Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems)

Baptiste Lepers, *Université de Neuchâtel;* Willy Zwaenepoel, *University of Sydney*

https://www.usenix.org/conference/osdi23/presentation/lepers

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems)

Baptiste Lepers
University of Neuchâtel

Willy Zwaenepoel
University of Sydney

Abstract

We demonstrate that hardware management of a tiered memory system offers better performance for many applications than current methods of software management. Hardware management treats the fast tier as a cache on the slower tier. The advantages are that caching can be done at cache line granularity and that data appears in fast memory as soon as it is accessed. The potential for cache conflicts has, however, led previous works to conclude these hardware methods generally perform poorly.

In this paper we show that low-overhead conflict avoidance techniques eliminate conflicts almost entirely and thereby address the above limitation. We explore two techniques. The static technique tries to avoid conflicts between pages at page allocation time. The dynamic technique relies on monitoring memory accesses to distinguish between hot and cold pages. It uses this information to avoid conflicts between hot pages, both at page allocation time and by dynamic remapping at runtime.

We have implemented these techniques in the Linux kernel on an Intel Optane machine in a system called Johnny Cache (JC). We use HPC applications, key-value stores and databases to compare JC to the default Linux tiered memory management implementation and to HeMem, a state-of-the-art software management approach.

Our measurements show that JC outperforms Linux and HeMem for most applications, in some cases by up to 5×. A surprising conclusion of this paper is that a cache can provide close-to-optimal performance by minimizing conflicts purely at page allocation time, without any access monitoring or dynamic page remapping.

1 Introduction

Tiered memory systems combine DRAM with a slower, but more abundant, storage tier (SSD, PMEM, CXL memory extension modules [8], ...). Most systems rely on a software daemon that monitors accesses to the data. Frequently accessed data is migrated to DRAM, while less frequently accessed data is migrated to the slower tier [1, 9, 11, 14, 20, 23, 25]. Tiered memory systems have also been implemented purely in hardware, using DRAM as an "L4" cache that sits between the CPU and the slower tier [13].

Previous work has argued that hardware implementations of tiered systems are inefficient because the hardware lacks a high-level view of the application requirements and because caching strategies have to be kept simple to be executed in hardware. For instance, in tiered DRAM+PMEM systems, software daemons have been shown to outperform the "memory mode" of Intel CPUs (in "memory mode", the CPU uses DRAM as a directly-mapped cache for PMEM) [20].

This paper is based on the observation that the previously mentioned limitations of hardware caching are not fundamental and can be addressed at the operating system level. In particular, we demonstrate that the poor performance observed in earlier hardware-based systems is due to cache conflicts resulting from Linux's page allocation policy, and that simple improvements to the page allocation policy can reduce cache conflicts with little or no overhead.

Linux's page allocation does not take into consideration the location of pages in hardware caches. As a consequence, Linux suffers from the birthday paradox: the DRAM cache is large, but many pages tend to map to a subset of the available cache locations. We propose the following simple static page allocation policy to reduce conflicts: we allocate a new page such that its physical address maps to a cache slot with the fewest pages currently mapped to it. For example, if we have a cache with 2 million slots and 4 million pages to be allocated, we allocate 2 pages to each slot. The static policy has no noticeable overhead, but it vastly reduces conflicts.

We also investigate a dynamic policy that takes into account the access frequency of pages, distinguishing between hot and cold pages. The dynamic policy allocates a new page to the cache slot with the lowest access frequency and reacts to workload changes by dynamically remapping pages when it detects conflicts. Surprisingly, we find that in many workloads the static policy already results in few conflicts, and the overheads of the dynamic policy offset the benefits of any further gains in conflict reduction.

We compare our conflict avoidance policies to software migration, as proposed by, among others, HeMem [20]. With software migration, access frequency is monitored as well, producing the same set of hot and cold pages and incurring the same monitoring overhead as our dynamic approach. Software migration, however, uses this information for an entirely different purpose, namely to migrate hot pages from slow to fast memory, and vice versa for cold pages, unlike our dynamic policy which uses it to reduce conflicts.

We have implemented the static and dynamic policies at the kernel level in a subsystem named Johnny Cache (JC), and we refer to these systems as JC-static and JC-dyn, respectively. We have evaluated these systems on a tiered DRAM+PMEM system against the Linux page allocation mechanism and against HeMem, a state-of-the-art software-based page migration system [20]. JC outperforms Linux and HeMem for the vast majority of applications, in some cases by up to 5×. We document these results in more detail in the paper and also discuss the limitations of a cache-based approach. In addition, we find that JC-static often suffices to obtain good performance. Methods involving profiling such as HeMem and JC-dyn suffer from profiling and migration overheads and the inability to detect hot pages in some workloads. In contrast, avoiding conflicts in the DRAM cache at allocation time, as done by JC-static, is robust and sufficient to achieve near-optimal performance for most workloads.

In summary, the paper makes the following contributions:

- The observation that hardware-managed DRAM caches can be made efficient by minor modifications to the operating system page allocation algorithms.
- The idea of placing conflict avoidance as a first principle of page management in tiered memory systems, instead of relying on migration of data.
- The design, implementation and evaluation of page placement policies that outperform state-of-the-art page migration systems.

The rest of the paper is organized as follows. Section 2 explains how tiered-main memory systems are managed in software and in hardware. Section 3 presents the design of our policies, Section 4 presents their implementation, and Section 5 their evaluation. Section 6 provides further discussion of the strengths and weaknesses of various approaches. Section 7 presents related work and Section 8 concludes.

2 Tiered main memory systems

In this section, we give an overview of existing software- and hardware-managed tiered memory systems, and we compare their overheads.

2.1 Software-based migration

In software-managed tiered memory systems, the operating system chooses which pages are allocated in DRAM and which pages are allocated in the slower tier. The kernel usually allocates as many pages in DRAM as possible and, when DRAM is full, subsequent pages are allocated in the slow tier. A daemon is in charge of migrating frequently accessed pages (hot pages) from the slow tier to DRAM, and infrequently accessed pages (cold pages) from DRAM to the slow tier. The techniques vary but aim at inferring the set of hot pages with high accuracy and low overhead. For instance, HeMem [20] uses the hardware performance unit of Intel CPUs to track memory accesses and migrates pages between DRAM and PMEM using DMA to minimize CPU overheads.

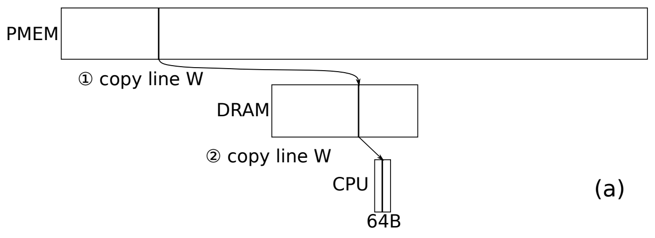
Software-based migration gives the operating system full control over page placement, but it comes with some downsides. First, data migrations are costly because they can only happen at page granularity (4KB or 2MB), and each migration requires modifying the page table, modifying the kernel VMA metadata and flushing the TLBs. Migrations may also cause latency spikes in write-heavy applications because pages have to be write-protected while being migrated. Second, since access frequency is collected on a per-page basis, for applications that mix hot and cold data in the same page, DRAM may need to be used for cold data to allow fast access to hot data in the same page. Finally, page migrations happen asynchronously: data may be accessed for a while in the slow tier before being migrated to DRAM. As a consequence, the performance of software-based migration is heavily dependent on the fast and accurate detection and migration of the working set. To do so, memory access must be sampled with high frequency, a costly proposition.

