Towards Just-In-Time Compiling of Operating Systems

September 28, 2023

Maximilian Ott, Phillip Raffeck, Volkmar Sieh and Wolfgang Schröder-Preikschat

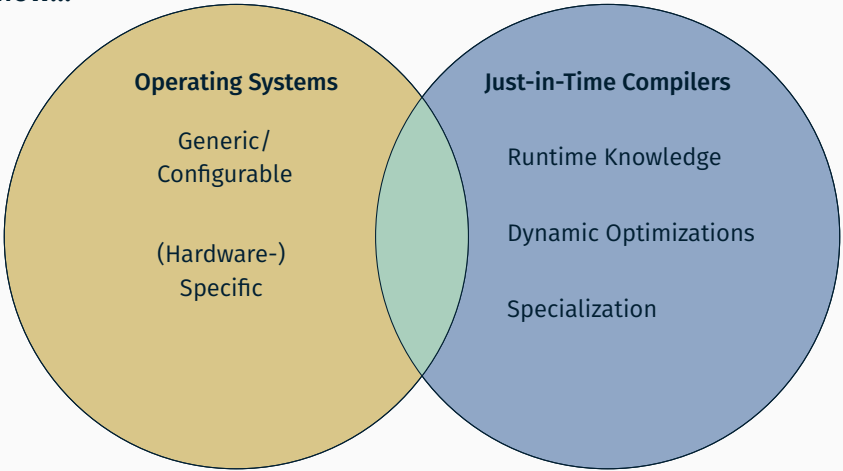
Friedrich-Alexander-Universität Erlangen-Nürnberg

Friedrich-Alexander-Universität
Faculty of Engineering

Chair in Distributed Systems
and Operating Systems

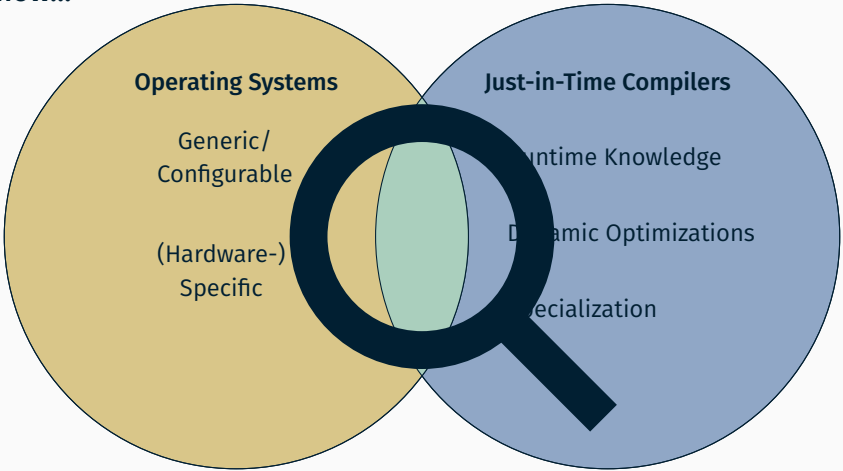
JIT + OS = ?

We all know...



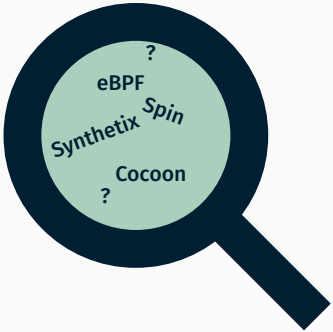
JIT + OS = ?

We all know...



JIT + OS = ?

We all know...

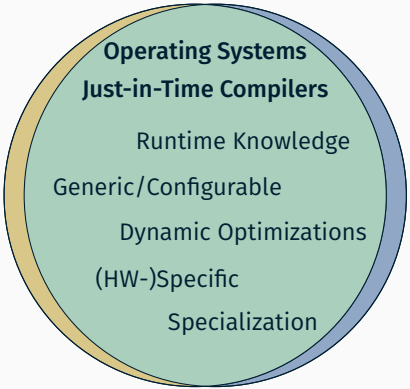


Dynamic specialization so far:

- **SPIN**: Change OS functionality via dynamic linking from applications
- **Synthesis**/**Synthetix**: Use code-synthesis techniques to specialize system functions
- **Cocoon**: Apply target-dependent optimizations at boot time

JIT + OS = ?

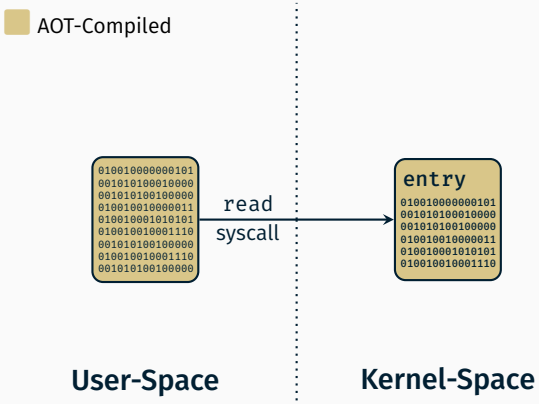
Our goals: Specialize **whole OS** at runtime based on **whole system state**:



- Utilize full potential of underlying system
- Re-compile for configuration transitions
  - → e.g. Adaptive Locking Mechanism
  - → e.g. Adaptive Scheduling Strategies

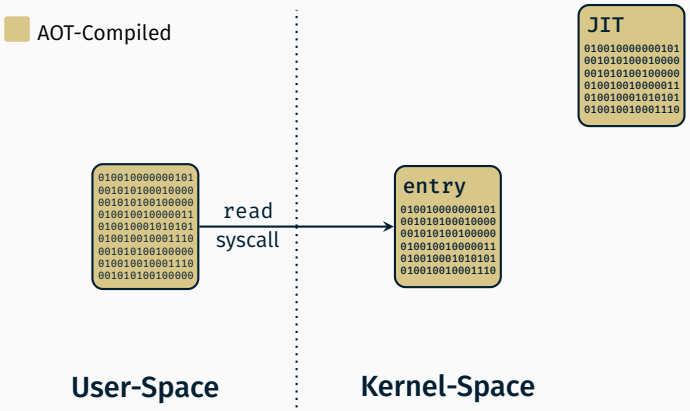
Why not simply...?

Plenty of JITs for User-Space Apps! Why not for the Kernel?



