Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes

Zain Ruan[*] Seo Jin Park[*] Marcos K. Aguilera[†] Adam Belay[*] Malte Schwarzkopf[‡]

*MIT CSAIL †VMware Research ‡Brown University

Operational reality of today’s datacenter

- Users provision fixed-sized, coarse-grained instances.

Instance types (624)

Actions

Find resources by attribute or tag

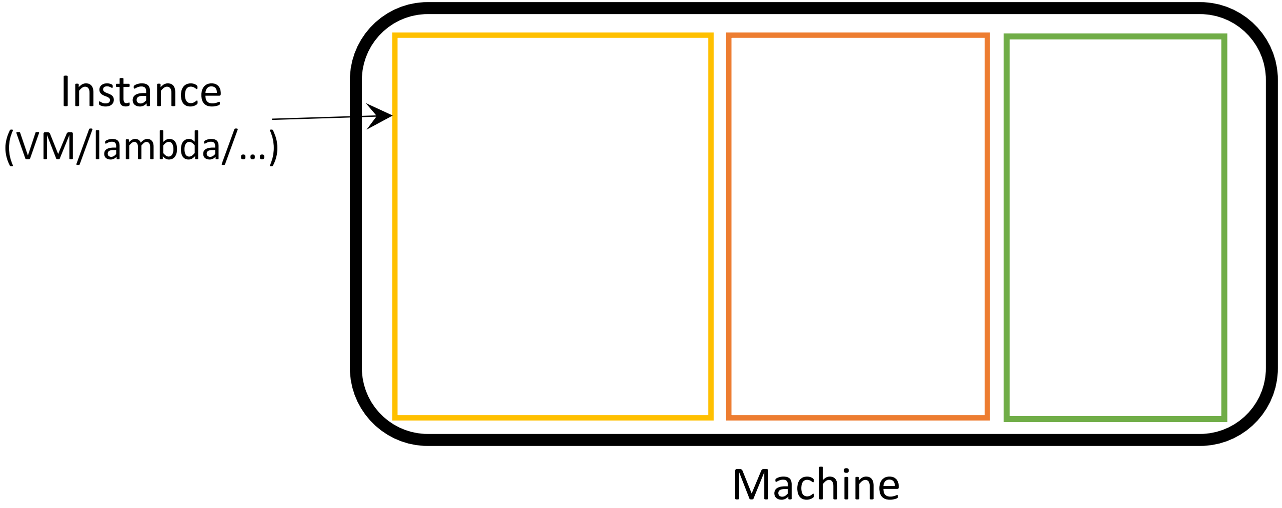
< 1 2 **3** 4 5 6 7 ... 13 >

Instance type	vCPUs	Architecture	Memory (GiB)	Storage (GB)	Storage type
d3.2xlarge	8	x86_64	64	11880	hdd
d3.4xlarge	16	x86_64	128	23760	hdd
d3.8xlarge	32	x86_64	256	47520	hdd
d3.xlarge	4	x86_64	32	5940	hdd
d3en.12xlarge	48	x86_64	192	335520	hdd

Operational reality of today's datacenter

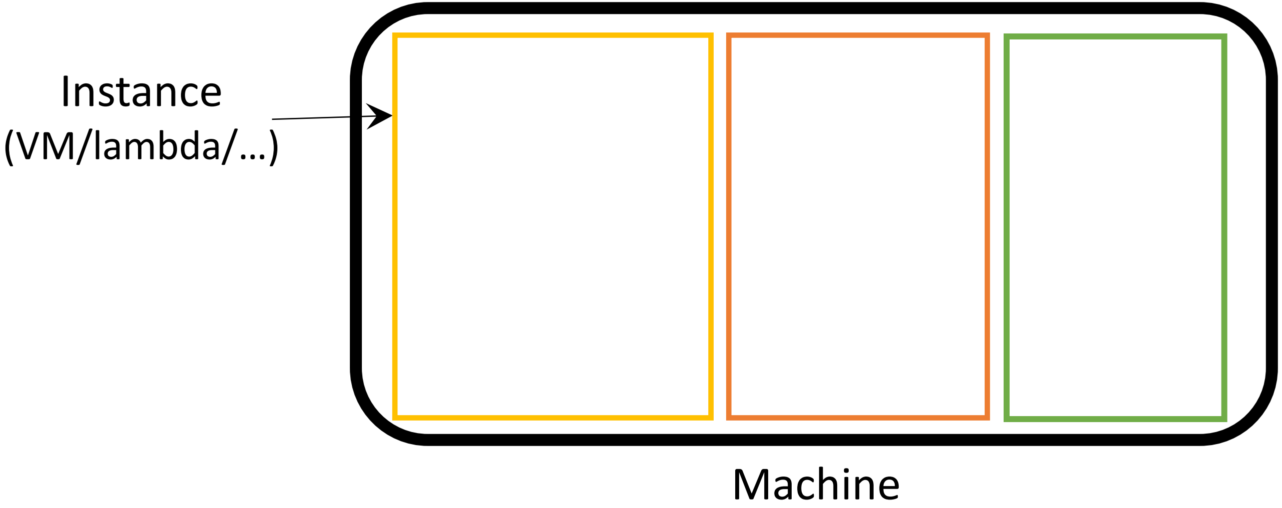
- Users provision fixed-sized, coarse-grained instances.

➢Operators bin-pack instances onto the available machines.



Inefficiency: resource overprovisioning

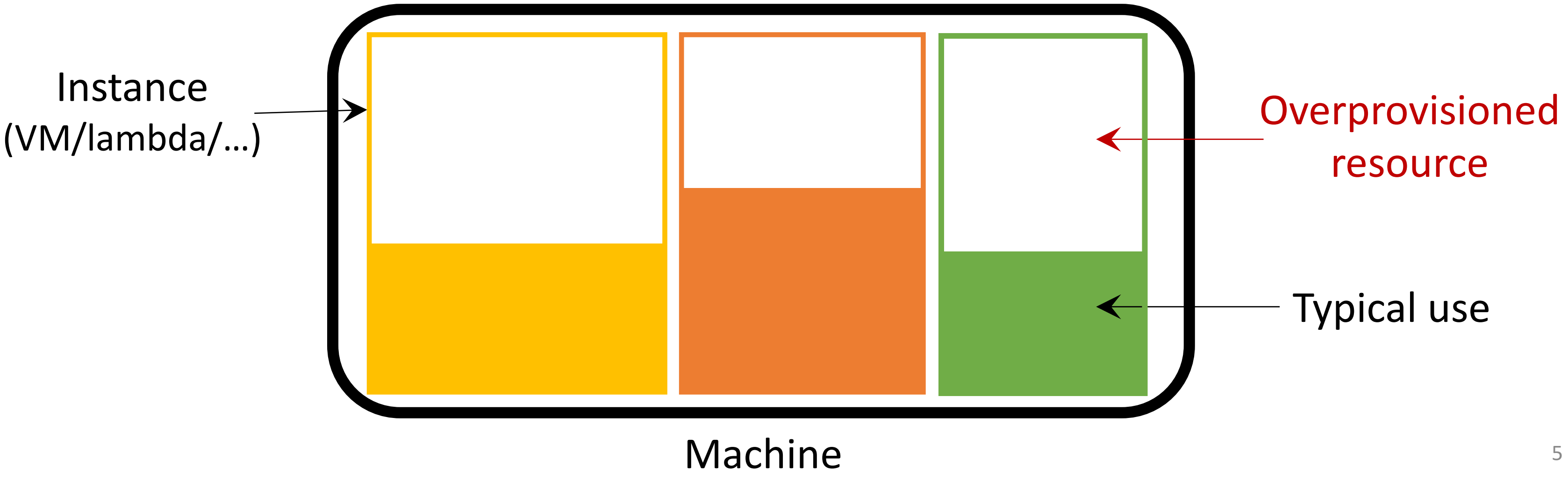
- Resource demands are often variable and hard to predict.



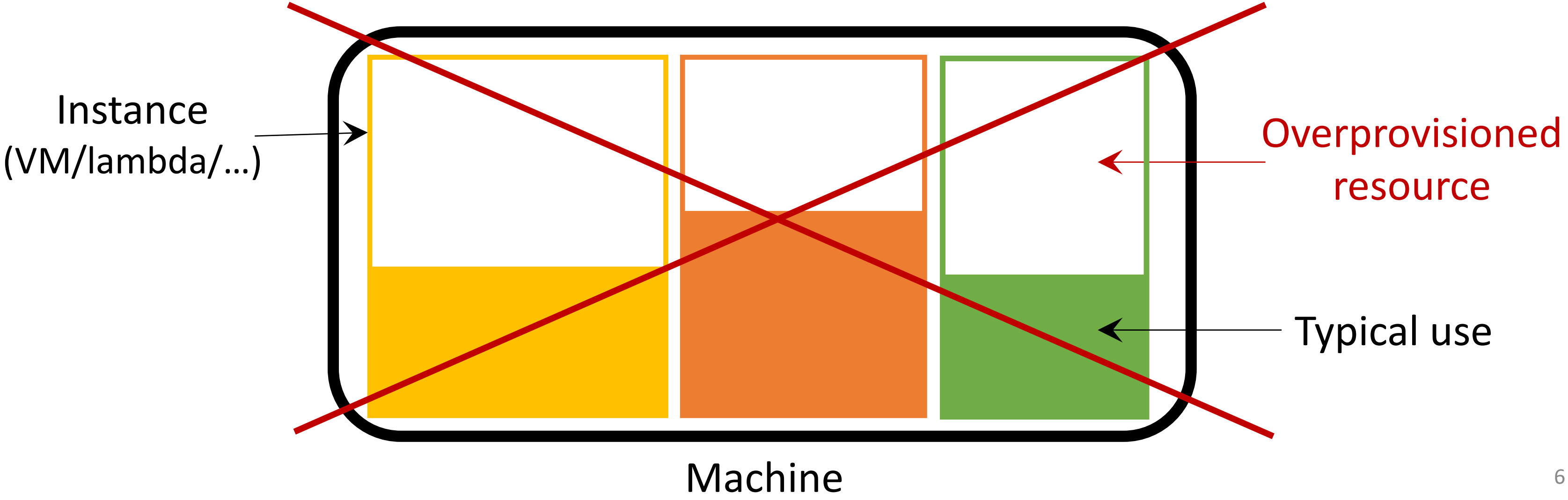
Inefficiency: resource overprovisioning

- Resource demands are often variable and hard to predict.

