Fast and Flexible Containerization with Pipsqueak

Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

* Microsoft Gray Systems Lab

OpenLambda

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Containers in the Cloud

(1) Traditional Server Containers

- Runtime & server deployed as a container
- Flexible runtime, but slow startup

(2) Serverless Computing

- Containers/customers **share** a host server
- Fast startup, but inflexible runtime

Containers in the Cloud

(1) Traditional Server Containers

- Runtime & server deployed as a container
- Flexible runtime, but slow startup

(2) Serverless Computing

- Containers/customers **share** a host server
- Fast startup, but inflexible runtime

(2’) Pipsqueak - Flexible Serverless

- Secure, built-in package support
- 9-2000x speedups for single-package workloads

Containers in the Cloud

(1) Traditional Server Containers

- Runtime & server deployed as a container
- Flexible runtime, but slow startup

(2) Serverless Computing

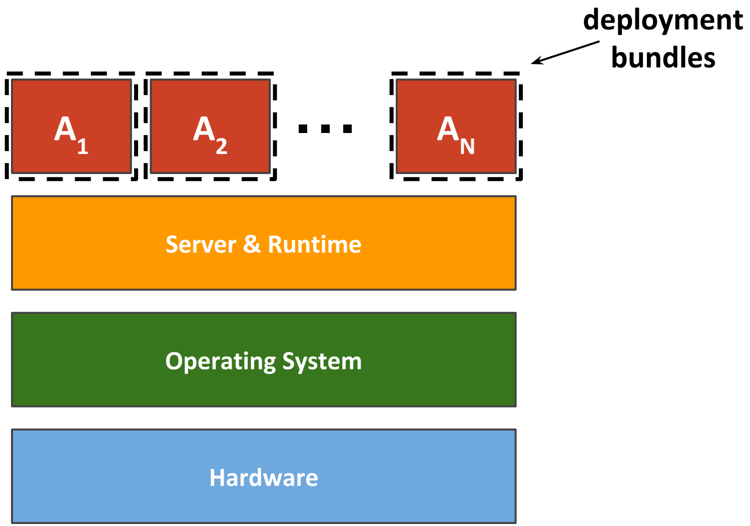
- Containers/customers **share** a host server
- Fast startup, but inflexible runtime

(2’) Pipsqueak - Flexible Serverless

- Secure, built-in package support
- 9-2000x speedups for single-package workloads

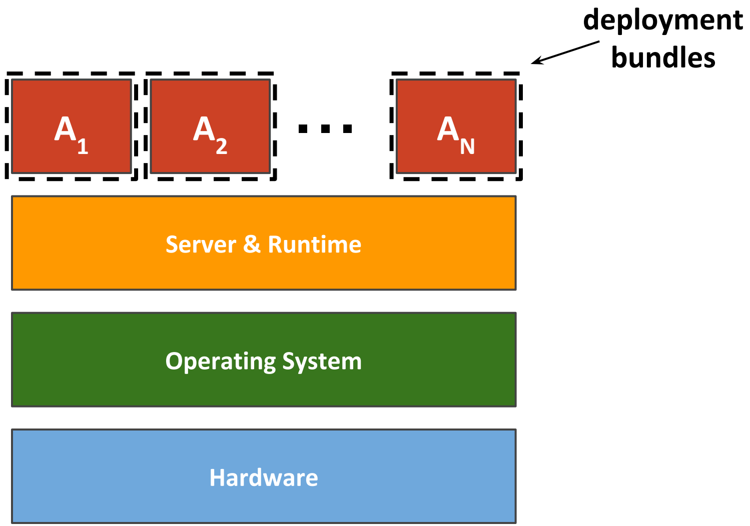
Microservices

- Applications are decoupled into modular pieces, or “services”
- Services are lightweight, making deployment and scaling less painful



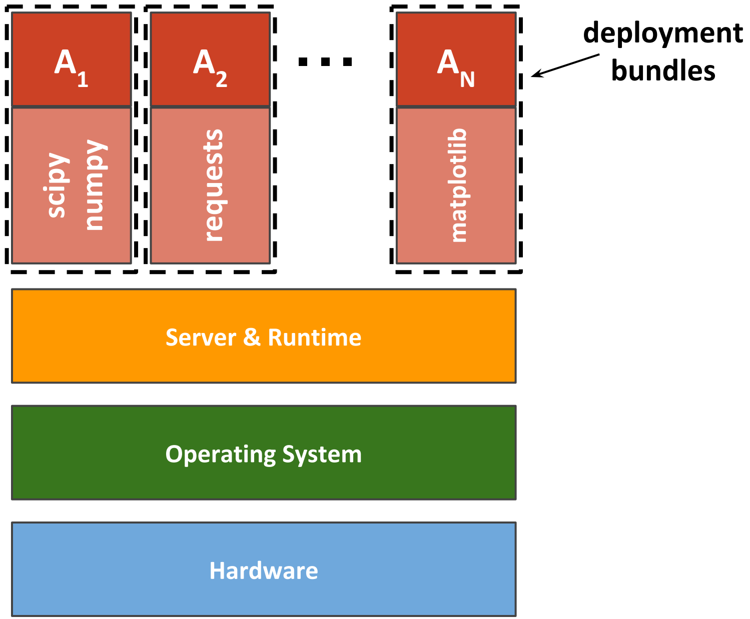
Microservices

- Applications are decoupled into modular pieces, or “services”
- Services are lightweight, making deployment and scaling less painful
  - Or are they?



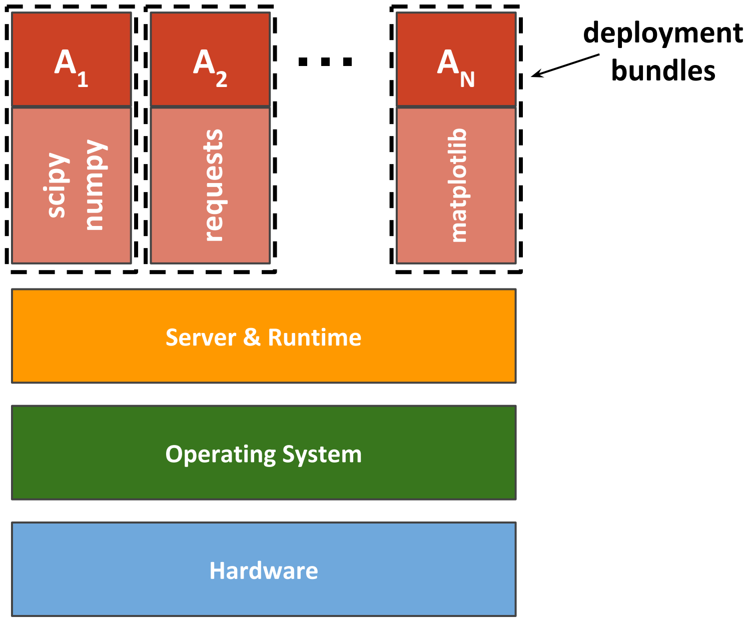
Microservices

- Applications are decoupled into modular pieces, or “services”
- Services are lightweight, making deployment and scaling less painful
    - Or are they?
    - *Problem*: developers depend on many userspace libraries



Microservices

- Applications are decoupled into modular pieces, or “services”
- Services are lightweight, making deployment and scaling less painful
  - Or are they?
  - *Problem*: developers depend on many userspace libraries



Matplotlib installation:

- 4.37s to download
- 5.24s to install
- 0.21s to import

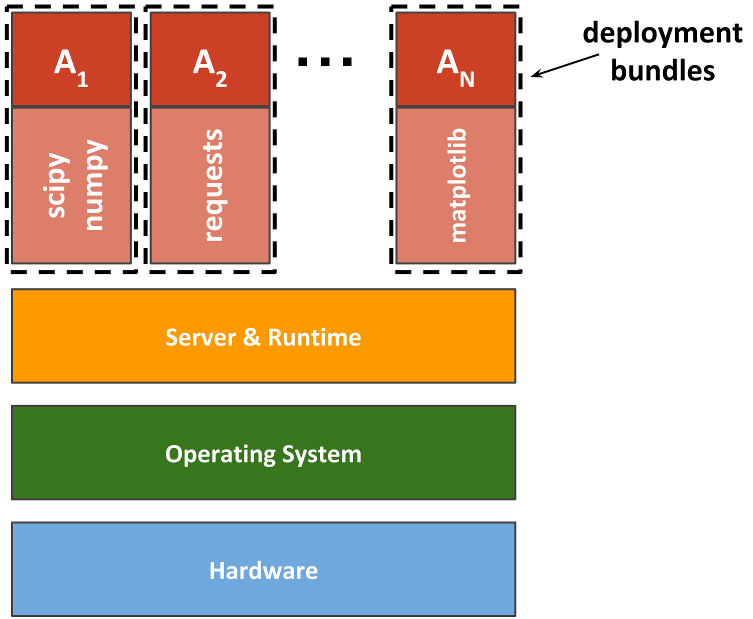
~~Microservices~~ MicroMonoliths

- Applications are decoupled into modular pieces, or “services”
- Services are lightweight, making deployment and scaling less painful
  - Or are they?
  - *Problem*: developers depend on many userspace libraries

Matplotlib installation:

- 4.37s to download
- 5.24s to install
- 0.21s to import

MicroMonolith - a conceptually small service that is inflated by large userspace libraries



Outline

Motivation

Python Packages

- Anatomy
- Analysis

Pipsqueak

- Handler cache
- Import cache

Evaluation

Conclusion

Installation Workflow

Download

numpy.tar.gz
requests.tar.gz
matplotlib.tar.gz
...