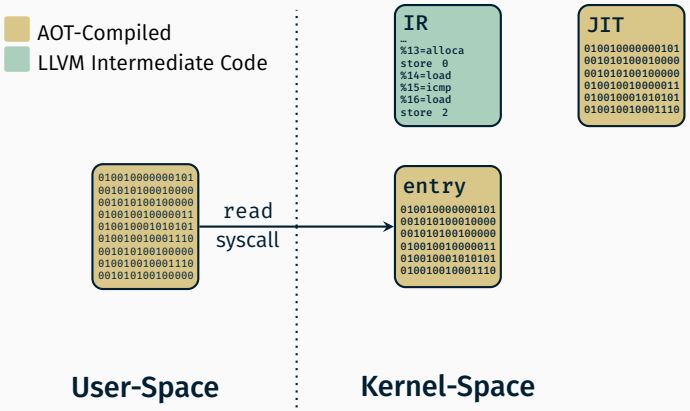
Why not simply...?

Plenty of JITs for User-Space Apps! Why not for the Kernel?

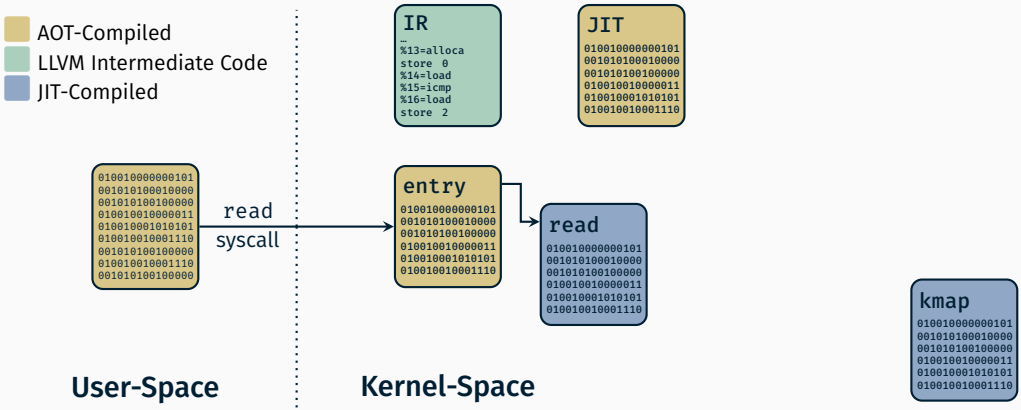


Why not simply...?

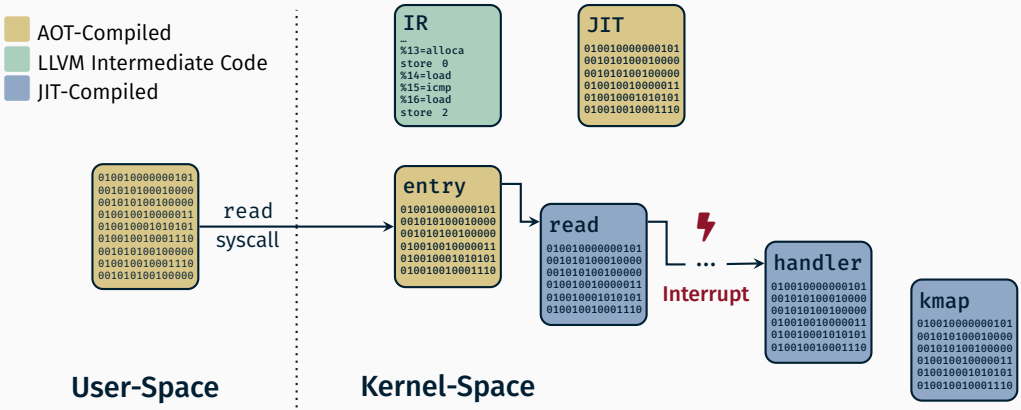
Plenty of JITs for User-Space Apps! Why not for the Kernel?



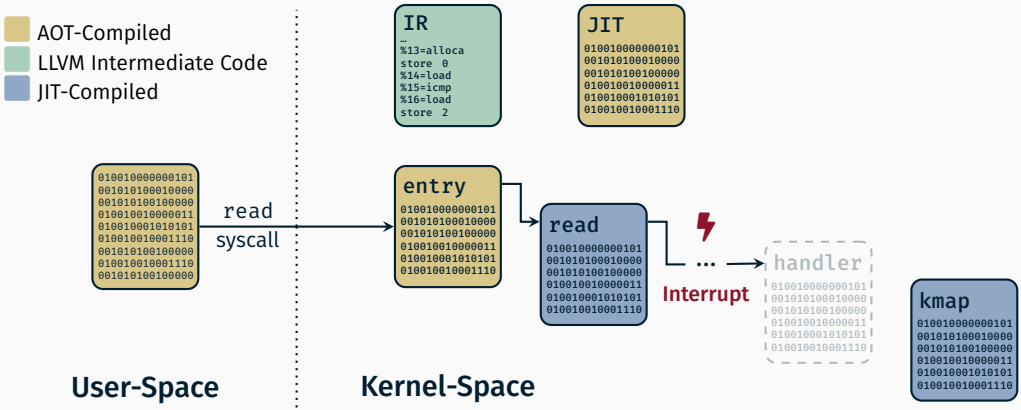
Plenty of JITs for User-Space Apps! Why not for the Kernel?



Plenty of JITs for User-Space Apps! Why not for the Kernel?

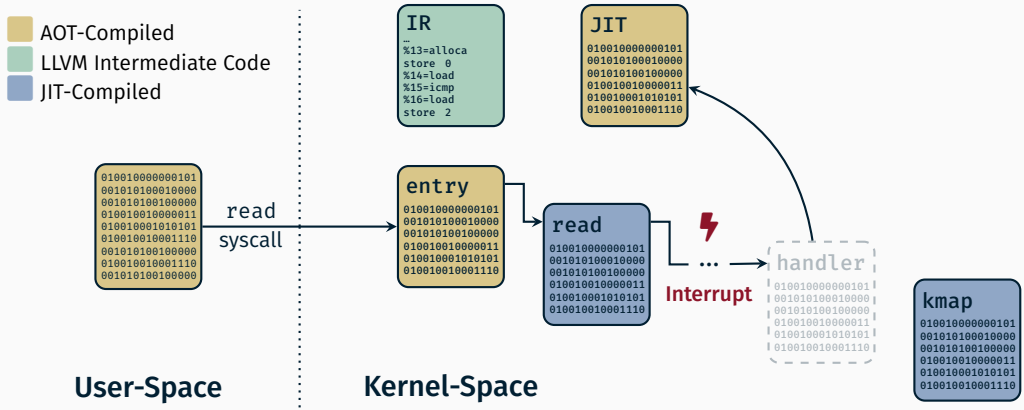


Plenty of JITs for User-Space Apps! Why not for the Kernel?