2.2 Hardware caching

In hardware, the CPU uses DRAM as a cache for the slow tier. In existing implementations [13], the DRAM is configured as a 1-way cache, indexed by physical address. Unlike software-based approaches, hardware caches are synchronous: all accessed data is cached in DRAM. In this section, we describe the implementation of the "memory mode" of Intel processors for tiered DRAM+PMEM systems.

When looking for a physical address W, the memory controller first checks if W is in the DRAM cache (at location "W mod *cachesize*"). If W is not present in DRAM, W is fetched from PMEM, copied to the DRAM cache and to the CPU cache (see Figure 1(a)).

A conflict occurs when W maps to a cache slot that is already occupied by X, in which case X must first be removed from the cache. In the best case, X is clean, and the cost is equal to that of a PMEM read. If X is dirty, then X must be written back to PMEM before W can be loaded in the cache, making the cost the sum of a PMEM write and a PMEM



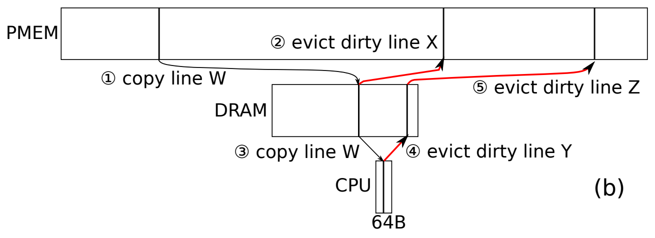


Figure 1: Caching in memory mode. (a) In the best case data found in PMEM is cached in DRAM and in the CPU caches, and in the (b) worst case the caching may result in evictions, causing up to two writebacks to PMEM.

read. A worst-case scenario can arise as described in Figure 1(b). In addition to the writeback of X, a dirty line Y may be evicted from the CPU cache, resulting in a writeback of Y to the DRAM cache, which itself may result in a writeback of another dirty line Z to PMEM (the DRAM cache is not inclusive).

Regardless of the precise sequence of events, conflicts are expensive. Table 1 compares the latency of performing 8B random reads or random writes in DRAM, in PMEM, and in the worst-case scenario presented in Figure 1(b). Reading from PMEM (in AppDirect mode) is 3.2× slower than reading from DRAM, and writing is 4.4× slower. A read causing two writebacks to PMEM is 9.7× slower than a read from DRAM, and a write causing two writebacks is 7.2× slower than a write to DRAM. (Causing two writebacks to PMEM is not exactly equivalent to performing two writes to PMEM, which would be 8.8× slower, because the CPU overlaps the evictions with other processing done by the application, resulting in slightly more in-CPU parallelism).

Memory mode thus performs suboptimally when frequently accessed data conflicts in the cache. If, however, conflicts can be avoided, then memory mode offers several advantages over software migration. First, caching avoids costly whole-page migration as well as virtual memory operations. Second, caches operate at the cache line level, while software migration can only migrate data at the page granularity. Therefore, they avoid wasting DRAM space if hot and cold data are located in the same page. Finally, caching is synchronous: hot data appears in DRAM on the first access.

Read in DRAM	96ns
Plain read from PMEM	305ns
Write in DRAM	130ns
Plain write from PMEM	578ns
Read/Write causing 2 writebacks	938ns

Table 1: Latency of memory access in various scenarios.

2.3 Comparison

Table 2 summarizes the costs of migrating data vs. caching data. These costs show that caching data in DRAM is *a priori* a more parsimonious solution: data is cached at cache line granularity (vs. page granularity), caching data requires no kernel metadata updates, and no memory profiling is necessary to infer which pages to cache.

3 Design

Our design is based on the idea that a DRAM cache is efficient, as long as conflicts in the cache are rare. Conflicts happen when two data items are mapped to the same cache location. Conflicts become problematic if the data items are accessed in turn. We have designed two policies that aim at minimizing conflicts in the cache. The hardware caches data at cache line granularity, but the kernel can only allocate data at page granularity, so our policies try to minimize conflicts between pages.

Static policy: The static policy minimizes the number of allocated pages that map to the same DRAM cache location. Assuming a DRAM cache that can store D pages, the static policy allocates the first D pages so that they map to different cache locations. The next D pages are allocated so that they possibly conflict with a single other page, and so on.

Dynamic policy: The dynamic policy samples memory accesses to compute the heat of every page and every cache location. When a new page is allocated, the kernel maps it to the coldest available location.

A conflict avoidance daemon monitors for conflicts between hot pages at the same cache location. When two pages, mapped to the same DRAM cache location, are both frequently accessed, one of the pages is remapped to a different cache location.

Rationale: The main advantage of the static policy is that it requires no monitoring of memory accesses, and so it runs with no overhead. The intuition behind the static policy is that minimizing the number of pages that overlap in the cache reduces significantly the likelihood of conflicts between hot data items. Indeed, in most workloads, at most of a few GBs of data is hot, even in workloads whose memory footprint vastly exceeds the available DRAM. Minimizing overlaps makes conflicts between hot pages unlikely. For instance, let's consider an application that allocates data twice the available DRAM size, 5% of which is hot. The static policy allocates

	Software migration	Hardware caching
Granularity	Page Size (4KB, 2MB)	64B
Cost of migrating/caching	Page copy from PMEM to DRAM Page copy from DRAM to PMEM 2 page table updates 2 VMA updates TLB flush	*Best case* 1 cache line copied from PMEM to DRAM *Worst case* 1 cache line copied from PMEM to DRAM 2 cache lines copied from DRAM to PMEM
Strategy	Software defined	All memory accesses are cached in DRAM

Table 2: Comparison of the cost of migration vs. caching.

pages such that exactly 2 pages map to each cache location. A given hot page has a 5% chance to compete for the cache with another hot page, and a 95% chance to compete with a cold page. In other words, most hot pages are "paired" with a cold page, and are thus unlikely to be evicted frequently from the DRAM cache.

The dynamic policy makes more informed choices at page allocation time, and the daemon fixes conflicts that may have been missed at allocation time. The dynamic policy borrows the notion of heat from software migration systems, but the heat is used to track and *avoid conflicts* between hot pages rather than to migrate hot pages to DRAM. We demonstrate in Section 5.2 that, in the general case, avoiding conflicts is less costly than migrating hot data to DRAM.

4 Implementation

In this section, we describe the implementation of the page allocation policies and the migration daemon. The code is available at `https://github.com/BLepers/JohnnyCache`.

4.1 Page initialization and associated metadata

Our policies are implemented in the kernel, as hooks in the kernel initialization function, the page initialization function, the page fault handler and the page unmap handler. To ease the development of policies, we implemented a framework that contains the logic common to the policies. In the remainder of the paper, we refer to the framework as Johnny Cache (*J*C).

In this paper, we assume a directly-mapped 1-way cache, in which data is cached at its physical address modulo the size of the cache – as implemented by Intel in the "memory mode" of tiered DRAM+PMEM systems. It would be easy to account for associativity in JC by changing the definition of a conflict: currently, a conflict involves 2 or more pages; in an N-way cache, a conflict would involve N+1 or more pages.

