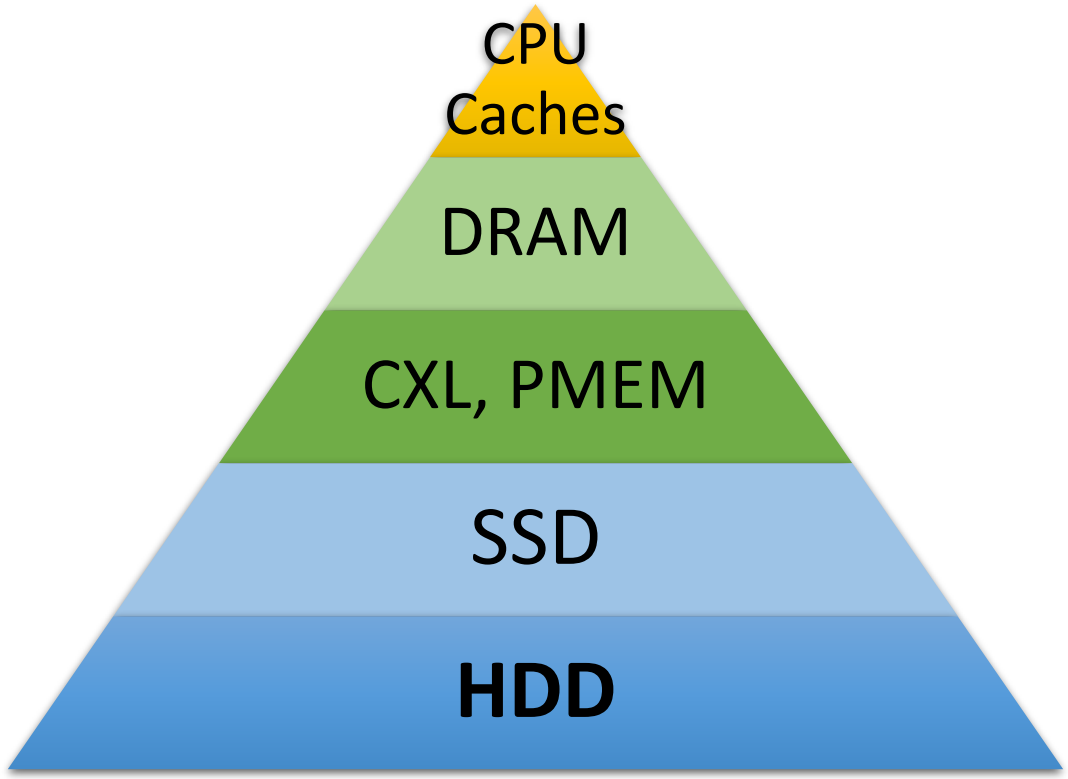
Johny Cache: the End of DRAM Cache Conflicts in Tiered Main Memory Systems