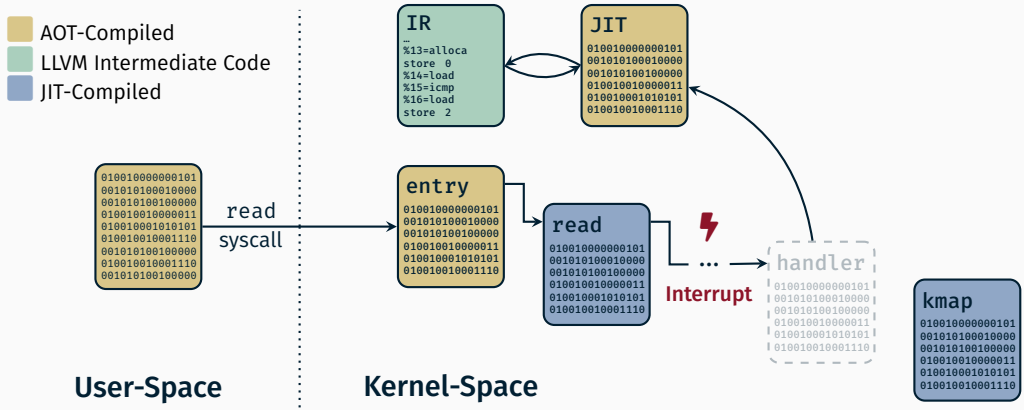
Why not simply...?

Plenty of JITs for User-Space Apps! Why not for the Kernel?

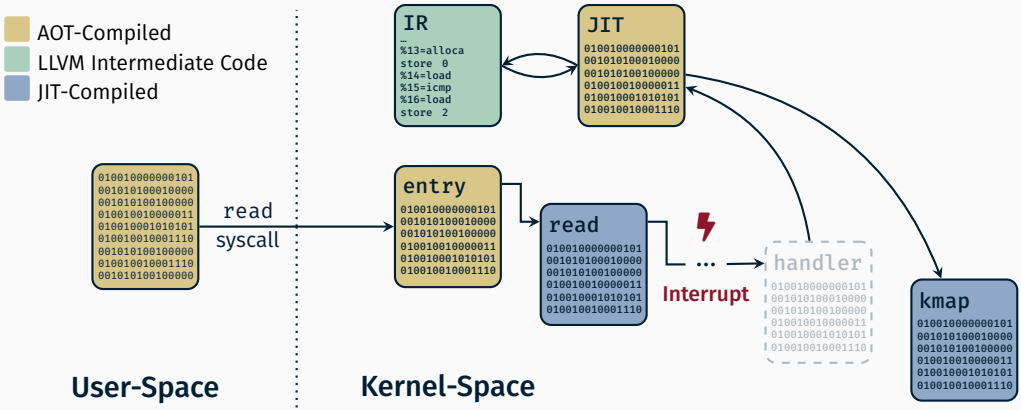


Why not simply...?

Plenty of JITs for User-Space Apps! Why not for the Kernel?

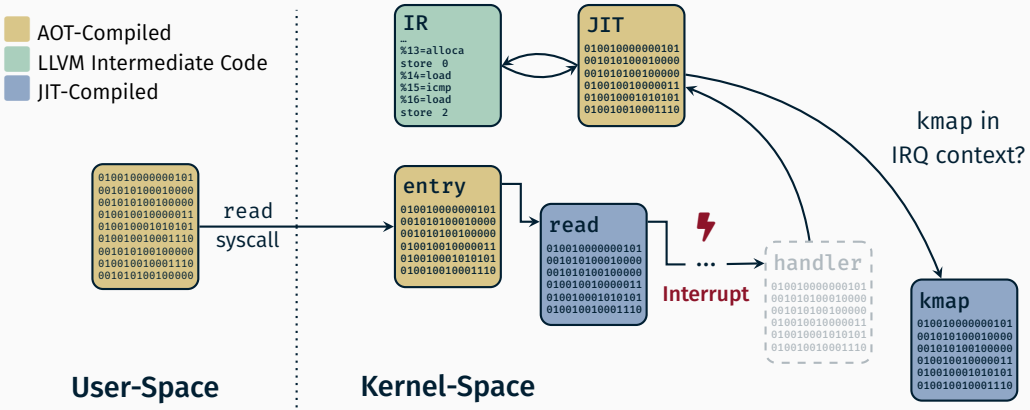


Plenty of JITs for User-Space Apps! Why not for the Kernel?

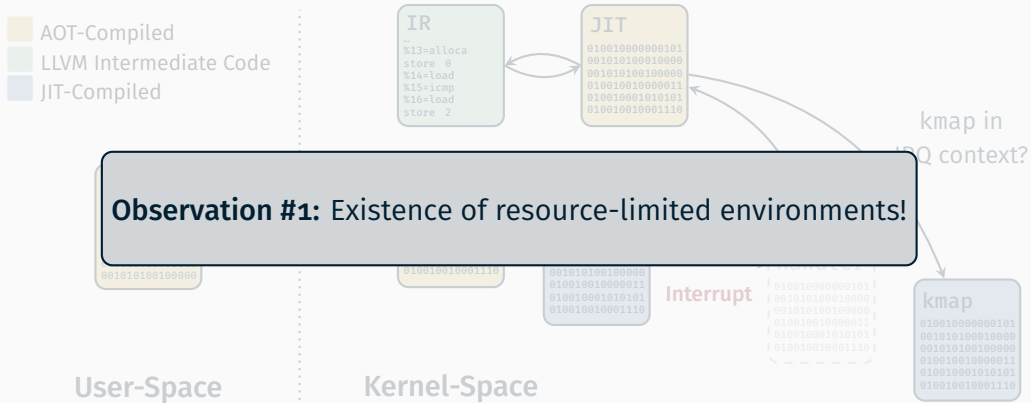


Why not simply...?

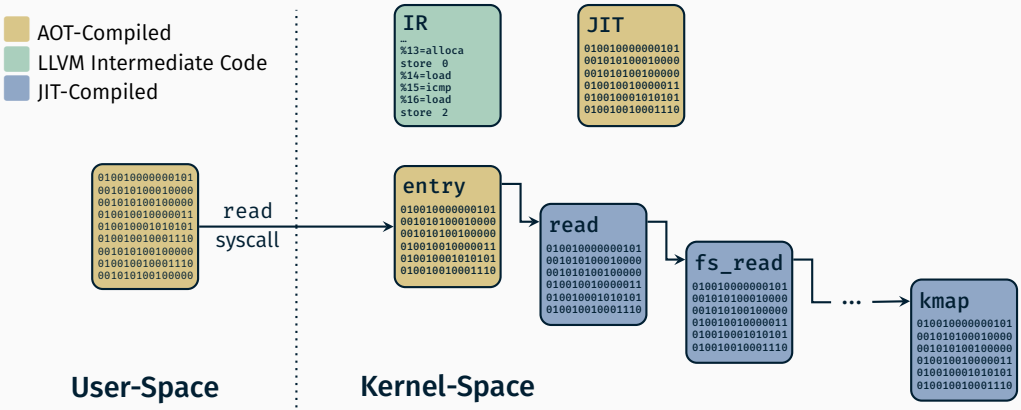
Plenty of JITs for User-Space Apps! Why not for the Kernel?