pip mirror

→

Install

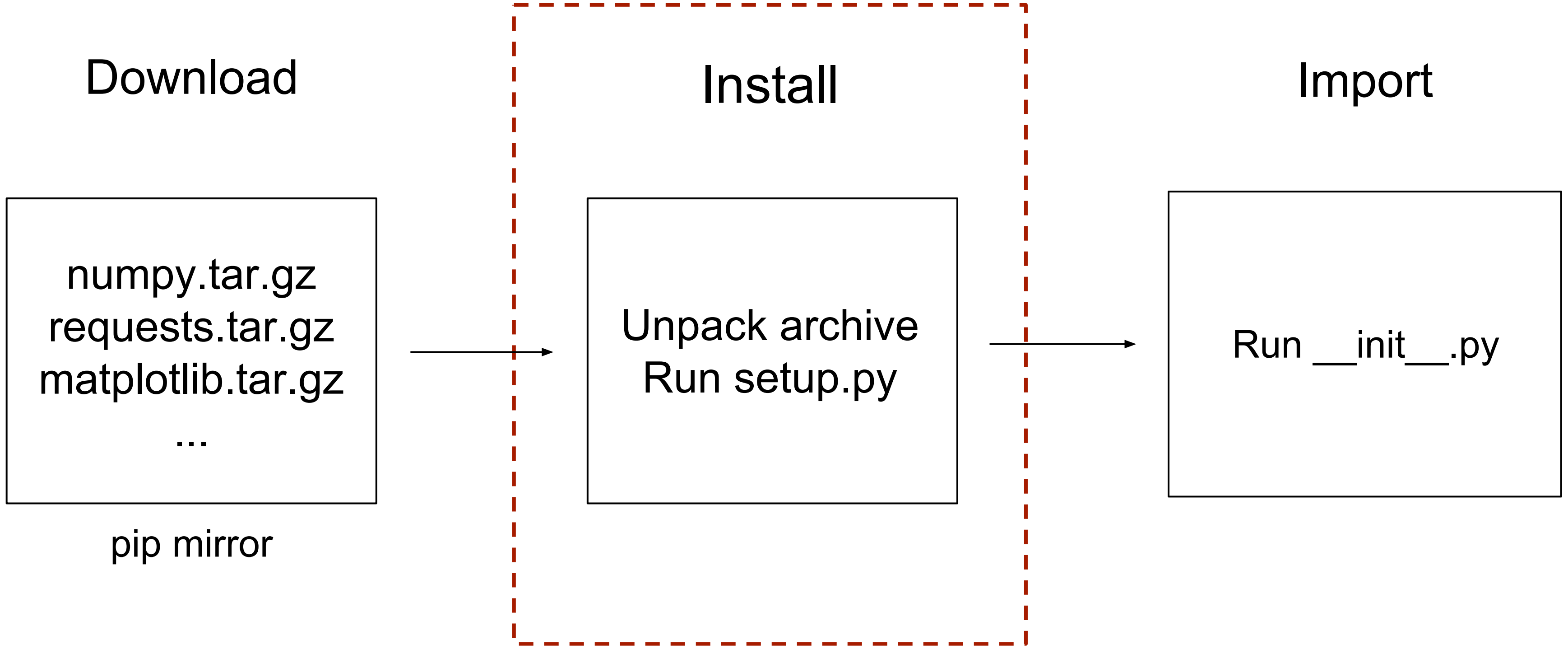
Unpack archive
Run setup.py

→

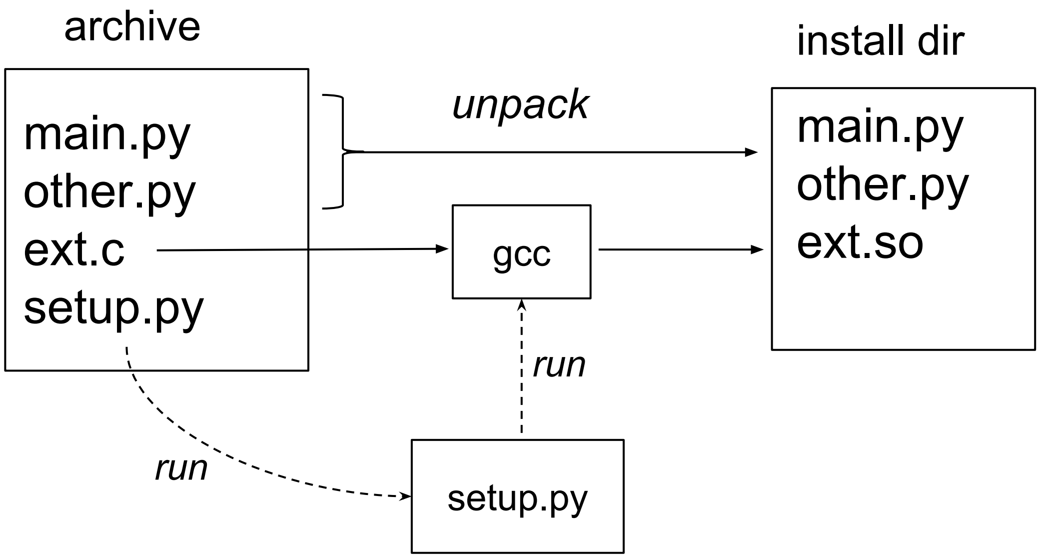
Import

Run __init__.py

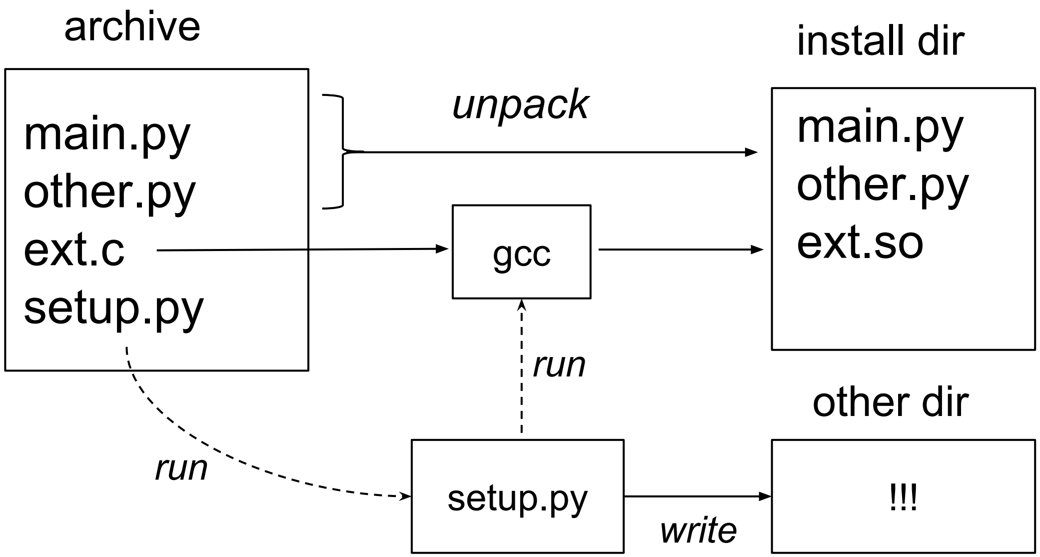
Installation Workflow



Install

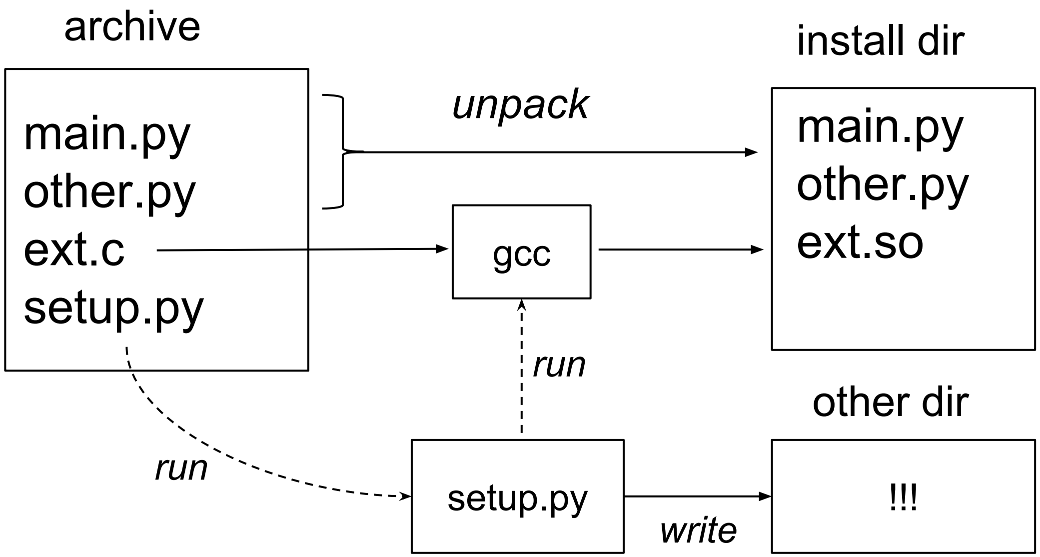

archive
main.py
other.py
ext.c
setup.py
unpack
install dir
main.py
other.py
ext.so
gcc
run
run
setup.py

Install



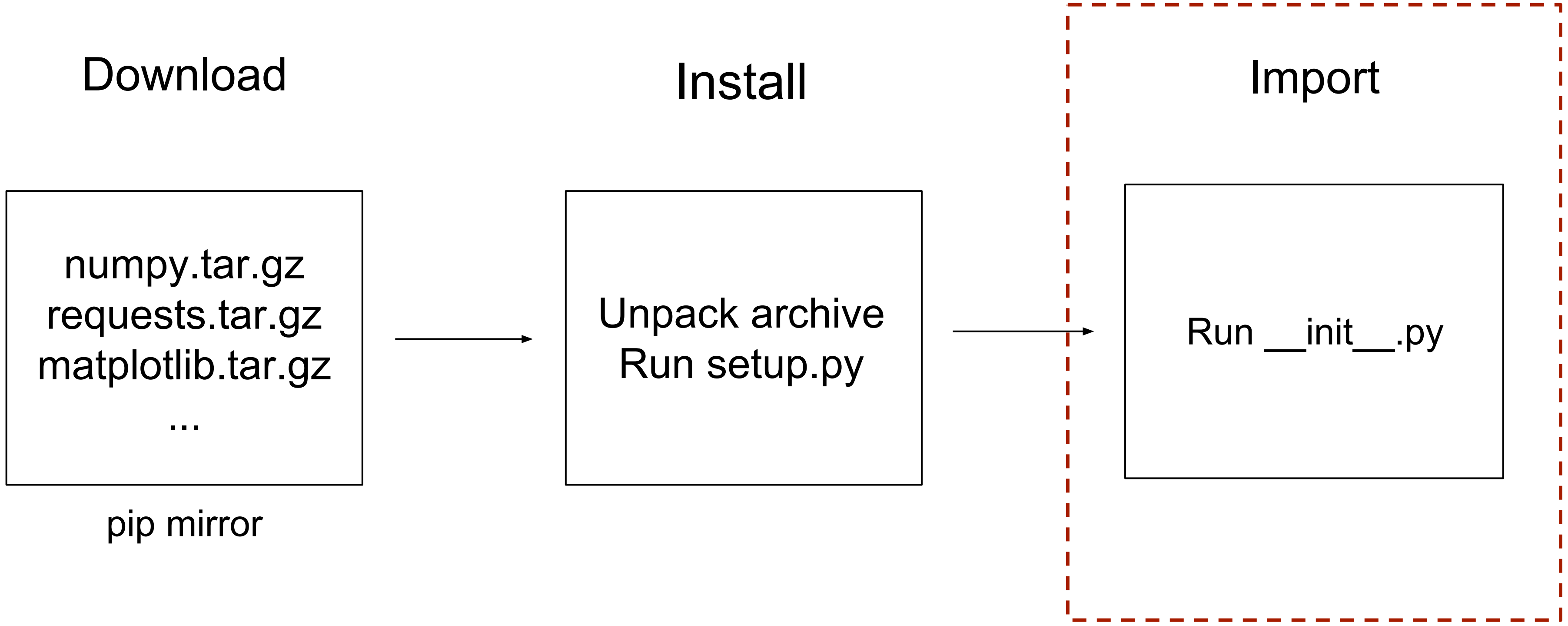
archive
main.py
other.py
ext.c
setup.py
unpack
install dir
main.py
other.py
ext.so
gcc
run
other dir
run
setup.py
write
!!!

Install



Installing pip packages must be considered **unsafe**

Installation Workflow



Import

1. Search for the named module
2. Bind the module’s metadata to the symbol table
3. Run __init__.py for the module and its dependencies

Import

1. Search for the named module
2. Bind the module's metadata to the symbol table
3. Run __init__.py for the module and its dependencies

Run arbitrary Python code, C code, etc

Import

1. Search for the named module
2. Bind the module's metadata to the symbol table
3. Run __init__.py for the module and its dependencies

Run arbitrary Python code, C code, etc

Would you trust these packages?

Import

1. Search for the named module
2. Bind the module's metadata to the symbol table
3. Run __init__.py for the module and its dependencies

Run arbitrary Python code, C code, etc

Would you trust these packages?

- "itsdangerous"
- "bugs-everywhere"
- "cocaine-tools"

Import

1. Search for the named module
2. Bind the module's metadata to the symbol table
3. Run __init__.py for the module and its dependencies

Run arbitrary Python code, C code, etc

Would you trust these packages?

- "itsdangerous"
- "bugs-everywhere"
- "cocaine-tools"

Importing pip packages must be considered **unsafe**

Outline

Motivation

Python Packages

- Anatomy
- Analysis

Pipsqueak

- Handler cache
- Import cache

Evaluation

Conclusion

Python Package Analysis

Analysis Questions

- What startup costs are associated with popular packages?
- How large are pip packages?

Python Package Analysis

Analysis Questions

- What startup costs are associated with popular packages?
- How large are pip packages?

Methodology

- Scraped 876K GitHub Python repositories and parsed import statements from all included .py files
- Setup mirror of pip repository (834K total packages)

Python Package Analysis

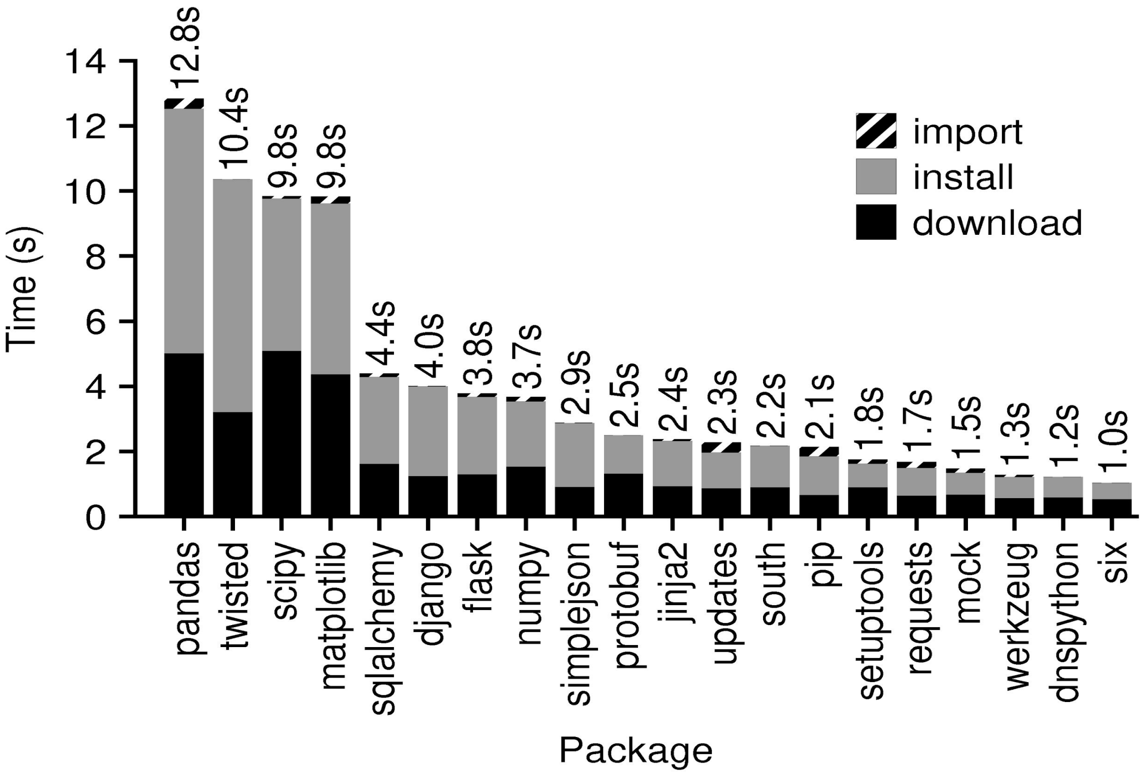
Analysis Questions

- What startup costs are associated with popular packages?
- How large are pip packages?