➢ Users have to overprovision resource for peak usage.

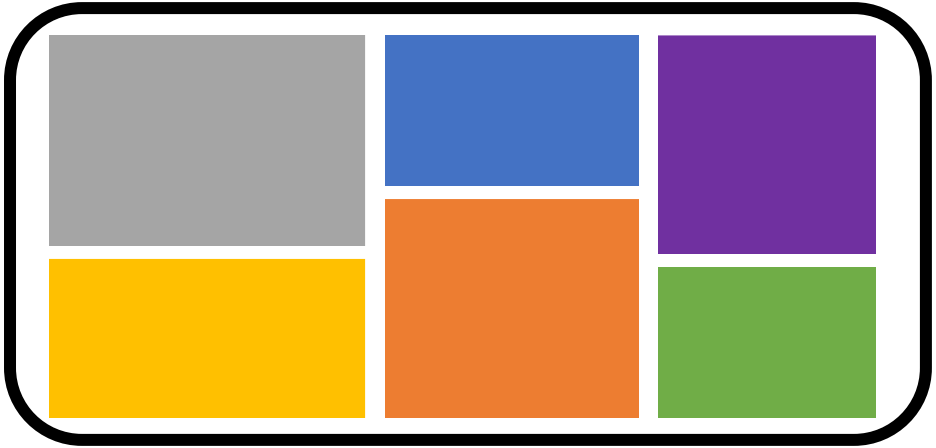


Can we avoid resource reservation?



Can we avoid resource reservation?

- Benefits: enables packing more apps.

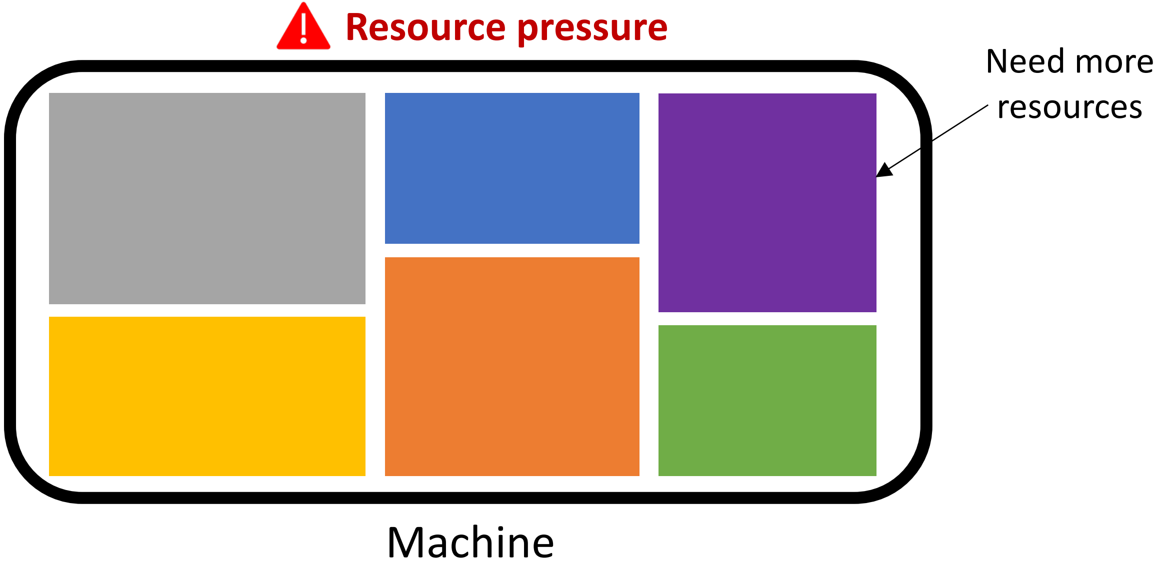


Machine

Can we avoid resource reservation?

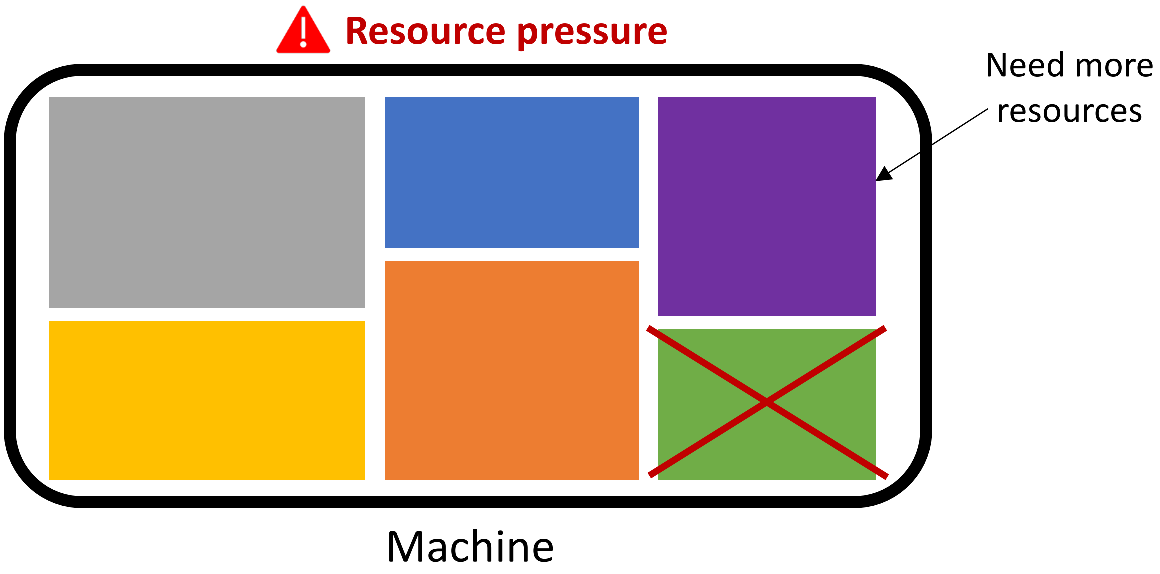
- Benefits: enables packing more apps.

➢ Problem: what if apps need more resource?



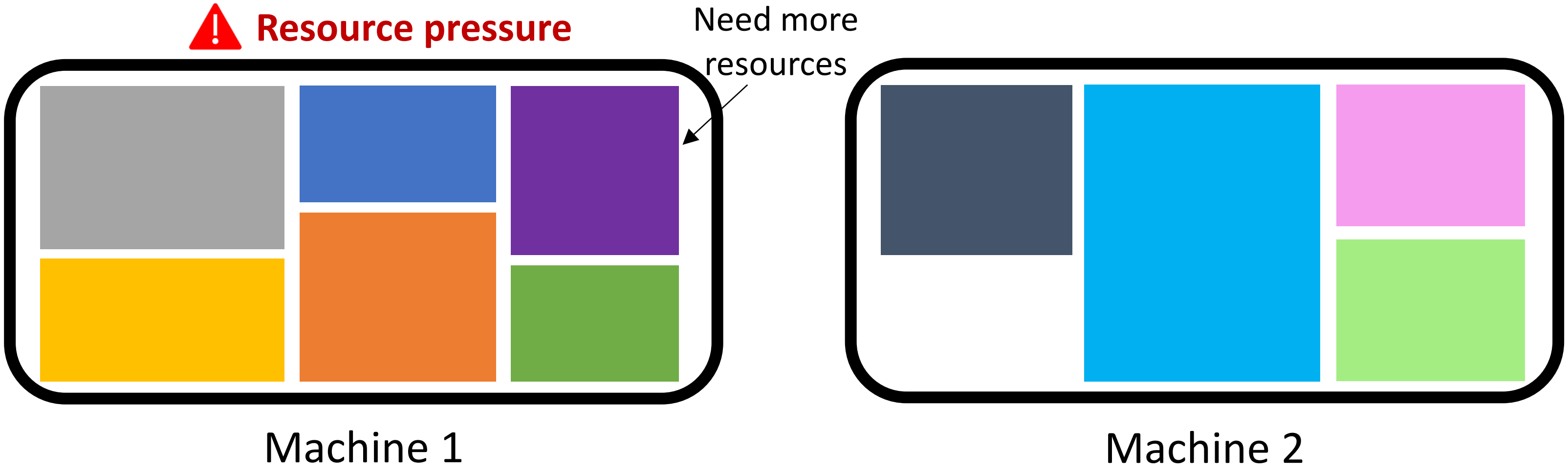
Strawman 1: kill applications

- Kill applications to make space.
- Unusable as it seriously disrupts victim's performance.



Strawman 2: migrate applications

- Migrate apps away from the overloaded machine.

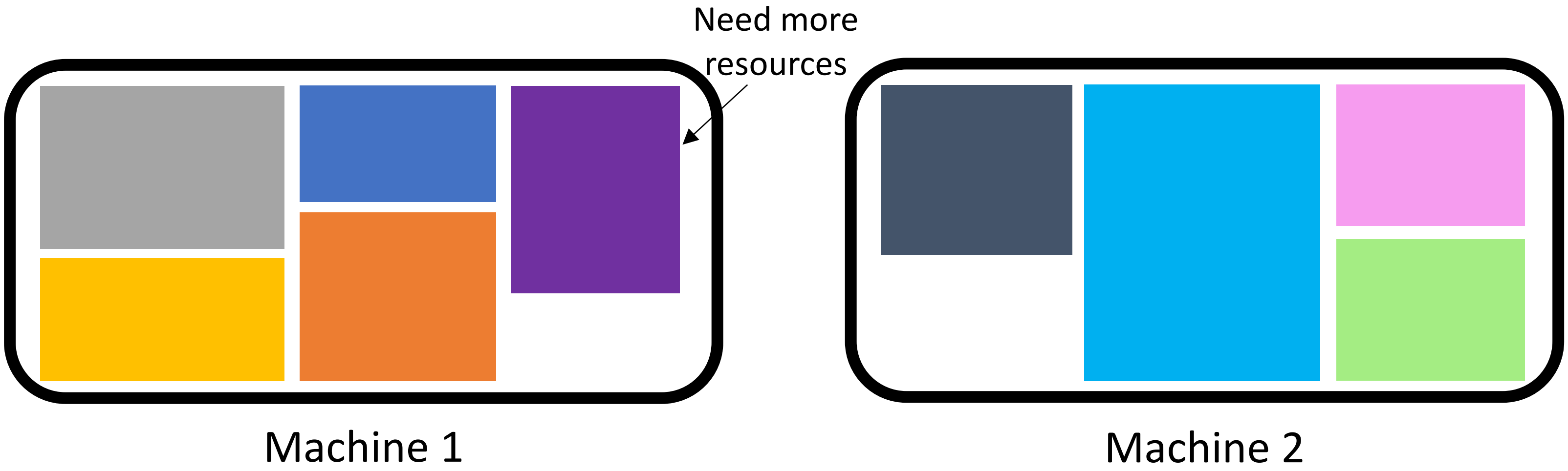


Strawman 2: migrate applications

- Migrate apps away from the overloaded machine.