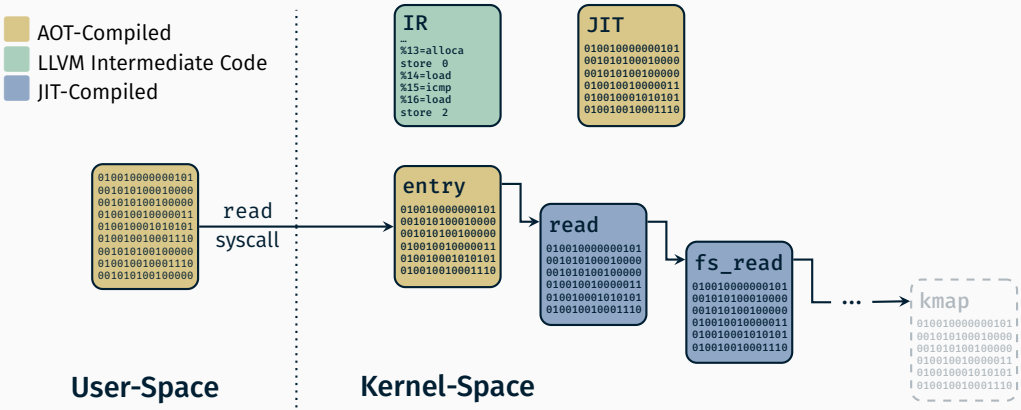
Plenty of JITs for User-Space Apps! Why not for the Kernel?



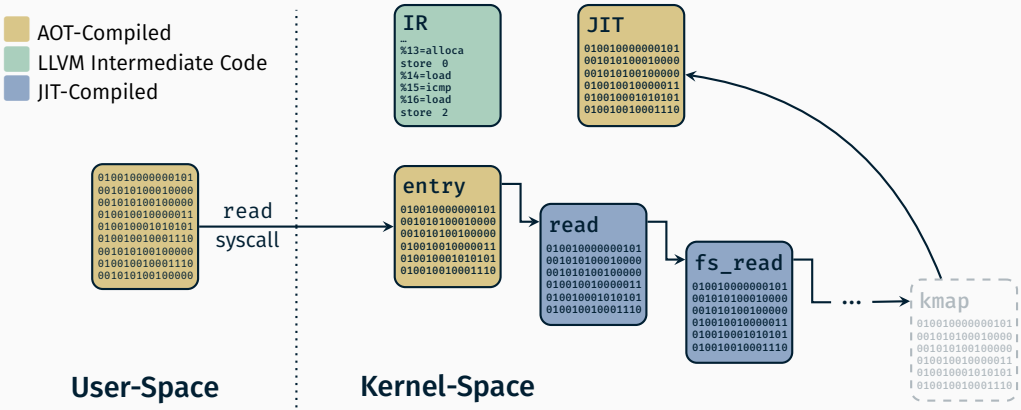
Plenty of JITs for User-Space Apps! Why not for the Kernel?



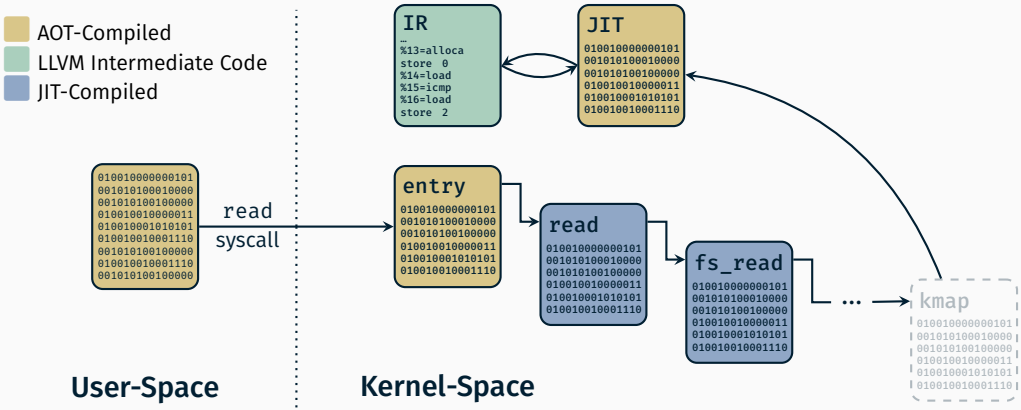
Plenty of JITs for User-Space Apps! Why not for the Kernel?



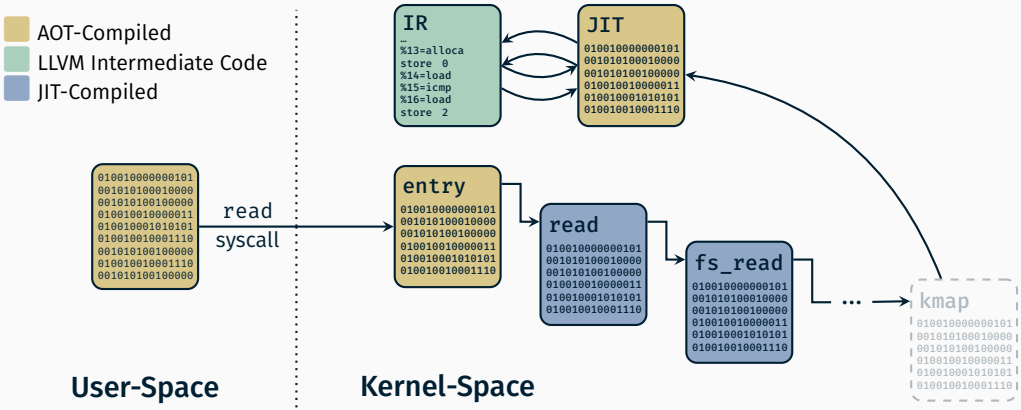
Plenty of JITs for User-Space Apps! Why not for the Kernel?