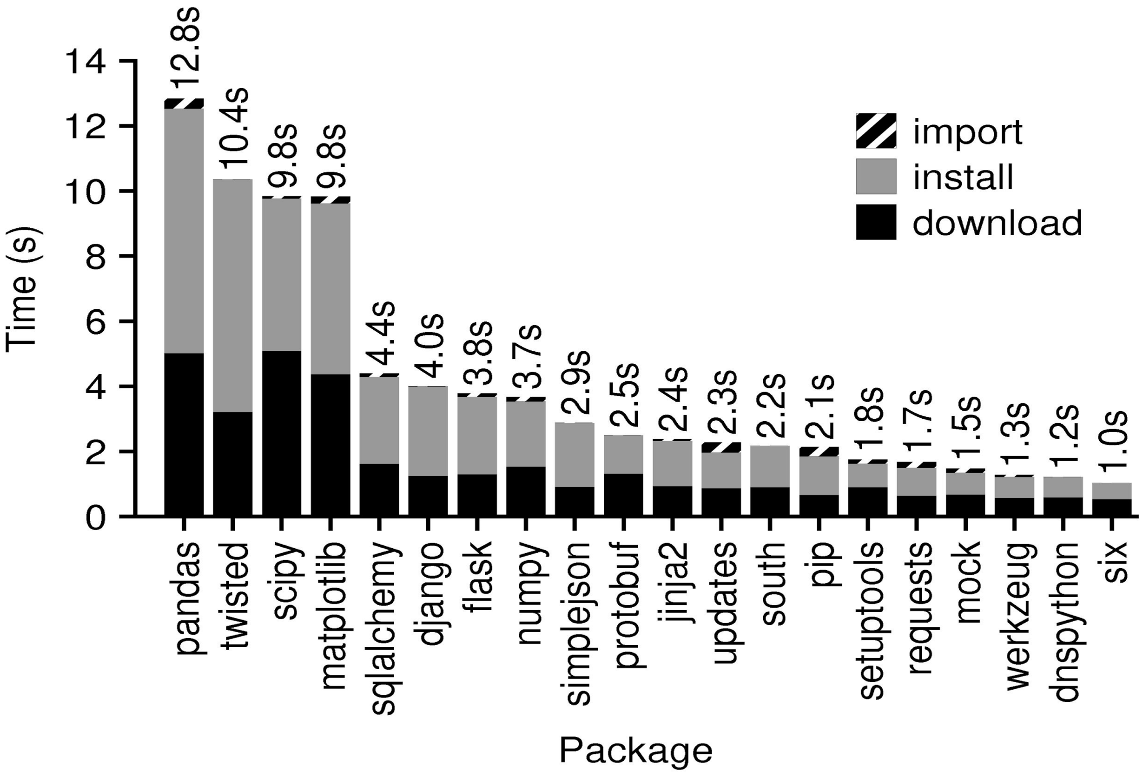
Methodology

- Scraped 876K GitHub Python repositories and parsed import statements from all included .py files
- Setup mirror of pip repository (834K total packages)

Startup Costs



Startup Costs



Average Times:

- Download: 1.6s
- Install: 2.3s
- Import: 107ms

Python Package Analysis

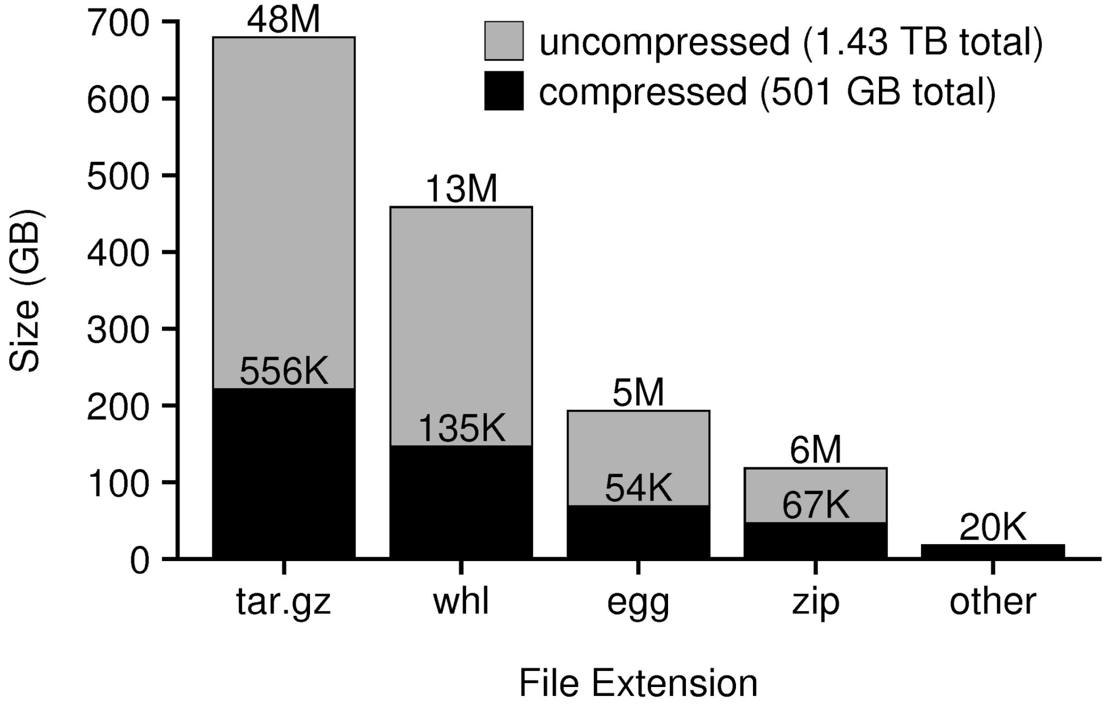
Analysis Questions

- What costs are associated with popular packages?
- How large are pip packages?

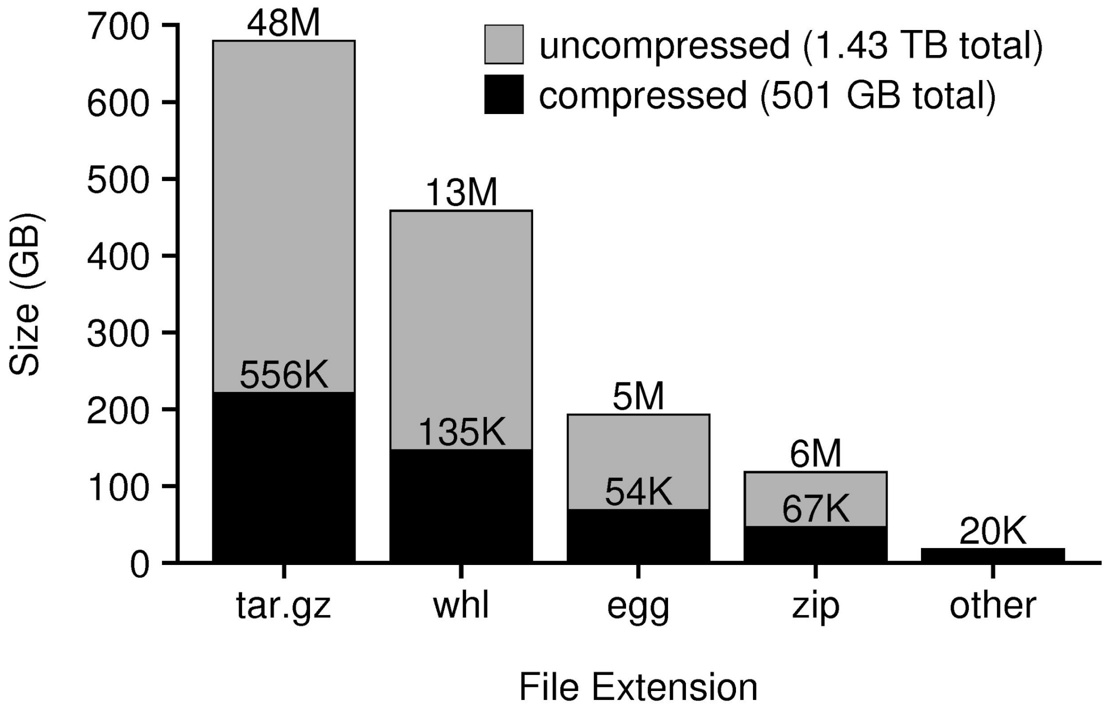
Methodology

- Scraped 876k GitHub Python repositories and parsed import statements from all included .py files
- Setup mirror of pip repository (834k total packages)

Pip Repository



Pip Repository



Average Sizes:

- Uncompressed: 1.8 MB
- Compressed: 630 KB

Outline

Motivation

Python Packages

- Anatomy
- Analysis

Pipsqueak

- Handler cache
- Import cache

Evaluation

Conclusion

Pipsqueak

Package sharing serverless compute platform

- Extension of OpenLambda
- Pre-initialize download, install, and import steps

Cache pre-initialized packages/interpreters across 3 tiers:

- **Unshared memory**: paused handler containers
- **Shared memory**: interpreter prototypes with pre-imported packages
- **Shared SSD**: pre-installed packages

Three Levels of Caching

Small & Fast

Handler Cache

- Reuse initialized containers *within* a customer

Import Cache

- Reuse initialized interpreters *between* customers

Install Cache

- Reuse installed packages *between* customers

Large & Slow

Three Levels of Caching

Small & Fast

Handler Cache

- Reuse initialized containers *within* a customer

Import Cache

- Reuse initialized interpreters *between* customers

Install Cache

- Reuse installed packages *between* customers

Pipsqueak Contribution

Large & Slow

Three Levels of Caching

Small & Fast

Handler Cache

- Reuse initialized containers *within* a customer

Import Cache

- Reuse initialized interpreters *between* customers

Covered Today

Install Cache

- Reuse installed packages *between* customers

Large & Slow

Outline

Motivation

Python Packages

- Anatomy
- Analysis

Pipsqueak

- Handler cache
- Import cache

Evaluation

Conclusion

Handler Cache

- Each customer’s handlers need to be sandboxed in a container, but we can reuse containers for multiple requests
  - Keep recently used containers in a “paused” state
  - Inspired by AWS Lambda mechanism
- Simple LRU policy
  - Evict on memory pressure

Outline

Motivation

Python Packages

- Anatomy
- Analysis

Pipsqueak

- Handler cache
- Import cache

Evaluation

Conclusion

Import Cache

- Maintain a set of Python interpreters with packages pre-imported in a sleeping state

