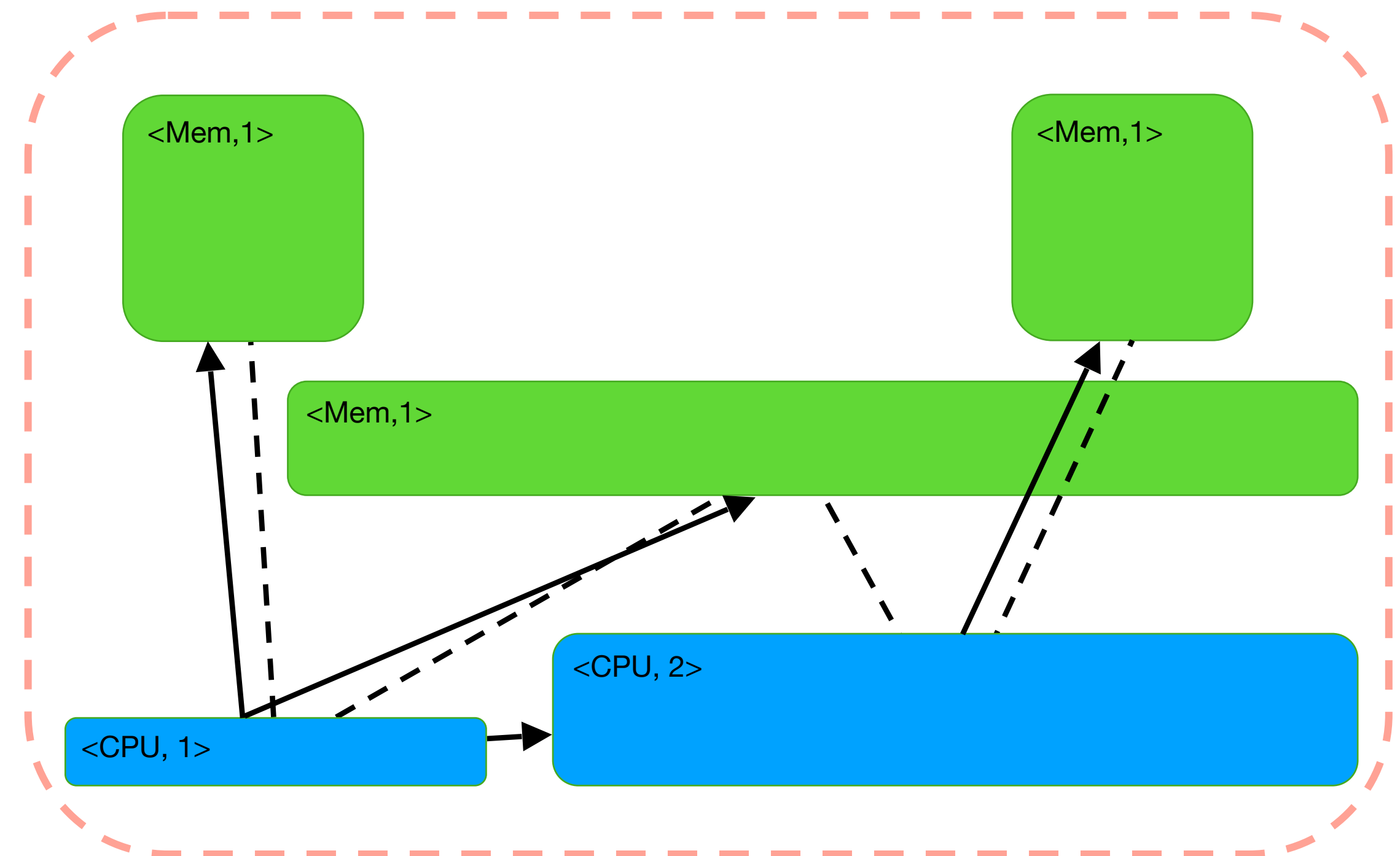


Resource Graph: A New Model for Capturing Application Resource Behavior

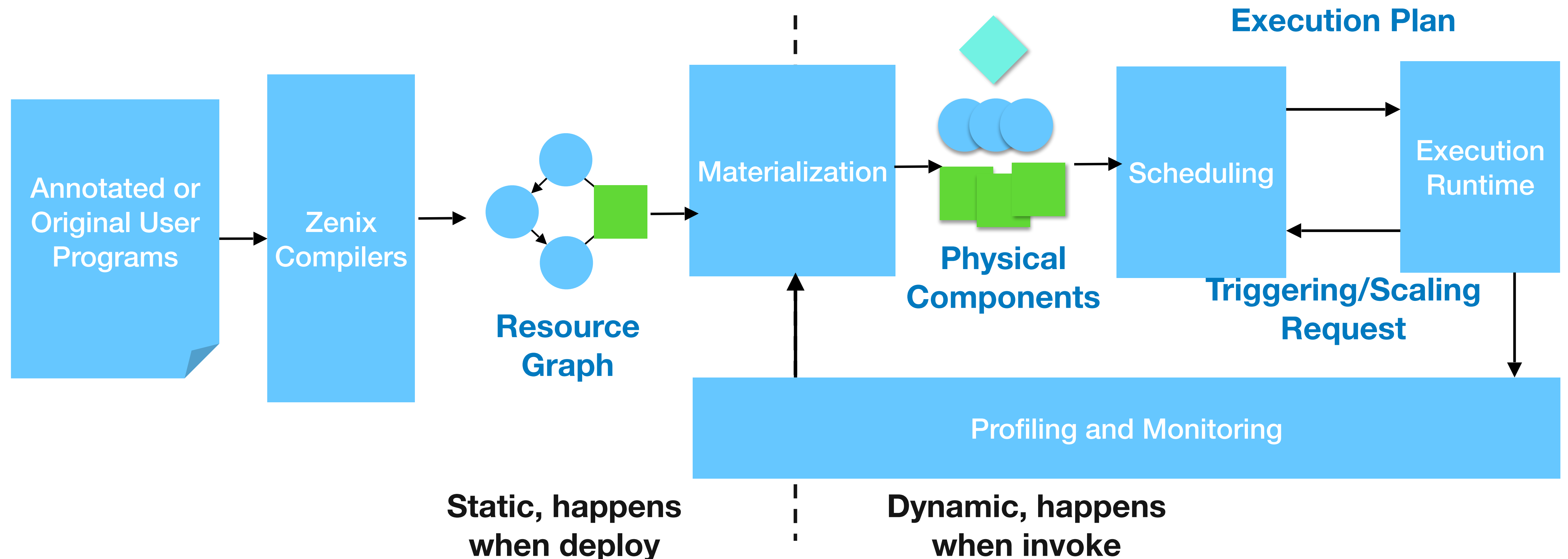
Edges in a resource graph:

→ **Triggering** another component

- - - - - **Communicating** with other components



Zenix: A Resource-Centric Serverless Computing Platform

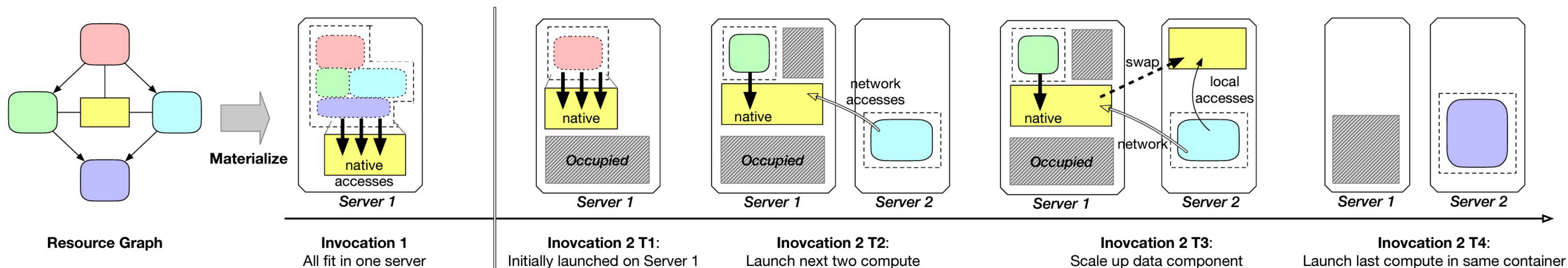


Resource Graph Generation

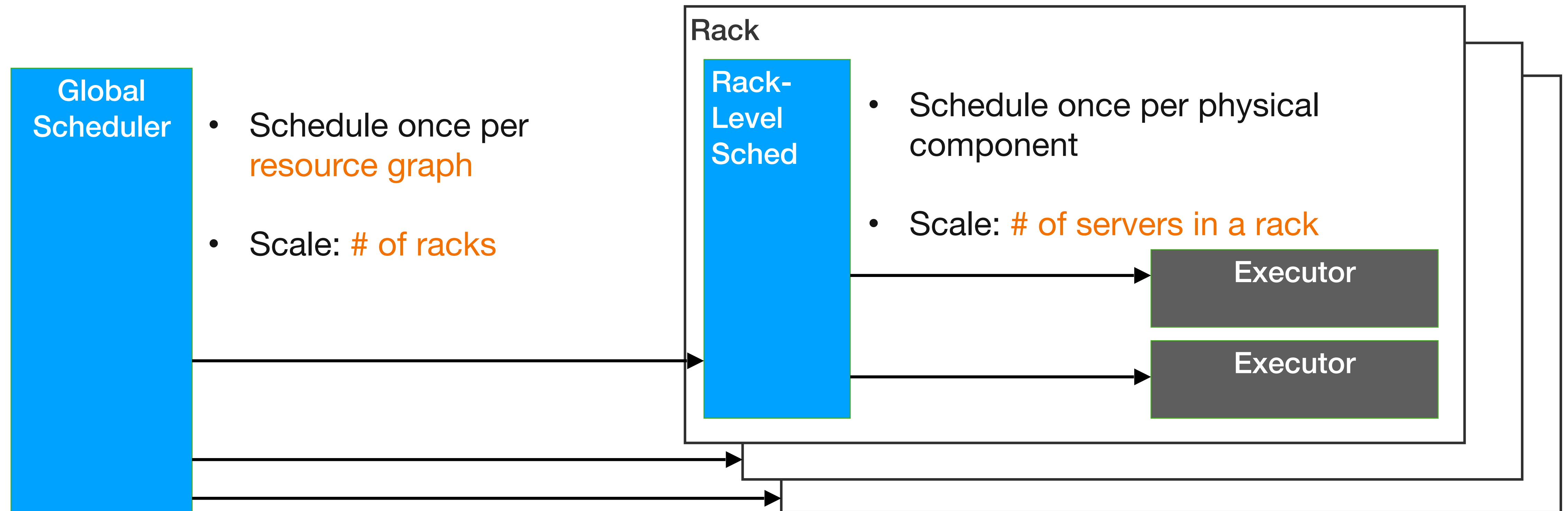
- Users annotate their code with *@compute (par=K)* and *@data*
- or, Zenix automatically inserts annotations at every function and data object
- Profile: object size, CPU utilization
- Determine graph components: group by CPU util and object lifetime
- Compilation: generating both a local and a disaggregated version (with Mira)