While DRAM caches data at cache line granularity, the kernel can only allocate and migrate data at page granularity. All the metadata maintained by JC are thus at the page level. When the kernel boots, we query the processor's memory controller to find out the size of the DRAM cache. In the remainder of this section, we refer to the maximum number of pages that the cache can hold as the cache *capacity*. Because the cache is directly-mapped, every page in the system maps to a unique index in the cache, which we call a *bin*. The bin of a page is its page frame number (physical address of the first byte of the page / size of a page) modulo the capacity of the cache. Furthermore, each bin of the cache has a *heat*. The definition of heat depends on the policy. For instance, for the static policy it corresponds to the number of allocated pages that map to that bin.

As is the case with the default page allocation policy of Linux, we use a lazy page allocation mechanism: pages are physically allocated only when they are first accessed. We thus hook the page fault handler to implement our page placement policies. The framework maintains a list of bins with available pages, sorted per heat. When a page fault occurs, a page from a bin with the lowest heat is returned, and the current allocation policy is informed of the page fault. Similarly, whenever a page is freed, the kernel unmap handler is called, and the current allocation policy is made aware of the unmapping.

Listing 1 summarizes the metadata and code of the page fault hook used by our framework. The overhead of keeping the metadata in memory is small (less than 50MB for a system with 128GB of DRAM and 1TB of PMEM).

Our policies are implemented at the kernel level and oblivious to the notion of a thread or an application. The policies try to minimize conflicts across the entire machine, and no partitioning of the cache is done (unlike page-coloring approaches). A major benefit of this approach is that conflicts are minimized globally. For instance, the conflict avoidance daemon remaps hot conflicting pages even if they belong to different applications.

4.2 Static policy

The static policy allocates a new page in a bin with the fewest allocated pages. The static policy consists of counting the number of allocated pages that map to a given cache bin. The policy is called on every page fault and page unmap by the framework. Listing 2 summarizes the code of the static policy. During a page fault, the policy increments by one the *heat* of the bin of the newly allocated page. Because the page fault handler of the common framework allocates a page from the bins which have the lowest heat, subsequent page faults will

Listing 1 JC framework.

```
// struct for each cache bin (page granularity)
struct bin* bins[CACHE_CAPACITY] = { ... };
// avail[heat] = list of bins with free pages
struct bin* avail_bins[HEAT_LEVELS];
// full[heat] = list of fully allocated bins
struct bin* full_bins[HEAT_LEVELS];

struct page* page_fault_handler(void) {
  for(i = 0; i < HEAT_LEVELS; i++) {
    bin = list_first(avail_bins[i]);
    if(bin) {
      struct page *p =
                 list_pop(bin->avail_pages);
      current_policy.page_fault(bin, p);
      if(list_empty(bin->avail_pages))
        list_move(bin, full_bins[bin->heat]);
      else
        list_move(bin, avail_bins[bin->heat]);
      return p;
    }
  }
  // OOM
}

void page_unmap(struct page *p) { ... }
```

likely avoid that bin. When a page is unmapped, the heat of the bin is decremented by one, increasing the likelihood of that bin being chosen for subsequent allocations.

Listing 2 In the static policy, the heat of a bin corresponds to the number of pages allocated to that bin.

```
void static_pf(struct bin *b, struct page *p) {
  b->heat++;
}

static_policy = {
  .page_fault = static_pf, .unmap = ...
};
```

4.3 Dynamic policy and migration daemon

The dynamic policy allocates a page in a bin with available pages and with the lowest heat. The dynamic policy monitors memory accesses to infer the heat of each page and bin. We monitor read accesses to the DRAM cache, read accesses to PMEM and all stores using the Processor Event-Based Sampling (PEBS) feature of Intel's CPUs. When a memory access is sampled, we increase the heat of the accessed page and accessed bin. We also artificially increase the heat of the bin of newly allocated pages to avoid multiple pages being mapped to the same bin during bursts of allocations. To avoid heat continually increasing over time, we trigger page cooling as soon as a page becomes "super hot", i.e., when its heat becomes double that of the threshold to detect a hot page.

In theory, our dynamic policy could monitor conflicts in the cache instead of monitoring memory accesses, but no such event exists in Intel CPUs. Our heat detection and cooling approaches are identical to those used by HeMem [20], which allows for a fair comparison between software-based migration and conflict-avoidance (both solutions use the same PEBS events, the same definition of heat and the same cooling function).

The migration daemon monitors conflicts between allocated pages. When two hot pages are present in the same bin, one of them is remapped to a physical location in a different bin. The daemon periodically looks for pages in the upper heat buckets and remaps them. The remapping operation calls the page fault handler which allocates a new page in a cold bin and calls the unmap function, which decreases the heat of the original bin. Algorithm 3 summarizes the approach of the dynamic policy and migration daemon.

HeMem triggers migrations as soon as a hot page is detected. To allow for a fair comparison, we trigger the daemon as soon as we detect a bin containing two hot pages (i.e., as soon as we detect a conflict that involves two hot pages).

Listing 3 The dynamic policy and migration daemon that remaps pages from highly accessed bins.

```
// Migration daemon
void migration_daemon() {
  wait();
  for(i=HEAT_LEVELS-1; i>MIN_CONTENTION; i--) {
    foreach(bin, avail_bins[i]) {
      if(bin->nb_hot_pages >= 2)
        remap(get_hot_page(bin));
    }
    foreach(bin, full_bins[i]) {
      if(bin->nb_hot_pages >= 2)
        remap(get_hot_page(bin));
    }
  }
}

// Called on every sampled memory access
void add_sample(struct bin *b, struct page *p){
  b->heat++;
  ... // increase the page's heat &
  ... // update the metadata
  if(b->nb_hot_pages > 2)
    migration_daemon.wakeup();
}

void dyn_pf(struct bin *b, struct page *p) {
  b->heat++;
  ... // increase the page's heat &
  ... // update the metadata
}

dynamic_policy = {
  .page_fault = dyn_pf,
  .unmap = ...
};
```

5 Evaluation

The evaluation aims at answering the following questions:

- What is the performance of JC compared to state-of-the-art tiered memory management systems?
- What is the overhead of JC, in terms of performance and latency spikes, compared to other systems?
- What are the limitations of JC? Which applications benefit from hardware caches, and which benefit from page migration?

We show that the static page allocation policy of JC achieves close-to-optimal performance in many applications, and sometimes outperforms the dynamic policy (and related work) when minimizing CPU overhead is essential for performance. The surprising conclusion of this evaluation is thus that hardware caches often outperform existing software-based migration strategies, provided minimal changes in the kernel page allocation policy are put in place.

5.1 Setup

Hardware configuration. All the experiments presented in this paper are run on a two-node NUMA machine, with 40 Intel Xeon Gold 6230 cores running at 2.10GHz (20 cores per NUMA node), 128GB of DRAM, and 8*128GB Intel Optane NV-DIMMs (64GB DRAM and 512GB PMEM per NUMA node). In memory mode, each NUMA node has a DRAM cache of 48GB (16GB is used by the CPU for the cache metadata).

Workloads. We borrow the workloads used to evaluate HeMem [20], a state-of-the-art software page migration system. The GUPS microbenchmark allocates a large array, zeroes it, and then threads perform updates to a random subset of 8-byte array elements. BC, from the GAP benchmark suite, computes the betweenness centrality algorithm on a powerlaw graph [4]. Silo [22] is an in-memory database running the standard TPC-C benchmark suite. Finally, Masstree [16] is an in-memory key-value store, running a YCSB workload[1]. We also present results from the NAS benchmark suite [3]. JC equals or more commonly outperforms the related work in all these benchmarks, with the exception of the MG.E application from the NAS benchmark suite, which allows us to demonstrate the limitations of our approach. As in the original HeMem evaluation, NUMA effects of tiered memory are beyond the scope of this paper, and we run the applications on a single NUMA node.