Import Cache

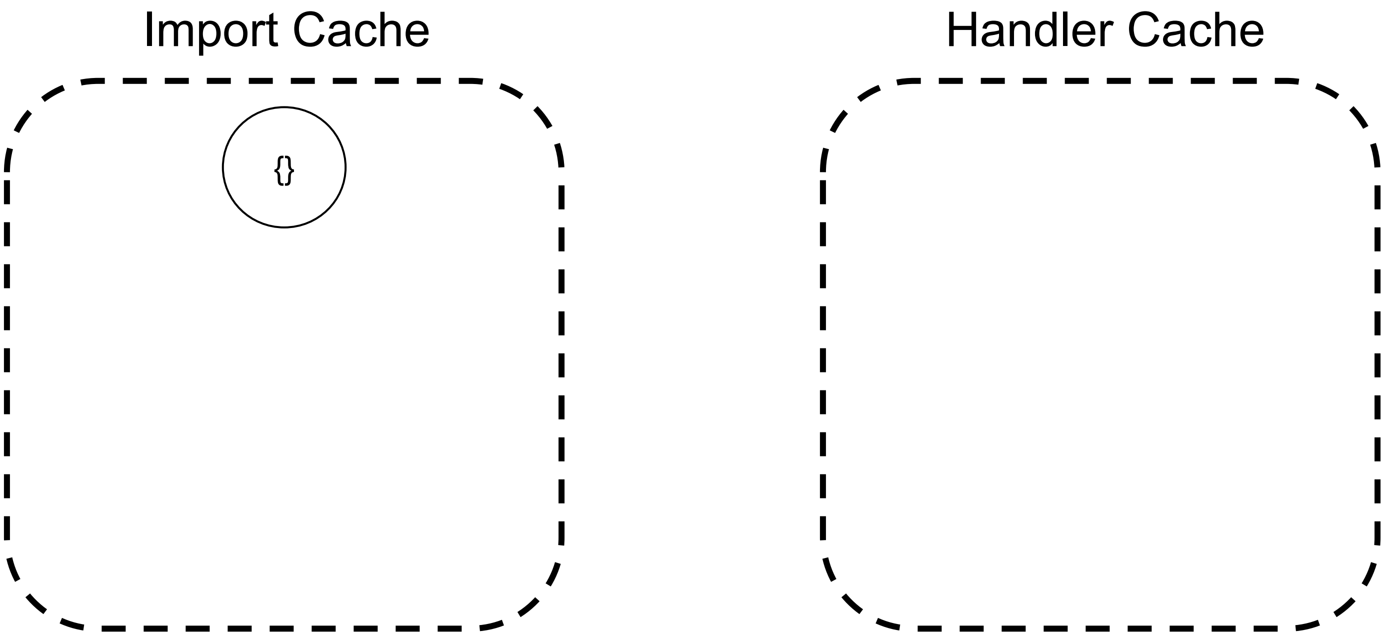
- Maintain a set of Python interpreters with packages pre-imported in a sleeping state
- Using a cache entry:
  a. Wake up & fork a sleeping Python interpreter
  b. Relocate child process into handler container
  c. Handle requests

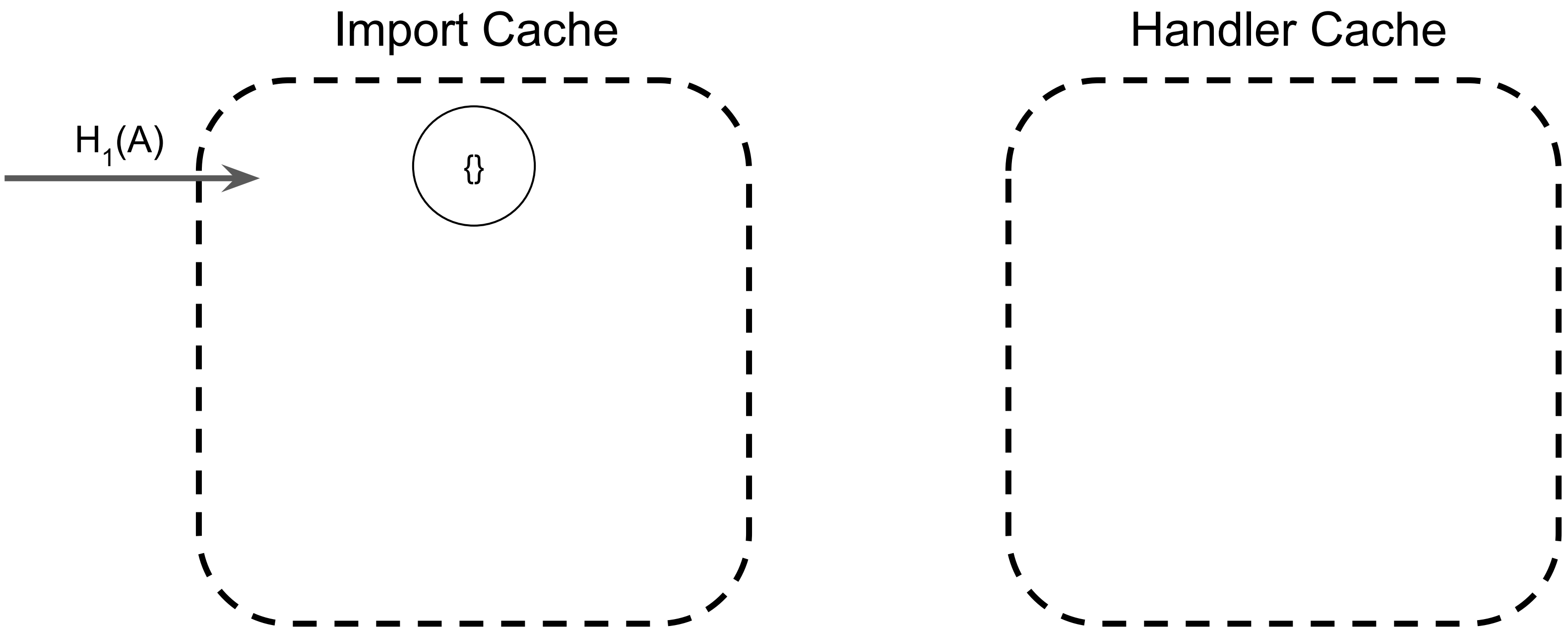
Import Cache

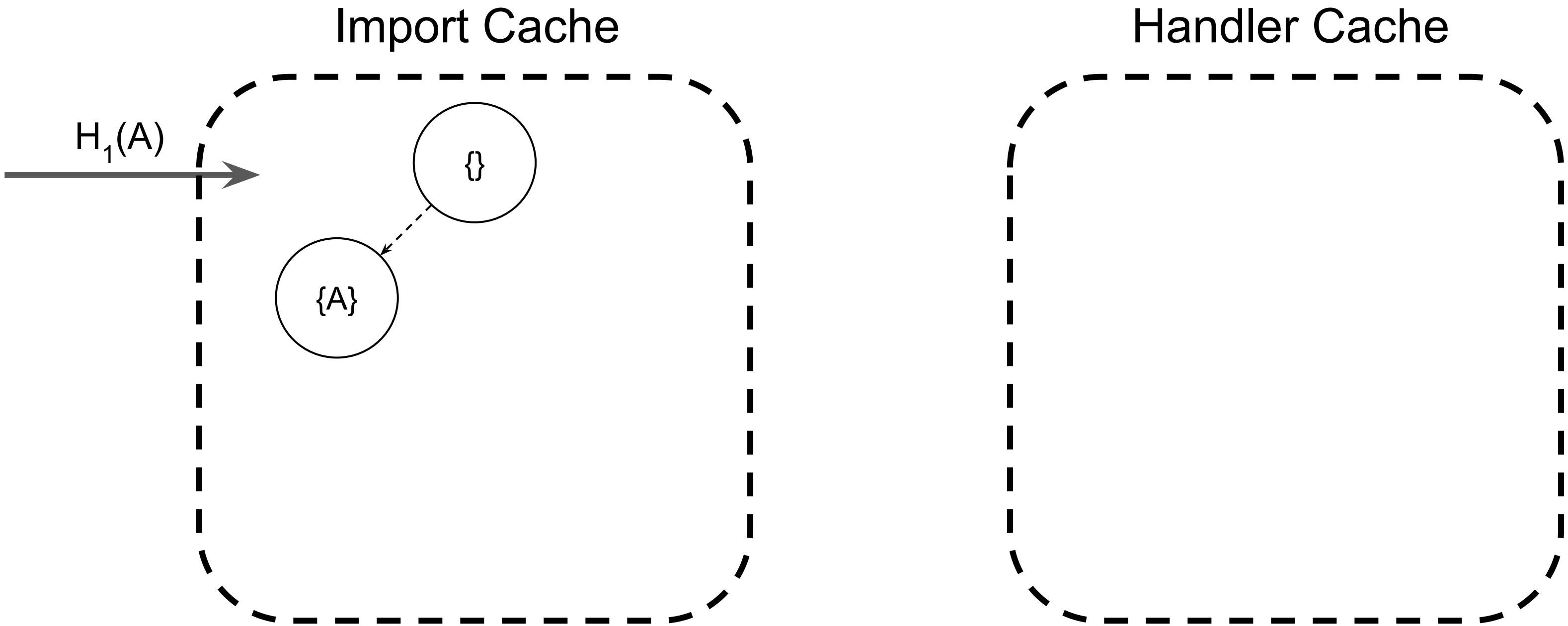
- Maintain a set of Python interpreters with packages pre-imported in a sleeping state

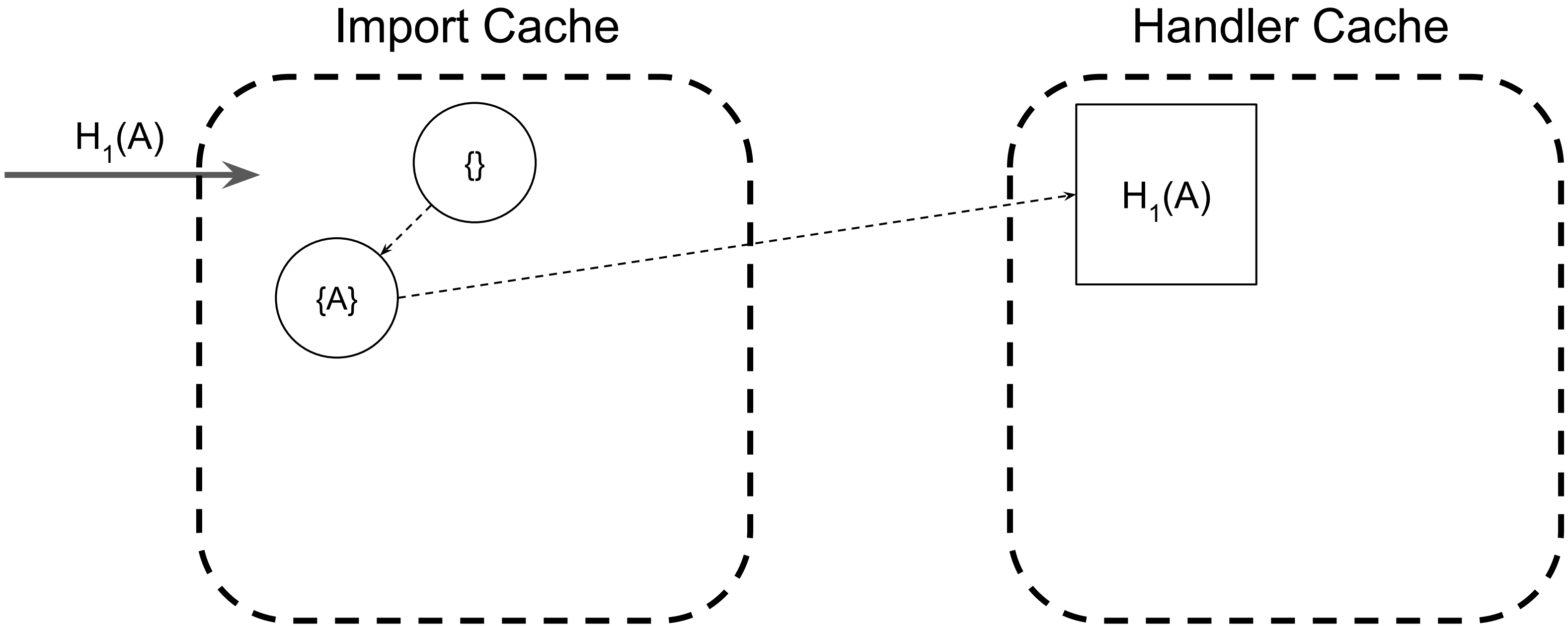
- Using a cache entry:
    a. Wake up & fork a sleeping Python interpreter
    b. Relocate child process into handler container
    c. Handle requests

- Creating a cache entry:
    a. Wake up & fork a sleeping Python interpreter
    b. Relocate child process into cache container
    c. Import Python packages & sleep