Adaptive Materialization

- Co-locate if possible (no reservation, but mark future resources as low priority for other scheduling)
- Co-locate compute components and their accessed data components => low comm overhead
- Co-locate triggered and triggering compute components => no startup overhead
- Sizes each component according to profiled history
- Adaptively scales components on demand



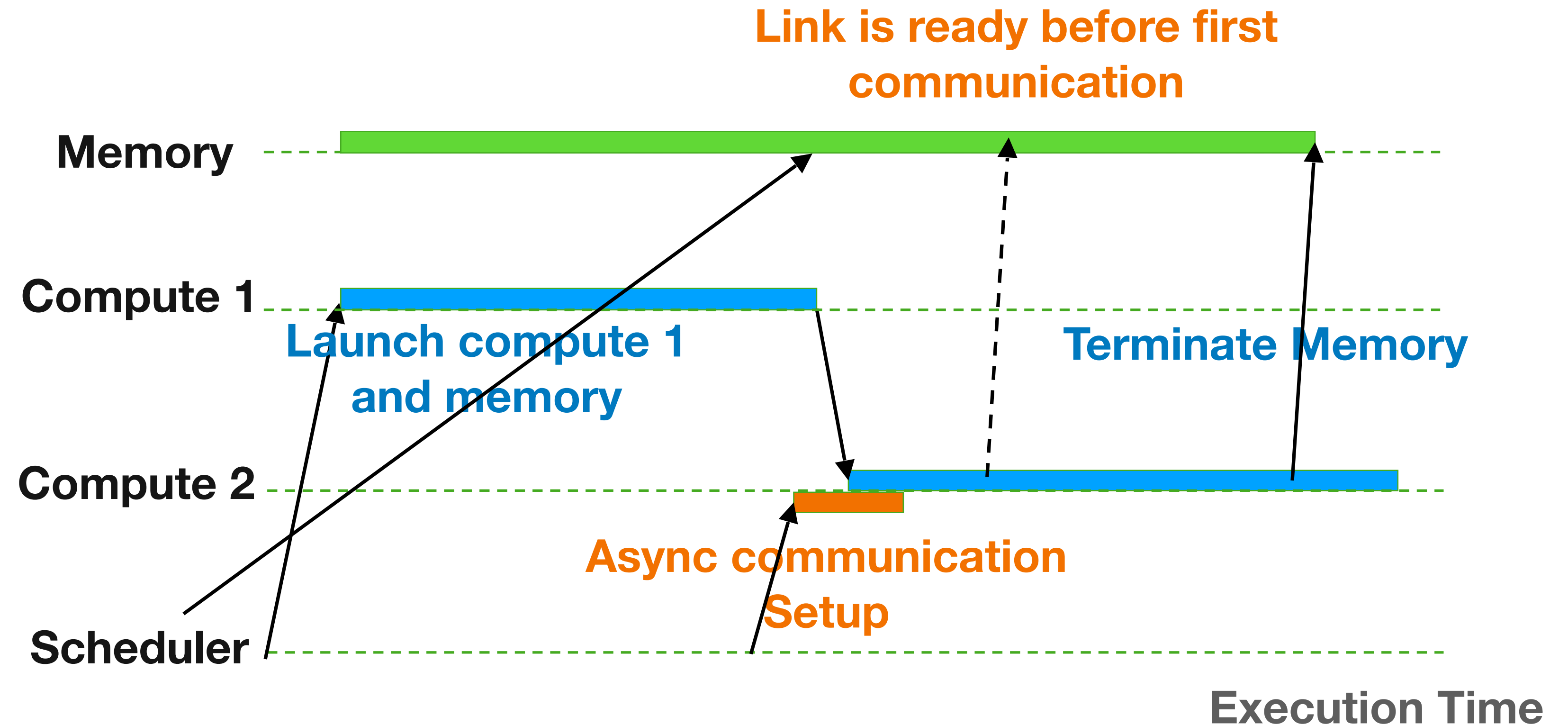
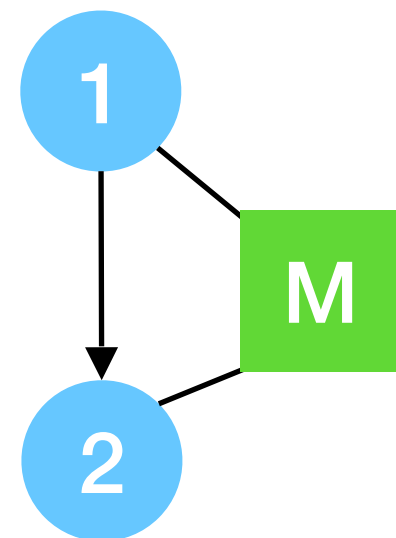
Two-Level Scheduling Mechanism



Decompose scheduling tasks into two levels solves **scheduling scalability problem**

Async Communication Mechanism

Communicating components in graph



Utilize **pre-setup scheduler links** to build connections between auto-scaled components: sub-100 μ s setup time, and largely hidden

Memory Accesses

- Native accesses when compute and data components scheduled locally
- Remote accesses via Zenix API calls (possibly to multiple servers)
- Local accesses via transparent swapping (when compute component scales beyond allocated size)
- Native/swap or API calls chosen at scheduling time and use different compilations

Implementation

- Compiler implemented on Mira (1.4K LOC)
- Scheduling/Monitoring built on top of OpenWhisk with 9K Scala LOC
- Two proof-of-concept add-on compiler: annotation-based and library based
- TCP/RDMA-enabled container runtime library with 6.4K C++/Python LOC.