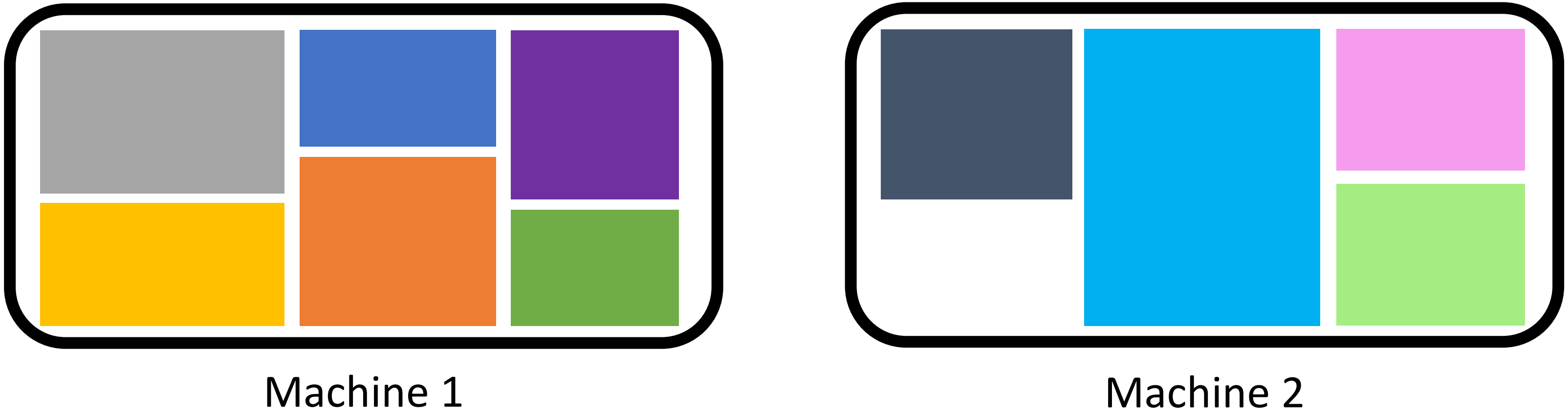
➢ Challenge: migration disrupts app's performance.

  -  E.g., takes seconds/minutes to migrate a VM.



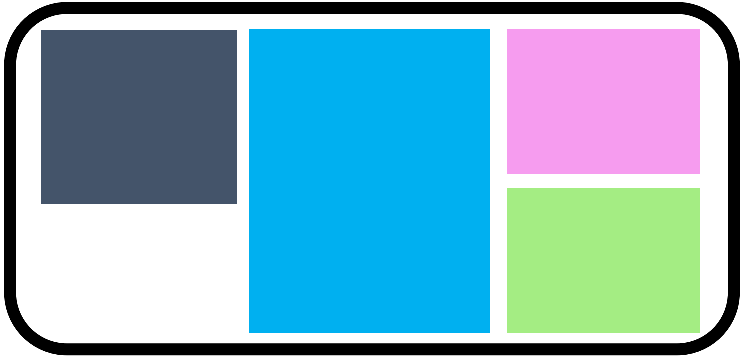
Design goal: achieving resource fungibility

- **Fungibility**: the ability to interchangeably use resources across machines w/o disruption.



Key Idea: fine-grained decomposition

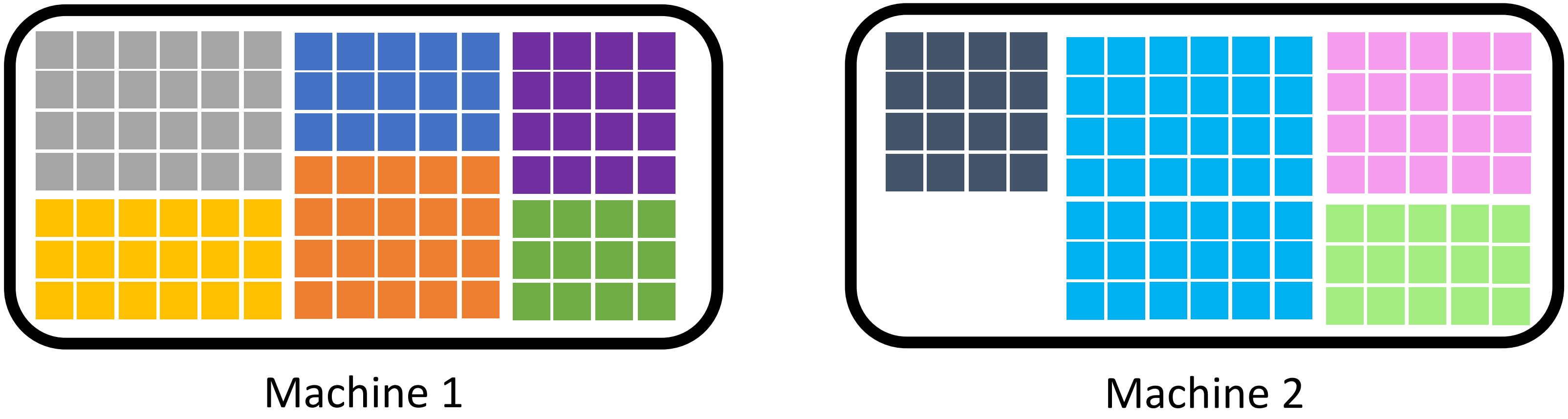
Machine 1



Machine 2

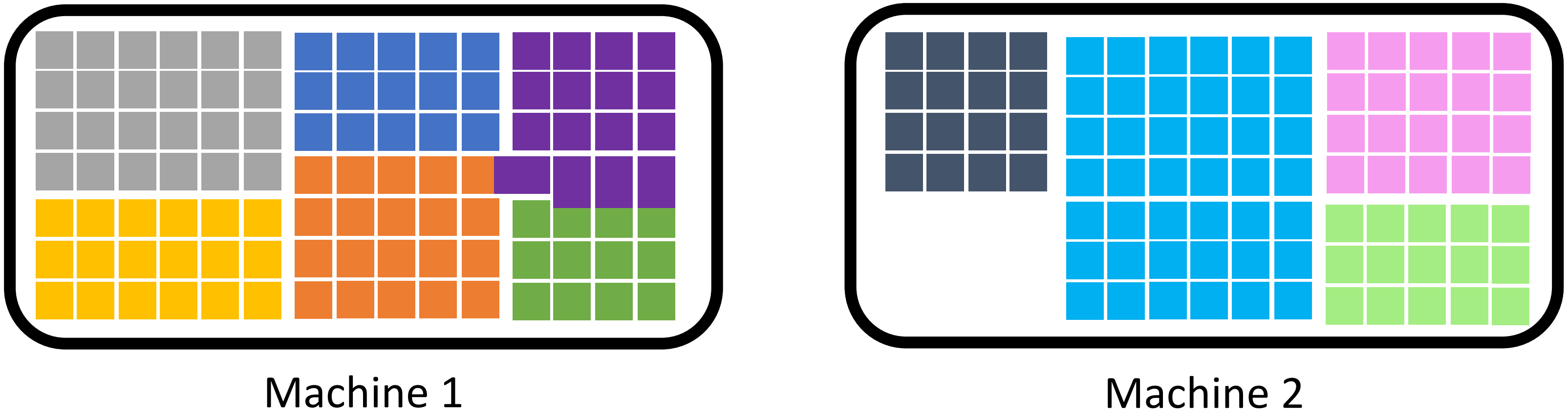
Key Idea: fine-grained decomposition

- Decompose apps into granular units.



Key Idea: fine-grained decomposition

- Decompose apps into granular units.
- Upon resource pressure, rapidly migrate units.
  - Only need to migrate the right amount of units.
  - ***~100 μs***/unit, orders of magnitude faster than migrating a VM.



Challenges and design overview

Challenges	Nu's Design
Migration can disrupt app's performance	Decompose apps into small rapidly-migratable units

Challenges and design overview

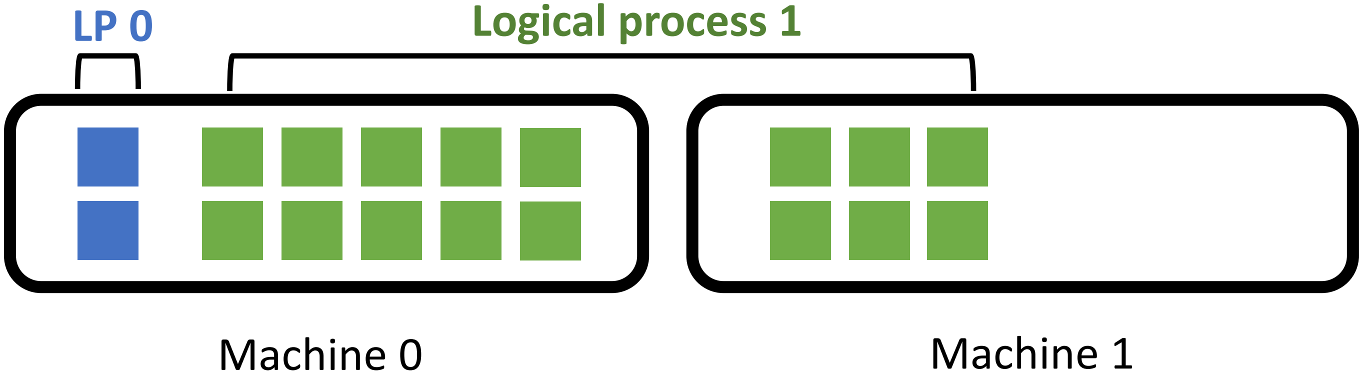
Challenges	Nu's Design
Migration can disrupt app's performance	Decompose apps into small rapidly-migratable units
Programming with small units is challenging	
Decomposition can increase the communication cost	