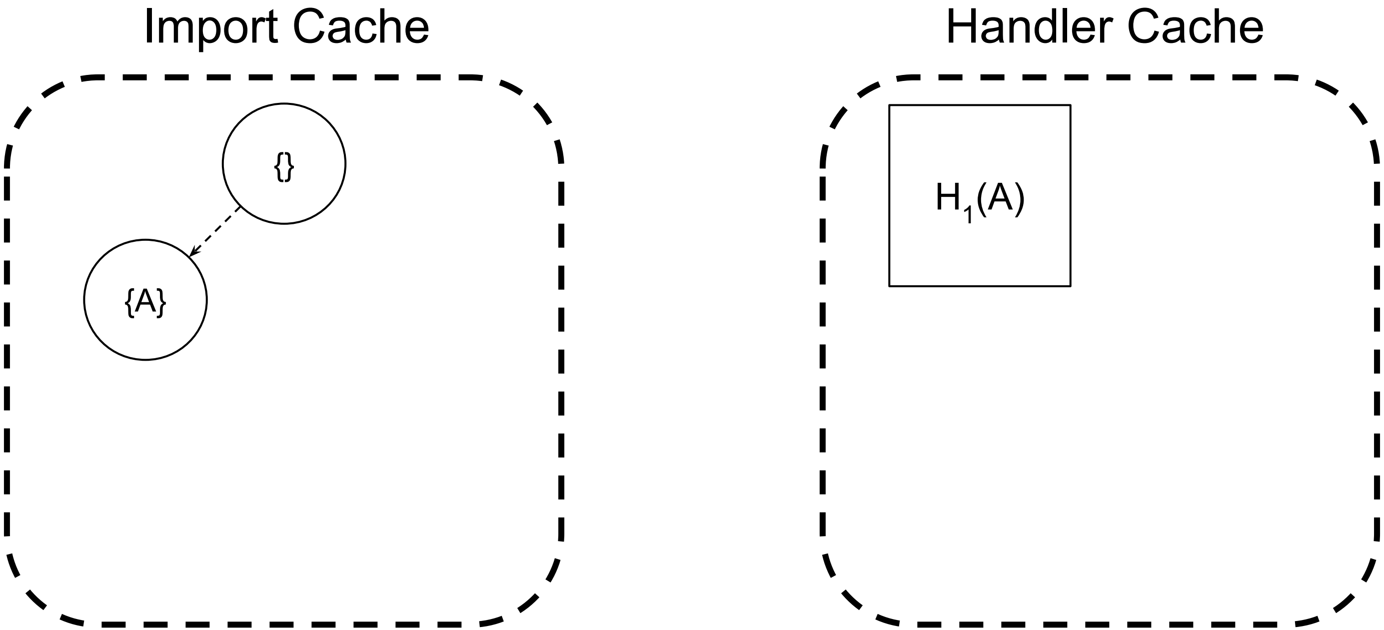

Import Cache
{}
Handler Cache

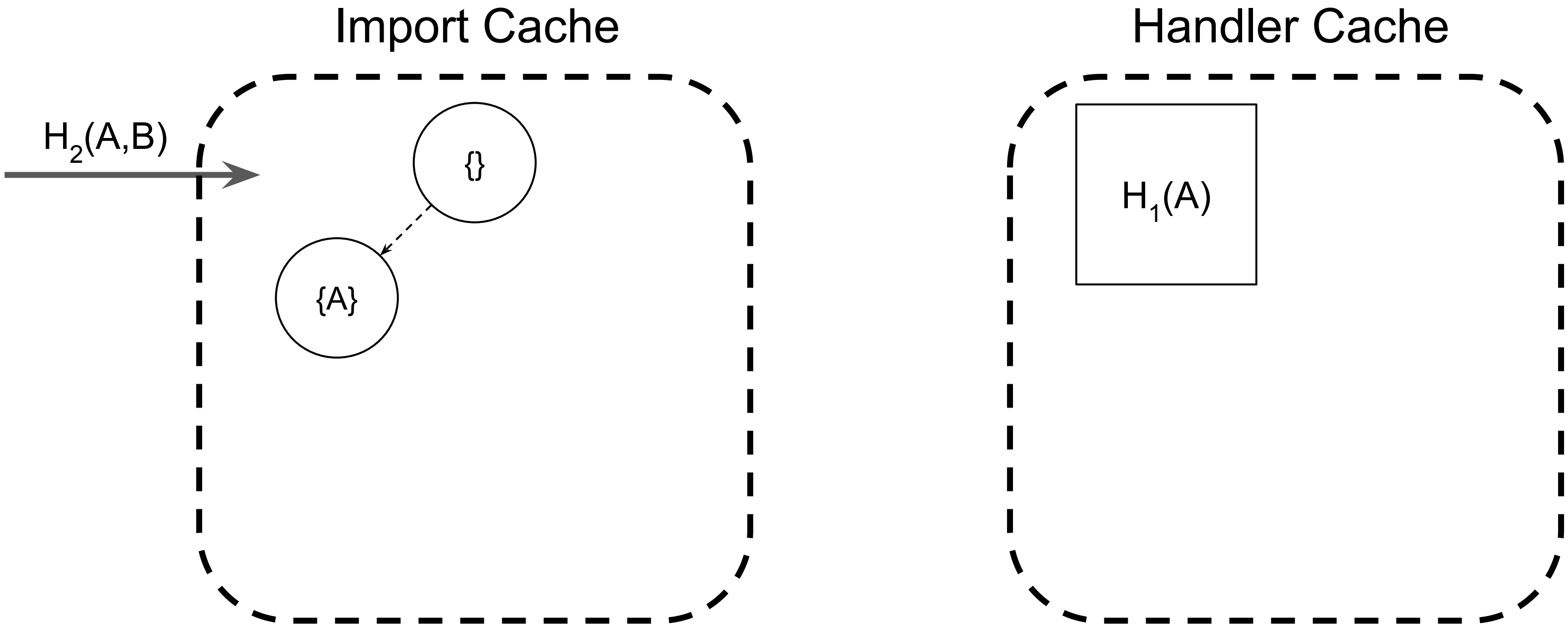

Import Cache
Handler Cache
H1(A)
{}


Import Cache
Handler Cache
H1(A)
{}
{A}

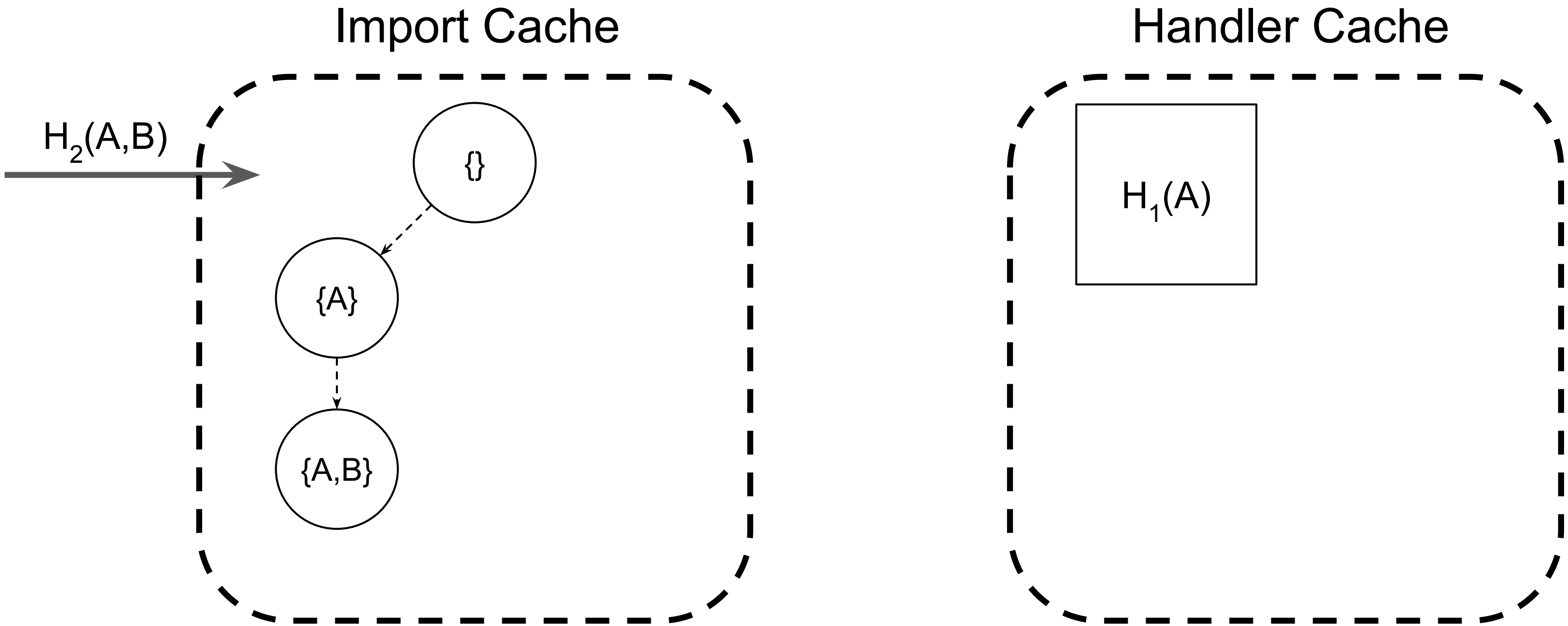


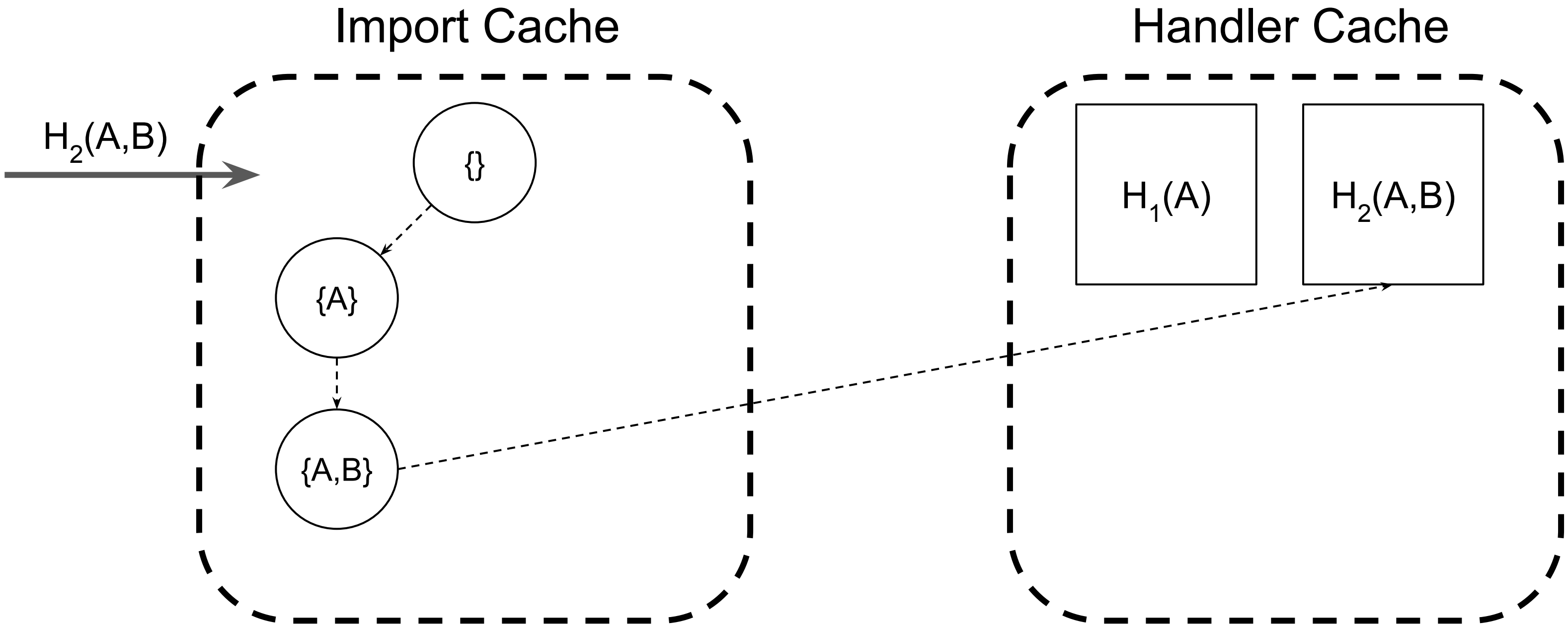
Import Cache
Handler Cache
$H_1(A)$
{}
{A}
$H_1(A)$