Evaluation Results - ML Training

- Logistic regression with two inputs
- Comparing with AWS Lambda, Lambda+Step Function with Orion resource tuning and with Redis storing intermediate results, OpenWhisk, and FastSwap
- Zenix reduces resource consumption by 40% to 85%

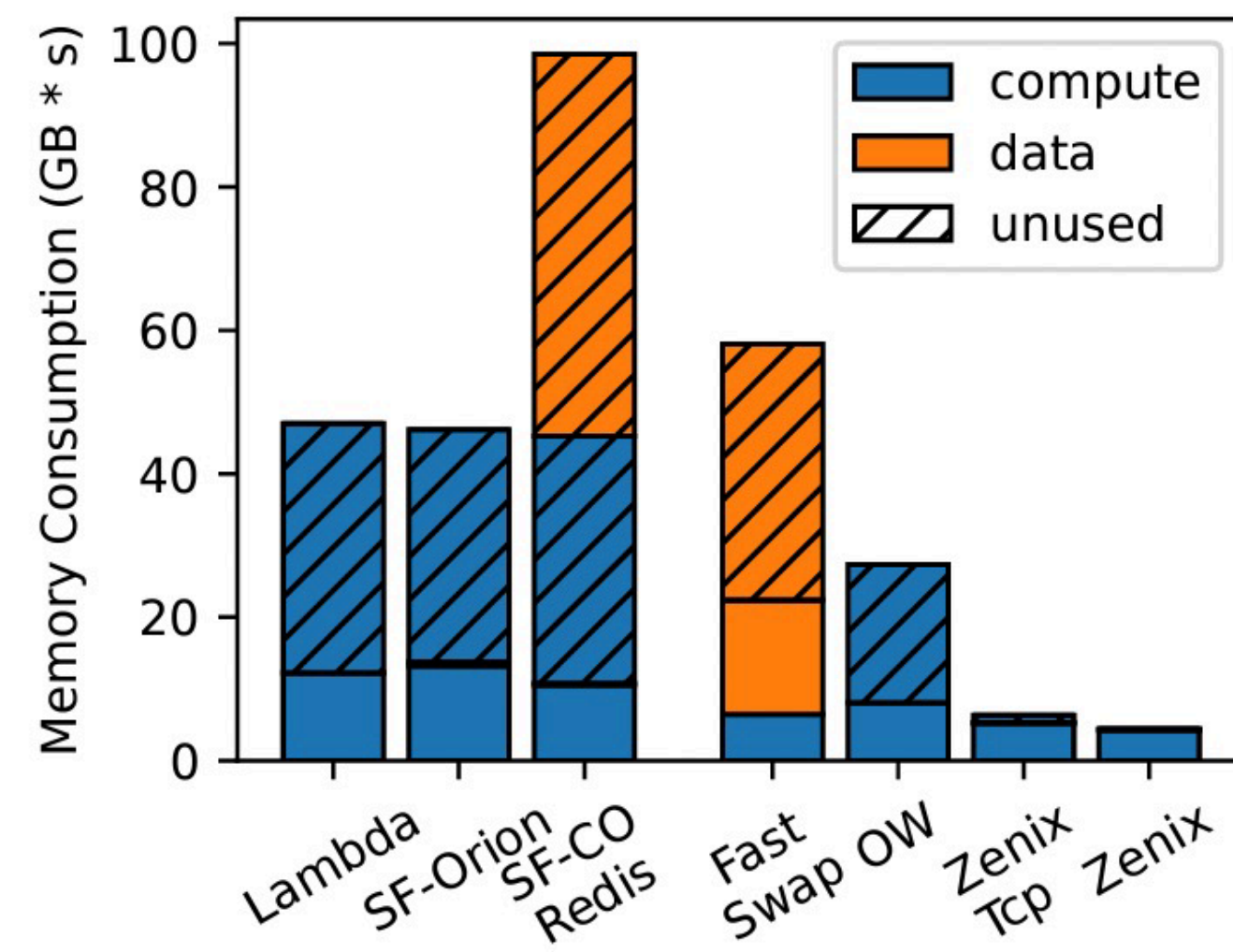


Figure 8: LR Memory Consumption with Small Input.