Software configuration. We compare JC against unmodified Linux and HeMem. *Linux* uses the machine in memory mode with the default Linux page allocation policy. With the default page allocation policy, pages are allocated on the local NUMA node, but contiguous virtual memory ranges may end up fragmented in physical memory. Any array larger than 2 pages may thus conflict with itself in the DRAM cache (the larger the array, the more likely 2 pages of the array conflict). We refer to *JC-static* as the machine in memory mode with our static page allocation policy and to *JC-dyn* as the machine in memory mode with the dynamic page allocation policy plus the page remapping daemon. We benchmark HeMem using the provided artifact [19].

[1]HeMem benchmarked FlexKVS [17], an in-memory key-value store which we could not evaluate due to the lack of an RDMA network card on our server.

5.2 GUPS

The GUPS microbenchmark, from HeMem [20], allocates a large array, a subset of which is hot. 90% of the updates are done on the hot section of the array, and 10% on the cold section. We configure the array to be 96GB, twice the DRAM cache size and measure performance when 10% of the array is hot (9.6GB).

The performance of HeMem and JC-dyn is dependent on their ability to detect hot and cold pages. Intuitively, the more threads perform memory accesses, the easier it is to detect hot pages. To assess the impact of the workload on the detection of hot and cold pages, we thus vary the number of threads. Furthermore, HeMem and JC-dyn use two separate threads to sample memory accesses and to migrate pages. To assess the overhead of these threads, we either run them on separate cores or on the cores used by GUPS. We refer to these configurations as (M+N) where M is the number of cores used by GUPS, and N is either 0 or 2 and reflects the number of cores dedicated to monitoring and migration in HeMem and JC-dyn. We use three such configurations: (16+2), (8+2) and (8+0). These results are presented in Section 5.2.1.

In the original GUPS implementation the hot and cold data items are located in separate regions of the array. While this microbenchmark reflects the partitioning done by some applications, it implies that all hot items are located in a small number of pages, and all cold items are located in the other pages. To reflect the behavior of applications that do not partition their hot and cold items in this manner, we also run GUPS with hot items scattered randomly in the array. These results are presented in Section 5.2.2. Finally, in Section 5.2.3 we explore the performance of the different systems with a larger data set of 480GB.

We measure the throughput achieved for a given combination of system and workload. When the hot data fits in DRAM, we present the results as the percentage of the throughput achieved when all hot data is manually allocated in DRAM. Otherwise, we present the result in terms of millions of updates per second.

5.2.1 Random updates to clustered hot values

We first benchmark GUPS with hot values clustered on a few pages, the scenario favoring page migration.

The allocated array is twice the size of the DRAM cache so, initially, half of the array is in DRAM, and all systems start with low throughput. Figure 2 presents the performance of JC-static, JC-dyn, HeMem, and Linux over time, as a percentage of the performance achieved when all hot pages are manually allocated in DRAM.

Steady-state performance in configuration (16+2): Both HeMem and JC-dyn achieve 100% of DRAM performance, JC-static 85% and Linux 60%. There is (conflict-free) space in the cache for all hot pages, so JC-dyn eventually eliminates all conflicts, and HeMem eventually moves all hot pages to DRAM, explaining their performance being equal to DRAM.

The good performance of JC-static is explained by the low number of conflicts on hot data. The array is twice the size of the cache, and JC allocates exactly 2 pages per cache bin. Let P be a hot page. P conflicts with a single other page Q, and Q only has a 10% probability of being hot. JC thus only suffers from conflicts between 10% of the hot data (0.96GB).

The low performance of Linux is explained by the large number of conflicts when no care is taken to properly spread the pages in the cache. Just as with the "birthday paradox", even though a year has many days (*even though the cache has many bins*), in a small group of people, many are likely born on the same day (*many pages are mapped to the same cache bin, even when allocating only 100GB*). On average, Linux uses only 32GB of the DRAM cache because the allocated pages map to a subset of the available cache bins, while JC takes advantage of the full cache. Figure 2(d) presents the performance of an average run of Linux, but, depending on page placement, performance varies between runs from 20% to 80% of DRAM performance. These extreme values are rare, with most runs achieving around 60%.

Steady-state performance in configuration (8+2) - The difficult configuration of heat detection systems: The performance of JC-static and Linux relative to DRAM performance remains the same as in configuration (16+2). HeMem does not reach steady state even after 2 minutes, only reaching 40% of DRAM speed. JC-static is between 4× faster than HeMem (at the beginning of the execution) and 2.2× faster (after 2 minutes of execution). JC-dyn also does not reach steady state, but its performance is closer to DRAM performance (85% at the beginning of the execution, 90% at the end).

With 8 threads, fewer samples are generated, and the cooling mechanisms of HeMem and JC-dyn reduce the heat of pages faster than it increases as a result of accesses. HeMem and JC-dyn trigger page cooling as soon as any given page becomes "super hot", i.e., when its heat becomes double the threshold to detect a hot page (see Section 4). We implemented other cooling algorithms, but none ended up working

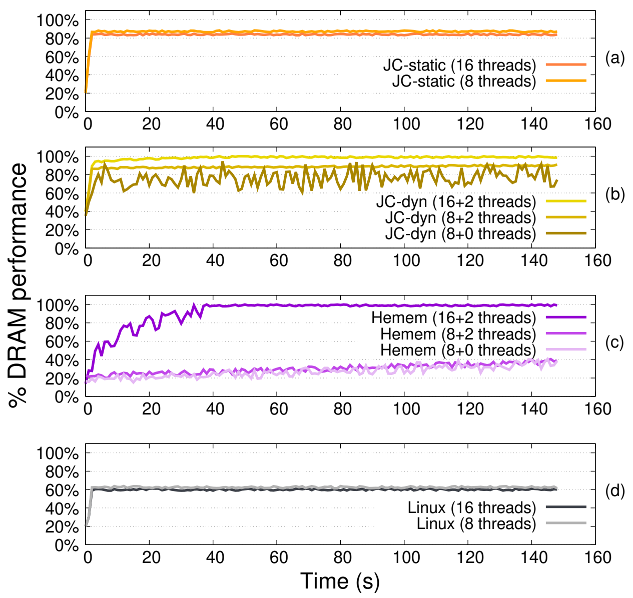


Figure 2: Hot values clustered in the array. Performance achieved by JC, HeMem, and Linux compared to the performance achieved when all the hot values are manually allocated in DRAM (optimal page placement). (a) JC-static, which does not use any profiling, performs close-to-optimally. (b) The profiling threads of the migration daemon can negatively interfere with the application threads (brown line). (c) The performance of HeMem is highly dependent on its ability to detect hot pages. When GUPS is launched with a low number of threads, hot pages are rarely detected and HeMem performs suboptimally. (d) The default page placement policy of Linux performs suboptimally because of conflicts.

in all configurations of GUPS. For instance, forcing cooling to happen periodically but less frequently results in most of the array being detected as hot in some other configurations (e.g., on smaller arrays). Replacing periodic cooling with other algorithms such as LRU also performs poorly in some configurations (e.g., when the hot set size exceeds the DRAM size). Most of the related work on page migration explores new ways of measuring heat accurately and with low overhead [1, 9, 11, 12, 14, 20, 23–25] but, in our experience, these heuristics require fine-tuning for each application and configuration.