Baptiste Lepers
Willy Zwaenepoel

THE UNIVERSITY OF SYDNEY

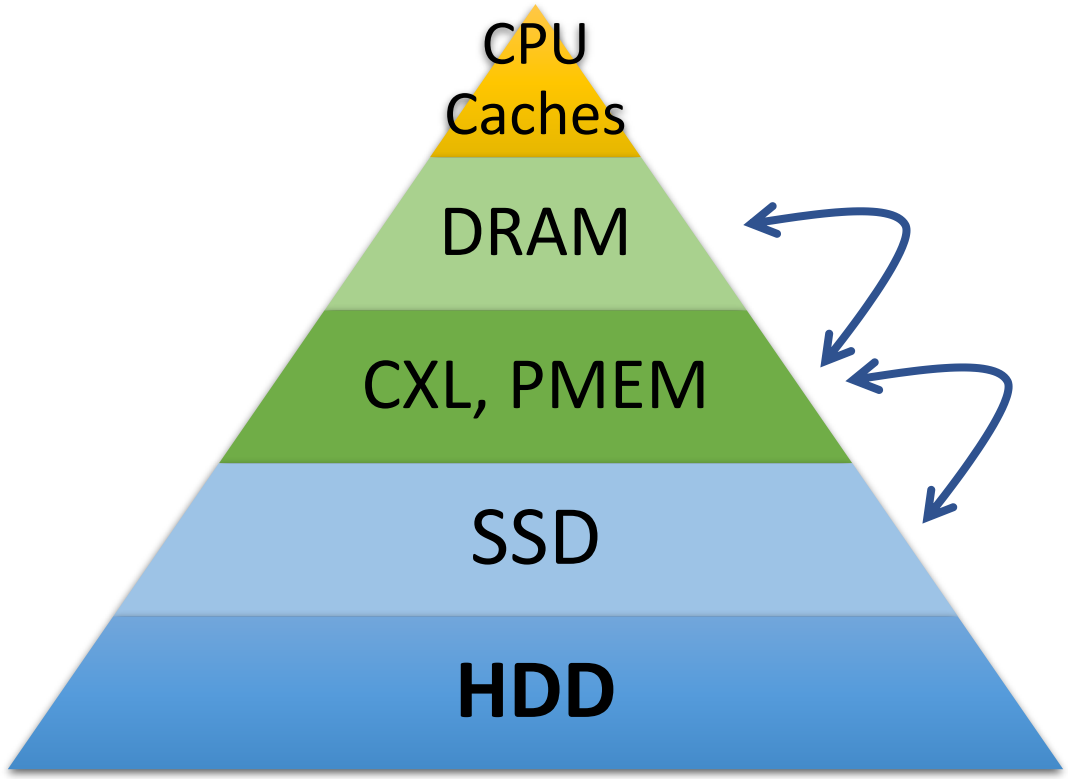
unine
Université de Neuchâtel

Tiered-memory systems



Goal:
hottest data should be at the top

Traditional approach



Software monitors memory accesses

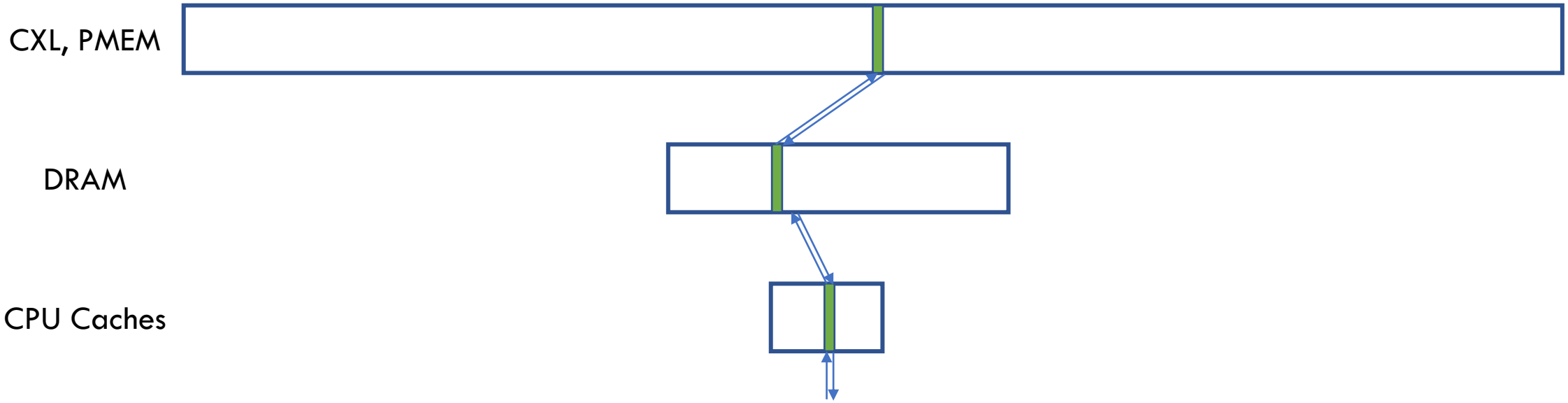
Software decides to migrate pages

✓ Full control over data placement

✗ Profiling & migration overheads

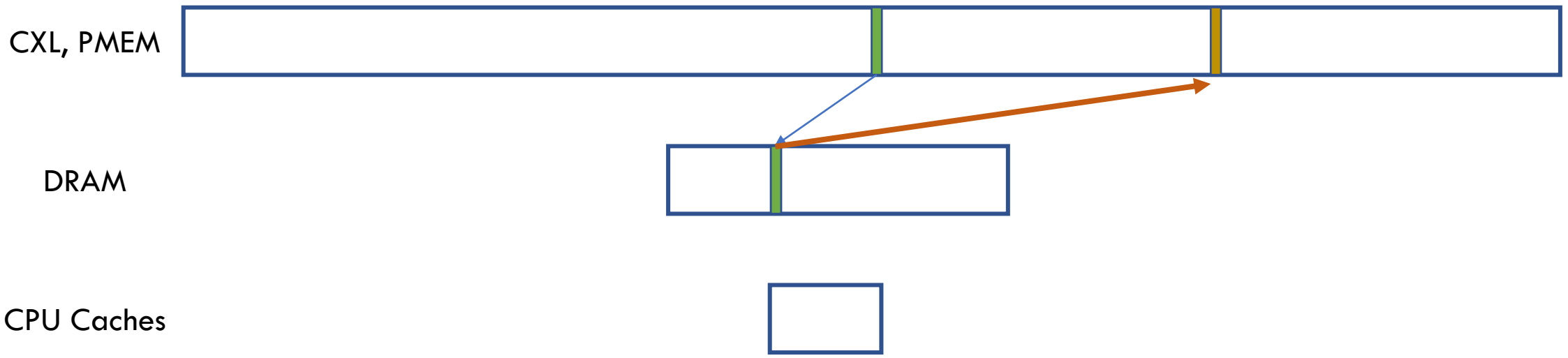
✗ Page granularity

Hardware approach



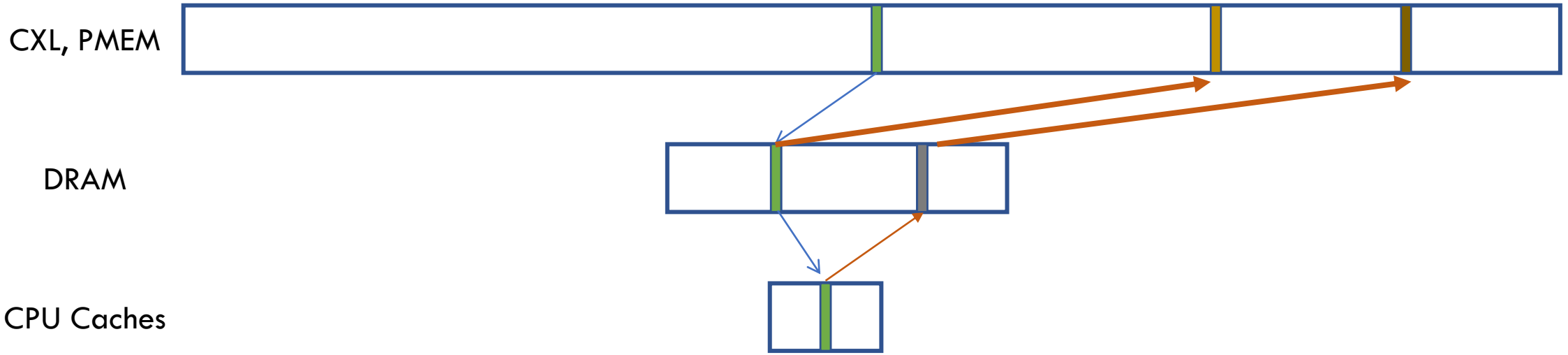
Hardware uses DRAM as a “L4” cache.

Hardware approach



Problem: conflicts cause writebacks.

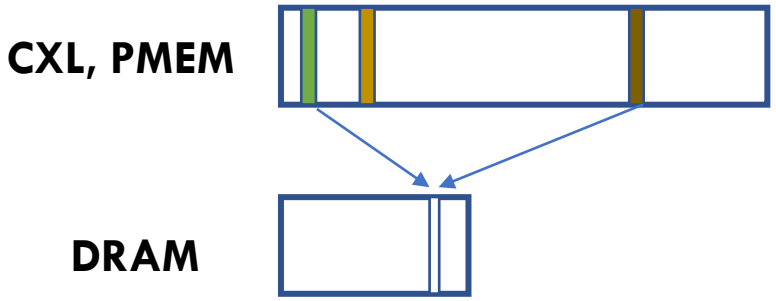
Hardware approach



Worst-case scenario:
2 writebacks to the slower tier

Problem with Linux

Linux is oblivious of hardware caching



Birthday paradox:

Multiple pages likely to be cached in the same slot ➔ many conflicts

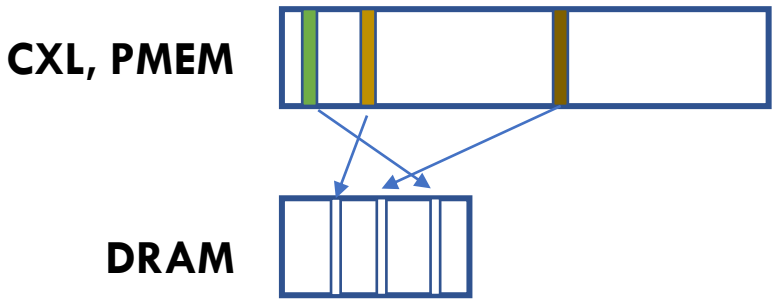
This talk

When conflicts are minimized

Hardware caching performs well

Conflict minimization

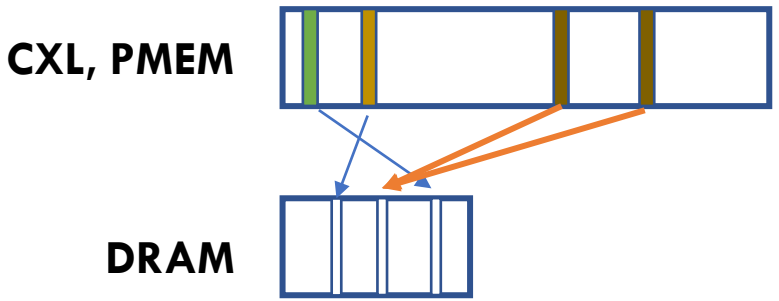
Static policy : conflict-aware page allocation