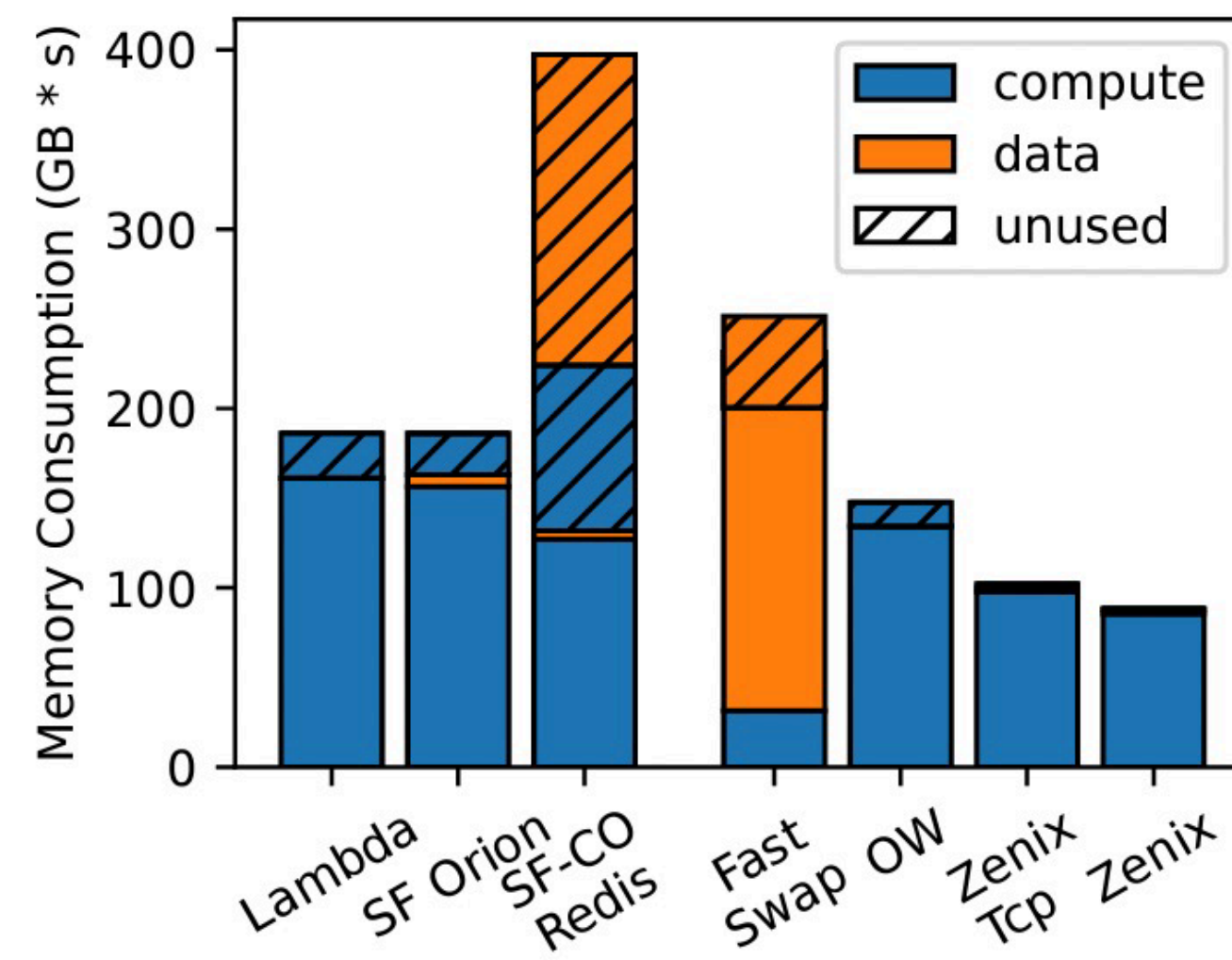


Figure 9: LR Memory Consumption with Large Input.

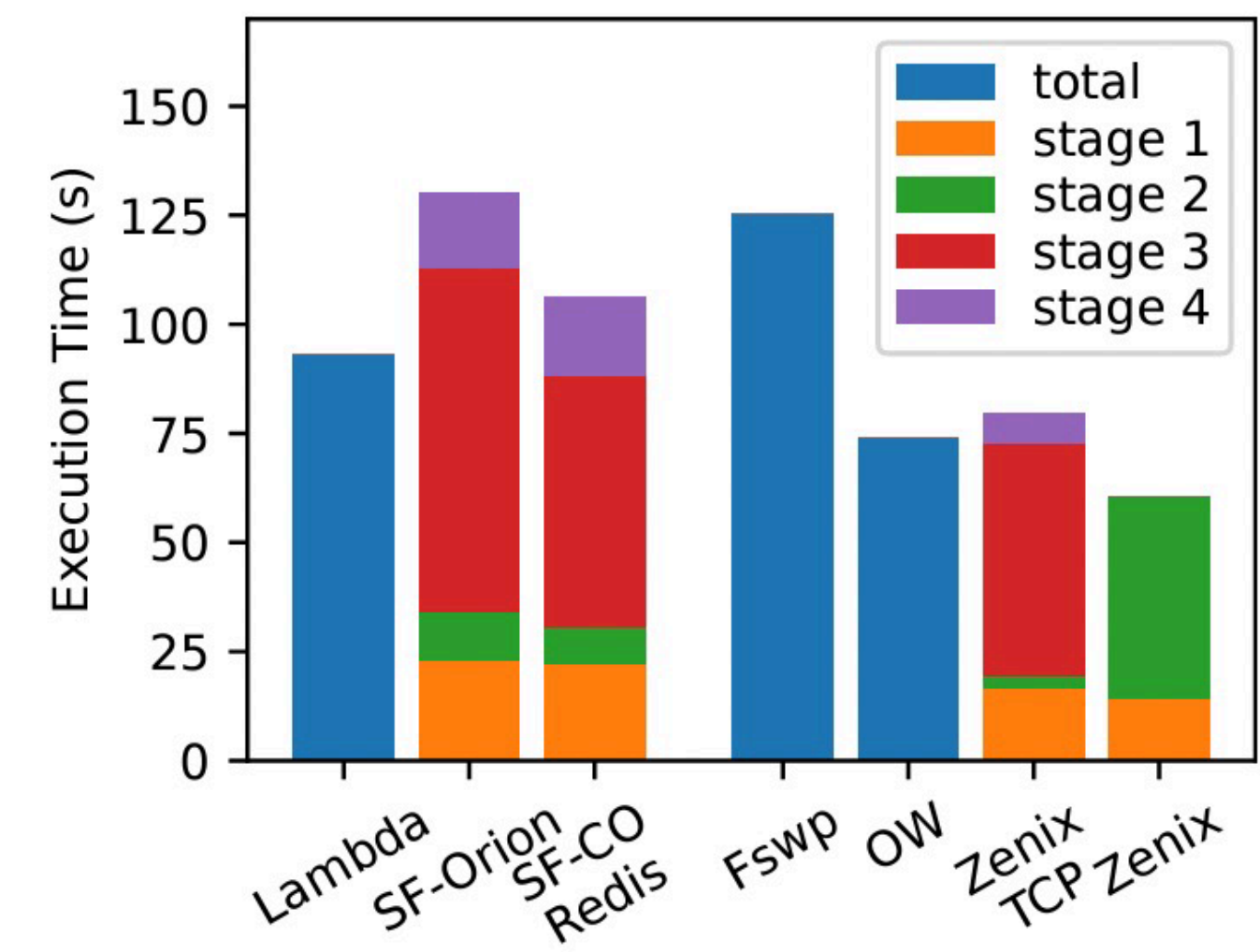


Figure 10: LR Execution Time Breakdown with Large Input.