It is possible to tune the sampling rate of memory accesses to gather more samples in a given amount of time, but doing so is also fraught with problems. For instance, doubling the sampling rate actually decreases the performance of GUPS running with 16 threads by 20%, due to profiling overheads. Some literature describes attempts to use dynamic sampling rates, but these algorithms also need to be precisely tuned for each machine or workload (e.g., to detect pages that need to be migrated between NUMA nodes, Carrefour [11] adjusts its sampling rate based on the workload, but all its parameters are fine-tuned for each machine).

The inability to detect most hot pages negatively impacts the performance of HeMem. In comparison, caches "work well", even without any heat detection or page migration. Once an item is updated, it is cached in DRAM. After a few seconds, most items have been updated, and the cache has reached its warmed-up state. JC-static and JC-dyn perform close-to-optimally without any fine-tuning, regardless of the number of GUPS threads. Just as HeMem, JC-dyn only manages to detect a subset of the (conflicting) hot pages. The partial detection of hot pages explains why JC-dyn is unable to reach optimal performance, but also explains why it performs slightly better than JC-static.

Performance over time in configuration (16+2) - Caches reach steady-state performance faster: As expected, the performance of JC-static and Linux remains constant after allocation is completed, since the number of conflicts remains the same throughout the execution. HeMem and JC-dyn require some time to reach maximum performance, in the case of JC-dyn to migrate pages to avoid conflicts, in the case of HeMem to migrate hot pages initially allocated in PMEM to DRAM and vice versa for cold pages. The time to reach this steady state performance is, however, much longer for HeMem than for JC-dyn, 38 seconds versus 2 seconds.

HeMem needs to sample memory accesses in order to perform informed migration decisions, and each migration consists in evicting a cold page to PMEM and promoting a hot page to DRAM. Migrations are thus inherently asynchronous and costly. In JC, once an item is updated, it is cached in DRAM. After a few seconds, most items have been updated, and the cache has reached its warmed-up state.

A surprising observation is that fixing conflicts requires fewer page migrations than migrating hot and cold pages. Indeed, before doing any page migration, only 0.96GB of the data conflicts, and these conflicts can be avoided by migrating 0.48GB of data. In HeMem, 4.8GB of the hot data is misplaced and needs to be brought to DRAM, which also causes 4.8GB of cold data to be migrated to PMEM. HeMem thus migrates 9.6GB (20× more data) to reach steady state performance.

Performance over time in configuration (8+0) - A background daemon can be counterproductive: JC-static and Linux do not use any profiling threads, and their performance is obviously the same as in configuration (8+2). The performance of both HeMem and JC-dyn becomes quite variable, and JC-dyn on average drops below JC-static in terms of performance. When the monitoring and migration threads execute on dedicated cores, JC-static and JC-dyn have similar performance, but, when they are scheduled on the same cores as the GUPS application, they have a non-negligible impact on performance. In that situation, JC-static outperforms JC-dyn by 10% on average and maintains a much more stable throughput over time.

5.2.2 Random updates to distributed hot values

In the previous experiment, all the hot items were clustered on the same pages, which is the best case scenario for page migration systems. However, hot items may not be clustered together in memory. To account for this behavior, we execute GUPS with hot items randomly scattered in the allocated array. As before, we allocate a 96GB array, 10% of which is hot. We execute GUPS with 16 threads, and dedicated cores for the profiling (16+2 configuration). Figure 3 presents the performance of HeMem, Linux and JC over time. JC is 4.5× faster than HeMem.

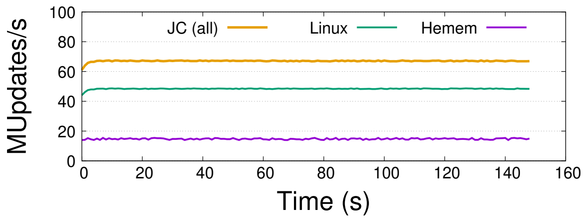


Figure 3: 16+2 threads, hot values distributed in the array and profiling running on dedicated cores. GUPS throughput over time when the hot set size is equal to 20% of the DRAM cache (higher is better).

In this experiment, because hot items are scattered in the array, most pages contain one or a few hot items. HeMem cannot bring all the hot pages in DRAM because the number of pages that contain hot items exceeds the number of pages that fit in DRAM. Interestingly, the hot data set does not need to be large for HeMem to be unable to migrate data to DRAM. Indeed, 1% of the data being hot (1GB) translates to 134 million 8-byte values, so all pages of the array likely contain many hot values (100GB is "only" 51K 2MB pages).

Just as HeMem, JC-dyn cannot perform any useful remappings, and as a result JC-static and JC-dyn perform equally well in this benchmark. Although hot items are scattered on all pages, since data is cached at the cache line granularity, the hot items rarely overlap in the cache. In this configuration of GUPS, JC reads on average 7× less data from PMEM than HeMem.

5.2.3 Performance on large datasets

In the previous experiments, GUPS was configured with a 96GB dataset (2× the cache size), 10% of which was hot. In this experiment, we configure GUPS to use 480GB (10× the cache size), the maximum workload size that fits on a single NUMA node, and we vary the percentage of hot data so that the hot data either fits in the cache or not. We run HeMem with dedicated cores for profiling and migration.

Because GUPS allocates all the available memory, JC's static page allocation policy and Linux have the same performance. Indeed, JC allocates pages in an order that minimizes conflicts but, in the end, both JC and Linux end up allocating all the pages of the system.

When the hot dataset is clustered and fits in the cache. We measure performance when 2% of the data is hot (9.6GB, same as in the smaller experiments). Figure 4 presents the performance of JC and HeMem compared to the performance obtained when the hot data is placed in DRAM.

As in the smaller dataset experiment, HeMem and JC's dynamic performance depends on their ability to detect hot pages. When GUPS is configured to run with 8 threads, HeMem does not reach steady state after 2 minutes of execution. In comparison, caches "work well", even without heat detection or page migration.

When GUPS is configured with 16 threads, both HeMem and JC-dyn eventually reach optimal performance. As seen earlier, caches reach optimal performance faster because avoiding conflicts requires fewer page migrations (1.7GB of the data initially conflicts in JC, 4.3GB of the data is initially misplaced in HeMem). The number of conflicting pages is higher in the 480GB experiment than in the 96GB experiments, even though the same number of pages are hot, because more pages map to the same slot. In the 96GB experiment, 2 pages map on a given slot, so a hot page has a 10% probability of conflicting with another hot page. In the 480GB experiment, a hot page conflicts with 9 other pages, each of which has a 2% probability of being hot, so it has a 16% probability of conflicting with another hot page. The higher number of conflicts explains why JC-static performs worse on bigger datasets than on smaller datasets: large datasets hinder the ability of JC-static to minimize conflicts at allocation time.

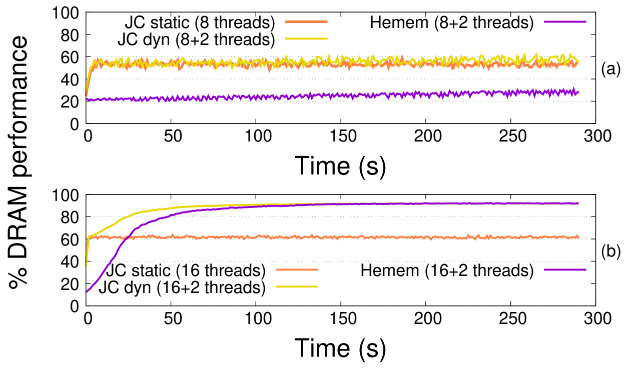


Figure 4: Hot values clustered in the array. 480GB dataset, 2% of which is hot (9.6GB). (a) With few threads, hot pages are rarely detected, and HeMem performs suboptimally. (b) With a large number of threads, JC reaches optimal performance faster than HeMem.

