Automating Context-Based Access Pattern Hint Injection for System Performance and Swap Storage Durability

SeongJae Park, Yunjae Lee, Moonsub Kim, and Heon Y. Yeom

Seoul National University

USENIX HotStorage'19 Jul.2019

Background

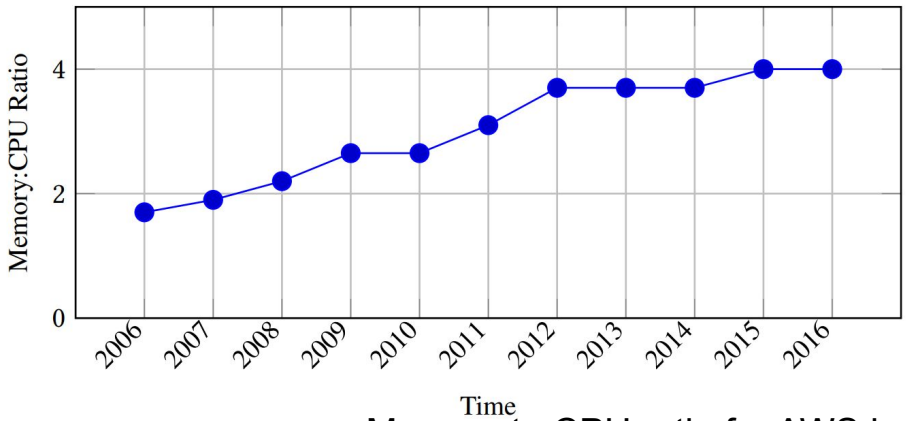
Design

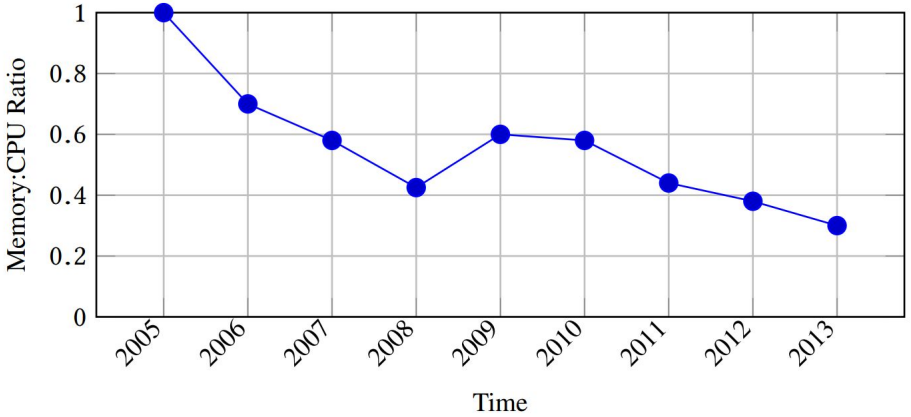
Evaluation

Summary

Memory Pressure is Inevitable

- Size of working sets is rapidly increasing
  - Common characteristic of modern workloads (e.g., cloud, big data, ML)
- Growth of DRAM space in a single machine is obviously slower than that
- Main memory alone will not be able to accommodate all of the working sets





Memory to CPU ratio for AWS instances (left) and physical servers (right)

Nitu, Vlad, et al. "Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter." Proceedings of the Thirteenth EuroSys Conference. ACM, 2018.