Minimize overlaps at page allocation time

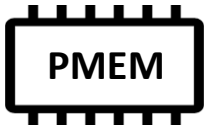
Conflict minimization

Dynamic policy : remap pages to avoid conflicts



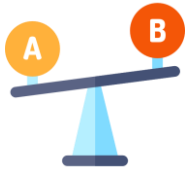
- Detect conflicts between hot pages
- Remap (migrate) pages to avoid conflicts

Evaluation

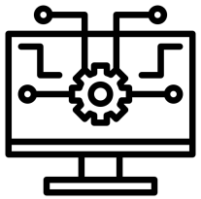


1TB

128GB

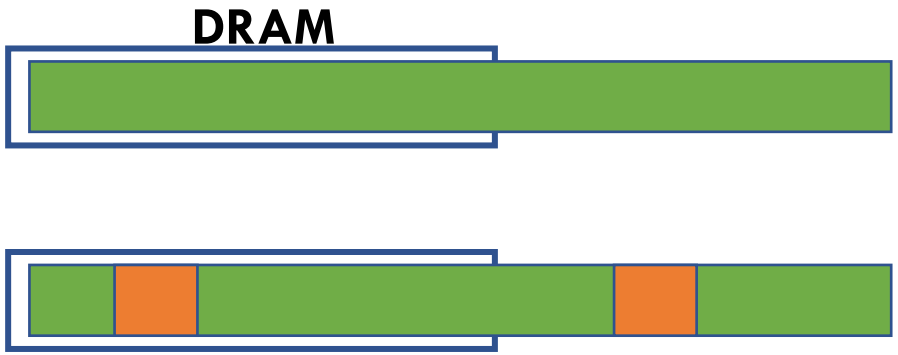


Johnny Cache: conflict minimization
HeMem (SOSP'21): hot page migration



HeMem's workloads

Evaluation: GUPS

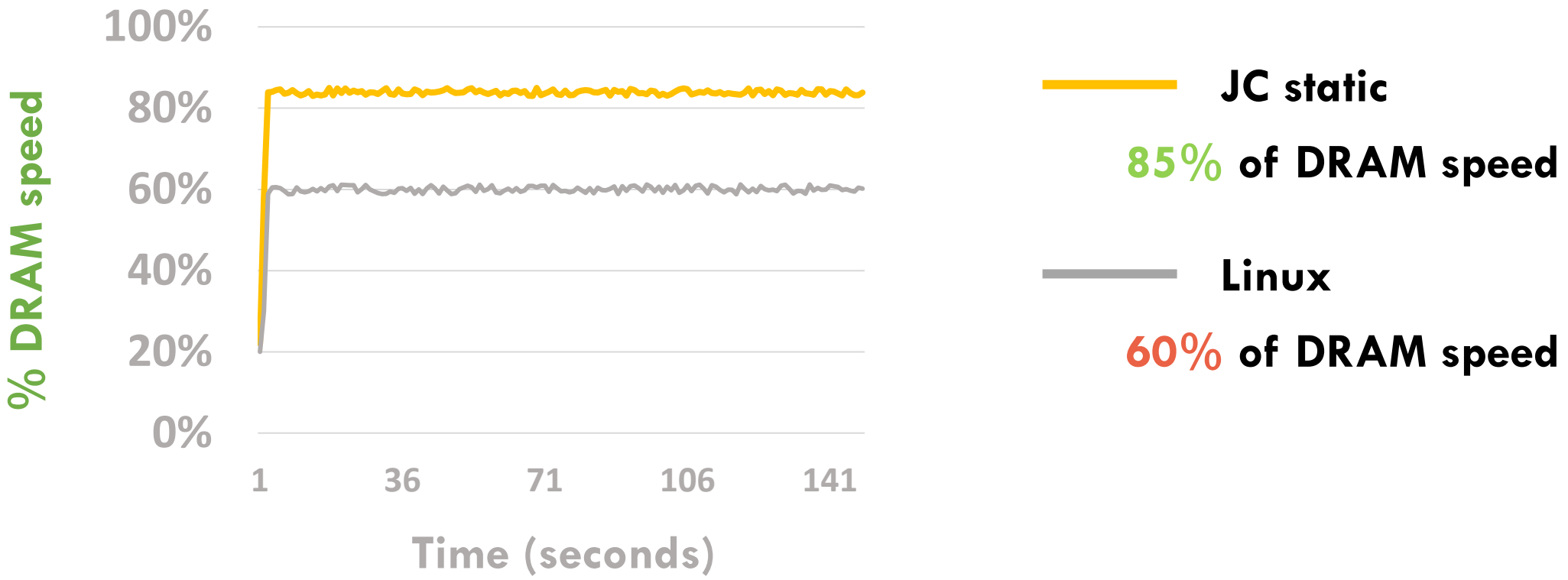


Array 2x the DRAM size

Random writes on 10 % of the pages

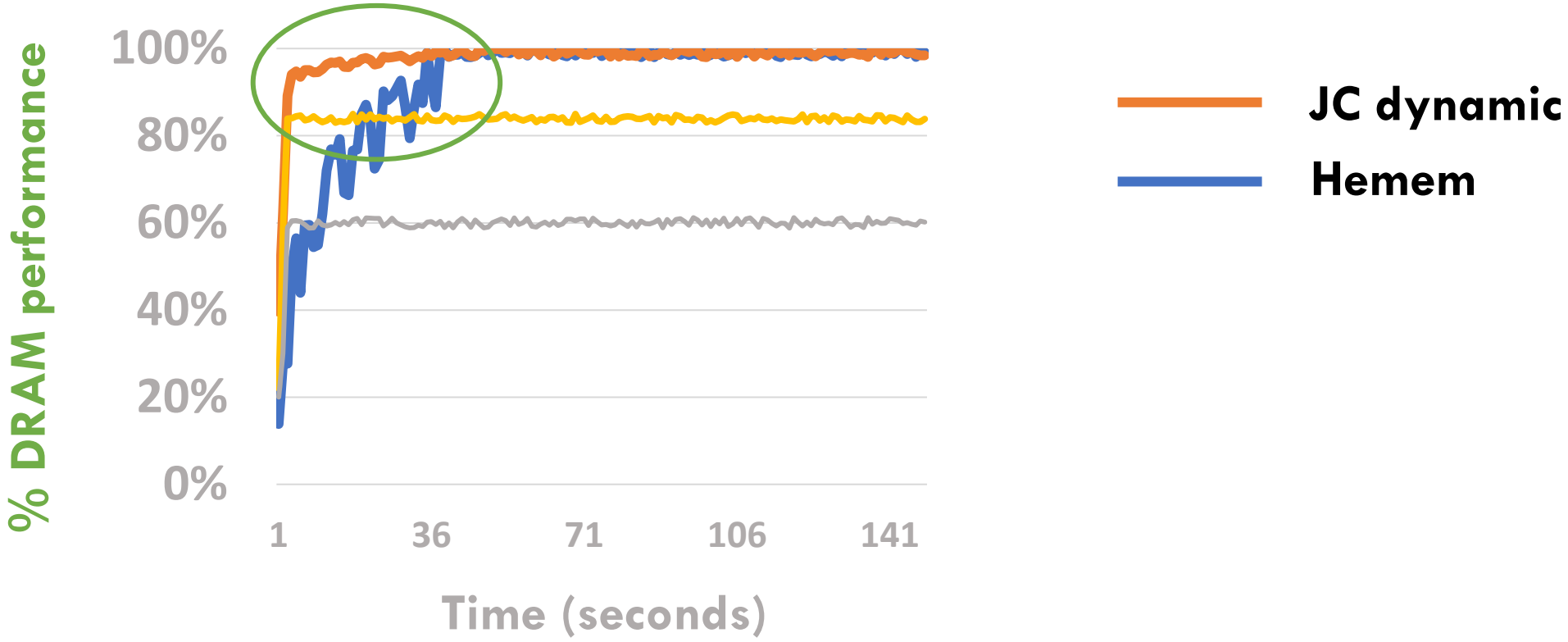
Experiment : Vary the number of threads doing random writes