When the hot dataset is clustered but does not fit in the cache. We measure performance when 50% of the data is hot (240GB, 5× the cache size). Figure 5 presents the performance of JC and HeMem compared to the performance that GUPS would get if all the data were to fit in DRAM.

Interestingly, HeMem's performance slightly increases at the beginning of the benchmark as it brings hot pages in the DRAM cache. JC's performance slightly decreases as the cache fills with dirty data. Regardless of the policy, GUPS end up doing most of its memory accesses in PMEM because most of the data does not fit in the cache. In their steady state, all solutions have the same performance.

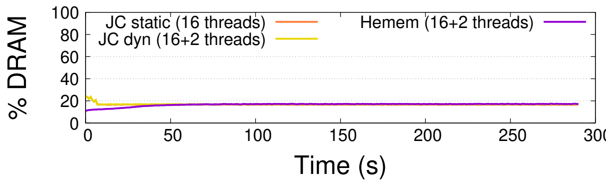


Figure 5: Hot values clustered in the array. 480GB dataset, 50% of which is hot (250GB). JC and HeMem do most of their accesses in PMEM because the hot dataset vastly exceeds DRAM capacity, which explains the low overall performance.

When the hot dataset is not clustered. Figure 6 presents the performance of GUPS when the hot data is not clustered.

As in the smaller experiment, HeMem cannot bring the hot data to DRAM and performs most of its accesses in PMEM. The performance of JC depends on the likelihood of conflicts. When a small percentage of the data is hot (2%, 9.6GB), then conflicts in the cache are unlikely. When most of the data is hot, JC also performs most of its accesses in PMEM and has the same performance as HeMem.

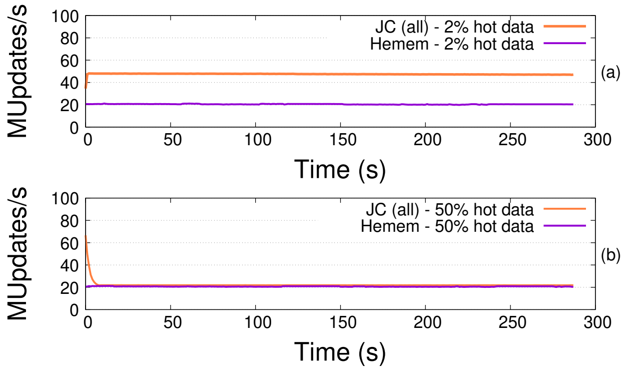


Figure 6: Hot values distributed in the array, 480GB dataset. The performance of JC depends on the percentage of hot values. Because hot values are spread on most pages, HeMem cannot improve performance and performs suboptimally.

5.2.4 Summary

In summary:

+ Hardware caches perform well, even without any active monitoring and page remapping. Software migration is highly dependent on its ability to detect hot pages.

+ Hardware caches reach steady-state performance faster than software migration.

+ Hardware caches often vastly outperform software migration when working with scattered small hot items.

5.3 BC

In this section, we evaluate the performance of BC, running with as many threads as cores, using the default Linux page allocator, HeMem and JC. Figure 7 presents the average duration of an iteration of the betweenness centrality algorithm.

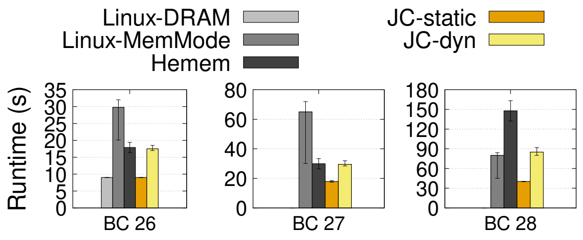


Figure 7: BC, average duration of an iteration, in seconds (lower is better). Linux-DRAM is the performance of BC when all data is manually allocated in DRAM. Linux-DRAM runs out of memory on BC 27 and BC 28. JC-static outperforms HeMem and JC-dyn.

When the hot data fits in DRAM With a scale of 26, BC allocates 35GB of memory and fits in DRAM. With a scale of 27, BC allocates 70GB, but then only actively accesses 45GB, so its "hot" dataset also fits in DRAM.

We confirm the results of the original HeMem paper: HeMem is faster than Linux in these two configurations. However, JC-static outperforms Linux and HeMem by up to 3.2× and 2× respectively. On BC 26, JC-static matches the performance of manually allocating all the data in DRAM (Linux-DRAM in Figure 7).

The performance differences are explained by the nature of the processing performed by BC. BC is an OpenMP application, and each of its threads performs a fixed fraction of the computation. The monitoring used by HeMem and JC-dyn uses two CPU-intensive threads, and these two threads compete for CPU with BC's threads. Because BC's threads frequently wait for each other in barriers, interrupting a single thread causes the whole application to be delayed at barriers. When run with HeMem or JC-dyn, BC 26 spends 50% of its time waiting at barriers. In comparison, JC-static has no overhead during the execution of BC, BC's threads progress at the same pace and spend only 2.5% of their time at barriers.

The BC example again illustrates the difficulty of fine-tuning software-migration systems. In BC, we found that the optimal performance was reached when dividing the default sampling rate by 10× and performing page cooling once every second. In that configuration, the Hemem and JC-dyn versions of BC 26 match that of JC-static in performance, and for BC 27 they improve from 60% slower to 10% slower. However, with such a low sampling rate, no hot page is detected in the previously tested configurations of GUPS, resulting in JC-static being 4× faster than HeMem in that benchmark.

The poor performance of Linux is explained by conflicts in the DRAM cache. Conflicts between hot pages are rare (on average 500MB of hot pages conflict in BC 26), but these conflicts are not evenly distributed between threads: some threads end up manipulating pages that mostly conflict, while others manipulate pages that mostly do not conflict. Threads impacted by conflicts slow down the whole application because the fast threads spend most of their time waiting at barriers. We measured that threads spend on average 63% of their time waiting at barriers in BC 26.

When the hot data does not fit in DRAM With a scale of 28, BC allocates 140GB of memory and accesses 90GB of it. In this configuration, JC-static is 5× faster than HeMem, and HeMem is slower than the default Linux page allocation mechanism.

The low performance of HeMem and JC-dyn is again explained by the interference between their monitoring threads and BC, and pressure put by page migrations on PMEM. HeMem copies data from DRAM to PMEM using DMA, which has low CPU overhead, but still increases contention on PMEM. HeMem ends up migrating 40GB of data during the execution of BC 28, continuously putting pressure on PMEM, and exacerbating the imbalance issues observed at scale 26 and 27. At scale 28, on average threads spend 65% of their time waiting at barriers. JC-dyn performs better than HeMem on BC 28 because it creates less contention on PMEM: JC-dyn only infrequently migrates data (on average 4GB per run). Its overhead comes mostly from monitoring memory accesses and cooling pages.

JC-static performs well because it is able to avoid most conflicts at allocation time. Indeed, BC mostly operates on two arrays: a 10GB array is frequently accessed, the other one less so. JC-static allocates the pages of the frequently accessed array so that they do not conflict with each other, effectively minimizing conflicts without the need for any migration. It may seem "lucky" that the hot data was allocated at once, which causes JC-static to place all hot pages in different cache bins, but we found this pattern to be extremely common in HPC applications (e.g., all the NAS applications start by allocating large arrays, only a subset of which are hot).