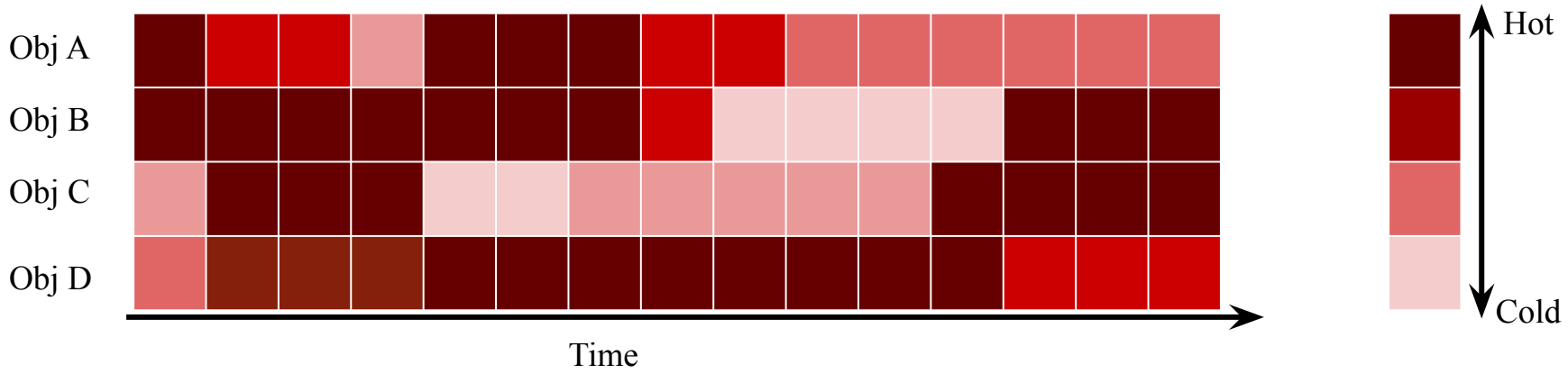
Modern Storage Devices as Secondary Memory

- New type of storage devices have rapidly evolved for last decade
- Much denser than DRAM, super-faster than traditional storages
- Systems utilizing fast storage devices as secondary memory are proposed
- Among those, we focus on swap system
  - Most stable and easy way to construct such a heterogeneous memory system
  - Our main idea is not for only swap but general heterogeneous memory systems, though

https://www.theverge.com/circuitbreaker/2017/10/31/16582018/intel-optane-p900-ssd-fast-dram-nand-flash-memory-desktop-computer
https://www.theverge.com/circuitbreaker/2018/2/20/17031256/worlds-largest-ssd-drive-samsung-30-terabyte-pm1643

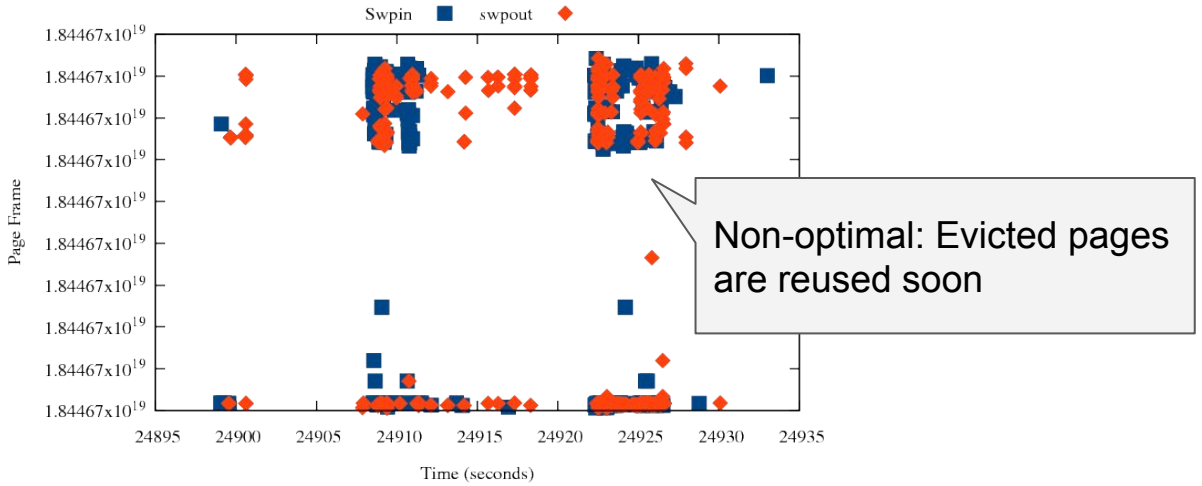
The Challenge: Optimal Data Objects Placement

- Modern storage devices cannot simply substitute DRAM
  - Obviously slower than DRAM
  - Number of writes to those is limited
- Hot data object should be placed in DRAM to minimize access to swap device
  - Non-optimal data placement can degrade performance and/or durability
- Complex and dynamic data access patterns make it hard to be optimal