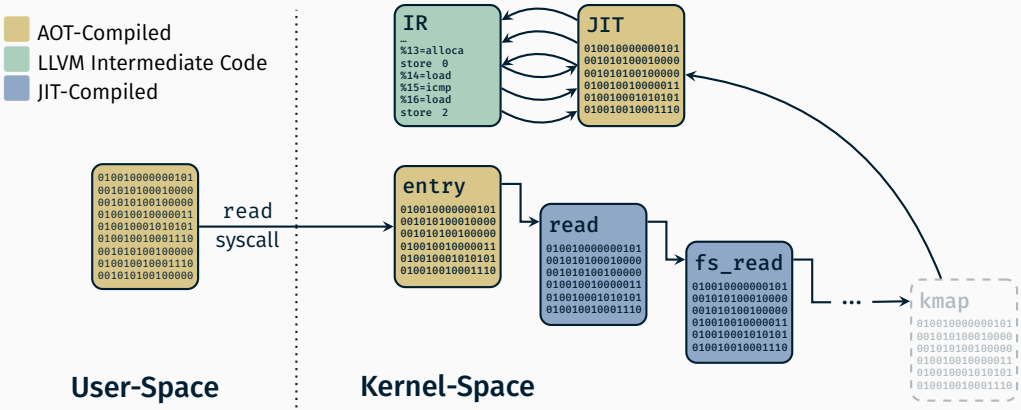
Plenty of JITs for User-Space Apps! Why not for the Kernel?



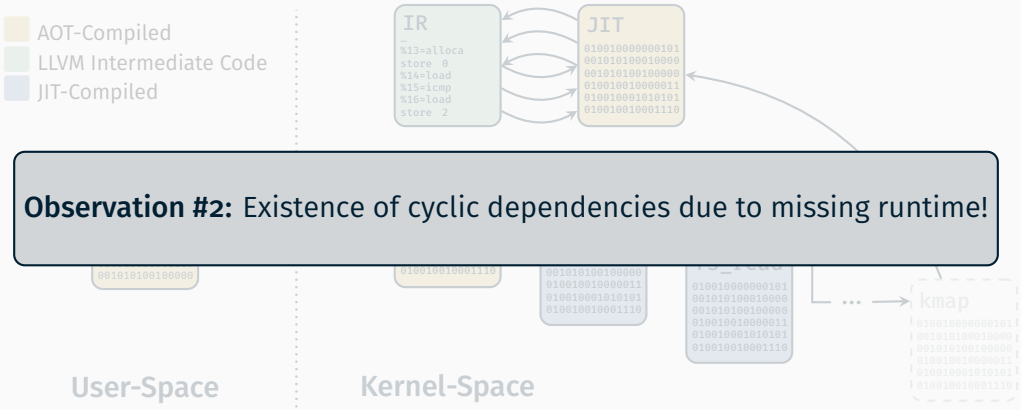
Plenty of JITs for User-Space Apps! Why not for the Kernel?



Plenty of JITs for User-Space Apps! Why not for the Kernel?



Plenty of JITs for User-Space Apps! Why not for the Kernel?



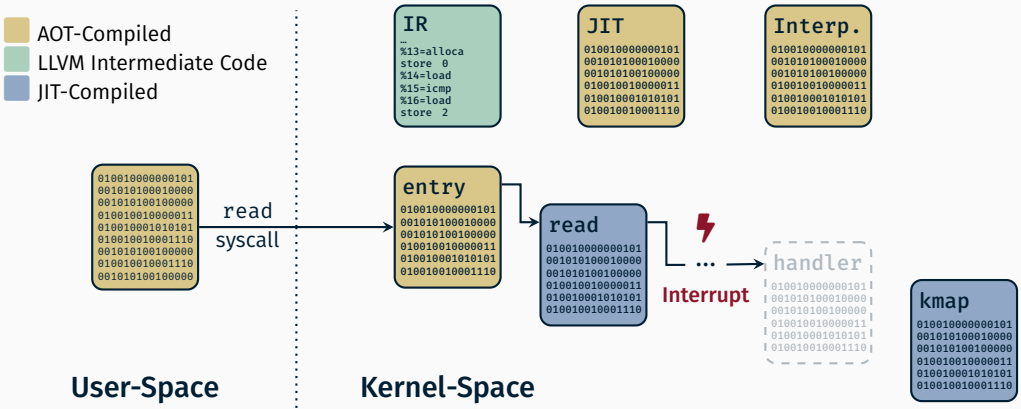
Observation #2: Existence of cyclic dependencies due to missing runtime!

DOSY: First steps towards on-demand compilation of operating systems.

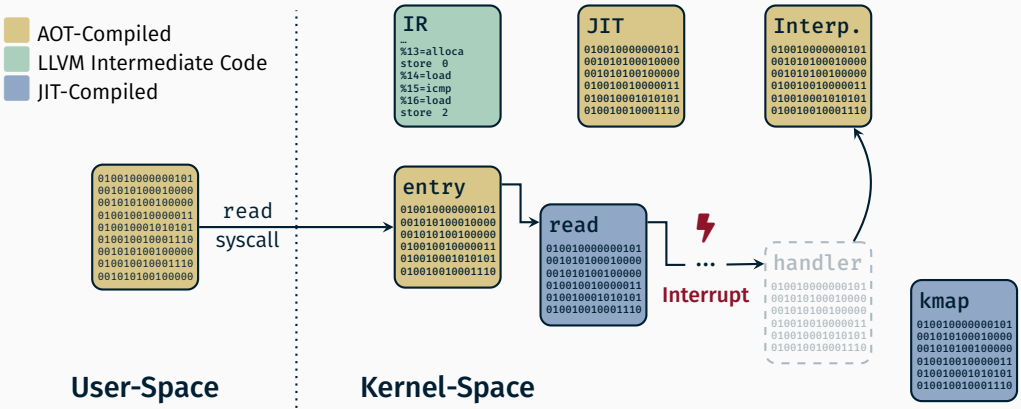
- **Design** for future symbiosis of OSs + Just-in-Time compiler[1] + Interpreter.
- **Implementation** of interpreter for resource-limited environments.
- **Static Analysis** to enforce required interpreter properties.

[1] currently Work-in-Progress

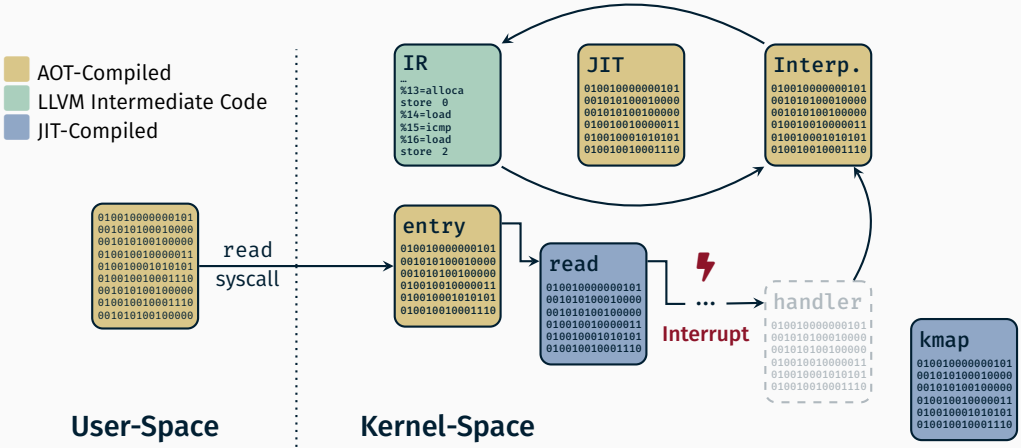
~~Why not simply...?~~ Custom-Tailored Interpreter!