Challenges and design overview

Challenges	Nu's Design
Migration can disrupt app's performance	Decompose apps into small rapidly-migratable units
Programming with small units is challenging	A familiar process-like programming model
Decomposition can increase the communication cost	An efficient locality-aware communication runtime

The logical process abstraction

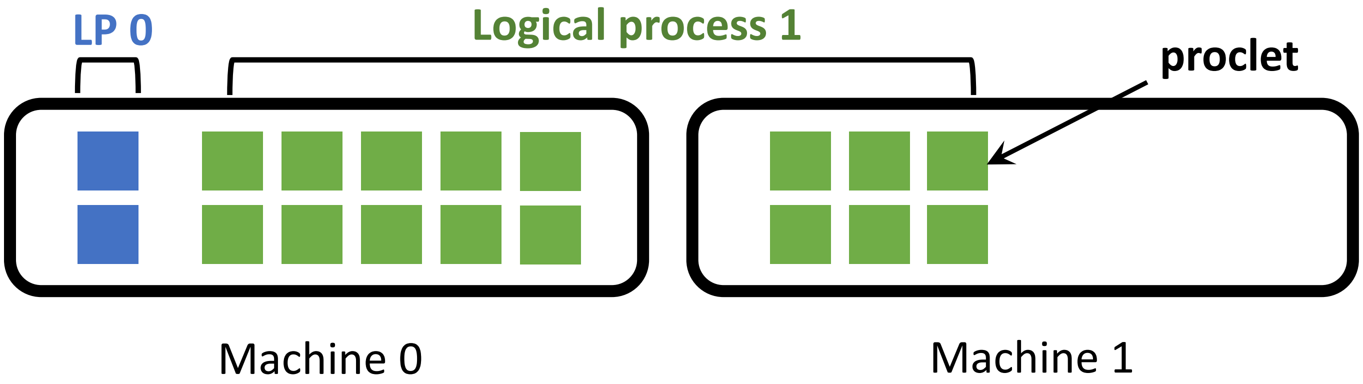
- Similar to the UNIX process, but can span across machines.



The logical process abstraction

- Similar to the UNIX process, but can span across machines.

➢ Consists of many smaller **proclets**.
  - An atomic unit of states and compute.
  - Can be independently migrated across machines.



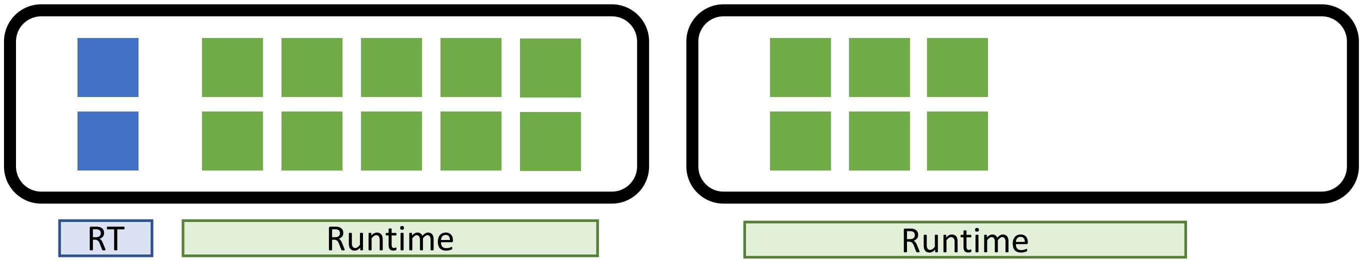
Proclet communication

- Proclets communicate through message passing.
  - No memory sharing → avoids expensive cache coherency.

Proclet communication

- Proclets communicate through message passing.
  - No memory sharing → avoids expensive cache coherency.

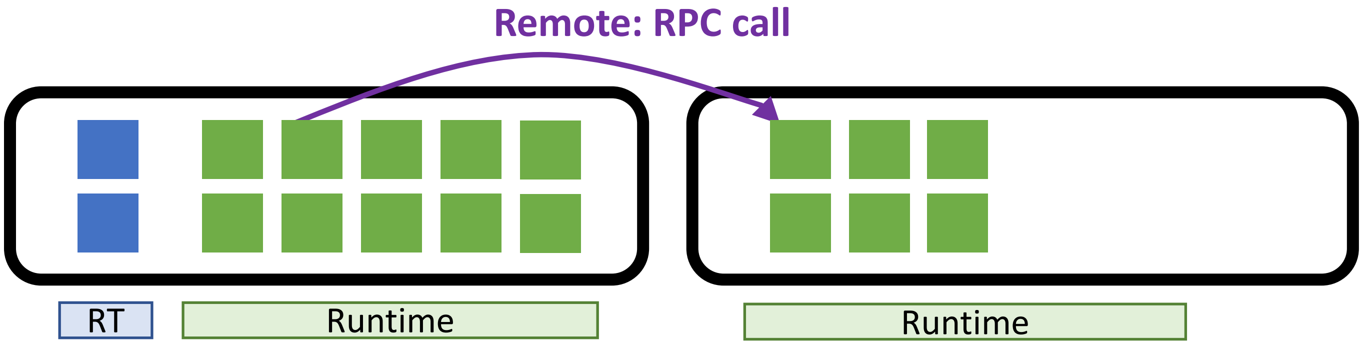
➢Runtime offers location transparency and optimizes performance.



Proclet communication

- Proclets communicate through message passing.
  - No memory sharing → avoids expensive cache coherency.

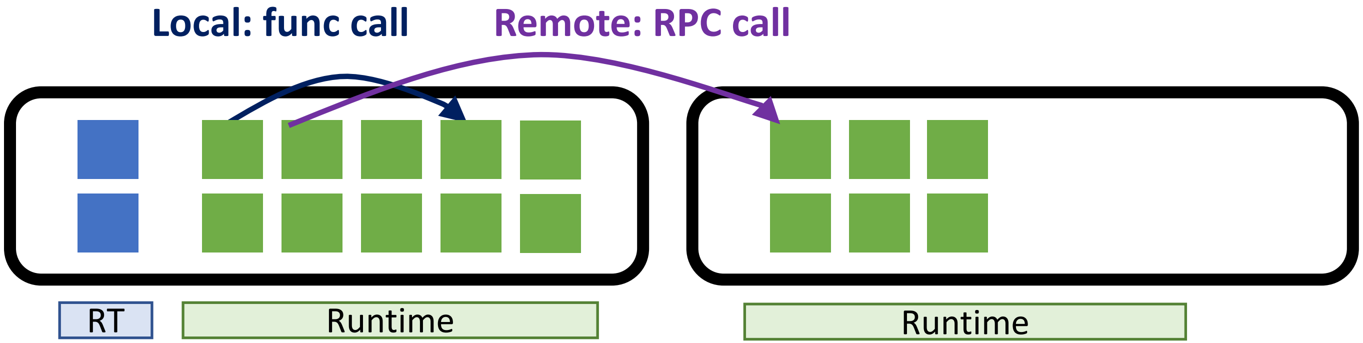
➢Runtime offers location transparency and optimizes performance.



Proclet communication

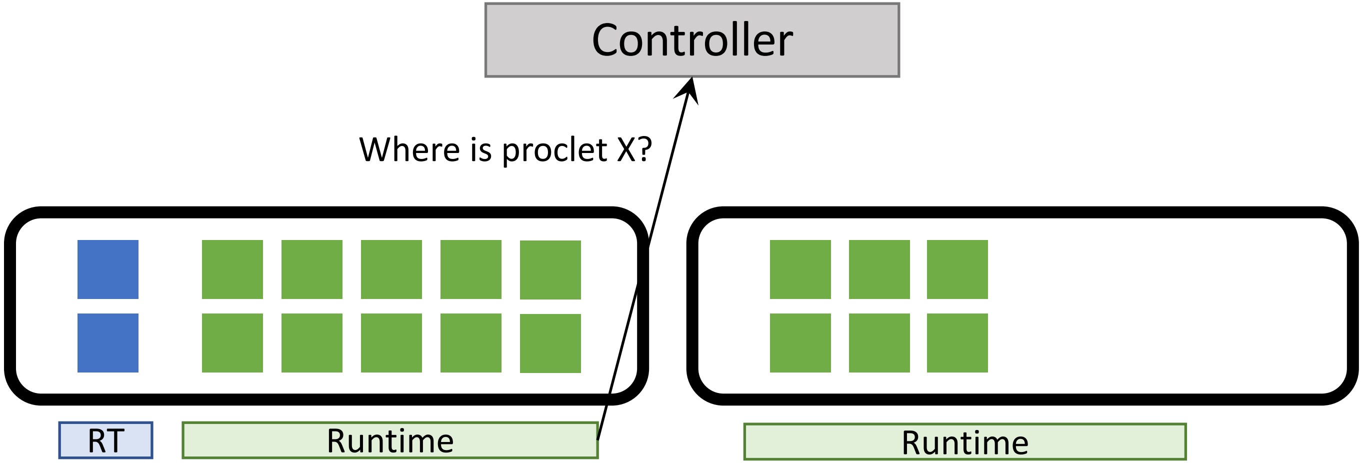
- Proclets communicate through message passing.
  - No memory sharing → avoids expensive cache coherency.

➢ Runtime offers location transparency and optimizes performance.