Evaluation: GUPS (16 threads)

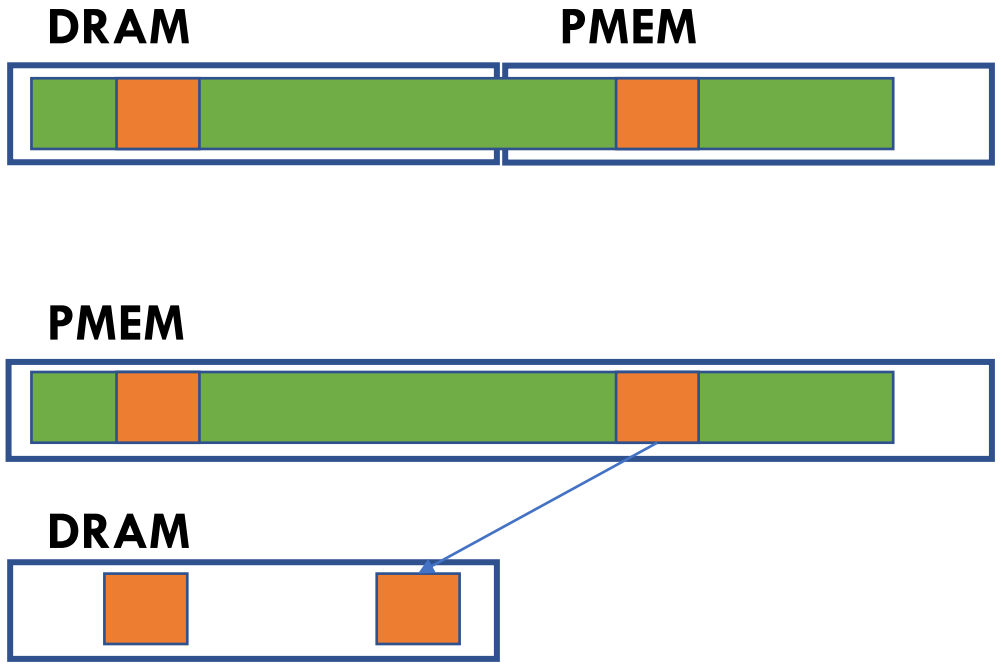


Conclusion #1: Near optimal by minimizing conflicts at allocation time

Evaluation: GUPS (16 threads)



Evaluation: GUPS (16 threads)