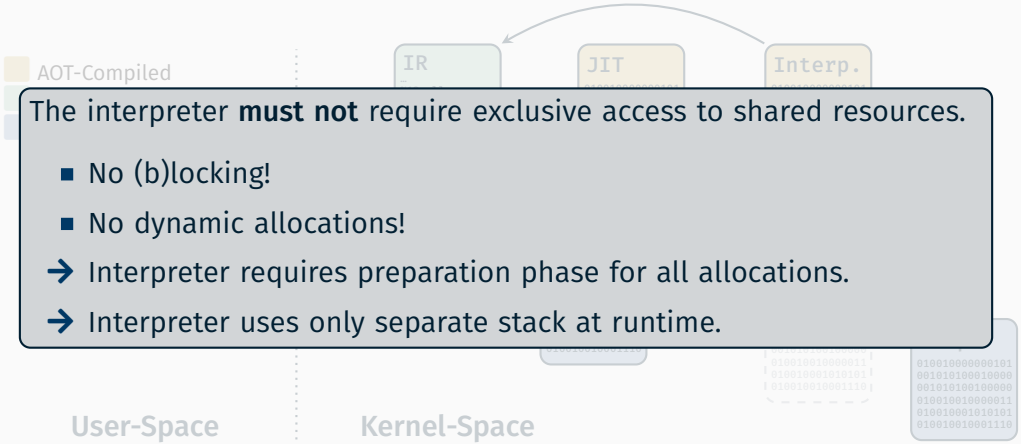
~~Why not simply...?~~ Custom-Tailored Interpreter!



~~Why not simply...?~~ Custom-Tailored Interpreter!



~~Why not simply...?~~ Custom-Tailored Interpreter!



Integration

DOSY is not restricted to a particular OS...

Integration

DOSY is not restricted to a particular OS… but certain concepts help!

Integration

DOSY is not restricted to a particular OS… but certain concepts help!

Desired properties:

- Strict separation of hardware in-/dependent subsystems

Integration

DOSY is not restricted to a particular OS… but certain concepts help!

Desired properties:

- Strict separation of hardware in-/dependent subsystems
- Strict separation of startup/runtime code

Integration

DOSY is not restricted to a particular OS… but certain concepts help!

Desired properties:

- Strict separation of hardware in-/dependent subsystems
- Strict separation of startup/runtime code
- Strict ”used-by” hierarchy

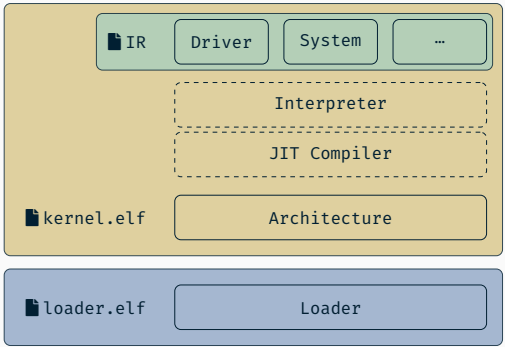
Integration

DOSY is not restricted to a particular OS… but certain concepts help!

JITTY OS: An Unix-/Linux-compatible kernel built with JIT in mind!

Desired properties:

- Strict separation of hardware in-/dependent subsystems
- Strict separation of startup/runtime code
- Strict ”used-by” hierarchy



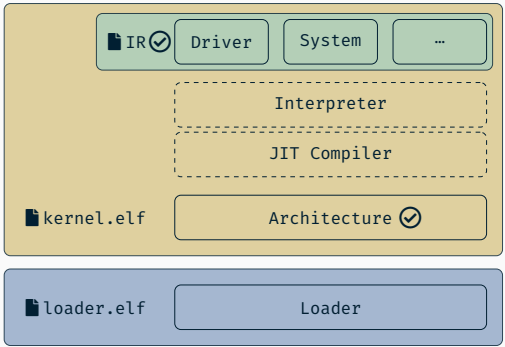
Integration

DOSY is not restricted to a particular OS… but certain concepts help!

JITTY OS: An Unix-/Linux-compatible kernel built with JIT in mind!

Desired properties:

- ☑ Strict separation of hardware in-/dependent subsystems
- ○ Strict separation of startup/runtime code
- ○ Strict ”used-by” hierarchy



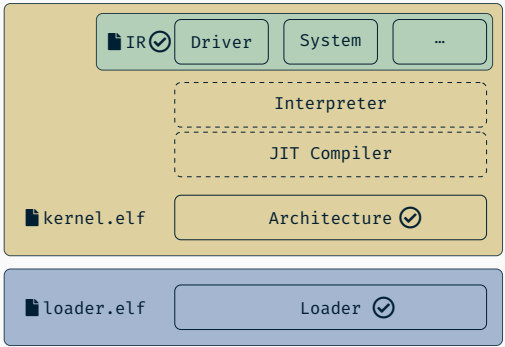
Integration

DOSY is not restricted to a particular OS… but certain concepts help!

JITTY OS: An Unix-/Linux-compatible kernel built with JIT in mind!

Desired properties:

- ✓ Strict separation of hardware in-/dependent subsystems
- ✓ Strict separation of startup/runtime code
- ○ Strict ”used-by” hierarchy



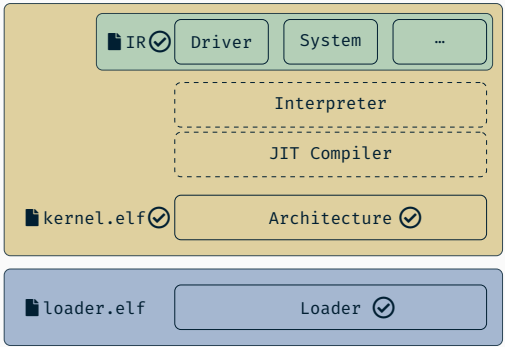
Integration

DOSY is not restricted to a particular OS… but certain concepts help!

JITTY OS: An Unix-/Linux-compatible kernel built with JIT in mind!

Desired properties:

- Strict separation of hardware in-/dependent subsystems
- Strict separation of startup/runtime code
- Strict ”used-by” hierarchy



Evaluation

Unfortunately, no results without JIT-Compiler in Kernel... Determine expected performance based on observations in user space.