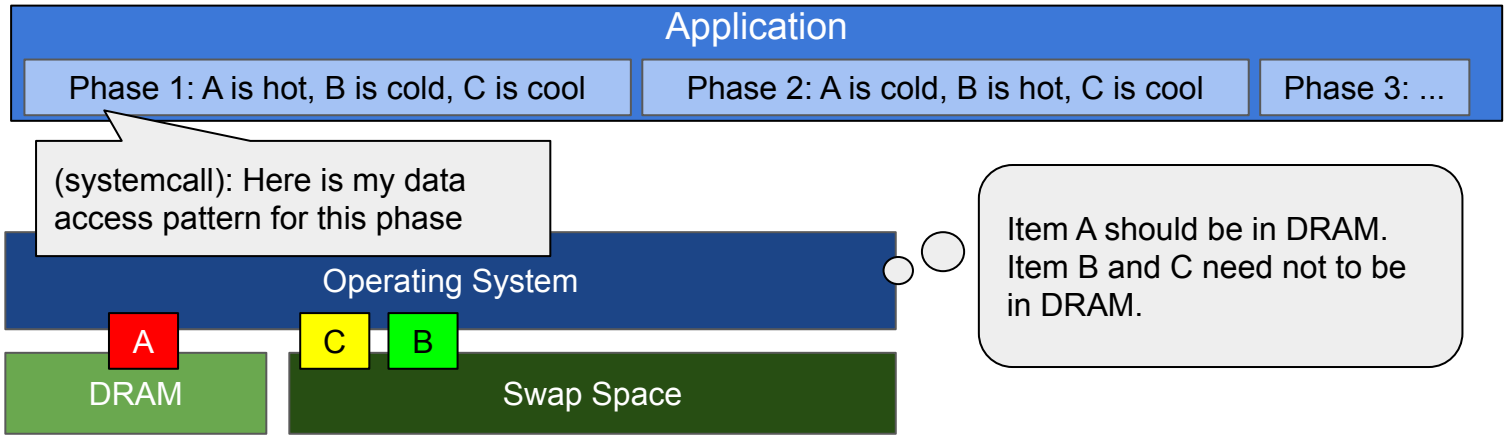
Existing Sol'n #1: Data Access Pattern Estimations

- Underlying system cannot know the access pattern of the application
- Thus, most systems estimate data access pattern using LRU-like schemes
  - Linux kernel swap system also uses pseudo-LRU page reclamation scheme
- Estimation-based schemes cannot make the optimal decision



Existing Sol'n #2: Self Access Pattern Notification

- Systems provide special system calls which can be used for the pattern hint
- Applications can voluntarily hint their data access pattern via the system calls
  - E.g., `mlock()` forces specific data objects to be locked in main memory
  - Proper use of such system calls can significantly improve performance and durability
- That said, proper use of these hinting system calls is challenging



Challenges for Hint-based Optimizations

- Data access pattern analysis
  - Code review and multiple experimental evaluations are essential
  - Natively time consuming and complex
- Hint code injection (program modification)
  - Program is constructed with multiple phases that each having unique access pattern
  - How to know which phase we are currently executing?
  - Injecting hint code for every phase including short ones can incur high overhead (Hinting system call itself has overhead)
- More efforts are required as the size and the complexity of program grows

Contribution of This Paper

- Automate the exhaustive access pattern analysis and optimizations
  - Use program context as the minimal phase of execution
  - Analyze the data access pattern using a combination of static/dynamic analysis
  - Inject the hinting system calls by prioritizing execution phases and objects
  - We call the prototype as DAPHICX: Data Access Pattern Hint Injecting Compiler Extension
- Evaluate the overhead and improvement with 8 realistic workloads
  - Shows consistent and significant improvement for performance and durability