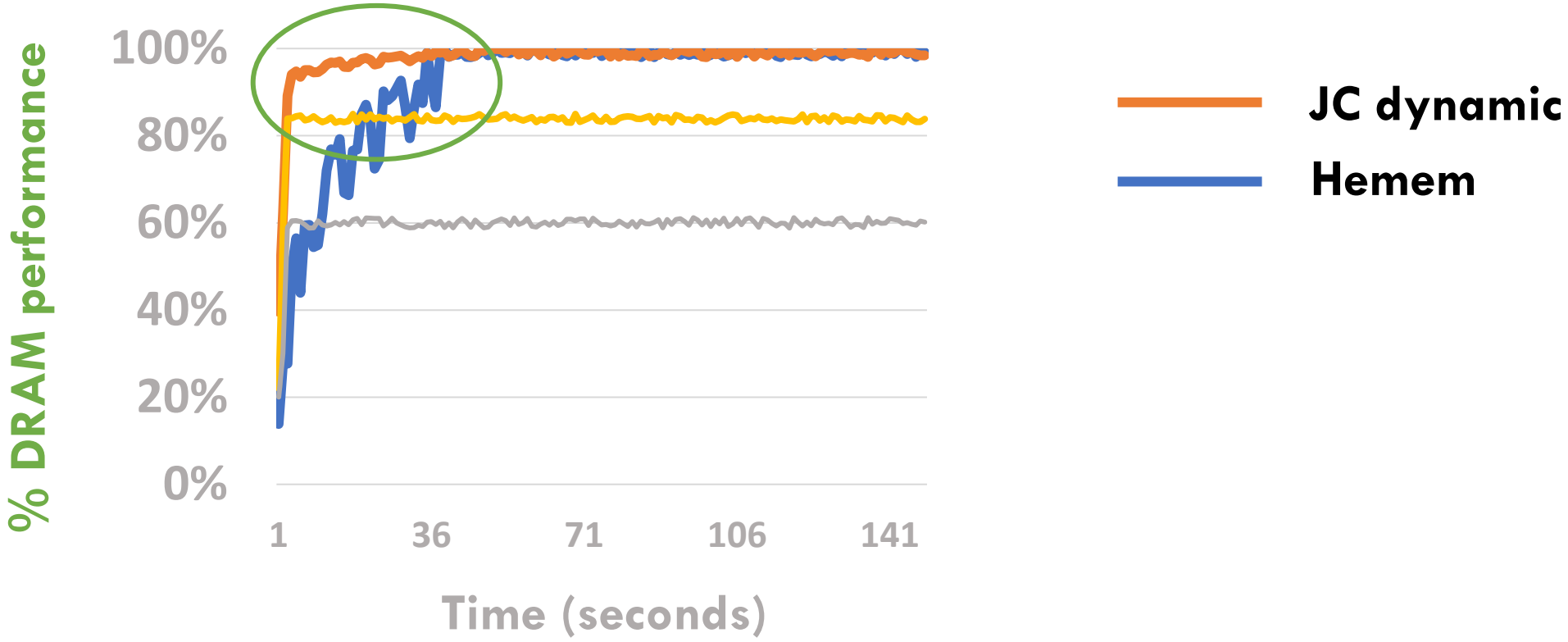


Hemem: 50% of the hot data is misplaced

JC: conflicts are unlikely

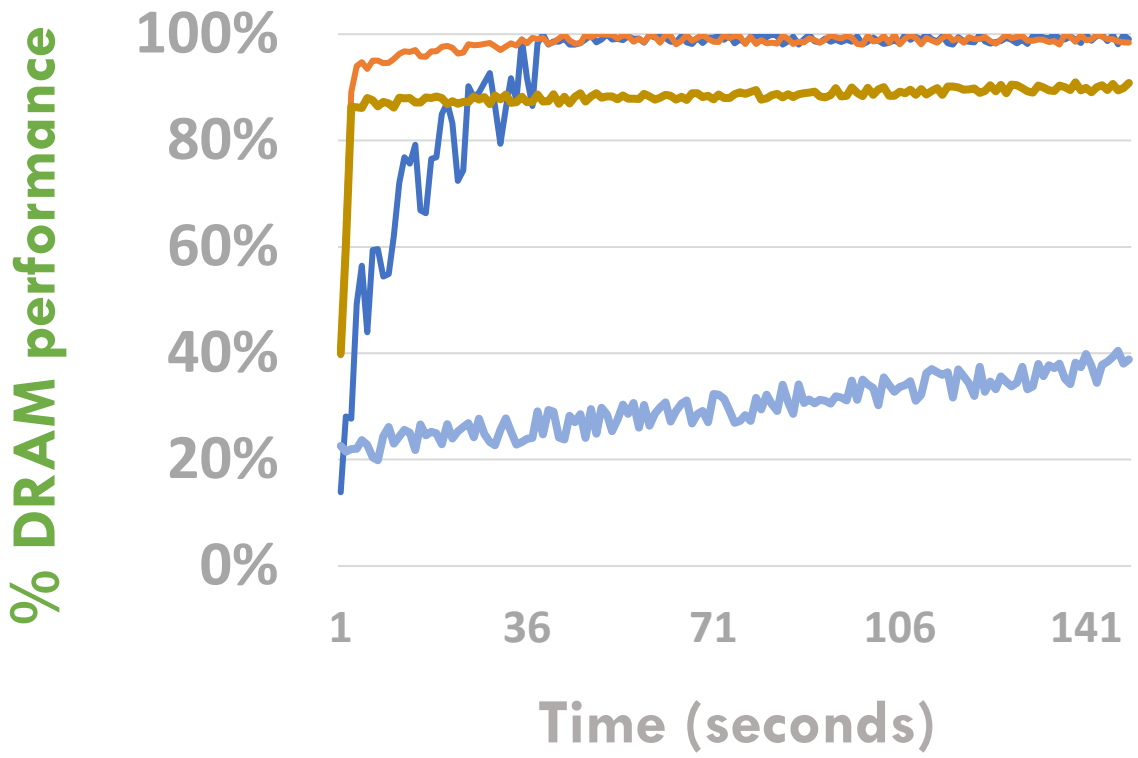
➔ JC migrates less pages to “fix” the situation

Evaluation: GUPS (16 threads)



Conclusion #2: Fixing conflicts is cheaper **than manual page migrations.**

Evaluation: GUPS (8 vs 16 threads)



JC dynamic 16 GUPS threads

JC dynamic 8 GUPS threads

Hemem 16 GUPS threads

Hemem 8 GUPS threads

Evaluation: GUPS (8 vs 16 threads)

Detecting hot pages:

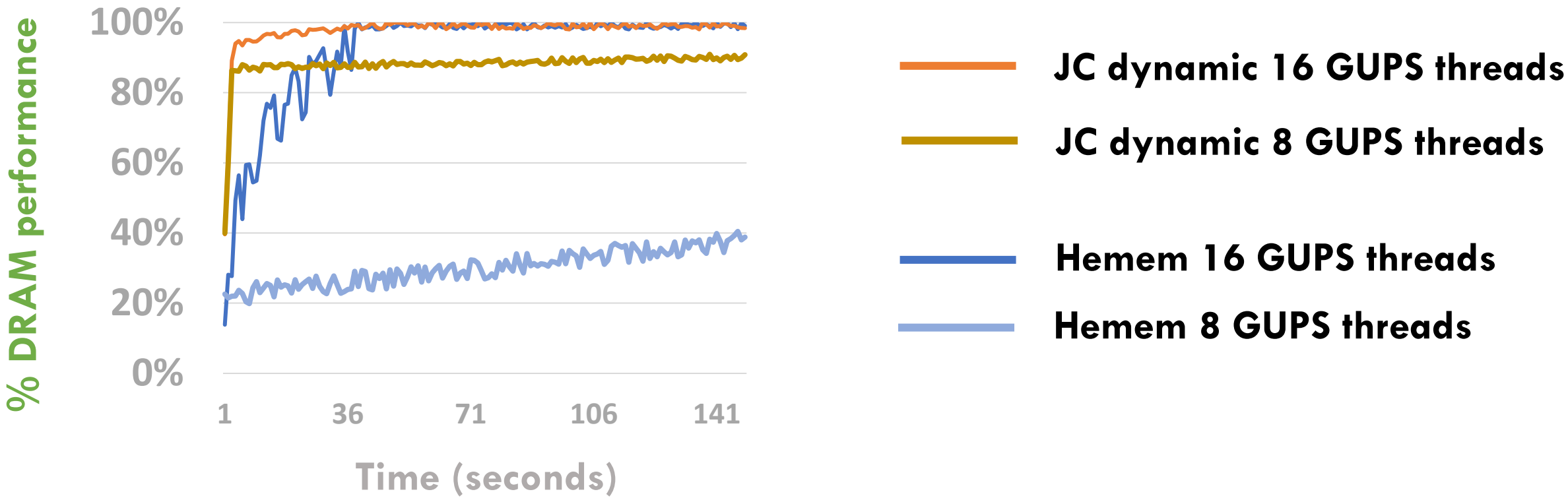
1/ Sample memory accesses

Number accesses > threshold
→ Pages marked as hot

2/ Periodically migrate hot pages and reset counters

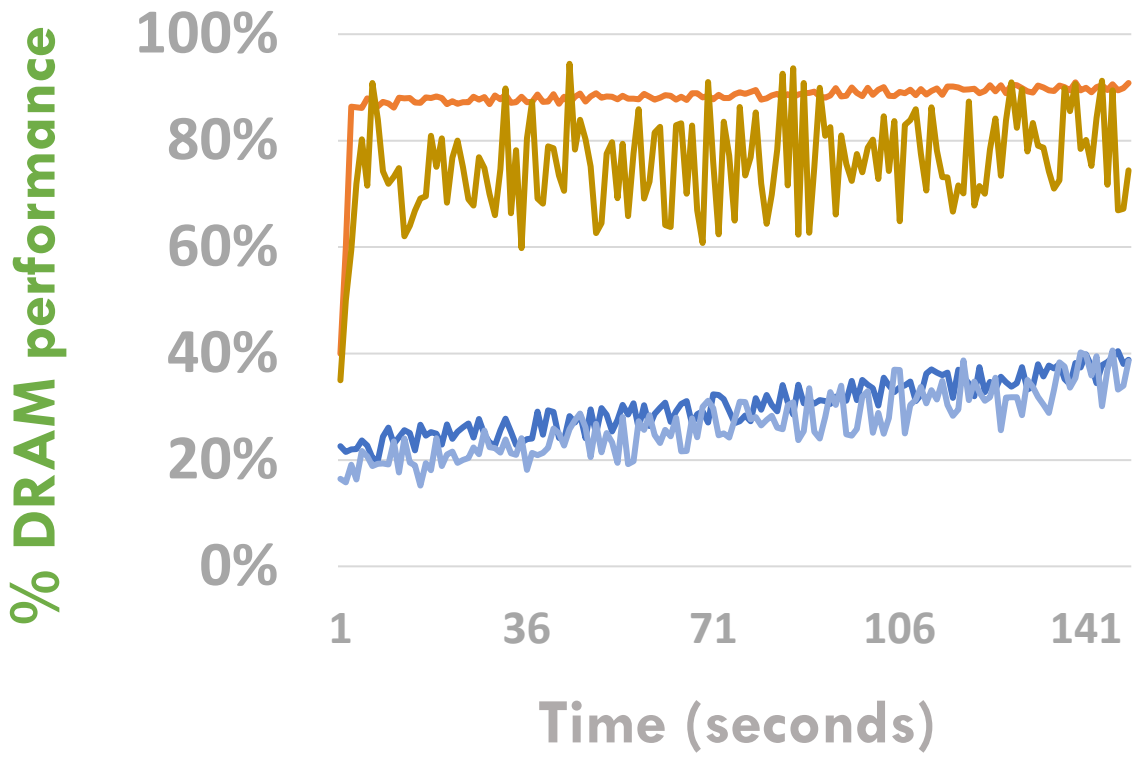
With few threads, not enough samples are generated, and pages are never hot!

Evaluation: GUPS (8 vs 16 threads)



Conclusion #3: Page migration is highly dependent on the detection of hot pages

Evaluation: dedicating cores to sampling?



JC dynamic - dedicated cores

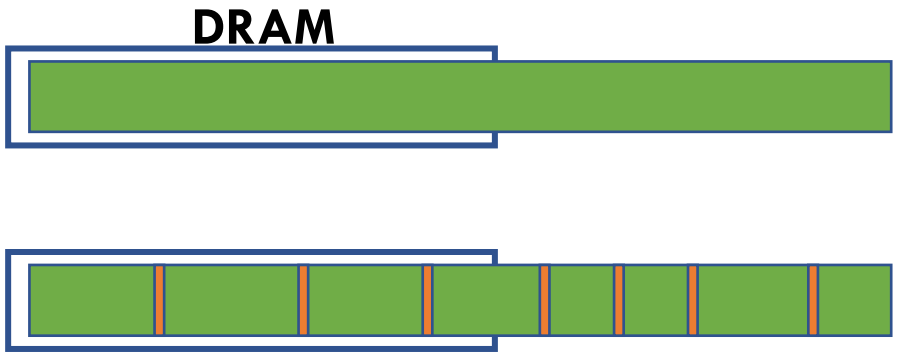
JC dynamic - interferences

Hemem - dedicated cores

Hemem - interferences

Conclusion #4: interferences when the sampling threads do not run on dedicated cores