Evaluation Results - Video Processing

- Video transcoding (ported from ExCamera) with three video sizes
- Compared to gg (serverless for video), vpxenc (running on a single server)

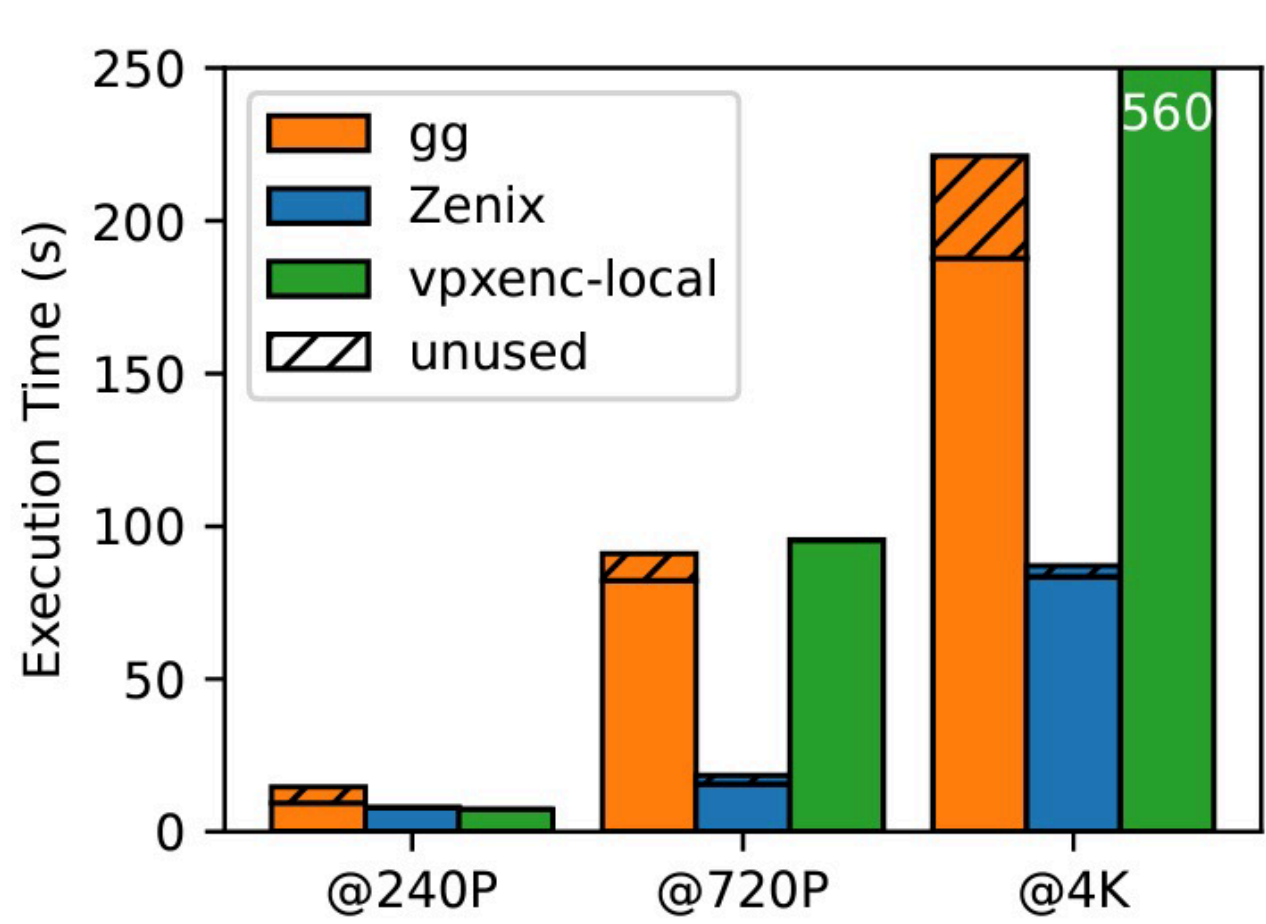


Figure 13: Video Transcoding Execution Time. With 3 video resolutions.