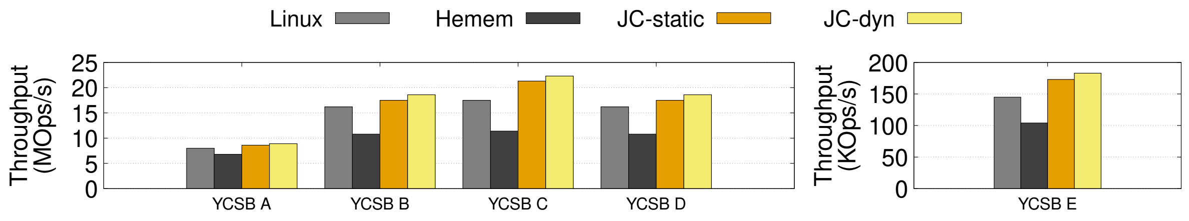


Figure 8: Masstree throughput (higher is better).

Summary Software-based migration requires monitoring memory accesses to perform informed migrations. The overhead of this monitoring can have cascading effects on HPC applications that rely on barriers to synchronize their threads. In contrast, it is possible to minimize conflicts in hardware caches at allocation time, without any profiling. Hardware caches thus vastly outperform software migrations when CPU overheads need to be avoided.

5.4 Masstree

We configured Masstree to execute with a 120 GB database, and we use 16 threads to avoid competition for CPU between Masstree's threads and the profiling and migration threads, which run on dedicated cores. The YCSB workload used by Masstree uses 128B items (1 billion items in total) and follows a Zipfian distribution: 20% of the dataset is accessed 80% of the time. Most of the accesses thus target the index (5GB) and a subset of the values (25GB).

5.4.1 Performance

Throughput Figure 8 summarizes the performance of Masstree on the YCSB workload. In both applications, JC outperforms Linux and HeMem by up to 2×.

The memory access behavior of Masstree is similar to that of GUPS when the hot items are randomly scattered in the allocated array. During initialization, items are inserted in the key-value store in random order. It is thus possible for a hot value to be allocated next to a cold value. Similarly, the nodes of the index are populated in random order, and it is possible for a hot node to sit next to a cold one. Because the hot data is scattered on all pages, it is not possible to bring the hot dataset to DRAM.

In the case of GUPS with distributed hot values, HeMem could not improve the performance of the application at all because hot items were uniformly hot. In Masstree, the index is slightly hotter than the values, and values are accessed in a Zipfian way. HeMem thus manages to migrate some of the "hottest" pages to DRAM, but 45% of the memory accesses performed by Masstree still hit PMEM. In comparison, hardware caches operate at the cache line granularity, and the hottest nodes and values are unlikely to conflict. On average, only 15% of the memory accesses hit PMEM with JC-static.

JC-dyn performs marginally better than JC-static because it detects that the pages used by the index are hotter than the pages used by the values. The difference with JC-static is negligible (14% of the data found in PMEM vs 15%).

It may seem surprising that migrations do not improve performance and that statically minimizing conflicts is enough to achieve close to optimal performance in Zipfian workloads, but conflicts between the hottest items are extremely unlikely (items are only 128B each in a 48GB cache). The benefit of adding active monitoring and conflict avoidance is thus negligible on average.

Latency The migrations performed by HeMem and JC-dyn have an impact on the observed latencies. Table 3 summarizes the latency spikes observed while running YCSB. While all systems have excellent 99p tail latency, the migration daemon pre-empts the Masstree threads, sometimes delaying the processing of a request by up to 4ms. Even though we use fewer threads than cores, the threads are not pinned to cores. The scheduler sometimes schedules two threads on the same core, explaining the pre-emption delays. The phenomenon happens when the scheduler tries to schedule threads that frequently block and unblock, such as the migration daemon.

Configuration	99p	Maximum latency
Linux	10us	10us
HeMem	10us	4ms
JC-static	10us	10us
JC-dyn	10us	4ms

Table 3: Maximum latency observed on the YCSB workload.

5.4.2 Performance over time, impact of the sampling rate

Both JC and HeMem perform better after a warm-up period: the DRAM cache needs time to cache accessed data, and HeMem needs time to detect and migrate hot pages to DRAM. Figure 9 presents the evolution of the performance of YCSB C. We initialize Masstree by inserting keys in random order, and then launch multiple iterations of YCSB C. Each iteration of YCSB C performs 10 million lookups, and keys are accessed following a Zipfian distribution. For HeMem, we compare 4 configurations with varying sampling rates. *HeMem-1K* corresponds to the highest sampling rate, with 1 sample analyzed every 1,000 memory accesses, and *HeMem-50K* to the lowest sampling rate. The default rate of HeMem is *HeMem-10K*. We also evaluate the impact of the sampling rate on JC-dyn (*JC-10K* and *JC-50K*).

Figure 9 illustrates the impact of the sampling rate on performance. When the sampling rate is too high, the overhead of sampling negatively impacts performance (*HeMem-1K*). Even when the profiling and migration threads run on dedicated cores, the other cores still handle the interrupts generated by the performance monitoring units of the CPU when a memory access is sampled. These interrupts explain the lower performance of HeMem-1K. When the sampling rate is too low, many accessed pages are never marked as hot and are never migrated to DRAM (*HeMem-50K*). In this benchmark, the optimal performance of HeMem is reached when the sampling rate is close to the default sampling rate (*Hemem-5K*, *Hemem-10K*).

JC is less impacted by such considerations because its performance is good even without any conflict avoidance daemon. JC-dyn (*JC-50K*) also fails to detect any conflicts, but its performance reaches 3% of our optimal configuration after 100s of execution. Even without any conflict avoidance daemon (*JC-static*), JC is only 5% slower than the optimal configuration.

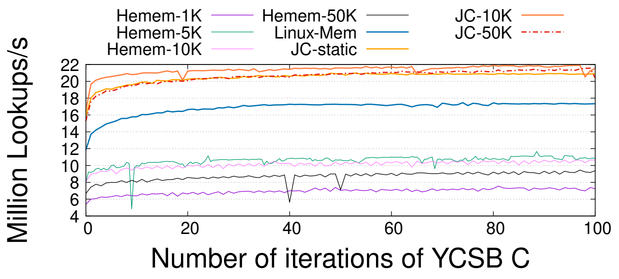


Figure 9: Performance of YCSB C. Every iteration of YCSB C runs 10 million queries.

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 10 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, but better than Linux.

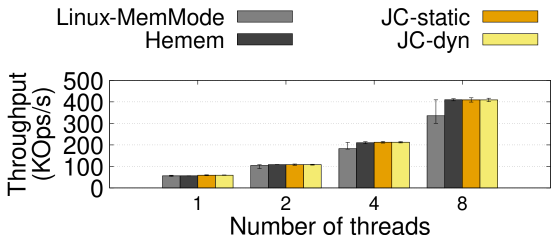


Figure 10: SILO (TPC-C) throughput (higher is better)

5.6 NAS benchmarks

Figure 11 presents the performance of the NAS benchmark suite running with Linux, HeMem and JC. We only include applications that executed in less than 24 hours 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 is 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 11). 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 at barriers.

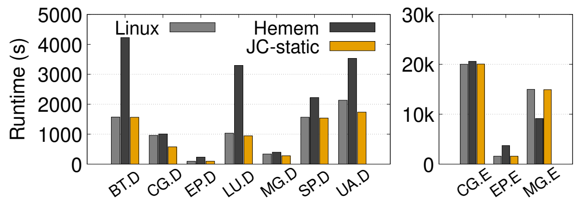


Figure 11: NAS application runtime (lower is better). JC outperforms Linux and HeMem except when the hot set size vastly exceeds the cache size (CG.E, MG.E).