Evaluation: GUPS++



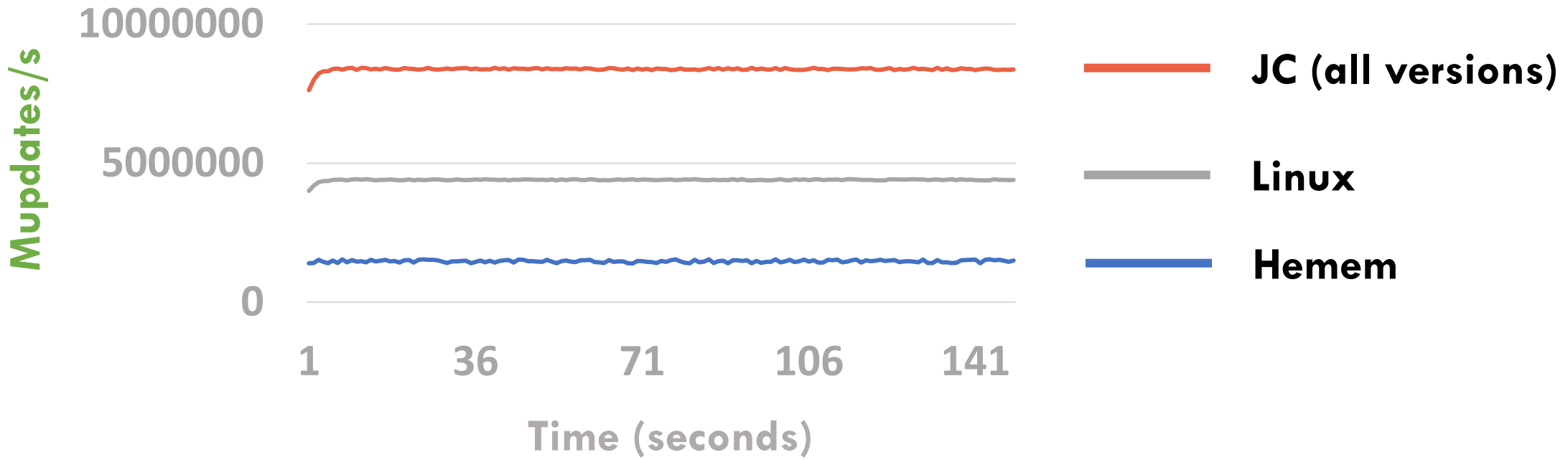
Array 2x the DRAM size

Random writes on 10 % of the ~~pages~~ items

Many pages now contain hot items

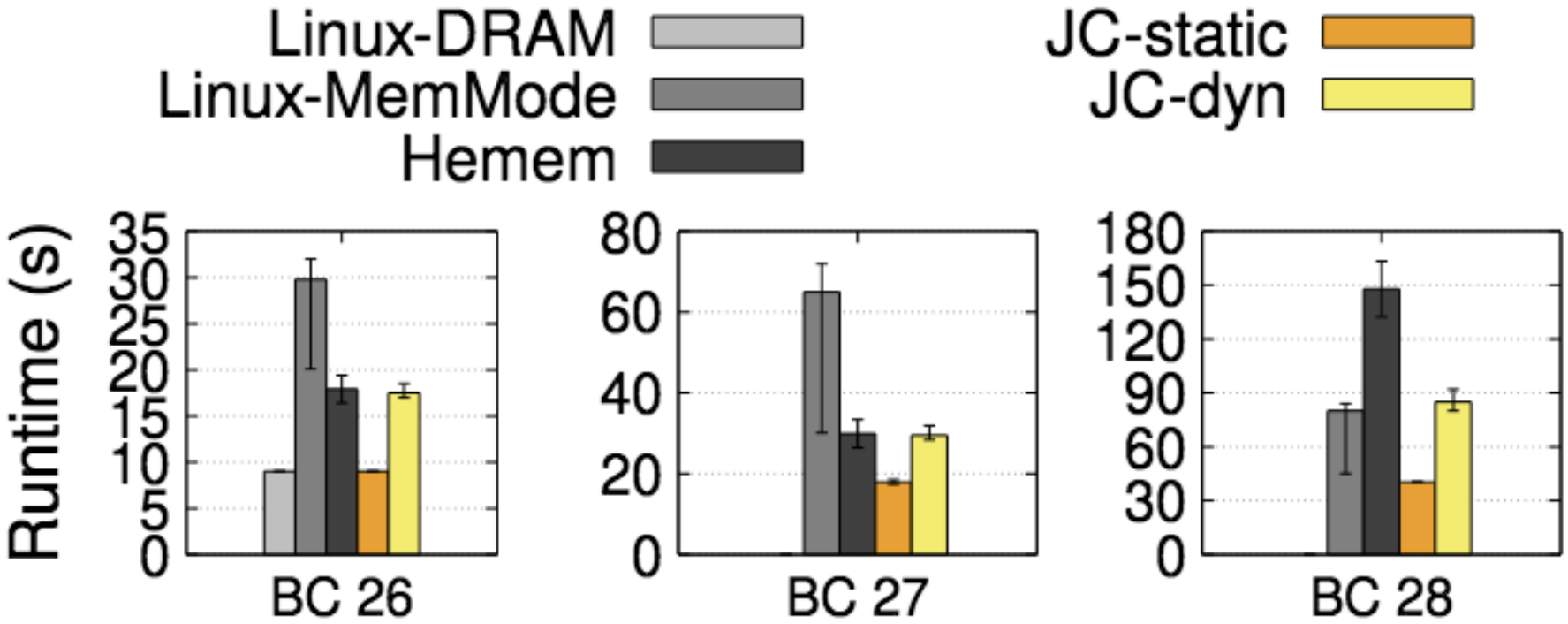
It is not possible to fit all hot pages in DRAM.

Evaluation: GUPS++



Conclusion #5: caches are also good with small items

Evaluation: HPC applications

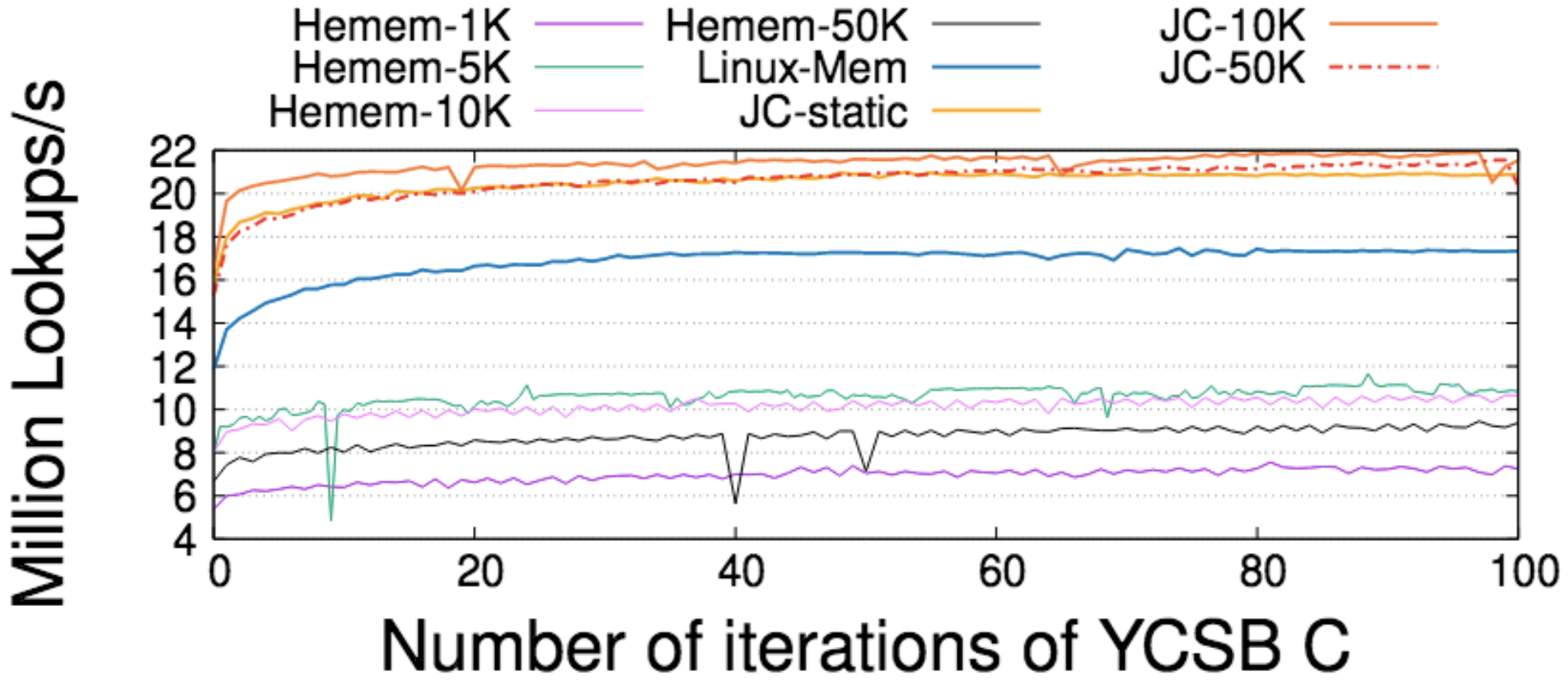


BC allocates large arrays, 1 is hot

➔JC static: the array does not conflict with itself

➔ Hemem & JC dynamic: overhead of sampling

Evaluation: key-value stores



Sampling struggles to find hot pages

Caches are less impacted than manual migrations

Evaluation

5.4.3 Summary

Hardware caches outperform page-based migrations when working with scattered small items. Again, caches tend to "work well" when conflicts are minimized at allocation time and, their performance is not strongly dependent on monitoring memory accesses to find and fix possible conflicts.

5.5 Silo

Silo is configured to execute a TPC-C workload on a 100GB database. The TPC-C workload is heavily skewed: most of the TPC-C data consists of the description of (sold) items, but most of the memory accesses are done on the customer and warehouse metadata. Figure 7 summarizes the performance of Silo, varying the number of threads.

Due to the order of initialization of the database, most of the hot working set used by Silo is allocated at the beginning of the execution. JC is able to allocate hot pages in a non-conflicting way, and Hemem allocates most of the hot pages in DRAM. Both JC and Hemem perform equally well on this workload.

Figure 7: SILO (TPCC) throughput (higher is better)

5.6 Limitations

Figure 8 presents the performance of the NAS benchmark suite running with Linux, Hemem and JC (we only included applications that executed in less than 24h on our machine).