Background

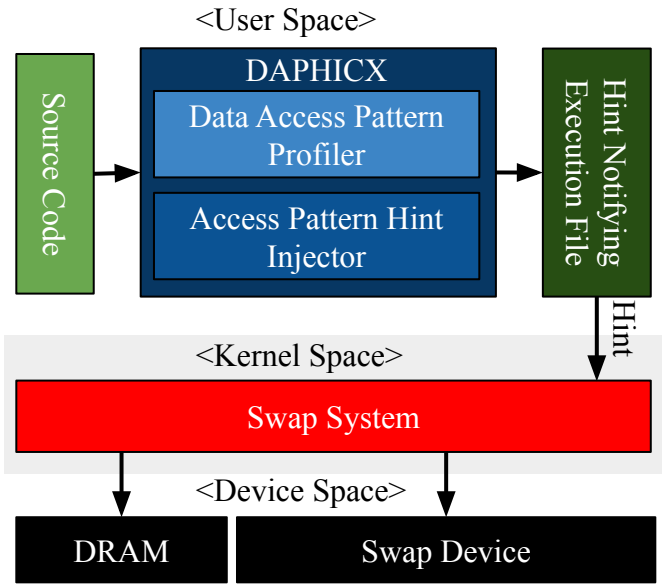
Design

Evaluation

Summary

The Workflow of DAPHICX

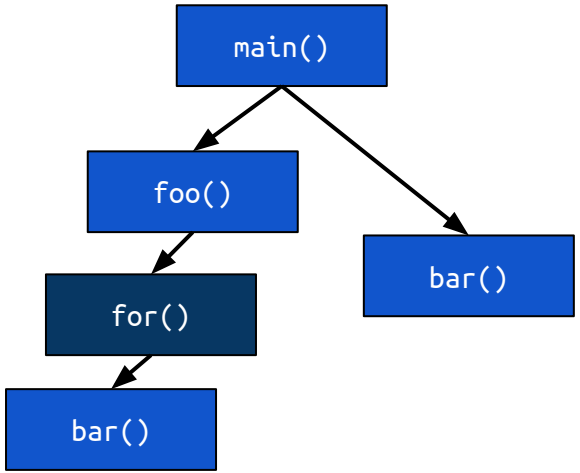
- Receive the source code of the target workload as an input
- Extract dynamic data access pattern
- Split given program into multiple execution phases that each has unique and meaningful access pattern
- Inject hints that notifying important objects for each phase



Static Analysis of Program Contexts and Objects

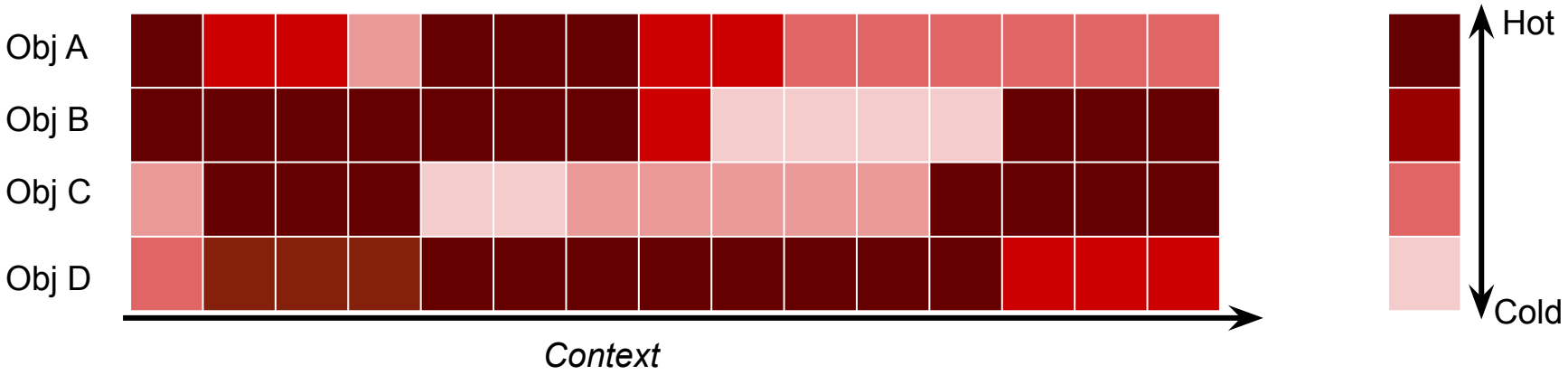
- Program context: Function or loop containing cumulated callsite information
  - Each context would have unique data access pattern
  - We use contexts as the minimal program execution phase
- Extract context and data objects information via static analysis of the code
  - Analyze function/loop entrance/exit for the context analysis
  - Analyze malloc()/realloc()/calloc()/free() invocations for the object analysis

```
01 int bar() {
02     return 42;
03 }
04 void foo() {
05     for (int i = 0; i < 3; i++)
06           bar();
07 }
08 int main(void) {
09     foo();
10     return bar();
11 }
```