Import Cache
{}
{A}
Handler Cache
$H_1(A)$

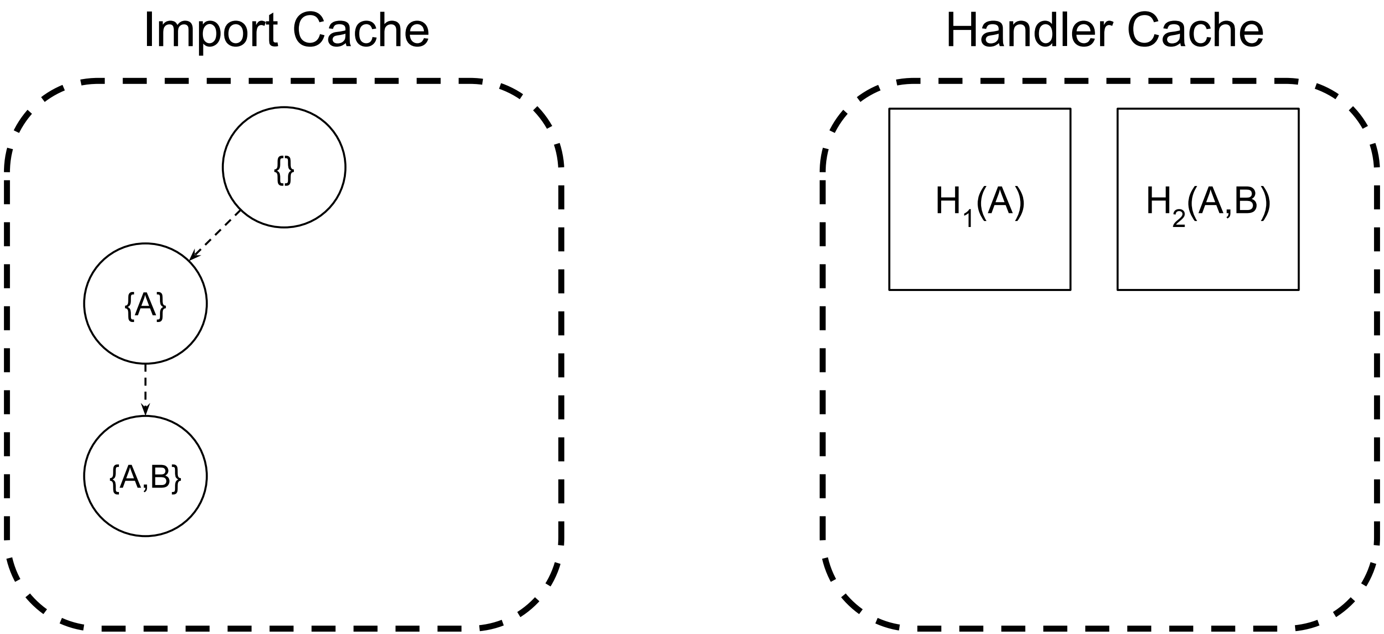


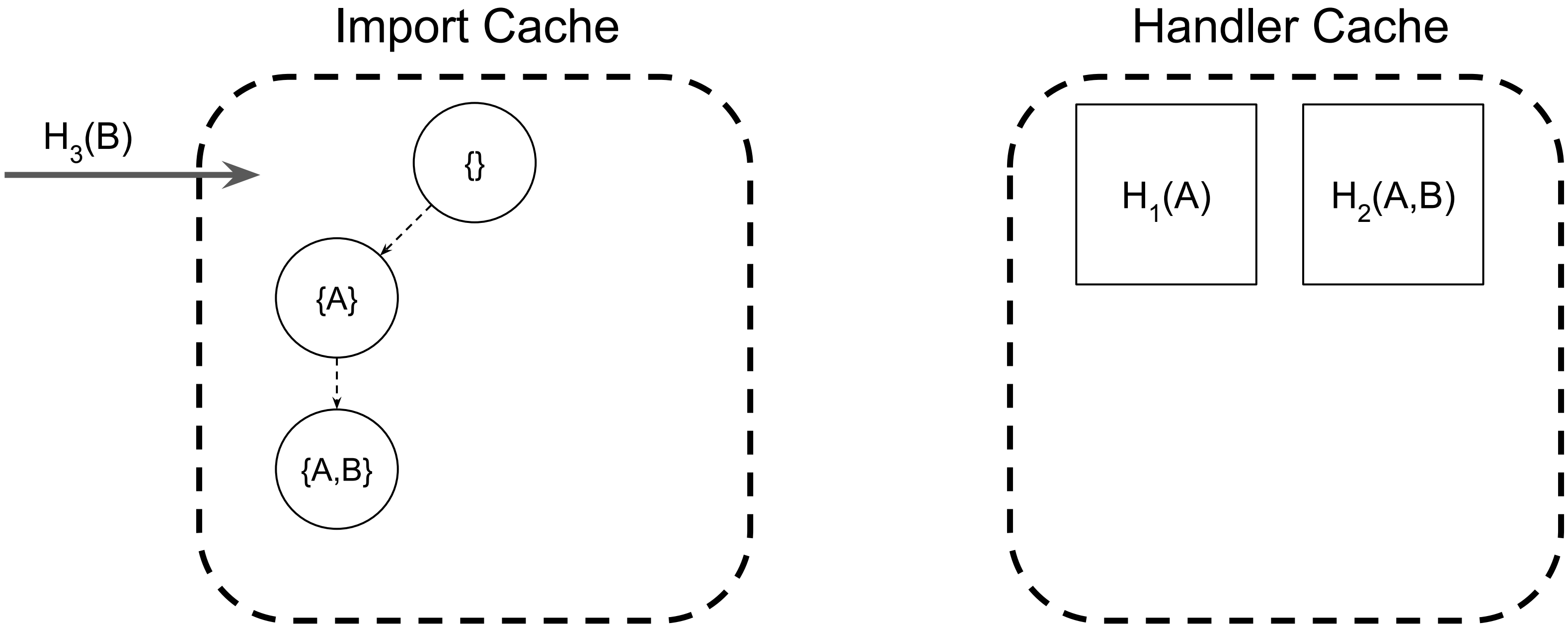
Import Cache
Handler Cache
$H_2(A,B)$
{}
{A}
$H_1(A)$

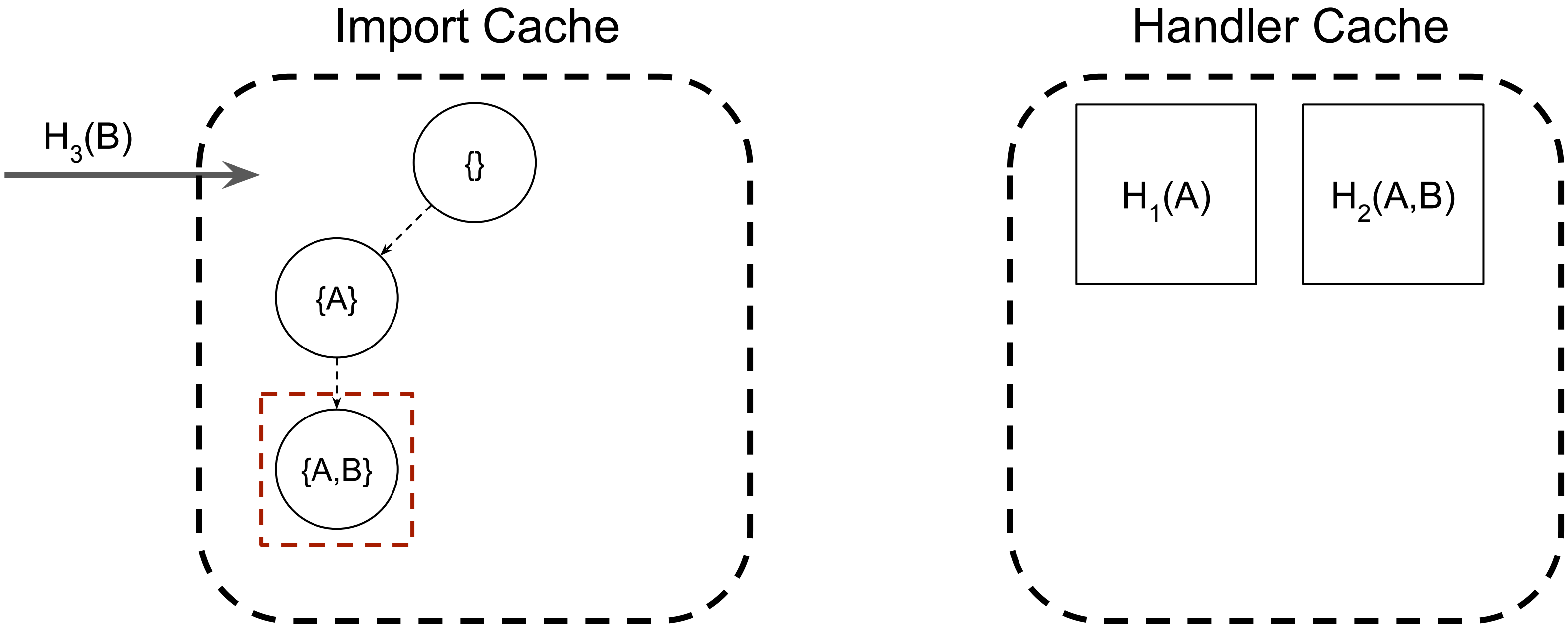

Import Cache
Handler Cache
H2(A,B)
{}
{A}
{A,B}
H1(A)

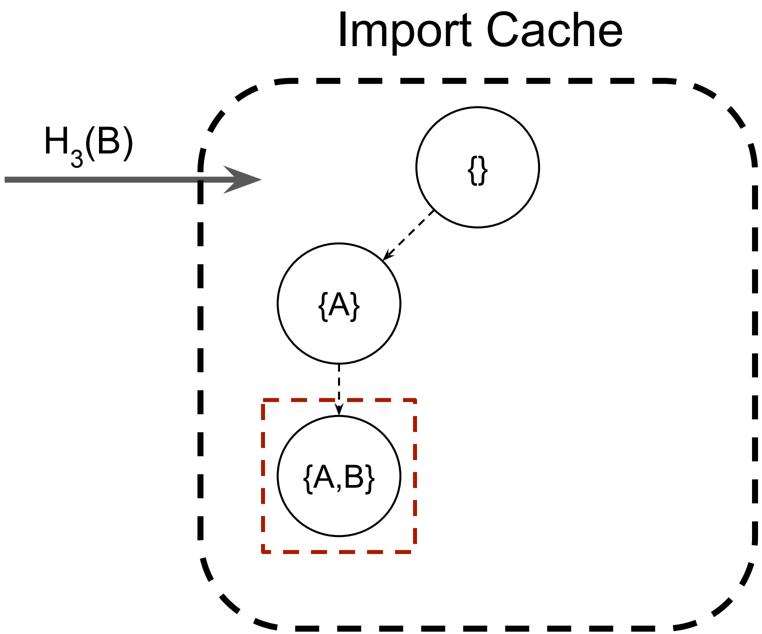


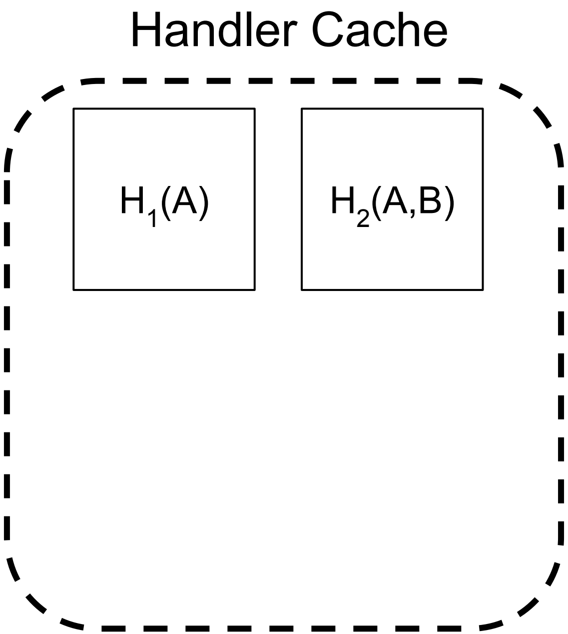
Import Cache
Handler Cache
H2(A,B)
{}
{A}
{A,B}
H1(A)
H2(A,B)


Import Cache
{}
{A}
{A,B}
Handler Cache
H1(A)
H2(A,B)

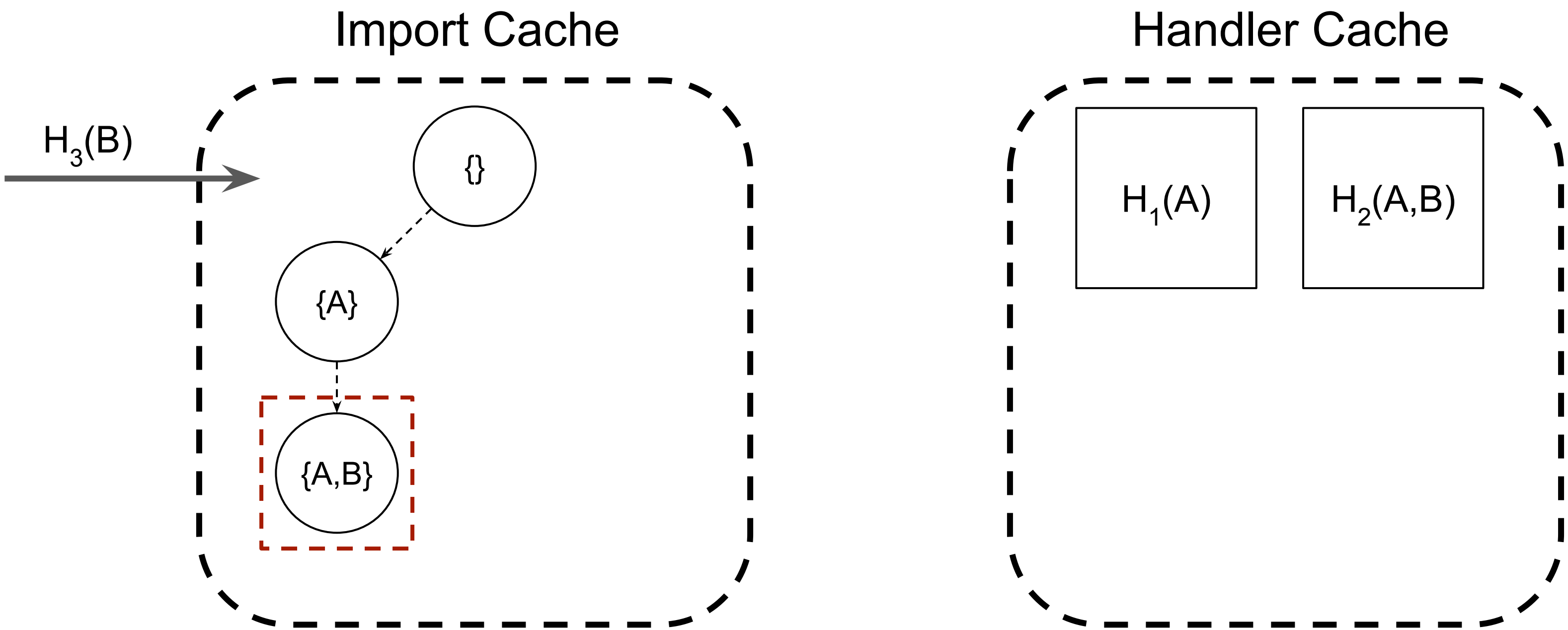

Import Cache
Handler Cache
H3(B)
{}
{A}
{A,B}
H1(A)
H2(A,B)


Import Cache
Handler Cache
$H_3(B)$
{}
{A}
{A,B}
$H_1(A)$
$H_2(A,B)$





What if package ‘A’ is malicious?