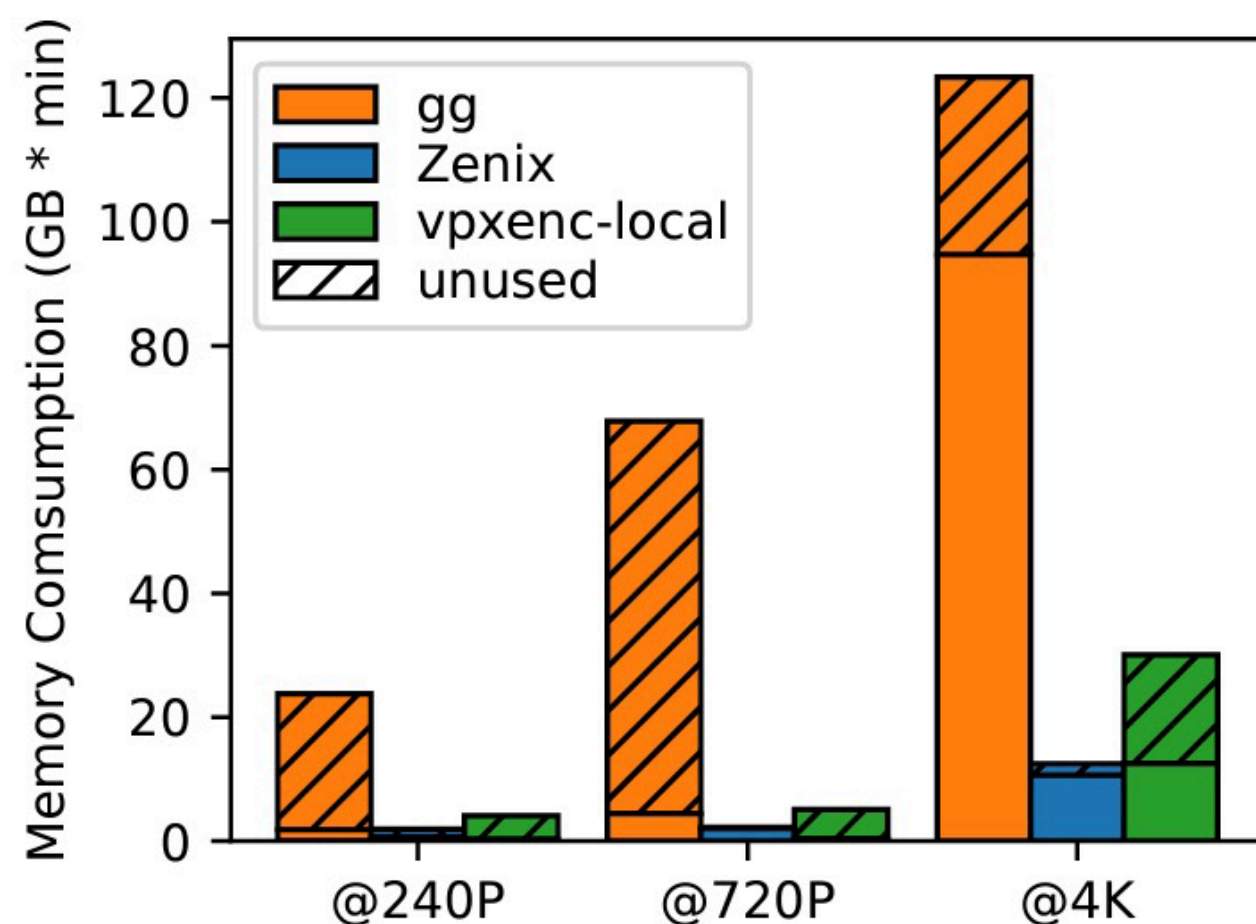


Figure 14: Video Transcoding Memory Consumption.