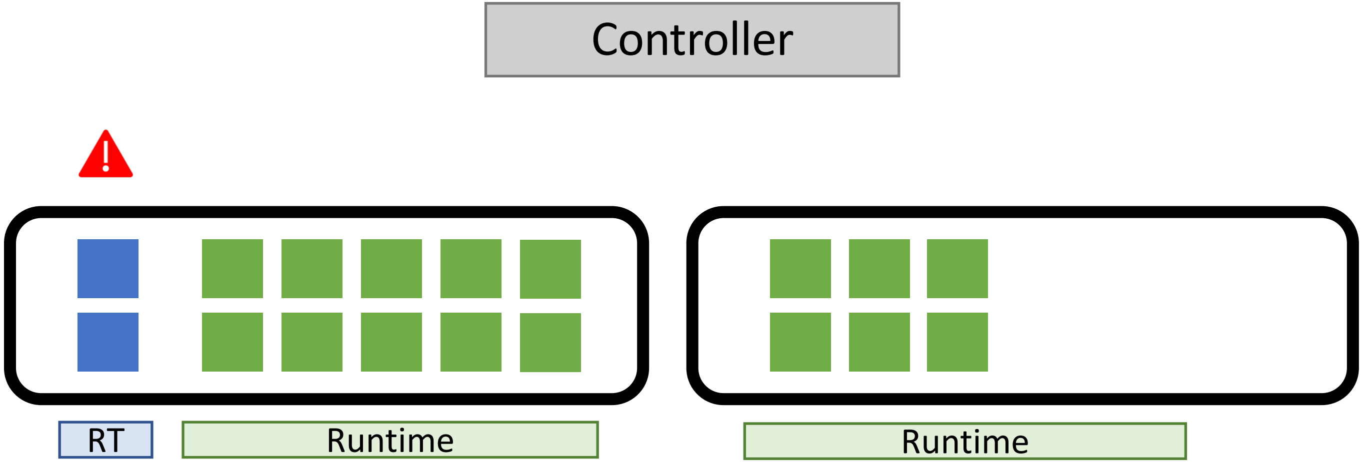
Centralized controller

- Tracks proclet locations and machine resources.
- Runtime caches location results to improve performance.



Proclet migration

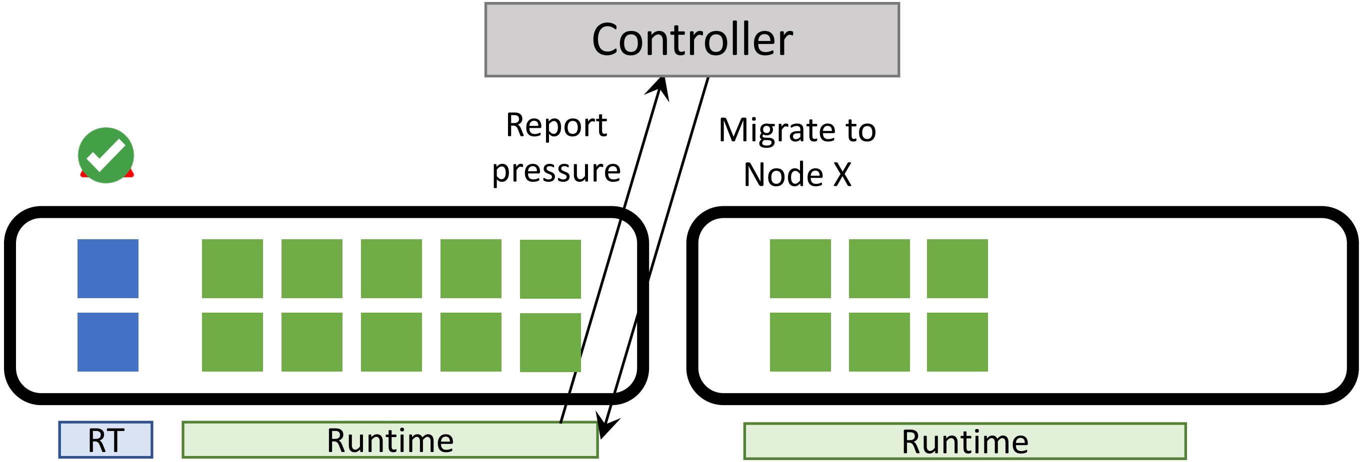
➢Runtime detects pressure and controller decides the new location.



Proclet migration

- Runtime detects pressure and controller decides the new location.

➢Rapidly migrate one proclet at a time.



Programming with Proclets

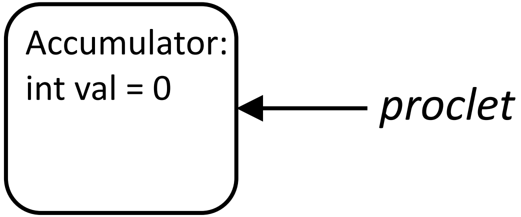
```
struct Accumulator {
    std::atomic<int> val_ = 0;
    void Add(int n) { val_ += n; }
    int Get() { return val; }
};
```

- Proclet class definition
  - Member variables are stored in proclet's heap
  - Public methods define the communication interface

Creating proclets

```
struct Accumulator {
    std::atomic<int> val_ = 0;
    void Add(int n) { val_ += n; }
    int Get() { return val; }
};
```

```
auto p = make_proclet<Accumulator>();
```

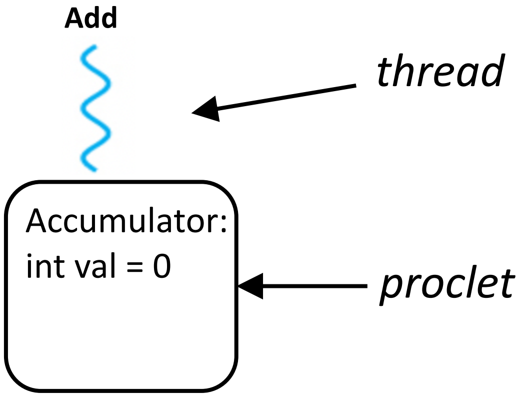


- Proclet smart pointer (like std::shared_ptr)
    - Can be passed as function arguments
    - Automatic lifetime management

Asynchronous Call

```
struct Accumulator {
    std::atomic<int> val_ = 0;
    void Add(int n) { val_ += n; }
    int Get() { return val; }
};

auto p = make_proclet<Accumulator>();
auto f0 = p.RunAsync(&Accumulator::Add, 1);
```

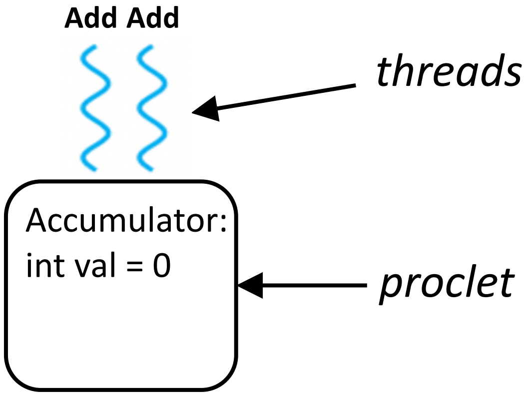


- Asynchronous call for latency hiding

Asynchronous Call

```
struct Accumulator {
    std::atomic<int> val_ = 0;
    void Add(int n) { val_ += n; }
    int Get() { return val; }
};
```

```
auto p = make_proclet<Accumulator>();
auto f0 = p.RunAsync(&Accumulator::Add, 1);
auto f1 = p.RunAsync(&Accumulator::Add, 1);
```

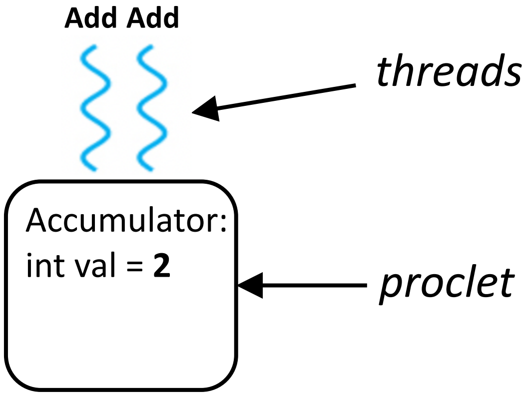


- Asynchronous call for latency hiding

Asynchronous Call

```
struct Accumulator {
    std::atomic<int> val_ = 0;
    void Add(int n) { val_ += n; }
    int Get() { return val; }
};

auto p = make_proclet<Accumulator>();
auto f0 = p.RunAsync(&Accumulator::Add, 1);
auto f1 = p.RunAsync(&Accumulator::Add, 1);
f0.get(); f1.get();
```

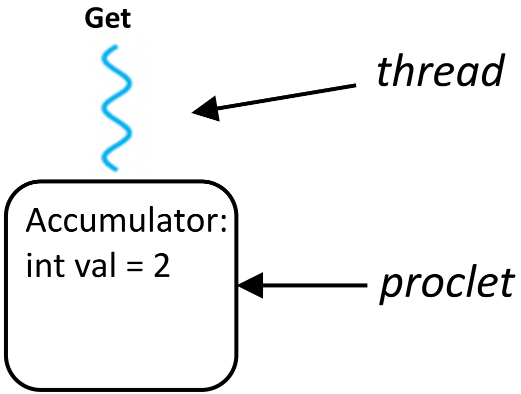


- Asynchronous call for latency hiding

Synchronous Call

```
struct Accumulator {
    std::atomic<int> val_ = 0;
    void Add(int n) { val_ += n; }
    int Get() { return val; }
};


auto p = make_proclet<Accumulator>();
auto f0 = p.RunAsync(&Accumulator::Add, 1);
auto f1 = p.RunAsync(&Accumulator::Add, 1);
f0.get(); f1.get();
auto val = p.Run(&Accumulator::Get); // = 2
```

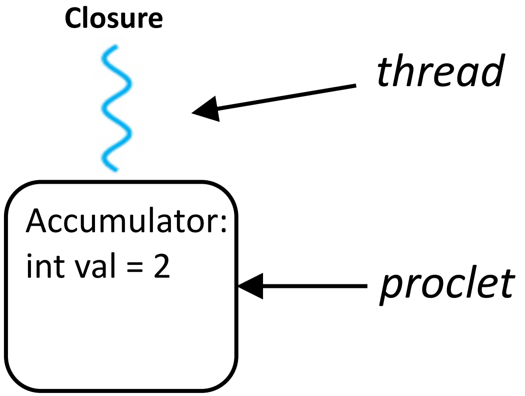


- ➢ Also supports simple synchronous call

Computation shipping

```
struct Accumulator {
    std::atomic<int> val_ = 0;
    void Add(int n) { val_ += n; }
    int Get() { return val; }
};

auto p = make_proclet<Accumulator>();
auto f0 = p.RunAsync(&Accumulator::Add, 1);
auto f1 = p.RunAsync(&Accumulator::Add, 1);
f0.get(); f1.get();
auto val = p.Run(&Accumulator::Get); // = 2
val = p.Run(+[](Accumulator &a) {
  a.Add(1); return a.Get(); }); // = 3
```