Evaluation

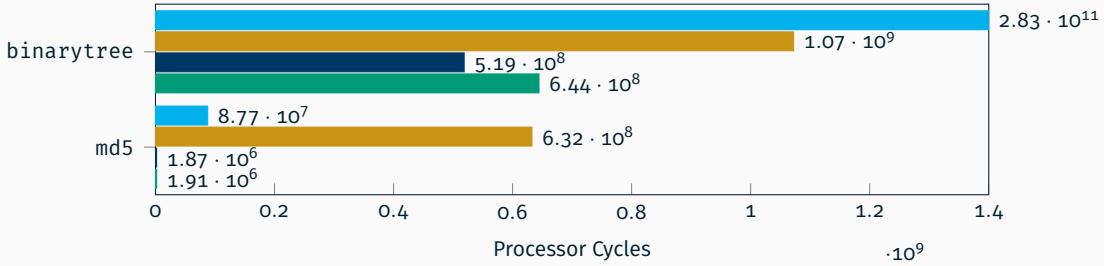
Unfortunately, no results without JIT-Compiler in Kernel… Determine expected performance based on observations in user space.

DOSY `interpreter`	≈ Our baseline
LLVM `lli` (basic JIT compiler)	≈ Expected costs of warmup phase
LLVM `clang -O3` (PGO)	≈ Expected performance of optimized JIT code
LLVM `clang -O0`	≈ Expected performance of unoptimized JIT code

Evaluation

Unfortunately, no results without JIT-Compiler in Kernel… Determine expected performance based on observations in user space.

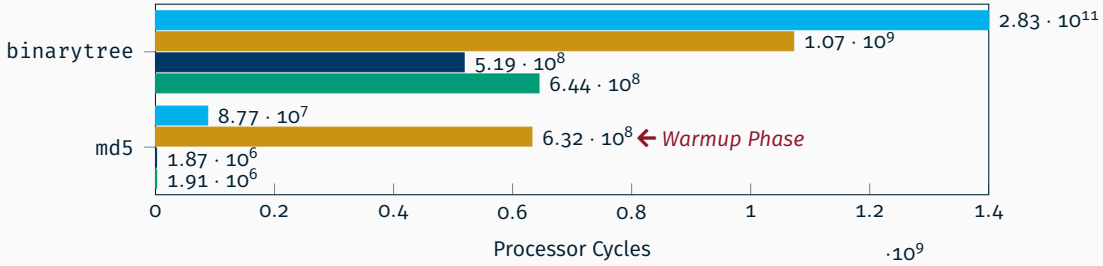
- DOSY `interpreter` ≈ Our baseline
- LLVM `lli` (basic JIT compiler) ≈ Expected costs of warmup phase
- LLVM `clang -O3` (PGO) ≈ Expected performance of optimized JIT code
- LLVM `clang -O0` ≈ Expected performance of unoptimized JIT code



Evaluation

Unfortunately, no results without JIT-Compiler in Kernel… Determine expected performance based on observations in user space.

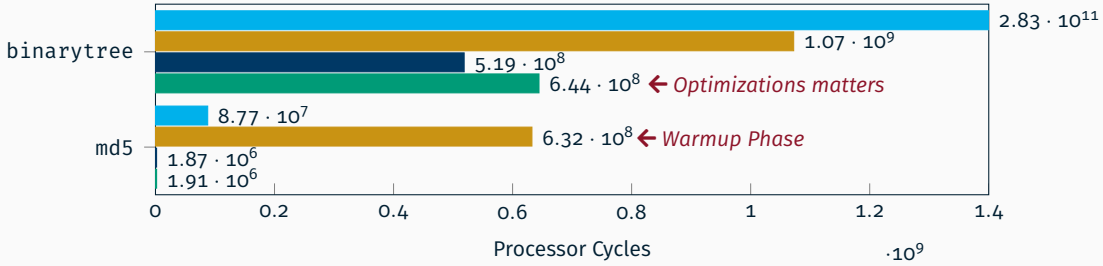
- DOSY `interpreter` ≈ Our baseline
- LLVM `lli` (basic JIT compiler) ≈ Expected costs of warmup phase
- LLVM `clang -03` (PGO) ≈ Expected performance of optimized JIT code
- LLVM `clang -00` ≈ Expected performance of unoptimized JIT code



Evaluation

Unfortunately, no results without JIT-Compiler in Kernel… Determine expected performance based on observations in user space.

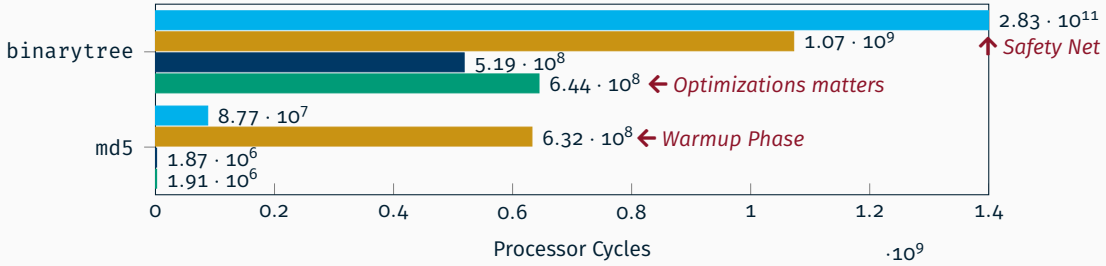
- DOSY `interpreter` ≈ Our baseline
- LLVM `lli` (basic JIT compiler) ≈ Expected costs of warmup phase
- LLVM `clang -03` (PGO) ≈ Expected performance of optimized JIT code
- LLVM `clang -00` ≈ Expected performance of unoptimized JIT code



Evaluation

Unfortunately, no results without JIT-Compiler in Kernel… Determine expected performance based on observations in user space.

- DOSY `interpreter` ≈ Our baseline
- LLVM `lli` (basic JIT compiler) ≈ Expected costs of warmup phase
- LLVM `clang -03` (PGO) ≈ Expected performance of optimized JIT code
- LLVM `clang -00` ≈ Expected performance of unoptimized JIT code



DOSY: Dynamic Operating System

Future symbiosis of...

- Operating Systems: JITTY OS 🕓, built to be *JITed*
- Just-In-Time Compiler
- Interpreter: DOSY Interpreter, used as *safety net*

...based on whole system state

Preprint version of the accepted PLOS '23 paper

References (1)

- B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers.
  Extensibility safety and performance in the SPIN operating system.
  In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95), pages 267–283, New York, NY, USA, 1995. Association for Computing Machinery.
- B. Heinloth, M. Ammon, D. Nguyen, T. Hönig, V. Sieh, and W. Schröder-Preikschat.
  Cocoon: Custom-fitted kernel compiled on demand.
  In (PLOS '19), pages 1–7, New York, NY, USA, 2019. ACM, ACM Digital Library.