What if package ‘A’ is malicious?

- “Subset only” rule

Outline

Motivation

Python Packages

- Anatomy
- Analysis

Pipsqueak

- Handler cache
- Import cache

Evaluation

Conclusion

Evaluation Questions

1. How much does package sharing improve latency?
2. How do the caching layers interact?

Microbenchmark

Not a stress test, want to examine differences in caching

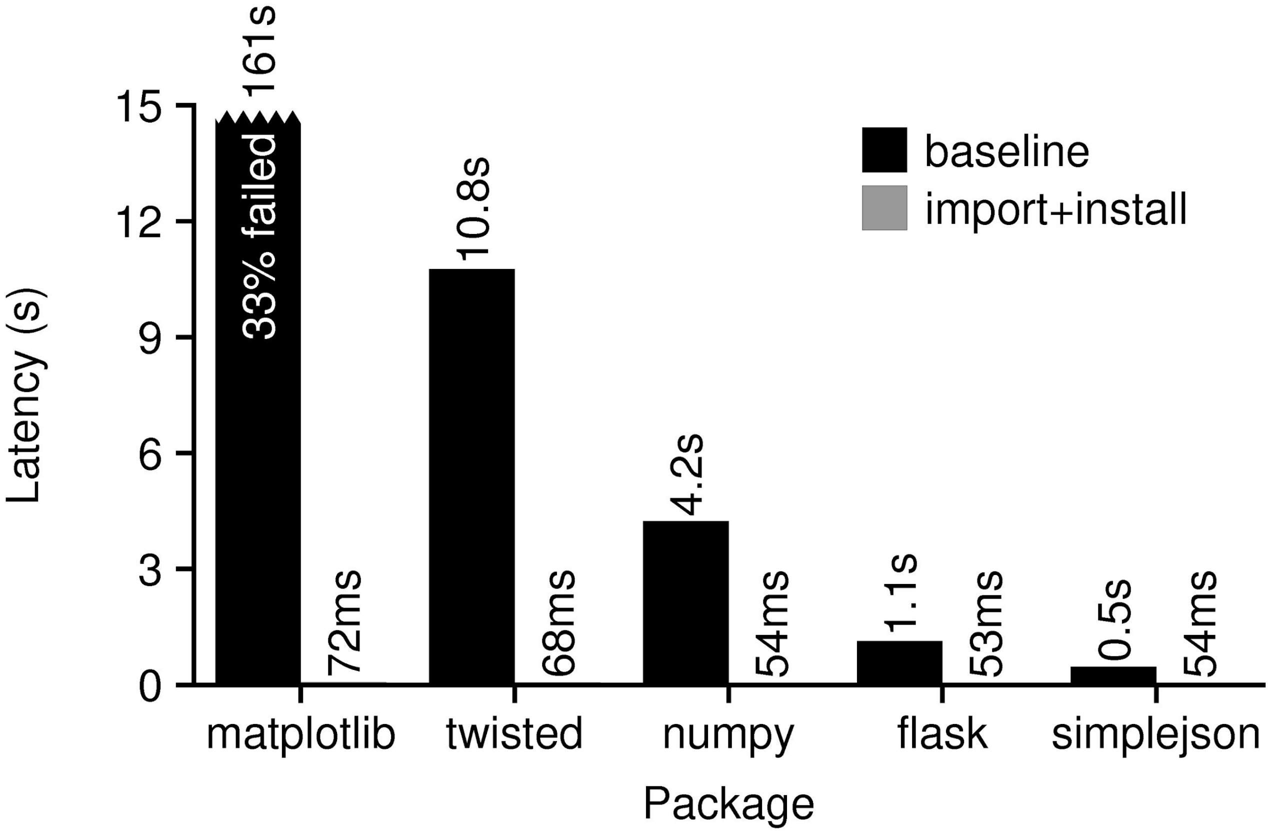
Experimental Setup:

- 1 OpenLambda worker machine
- 2 random requests per second
- 100 distinct handlers, all importing the same pip package