➢ Ships a closure with very low overhead

More details in the paper

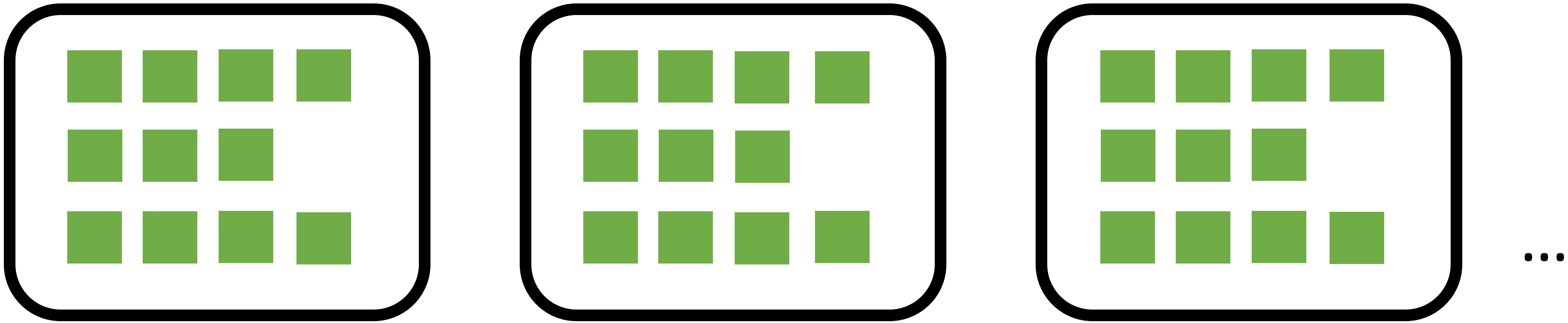
- Fault tolerance and proclet replication.
- Security and threat model.
- Placement and migration policy.
- Migration and RPC optimizations.

Evaluation

- Setup: 32 machines in a rack connected with 100GbE
- Applications:
  - Social network microservices (from DeathStarBench).
  - Key-value store.
  - Phoenix (a C++ MapReduce framework)
- Focus on answering followings:
  - Can we reconcile tensions between utilization and performance?
  - How fast can we migrate proclets across machines?

Able to achieve high utilization w/o disruption?

- Initially run the social network app across all 32 nodes.

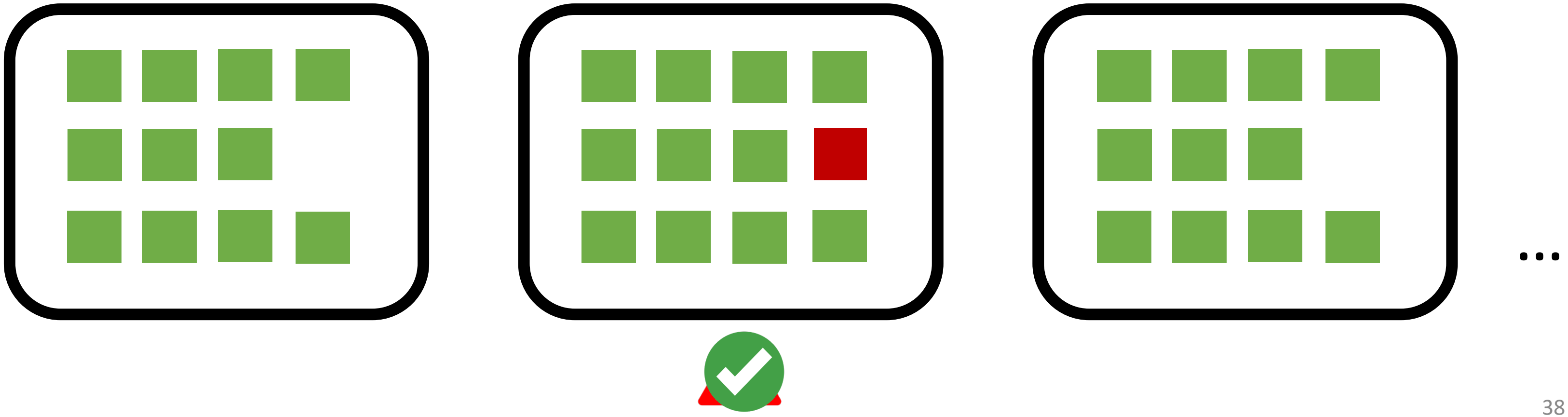


Able to achieve high utilization w/o disruption?

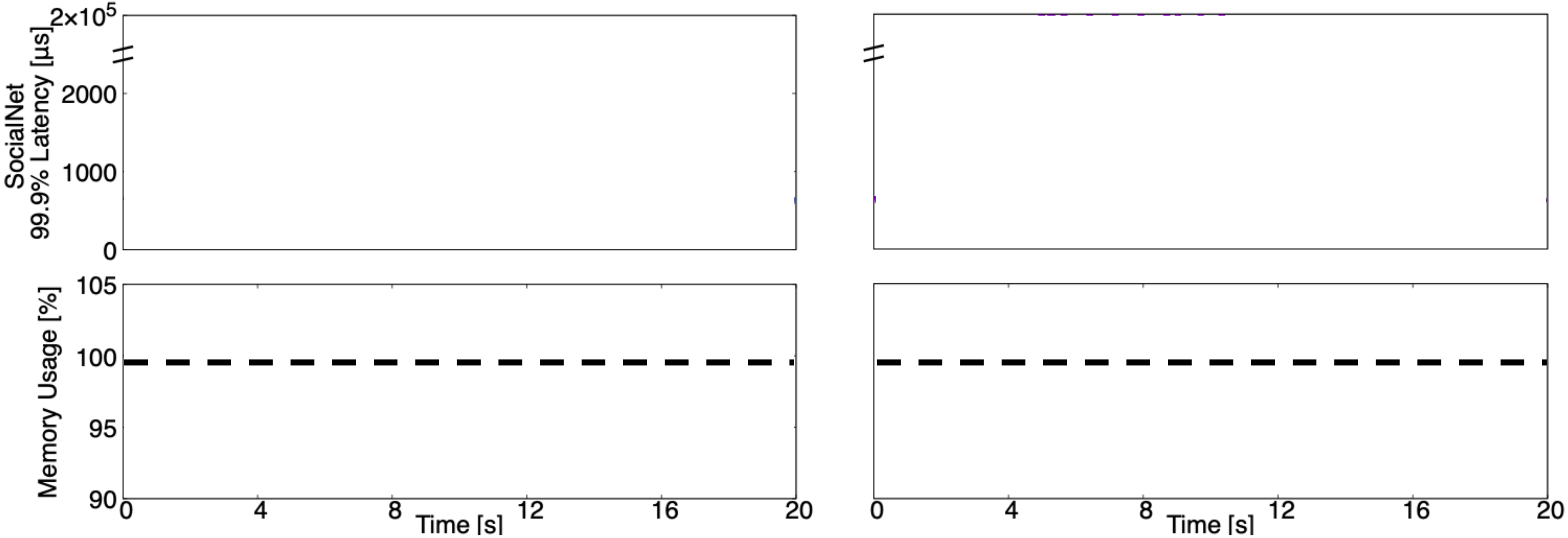
- Initially run the social network app across all 32 nodes.

➢ Then launch the memory antagonist at one node.

  - Allocates memory as fast as possible, around 7 GB/s.



Able to achieve high utilization w/o disruption?

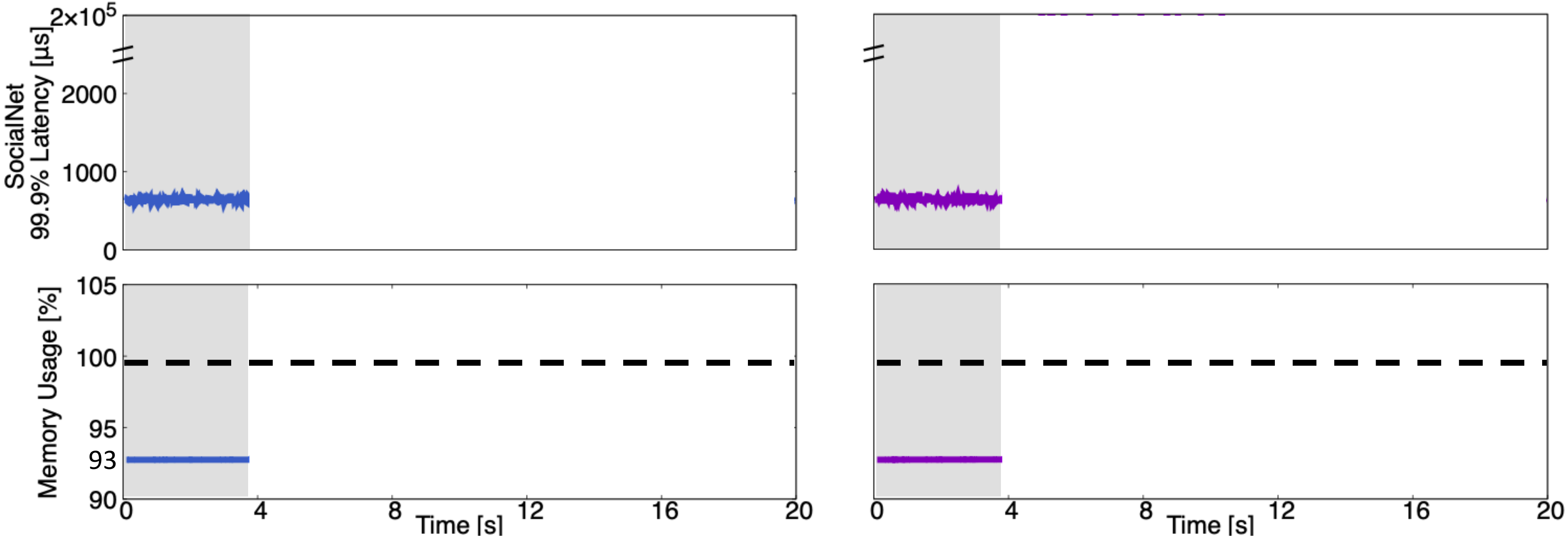


Nu

MigrOS [ATC'21]

(container live migration)

Able to achieve high utilization w/o disruption?

