1 Evaluation

1.1 Experimental Design

Our evaluation is based on two benchmarks with significantly different access patterns. The first is quicksort (Qsort). This benchmark first allocates a large array of random numbers, and then sorts it using the well-known quicksort algorithm. Quicksort is a divide and conquer algorithm that automatically partitions the input array into small local blocks before performing a final sort. tterns. Further- orms no file I/O,

1

Code-friendly description of raw data involved in this experiment (defines to the user what types of questions are possible). This field is collapsable. It is unclear if this should be a real piece of code or an abstract description.

ark (Gen). Gen begins by loading a large text file that represents raw genome data. Raw genome data consists of short, overlapping, sequences of base-pairs called "contigs", the goal is to align these overlapping contigs into a single contiguous sequence representing a genome. This is done by loading contigs into a large hash table and probing into it repeatedly to find matching sequences. This leads to very little locality and unpredictable access patterns. Furthermore, Gen performs file I/O on the input which allows for more complex OS interactions.

▼ ***Schema***

```
# Data Schema
result = NamedTuple(
        'mean', # Mean results from 10 runs
        'std', # Standard deviation from 10 runs
)

# mean and std have the same set of keys
result.mean.keys
[ 't_run', 't_bookkeeping', 't_rmem_write',
  't_rmem_read', 'n_fault', 't_fault',
        'n_swapfault', 'n_pfa_fault', 'n_early_newq',
        'n_evicted', 'n_fetched', 'n_kpfad',
        't_kpfad', 'slowdown']

# Datasets
# Paging to remote memory in SW ('baseline')
base_res = ingest_run('raw/results_baseline.csv', 'rv')

# Paging using the PFA to real memblade
pfa_res = ingest_run('raw/results_pfa.csv', 'rv')
```

2 End-to-End Performance

2

Linked to real experimental artifact. Removes some ambiguity from the experimental design. Collapsable to avoid distracting users that don't care

...o benchmarks were both run under a cgroup in Linux in order to reduce the av...ld need to share li... administrators us... today to control appli...ion memory consumption (e.g., in containers). In this experiment, we disable kpfad in order to isolate the batching of new-page management from the scheduling flexibility offered by kswapd's asynchrony. The PFA was co...ured to allow up to 64 outstanding page faults before bookkeeping was performed.

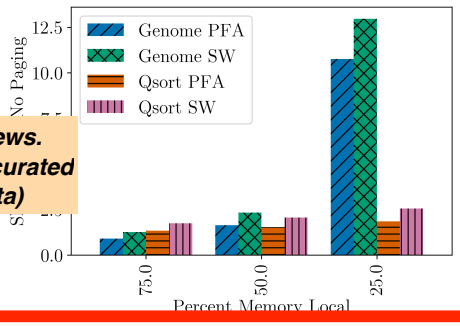
▼ Linux Config

```
CONFIG_PFA=y
CONFIG_PFA_EM=n
CONFIG_MEMBLADE_EM=n
CONFIG_PFA_VERBOSE=n
CONFIG_PFA_DEBUG=n
CONFIG_PFA_PFLAT=n
CONFIG_PFA_KPFAD=n
CONFIG_PFA_FREEQ_SIZE=64
CONFIG_PFA_NEWQ_SIZE=64
CONFIG_PFA_EVICTQ_SIZE=1
```

3

Live updated data from experimental results (uses tangle library from ExplanableExplanations)

Both applications use 64MB of memory at their peak. We then varied the cgroup memory limit from 100% (64MB) down to 25% (16MB), triggering increasing levels of paging. For both benchmarks, the PFA reduces end to end run time by up to 1.4x.



4

Tabbed interface to alternative data views. "Original" view is static and immutable, curated by author (still derived from base data)

Original | User | Explore

Figure 1: PFA vs Baseline without kswapd. Applications run approximately 20-40% faster when the PFA is enabled.