Most HPC applications follow the same pattern as BC: large arrays are allocated and initialized at the beginning of the application, and then only a subset of the arrays are used during the execution of the algorithm. When the hot arrays fit in the DRAM cache, JC and Linux outperform Hemem by up to 2.8× (class D size of the NAS benchmark, on the left of Figure 8). As BC, the NAS applications use OpenMP to parallelize their computation, and the profiling and migration threads of JC-dyn and Hemem have cascading effects on the performance of threads waiting in barriers.

On MG.E, Hemem runs 1.8× faster than JC, despite its profiling overhead. MG uniformly accesses a large array and does not benefit from DRAM caching: most of the cached data is evicted before being reused. In general, uniform accesses over large datasets, or streaming patterns tend to be poorly

Figure 8: NAS application runtime (lower is better). JC outperforms Linux and Hemem except when the hot set size vastly exceeds the cache size (CG.E, MG.E).

cached. In the worst case scenario, with DRAM caching, 100% of the memory accesses end up in PMEM – for instance, in large streaming workloads, the streamed content keeps replacing itself in the cache, and data is always evicted before being re-read. With software-base migration, some of the data is allocated in DRAM and some of the memory accesses are resolved in DRAM.

Such patterns are hard to address at the kernel level because no data is hot, and no conflict is particularly worthy of fixing. However, caching could be improved at the hardware level. CPUs already implement special cases for streaming workloads in their CPU caches: most recent CPUs implement QLRU, in which data that was cached by a streaming thread is evicted before data cached by threads performing random accesses [7]. Such a strategy could be implemented in a DRAM cache as well, for instance by avoiding caching large streams. The performance of the DRAM cache could also be improved by optimistically flushing dirty data to PMEM, when the PMEM is idle, to reduce the latency of future evictions of dirty data.

Summary When an application does not have a clear hot dataset, but rather streams or accesses large datasets that do not fit in the DRAM cache, DRAM caches are not as good as software migration.

6 Discussion

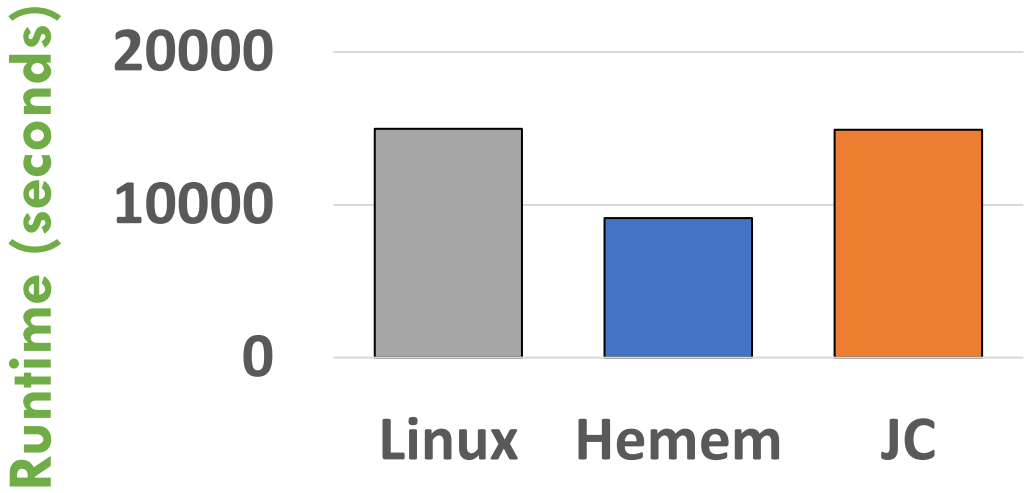
Recommendations From our experience, working with hardware caches and page migration systems, no solution fits all workloads, but the general rule of thumb is:

- Systems that rely on monitoring memory accesses are finicky to configure and can introduce huge performance overheads if not properly fine-tuned. In our experience, it is more likely for a migration daemon to be misconfigured than to perform well. This observation is not unique to this paper nor to the monitoring done by JC-dyn and Hemem. For instance, by default, most Linux distributions deactivate AutoNUMA, the page migration daemon of Linux because it negatively impacts most workloads. So, unless working with a known and predictable

11

More in the paper!
(To appear OSDI'23.)

Limitations – MG.E (NAS benchmark)

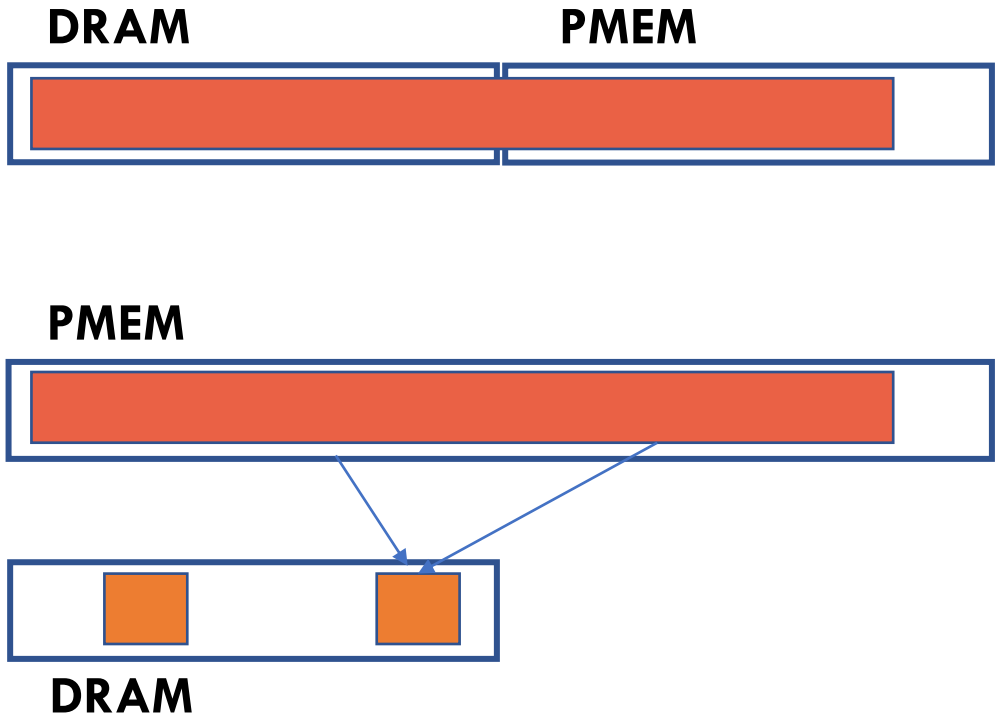


Write intensive streaming application:

- **Working set does not fit in the cache**
- **No spatial locality**

→ Caching is counter-productive.

Evaluation: GUPS (16 threads)



Hemem: 50% of the accesses in PMEM

JC: conflicts on 100% of the items

➔ Possible solutions at the hardware level
(E.g., CPU caches sometimes do not cache streaming patterns!)

Conclusion


It is possible to minimize conflicts at page allocation time


It is possible to minimize conflicts dynamically


Manual migrations are heavily dependent on sampling


Caches work well by default

Fixing conflicts is less expensive than manual migrations

Questions?