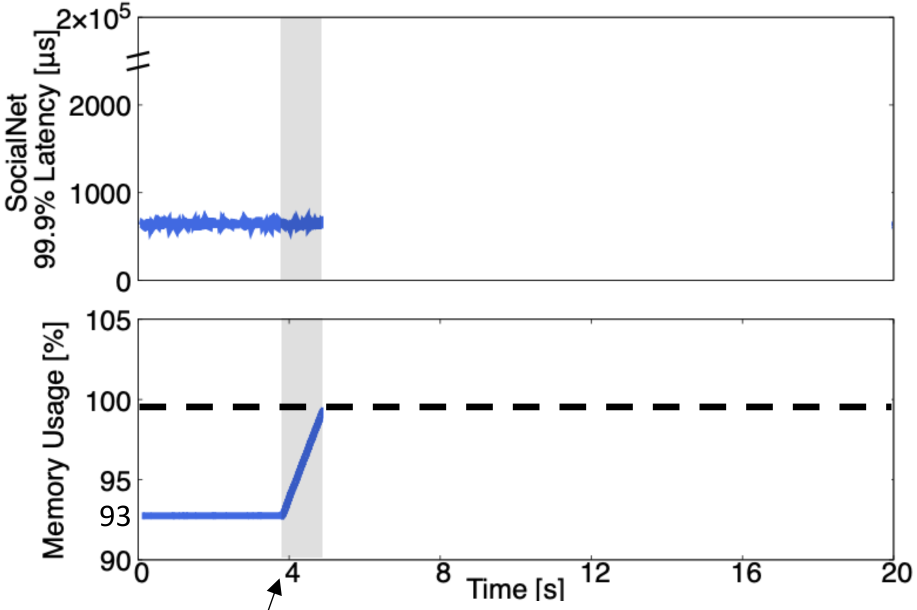


Nu

MigrOS [ATC'21]

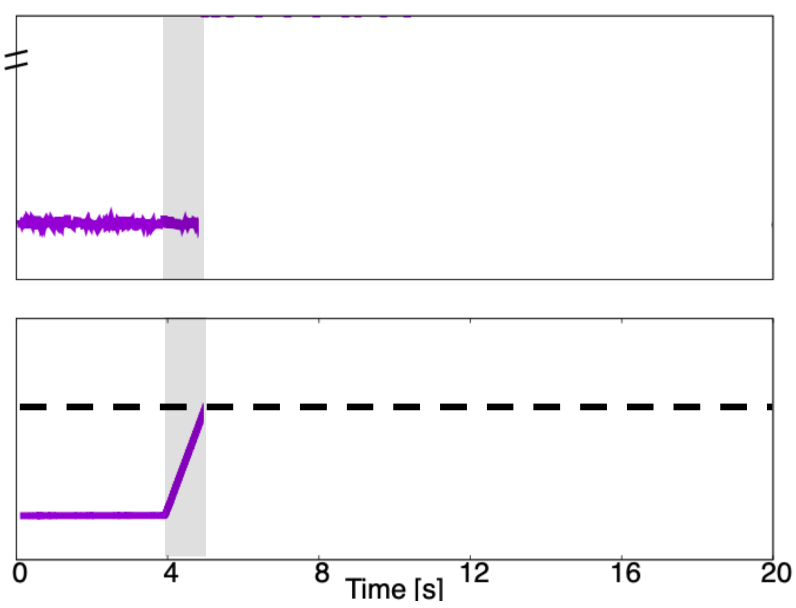
(container live migration)

Able to achieve high utilization w/o disruption?



Antagonist starts

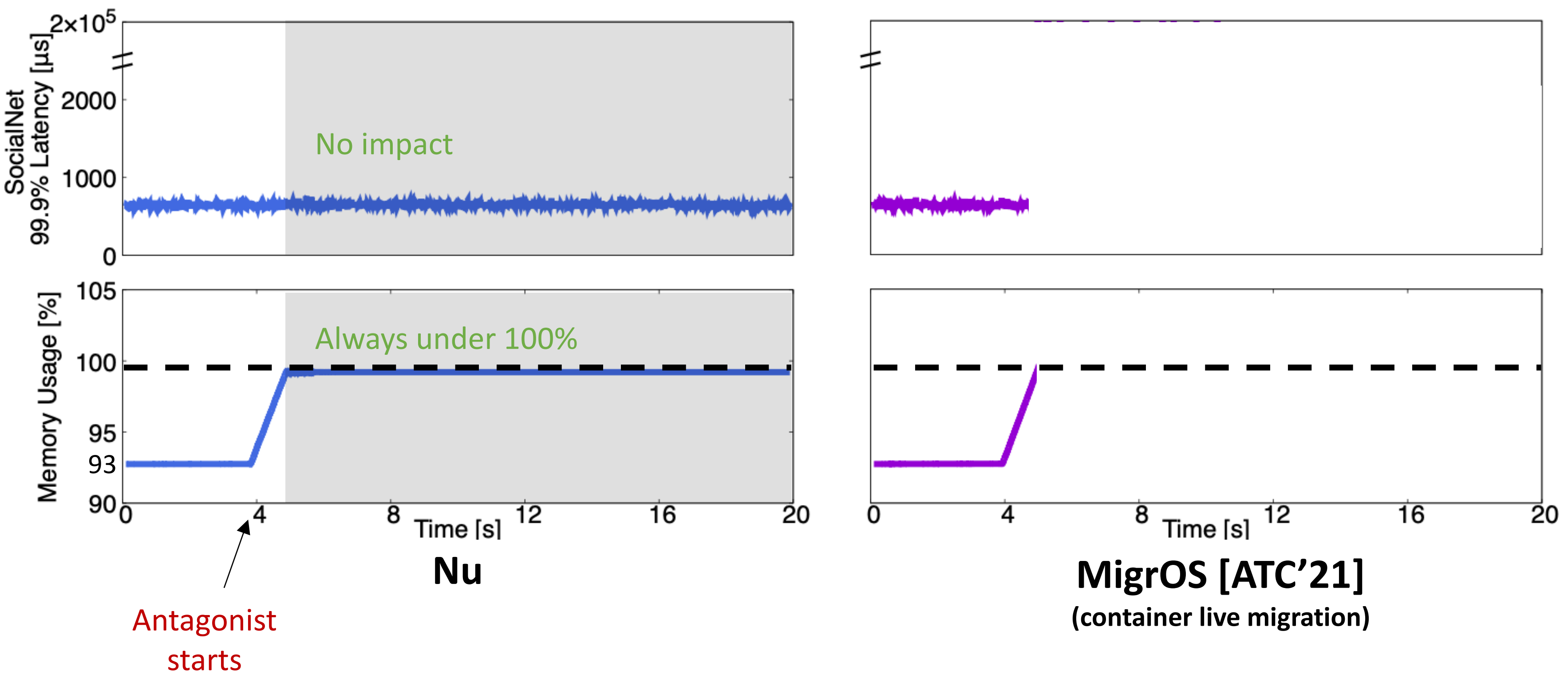
Nu



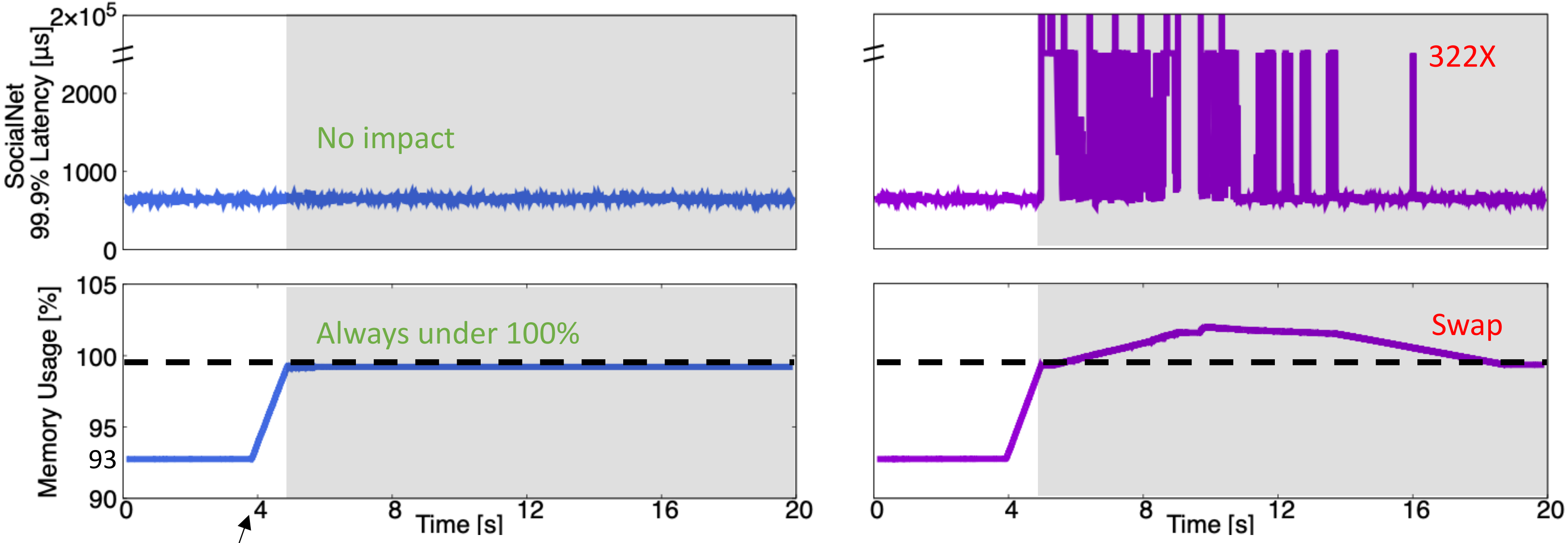
MigrOS [ATC'21]

(container live migration)

Able to achieve high utilization w/o disruption?



Able to achieve high utilization w/o disruption?