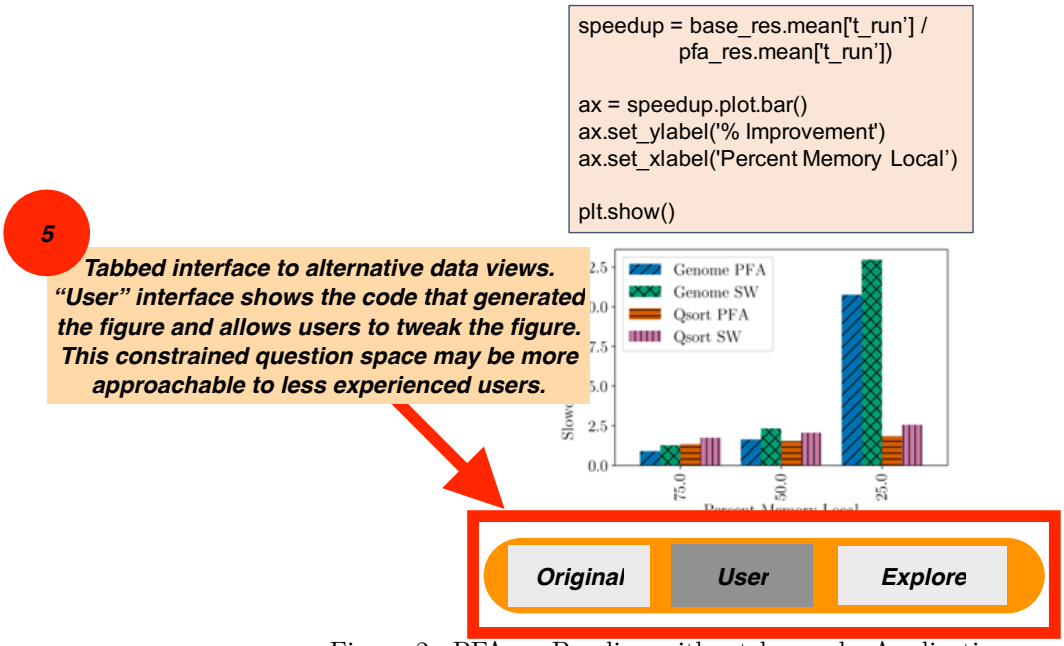


Figure 2: PFA vs Baseline without kswapd. Applications run approximately 20-40% faster when the PFA is enabled.

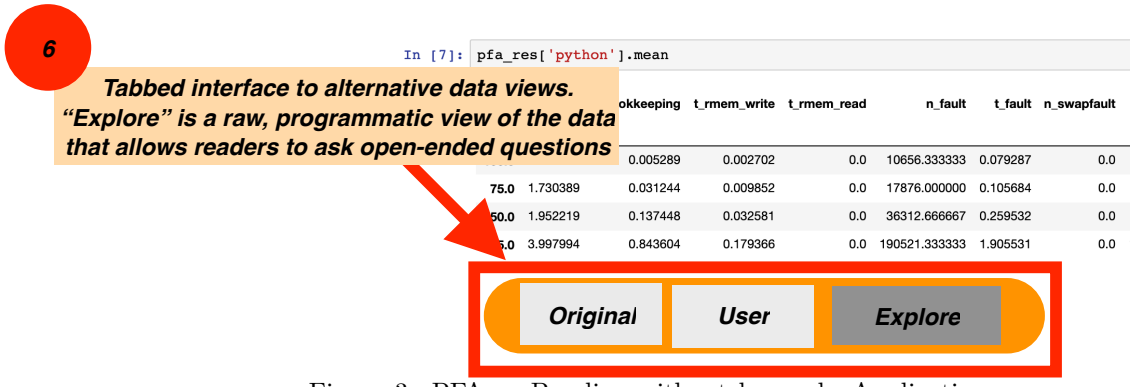


Figure 3: PFA vs Baseline without kswapd. Applications run approximately 20-40% faster when the PFA is enabled.