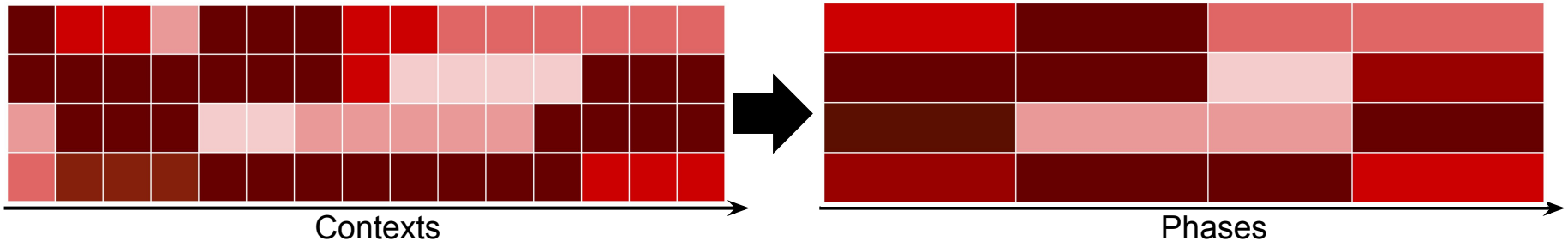
Dynamic Analysis of Data Access Pattern

- Build profiling program by injecting dynamic analysis code
  - Injected code counts number of accesses to each object from each program context
  - Hook every function/loop entrance/exit and every memory access
- Data access pattern of the workload is profiled by running the program
  - That is, the pattern represents the number of accesses to each object from each context



Verbose Contexts to Meaningful Phases

- Program context based access patterns are too verbose to be used as is
  - If a context has very short execution time, giving hint for it incurs only overhead
  - If adjacent contexts have similar access pattern, hint should be given only once
- Merge small/unnecessary program contexts into meaningful phase
  - If execution time of a contexts is smaller than given threshold, merge it with adjacent one
  - If two adjacent contexts has similar data access pattern, merge them
  - Contexts remaining after this merging step is called phases
- DAPHICX injects access pattern hints for each phase, not for each context



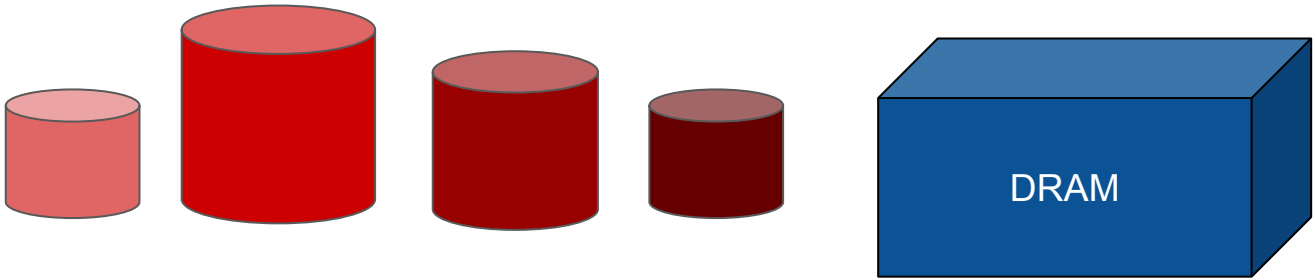
Prioritizing Data Objects

- If we lock a data object in memory,
  - Number of DRAM hits increases as the number of accesses to the object grows
    - Every accesses to the object will incur DRAM hit
  - Number of DRAM misses increases as the size of the object grows
    - Other objects which might incur DRAM hits would be evicted to the secondary memory for the data object pinning
- Based on this finding, we reason a simple object priority calculation model
  - Two constants (alpha and beta) control the growth rate of each metric

$$\text{Priority(object)} = \frac{\text{Number\_of\_accesses(object)}^{\alpha}}{\text{size\_of(object)}^{\beta}}$$