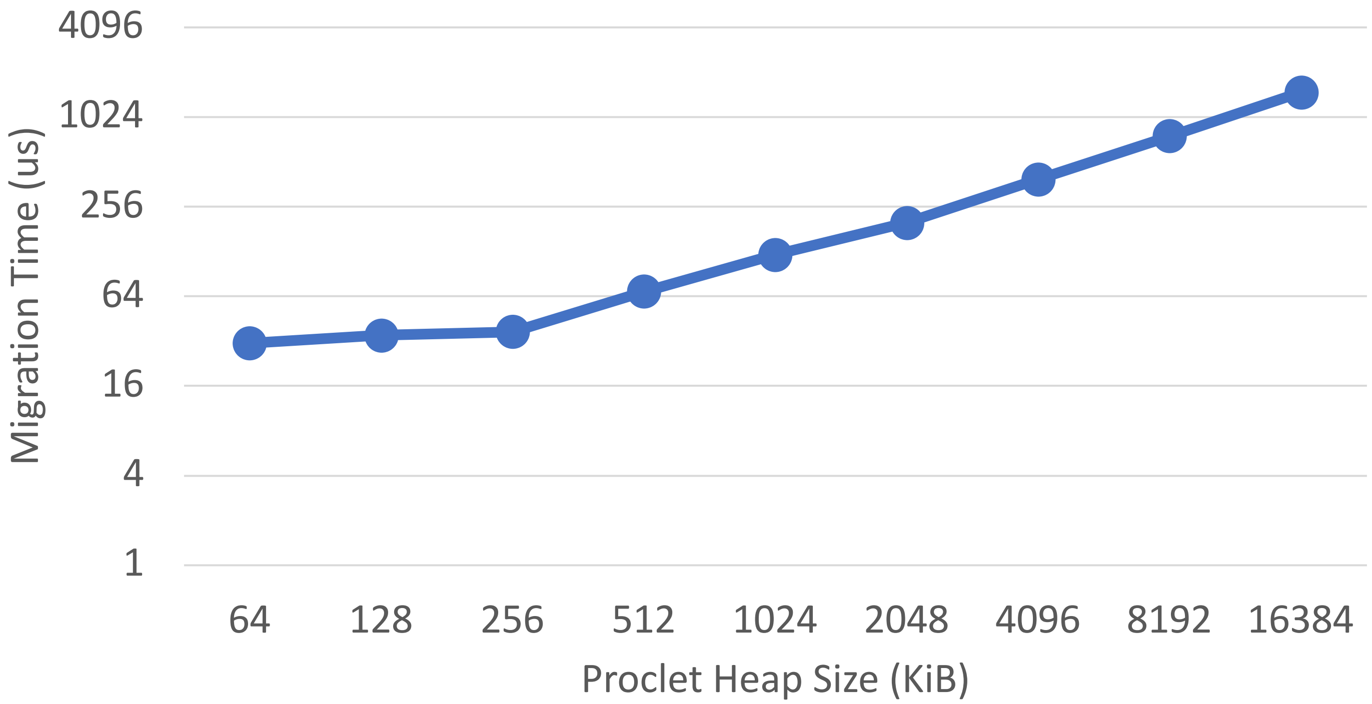
Antagonist starts

Nu

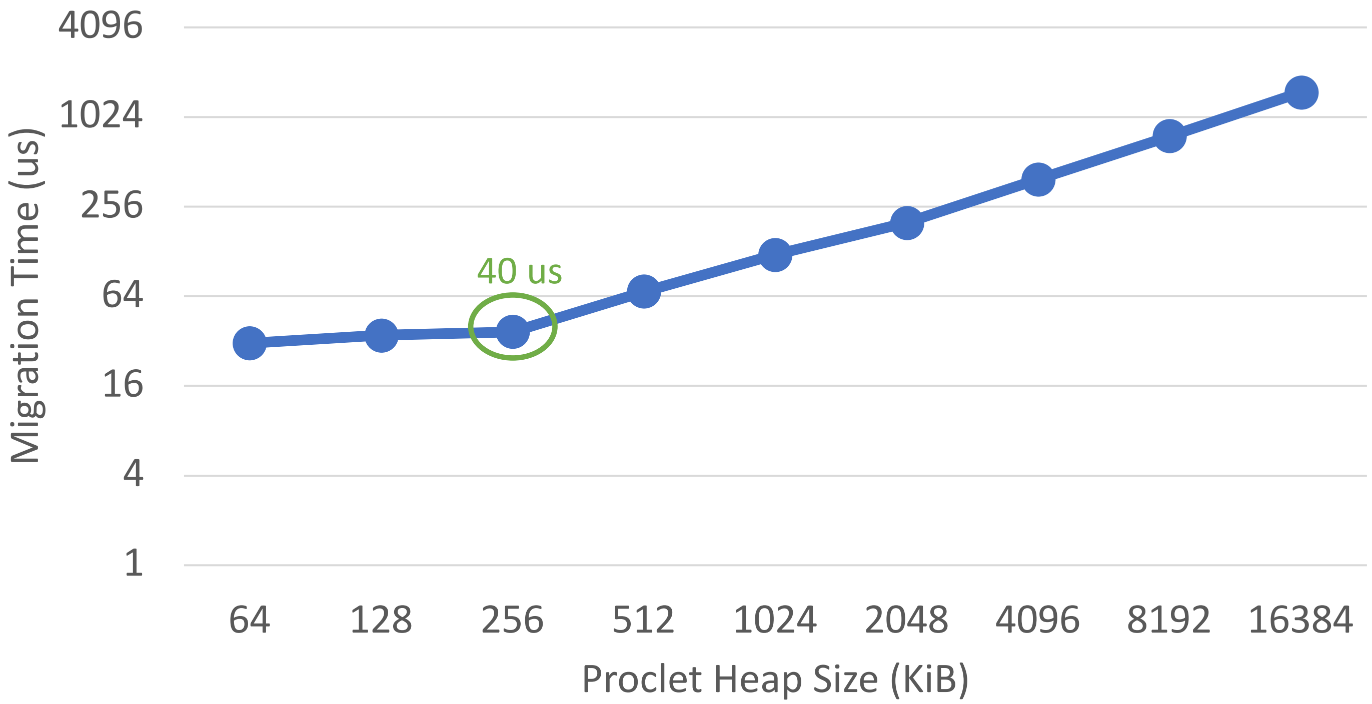
MigrOS [ATC'21]

(container live migration)

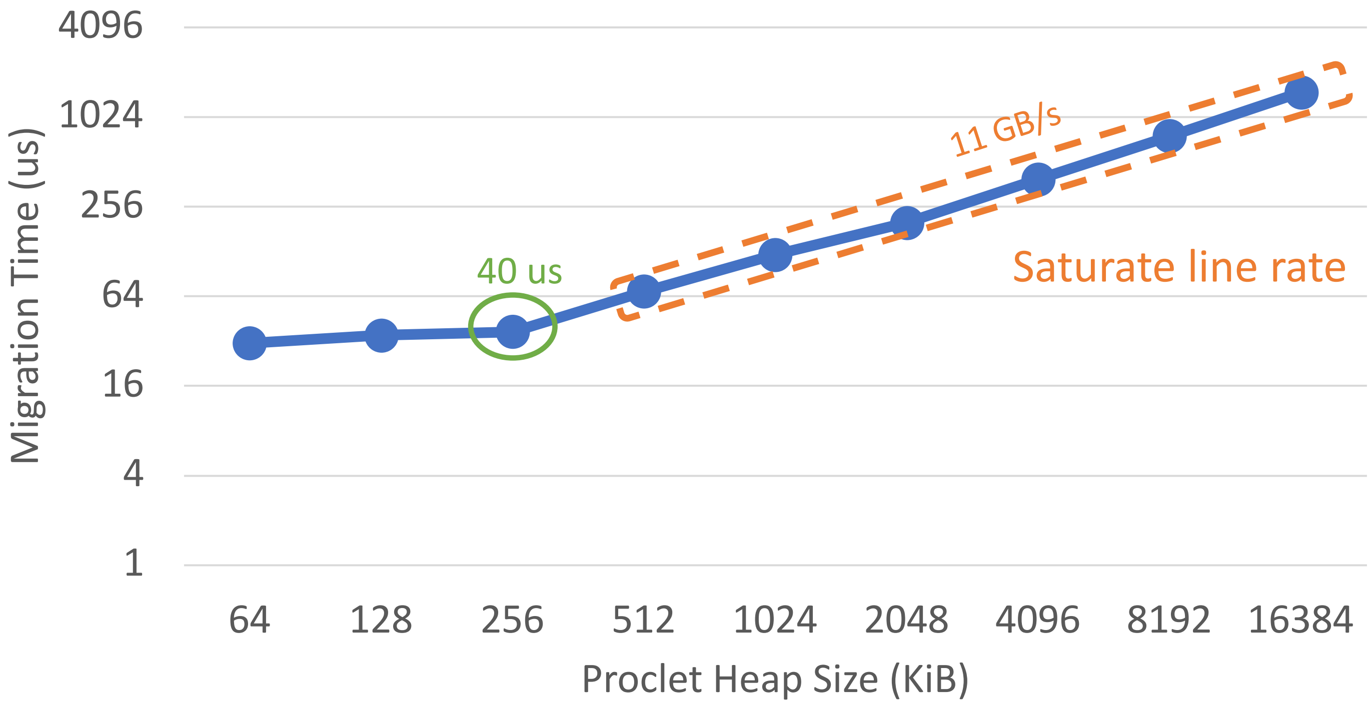
How fast can we migrate proclets?



How fast can we migrate proclets?



How fast can we migrate proclets?



More results in the paper

- Other applications: KV store, Phoenix.
- React quickly to CPU pressure as well.
- Scale linearly with the number of machines.
- Match/exceed the performance of existing implementations.

Related Work

- Other migration systems.
  - VM/container/process live migration: MigrOS [ATC’ 21]
- Other programming models.
  - Distributed objects: Orca [OOPSLA’ 93]
  - Serverless: Boki [SOSP’ 21]
  - Actor: Ray [OSDI’ 18].
- Other options for fungibility.
  - Resource disaggregation: LegoOS [OSDI’ 18]
  - Load balancing: Slicer [OSDI’ 16]

Conclusion

- Resource overprovisioning impacts datacenter utilization.
- Nu's logical process avoids overprovisioning through fungibility.
- Key ideas: 1) decompose apps into granular proclets.
  2) rapidly migrate proclets upon pressure.
- Nu achieves high utilization without performance disruption.
- Code is available at https://github.com/Nu-NSDI23/Nu.