References (2)

- C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang.
  Optimistic incremental specialization: Streamlining a commercial operating system.
  In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, page 314–321, New York, NY, USA, 1995. Association for Computing Machinery.
- C. Pu, H. Massalin, and J. Ioannidis.
  The Synthesis kernel.
  Computing Systems, 1(1):11–32, 1988.

References (3)

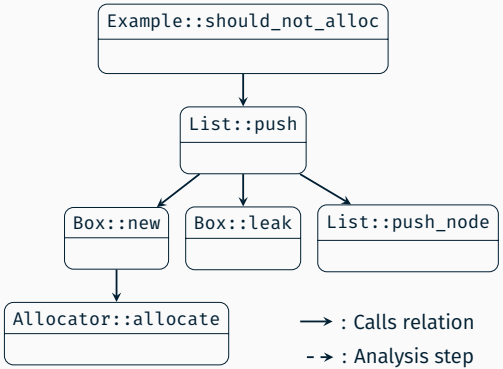

J. Schulist, D. Borkmann, and A. Starovoitov.
Linux socket filtering aka berkeley packet filter (bpf), 2019.
Retrieved 28. July 2023.

Static Analysis

Problem: Hidden allocations within interpreter.
Solution: Static analysis based on minimal annotations.

Basic algorithm:

1. Assign initial annotations
2. Query all traits and their implementation
3. Generate call graph for each function
4. Extended call graph based on collected trait information
5. Propagate annotations

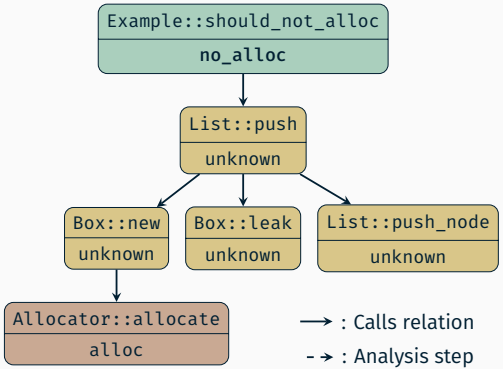


Static Analysis

Problem: Hidden allocations within interpreter.
Solution: Static analysis based on minimal annotations.

Basic algorithm:

1. Assign initial annotations
2. Query all traits and their implementation
3. Generate call graph for each function
4. Extended call graph based on collected trait information
5. Propagate annotations

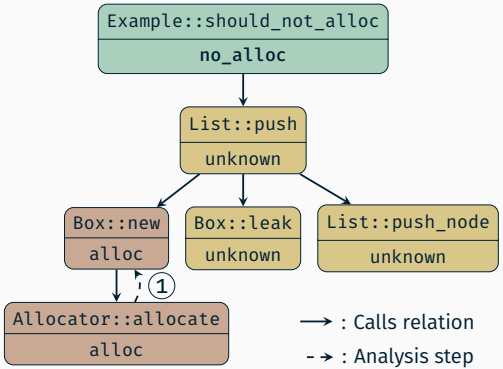


Static Analysis

Problem: Hidden allocations within interpreter.
Solution: Static analysis based on minimal annotations.

Basic algorithm:

1. Assign initial annotations
2. Query all traits and their implementation
3. Generate call graph for each function
4. Extended call graph based on collected trait information
5. Propagate annotations

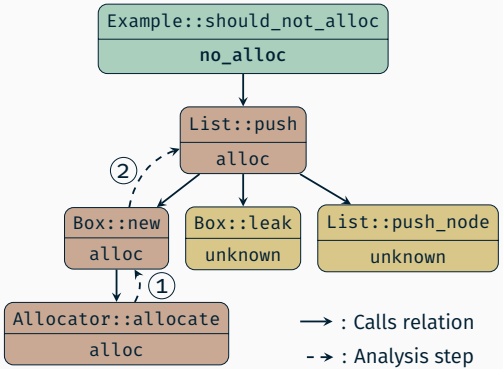


Static Analysis

Problem: Hidden allocations within interpreter.
Solution: Static analysis based on minimal annotations.

Basic algorithm:

1. Assign initial annotations
2. Query all traits and their implementation
3. Generate call graph for each function
4. Extended call graph based on collected trait information
5. Propagate annotations



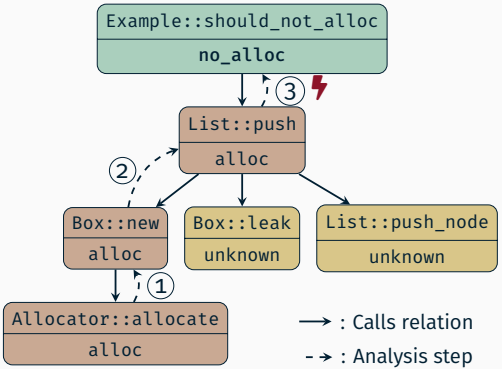
Static Analysis

Problem: Hidden allocations within interpreter.

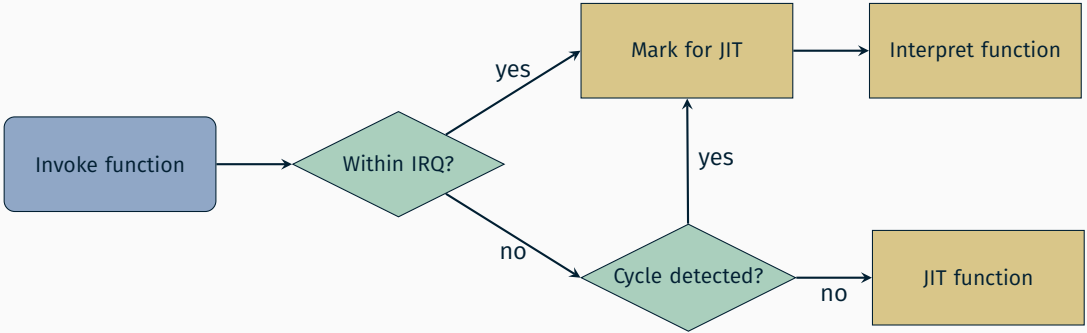
Solution: Static analysis based on minimal annotations.

Basic algorithm:

1. Assign initial annotations
2. Query all traits and their implementation
3. Generate call graph for each function
4. Extended call graph based on collected trait information
5. Propagate annotations



Invocation Mechanism



- Each function described by **Global Function Table** (GFT)
- Implementation similar to PLT/GOT

Global Function Table