Selecting Objects to be Locked in DRAM

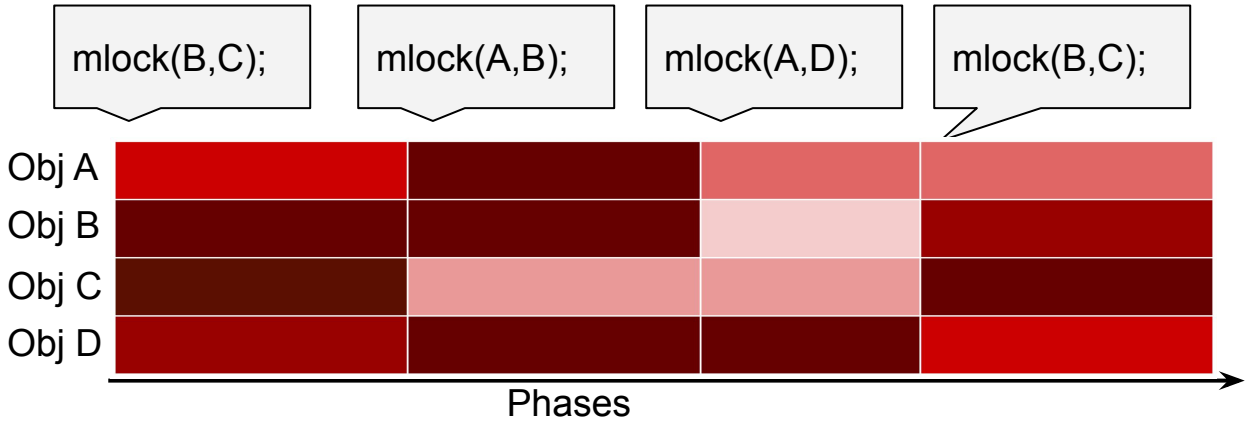
- Select objects to be placed in DRAM so that total priority of those be biggest
- Limited DRAM, different size and priority of objects: a Knap-sack problem
- DAPHICX utilizes a simple solution
  - Key difference with other knap-sack solution: The objects can be splitted
  - For each phase, select highest priority objects as many as possible
  - If next highest priority object cannot be locked in to remaining DRAM space, split the object in half, re-calculate priority, and repeat selection until DRAM is full



Objects sorted by their priority

Hint Injection

- DAPHICX generates optimized program by injecting hint notification code
- Injected hint code does
  - Check current phase by hooking phase-related function/loop entrance/exit
    - Phase checking overhead is minimized by hooking only phase-related ones
  - For entrance to each phase, invoke mlock() calls for selected data objects of current phase



Implementation

- The prototype implementation is based on LLVM
- 5,000 lines of code for data access pattern analysis
- 200 lines of code for hinting object selection
- 1,300 lines of code for the hint injection

https://upload.wikimedia.org/wikipedia/en/thumb/4/4c/LLVM_Logo.svg/1200px-LLVM_Logo.svg.png