Evaluation Questions

1. How much does package sharing improve latency?
2. How do the caching layers interact?

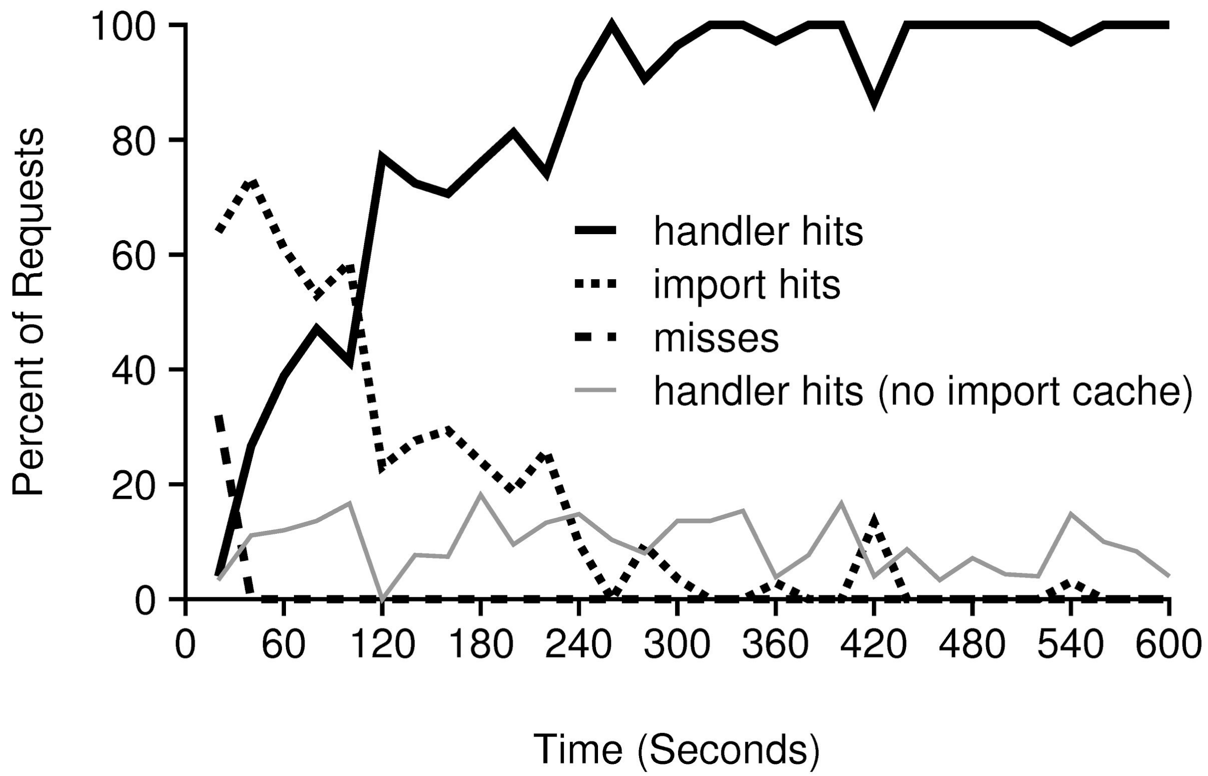
Microbenchmark



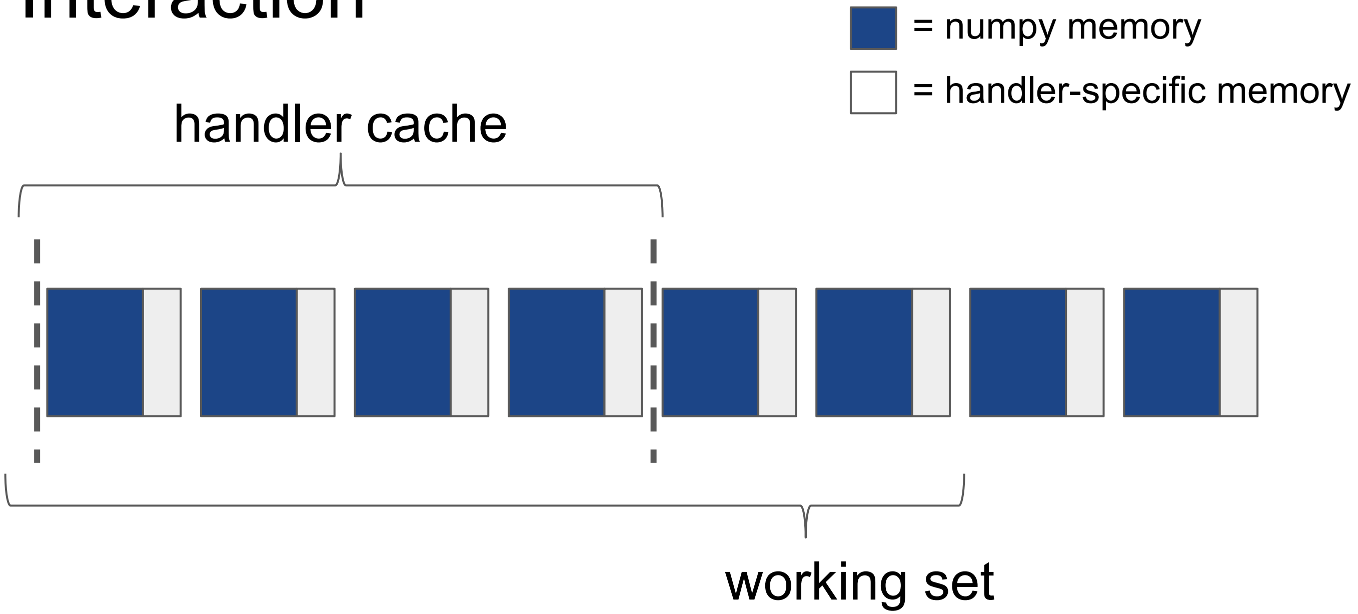
Evaluation Questions

1. How much does package sharing improve latency?
2. How do the caching layers interact?

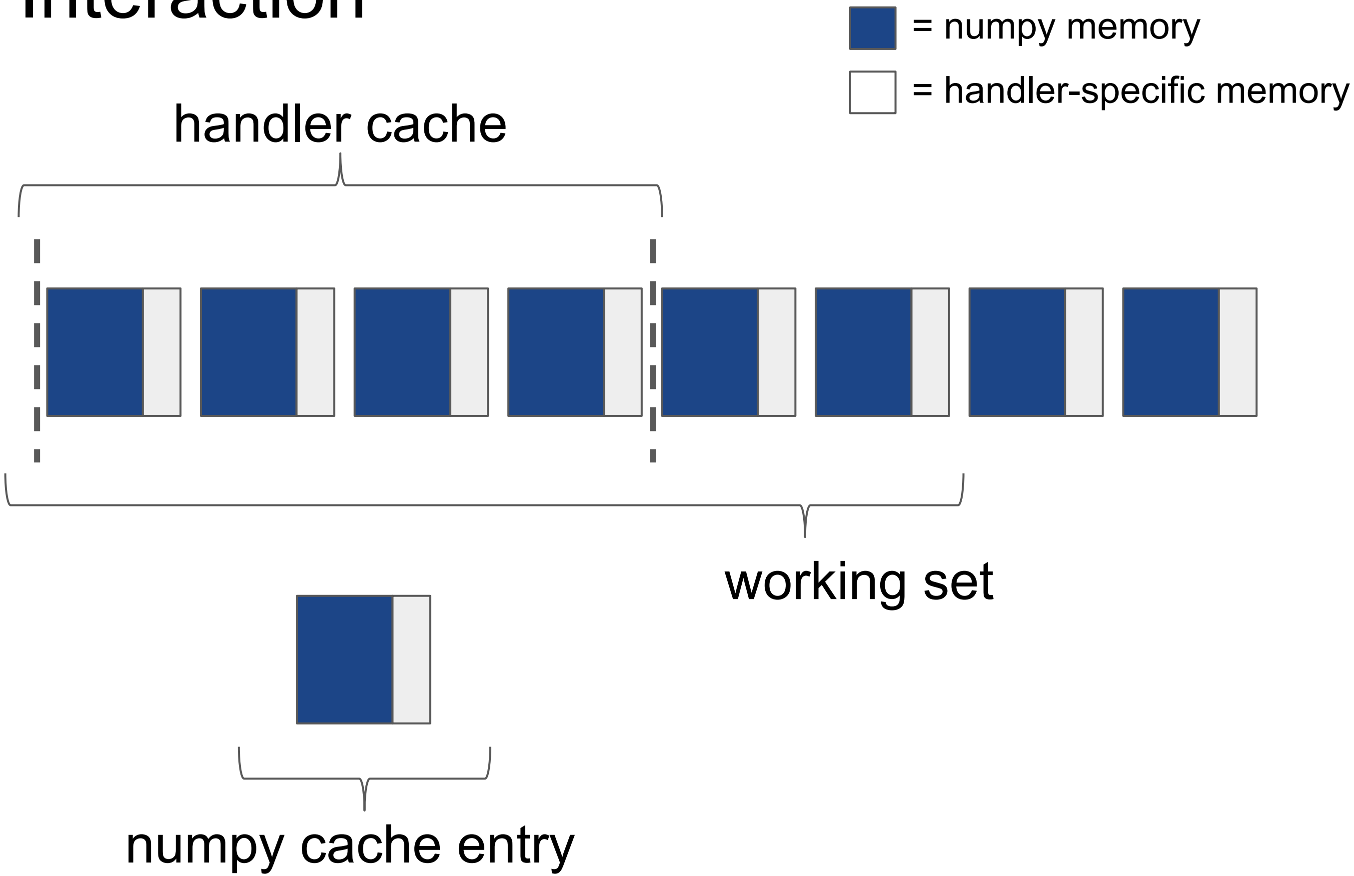
Cache Interaction



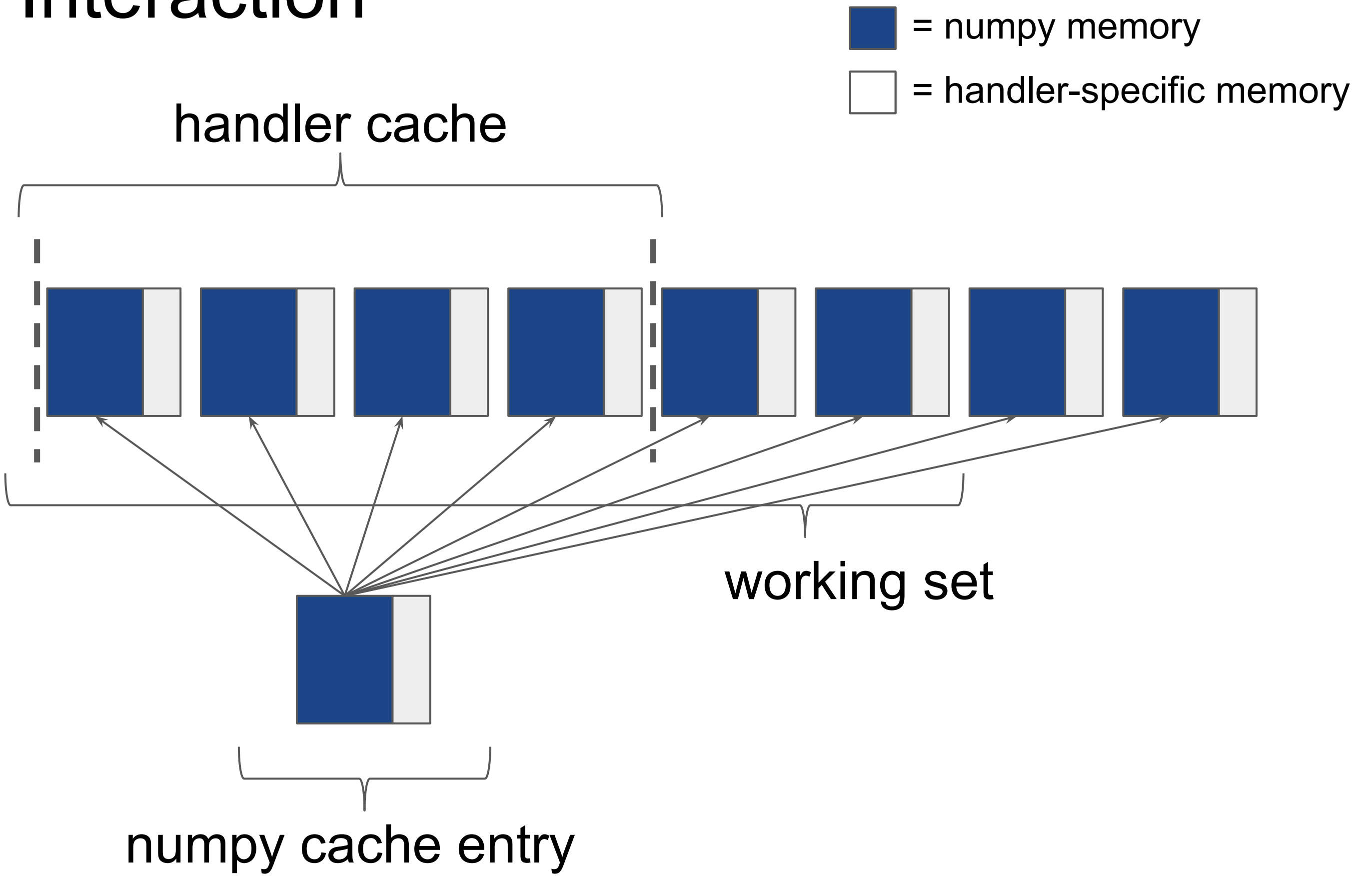
Cache Interaction



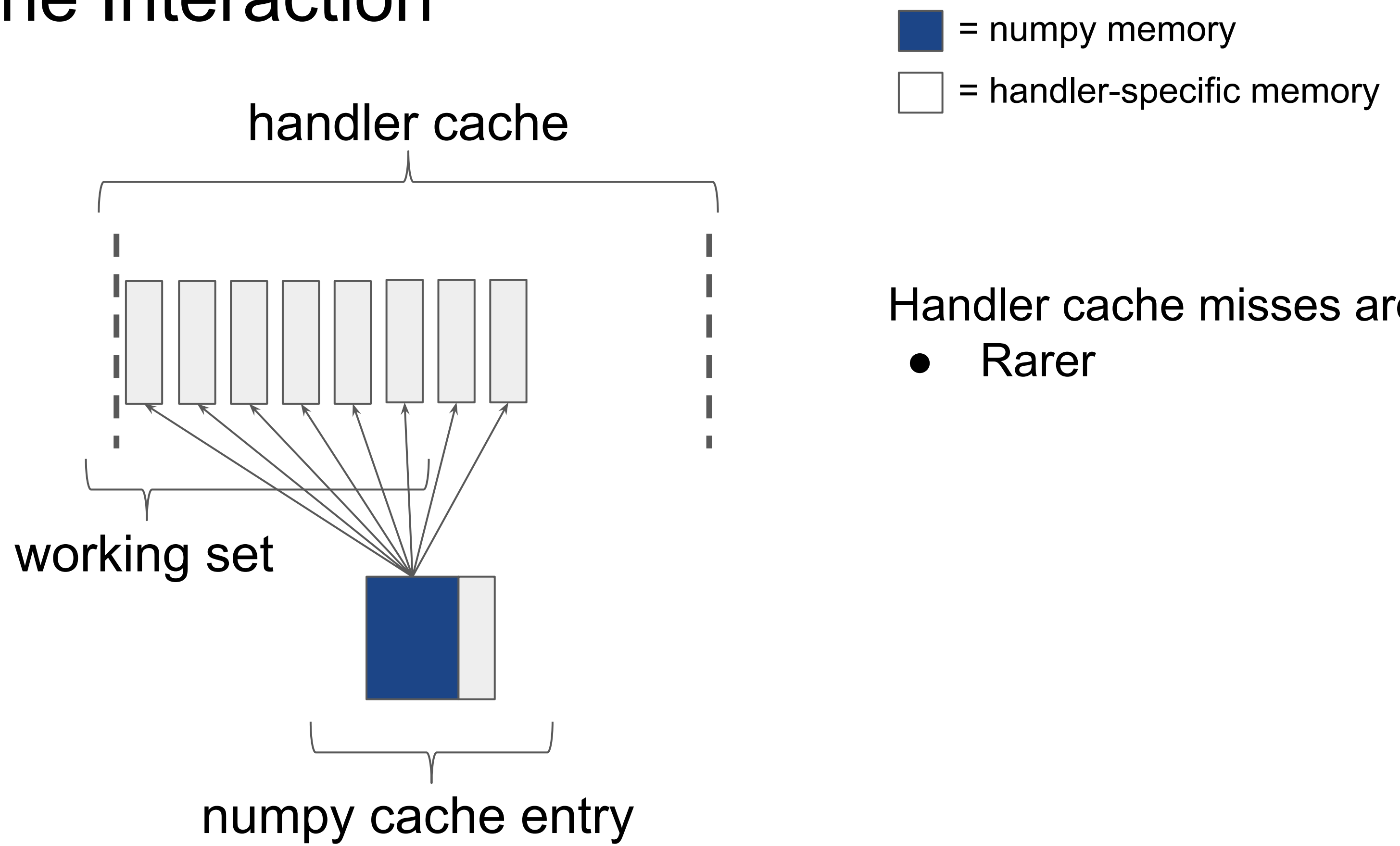
Cache Interaction



Cache Interaction



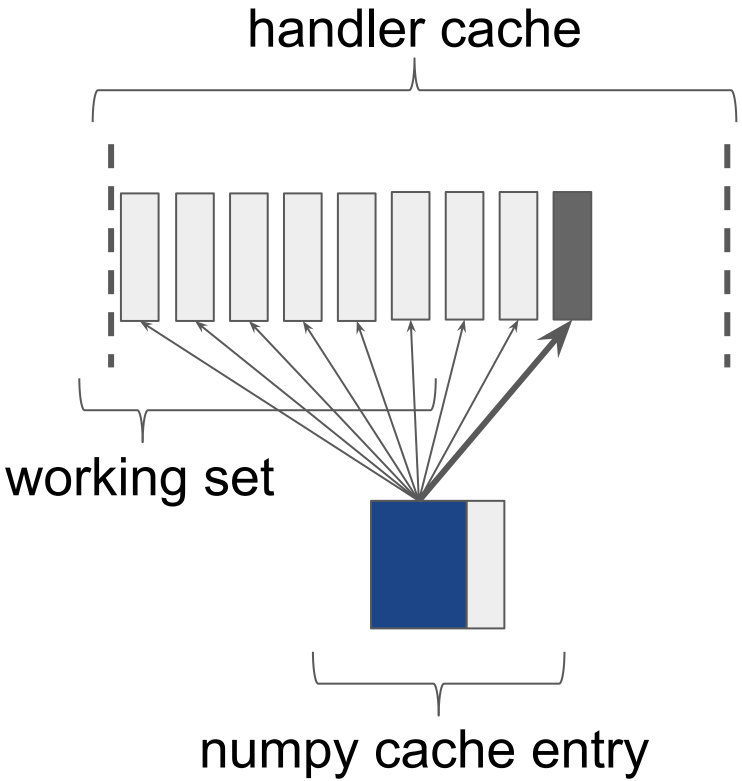
Cache Interaction

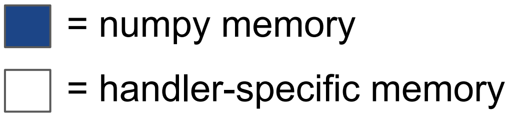


Handler cache misses are:

- Rarer

Cache Interaction





Handler cache misses are:

- Rarer
- Faster

Outline

Motivation

Python Packages

- Anatomy
- Analysis

Pipsqueak

- Handler cache
- Import cache

Evaluation

Conclusion

Conclusion

Problem:

- Lambda handlers are supposed to be small, but developers' reliance on user-space libraries inflates them

Our Solution:

- Share pre-initialized packages among handlers in multi-level cache

Results:

- 9-2000x speedups for single-package workloads

Questions?