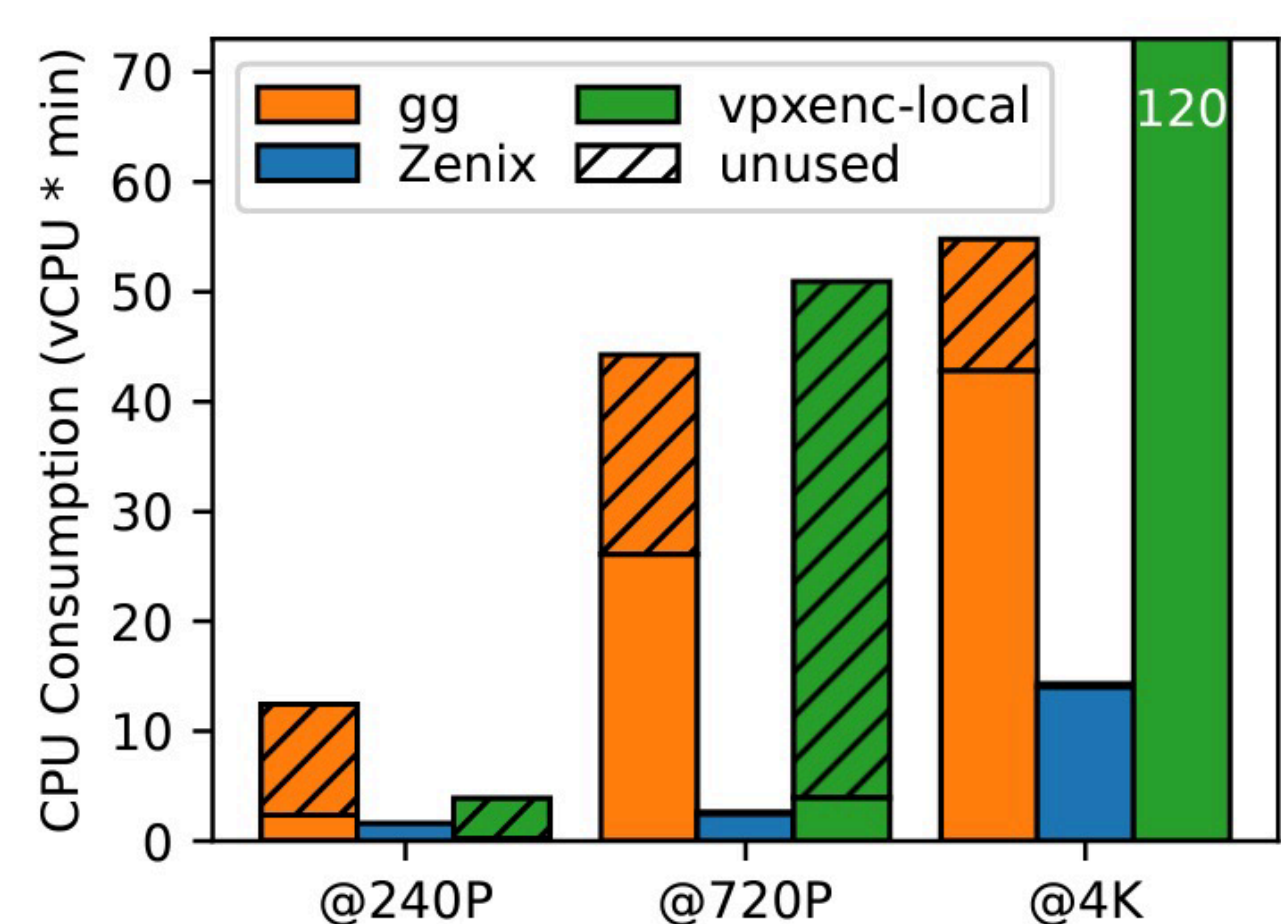
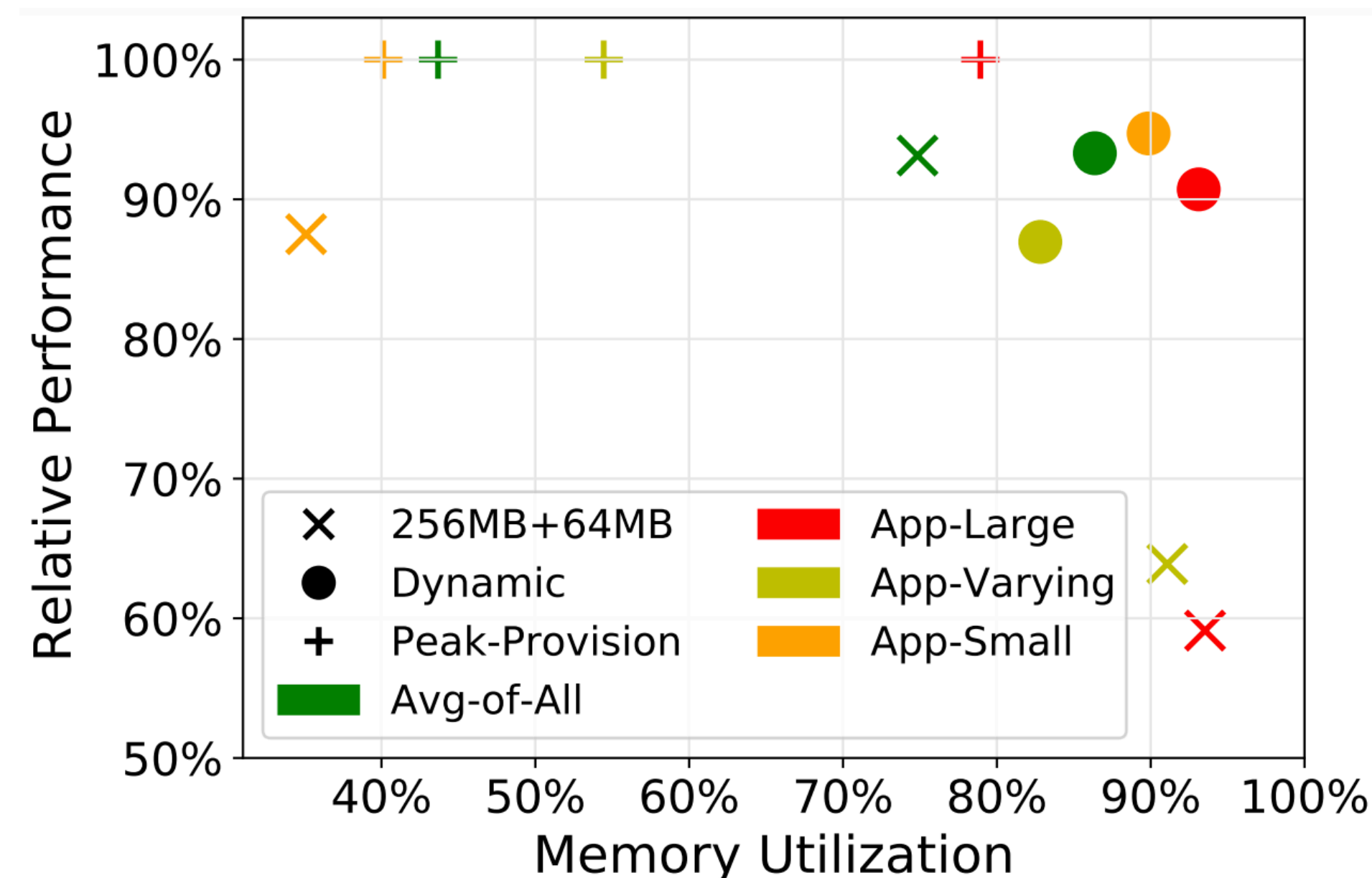


Figure 15: Video Transcoding CPU Consumption.

Evaluation Results - Resource Materialization Policies

- Zenix monitor system could adjust allocation policy at resource granularity to reduce the performance.
- For both small, single element applications and large multistage operations, Scad could achieve over 90% resource utilization and performance.



Summary of Zenix

Serverless computing can and benefit from being resource centric!

We built **Zenix**, a resource-centric serverless computing platform
with **resource graphs** executed with **disaggregation and materialization**
achieves high performance, scalability, and resource utilization

Check our paper for more details!
<https://arxiv.org/abs/2206.13444>