Background

Design

Evaluation

Summary

Evaluation Setup

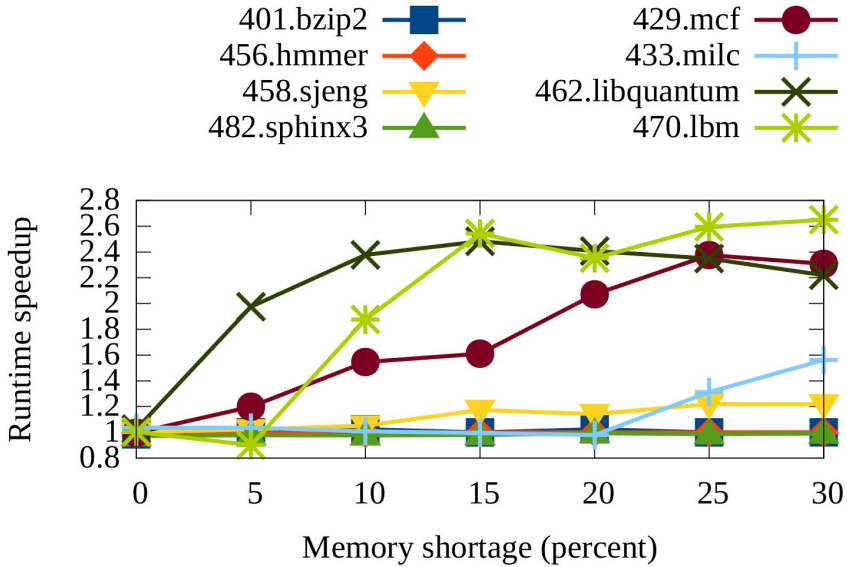
- Intel Xeon E7-8837 @ 2.67 GHz
- Intel Optane SSD (DC4800) as a swap device
- Workloads: memory-intensive workloads from SPEC CPU 2006
- Memory pressure: synthetically induced via cgroups
  - Shrink available memory down to 70% of working sets (30% shortage)
  - 30% shortage would be reasonable for 1.5:1 memory overcommit environment (OpenStack encourages this ratio)

https://images-na.ssl-images-amazon.com/images/I/51Q1zcv%2Bk%2BL._SY355_.jpg

https://www.storagereview.com/images/StorageReview-Intel-Optane-SSD_4800.jpg

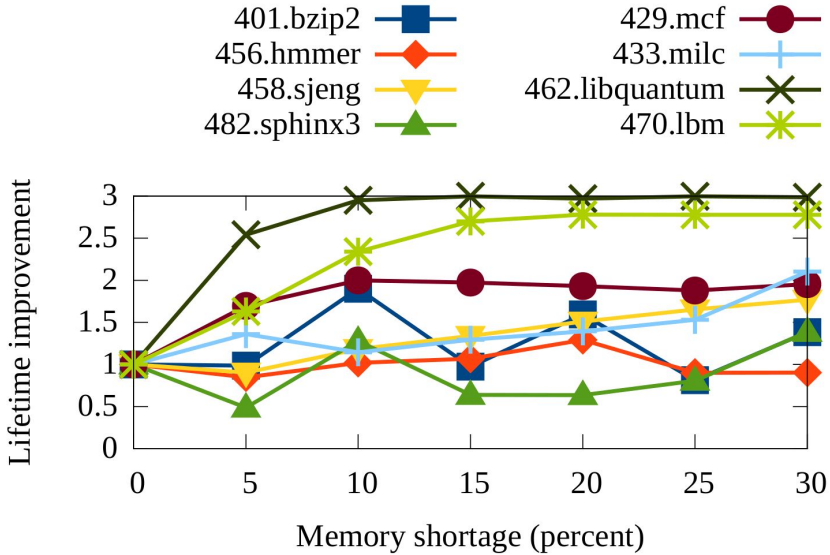
Performance

- Runtime speedup = runtime(original) / runtime(optimized)
- Injected hint code induces no overhead due to the merge of contexts
- For 30% memory shortage, 1.4x to 2.65x performance with 5 workloads



Durability

- Lifetime improvement = nr_swpout(original) / nr_swpout(optimized)
- Up to 2.98x durability improvement measured in best case
- Durability degrades with 482.sphinx3 due to its plethora of data objects
  - The size of hint information becomes too huge so that it can increase number of swap events



Summary

- Systems utilizing fast swap storages will widespread
  - Size of working sets is rapidly increasing while DRAM space is not
  - Modern storage devices have continuously evolved so that those could be secondary memory
- Optimization for swap systems is required
  - Modern storage devices are slower than DRAM and restricts total number of writes
  - Underlying swap system cannot know accurate data access pattern of running workloads
  - Manually analyzing and notifying the pattern via system calls are exhaustive to human
- We introduce an automated data access pattern hint injection compiler
  - Extracts program contexts and data access patterns via static/dynamic analysis
  - Injects access pattern hint code in the execution binary
  - Hint injected version achieves up to 2.65x performance and 2.98x durability

Any question, please!

E-mail is also always welcome:
SeongJae Park <sj38.park@gmail.com>