The NAS benchmarks also allow us to demonstrate the limitations of our approach. 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. It is well known that applications with uniform access to large data sets or streaming access patterns do not benefit from caching. For instance, in large streaming workloads, the streamed content keeps replacing itself in the cache, and data is always evicted before being re-read. In the worst case scenario, with DRAM caching, 100% of the memory accesses end up in PMEM. With software-based migration, some of the data is allocated in DRAM, and some of the memory accesses are resolved in DRAM.

Such applications are hard to support efficiently 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 PMEM is idle, to reduce the latency of future evictions of dirty data.

Summary JC-static equals or outperforms Linux and HeMem on most NAS benchmarks. When, however, 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 inferior to 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 HeMem and JC-dyn. 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 workload, we recommend using hardware caches with a static page allocation policy.
- When working with very large datasets that do not have a clear hot subset, caches should be avoided.

A surprising observation of this paper is that, for many workloads, large hardware caches perform close to optimally with a static page allocation policy, and that having a conflict avoidance daemon is unnecessary. This seemingly counter-intuitive observation is explained as follows: conflicts that would be fixed by a daemon happen between frequently accessed cache lines. The number of such cache lines has to be small compared to the size of the DRAM: at current DRAM speed, it takes a few seconds to read the full DRAM cache, so any dataset that is large compared to the DRAM cache size cannot be "frequently" accessed. Because the number of frequently accessed cache lines is small compared to the DRAM size, the likelihood of problematic conflicts is small and a conflict avoidance daemon is more often a source of overheads than useful.

It is possible to craft adversarial workloads for which the static page allocation policy underperforms, and in which the dynamic policy performs well. In hand-crafted corner-case workloads, we found that running the conflict avoidance daemon infrequently and with a low sampling rate was enough to detect the most problematic conflicts and get close-to-optimal performance.

Applicability to systems other than DRAM+PMEM To the best of our knowledge, Intel's Memory Mode is the only currently commercially available hardware DRAM cache, so we focused the performance evaluation on DRAM+PMEM systems. We believe that the findings of this paper apply more broadly. Indeed, we have shown that tracking memory accesses at the software level is costly (profiling overhead) and requires migrating a large amount of data (migration overhead). These observations are fundamental limitations of software migration and independent of the underlying technology. If anything, software migration cost is likely to increase in future hardware with larger and faster memory – higher sampling rates will be required to detect and migrate more pages faster, incurring even more CPU overhead.

In comparison, provided that conflicts are minimized, hardware caches tend to "work well by default". Because hardware caches perform close to optimally even without any active conflict avoidance daemon, they can be operated with limited or no CPU overhead, and are more likely to perform well on future hardware.

7 Related Work

Software-managed migration Previous work focused on managing tiered memory systems at the software level. HeMem [20] is the state-of-the-art page migration system for DRAM+PMEM systems. HeMem focused on reducing the overhead of page migration, but still suffers from profiling and metadata overheads. Over the years, multiple metrics have been explored to accurately infer the heat of pages. Thermostat [1] and AutoNUMA [9] compute heat by sampling the accessed bit of the page table. Nimble [25] uses the OS active/inactive page list. TMO [23] counts the number of cycles wasted waiting for unavailable resources. HeteroOS [14] uses hints from guest OSes to help the host OS perform informed page placement decisions. X-Mem [12] uses hints from the application developers to compute the hottest pages. UniMEM [24] uses performance counters and hints from the MPI runtime. Carrefour [11] gathers high-level performance metrics from the CPU (e.g., average latency of memory accesses) to tune the frequency of memory access sampling. All these works show that measuring heat accurately is a hard problem, but crucial to the performance of software migration. In this paper, we have shown that hardware caches are less sensitive to heat measurement and can even operate efficiently without if conflicts are statically minimized at page allocation time.

CPU cache management systems In this work, we assume the DRAM cache to be a 1-way directly mapped cache. This assumption holds true on current systems, and is likely to hold true in the future – for large caches DRAM, 1-way caches have been shown to outperform multi-way caches [18].

Multiple strategies have been proposed for maximizing the efficiency of CPU caches. In the early 90s, Kessler et al. [15] simulated the relationship between page placement and conflicts in caches and showed that it is possible to reduce the number of conflicts at allocation time. Bershad et al. [5] simulated the impact of page migration on the efficiency of caches. The generalization of caches with large associativity (many-ways CPU caches) allowed CPUs to keep a few conflicting cache lines in their caches and reduced the impact of system-level page placement on the performance of caches. These techniques have gradually been replaced by much coarser-grain page coloring techniques that partition the cache to avoid cache trashing between users or applications [6, 26] or by scheduling techniques to better share the cache between cache-intensive and cache-friendly applications [2, 21, 27]. It is interesting to note that current DRAM caches resemble the state of large CPU caches simulated in the 90s, and that page allocation policies matter in current tiered memory systems. In this work, we chose to avoid partitioning the cache. Adding page coloring on top of conflict minimization could be implemented to give a larger portion of the DRAM cache to an application.

The impact of page placement on cache performance has also been studied on Intel Xeon Phis. Xeon Phis can be configured to use a large MCDRAM pool as a hardware cache that sits in front of DRAM. Intel's Zonesort [28] aims at limiting conflicts in the MCDRAM pool at page allocation time. In its first release, ZoneSort [10] periodically sorted the list of available free pages in an order that limits conflicts with already allocated pages. The module incurred significant CPU overhead and only partially limited conflicts. A later version of ZoneSort [28] allocated pages from bins in a round-robin order, an approach which does not always minimize conflicts when pages are not freed in the same order as they are allocated. JC always allocates pages from the bin with the lowest heat. Zonesort was thought of as a temporary solution for applications that have not been adapted to the Xeon Phi architecture. In our paper, we show that the hardware management of a tiered memory system, combined with low-overhead conflict avoidance techniques, outperforms traditional page migration on a wide range of workloads. We believe that this novel counter-intuitive conclusion is important in the widening context of cacheable disaggregated memory.

8 Conclusion

We have demonstrated that hardware caches offer better performance than software management of tiered main memory systems, provided minor modifications of the operating system. We have shown that, surprisingly, statically minimizing conflicts at allocation time is sufficient to avoid most conflicts between hot pages in the cache.

Acknowledgements. We would like to thank our shepherd, Emery Berger, and the anonymous reviewers for all their helpful comments and suggestions. This work was supported in part by the Australian Research Council Grant DP210101984.

References

[1] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.

[2] Reza Azimi, David K Tam, Livio Soares, and Michael Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *ACM SIGOPS Operating Systems Review*, 43(2):56–65, 2009.

[3] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[4] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[5] Brian N Bershad, Dennis Lee, Theodore H Romer, and J Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 158–170, 1994.

[6] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. Compiler-directed page coloring for multiprocessors. *ACM SIGPLAN Notices*, 31(9):244–255, 1996.

[7] Zixian Cai, Stephen M Blackburn, and Michael D Bond. Understanding and utilizing hardware transactional memory capacity. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, pages 1–14, 2021.

[8] Many contributors. Samsung Electronics Introduces Industry's First 512GB CXL Memory Module. "https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module", 2022.

[9] Jonathan Corbet. AutoNUMA: the other approach to NUMA scheduling. "https://lwn.net/Articles/488709/", 2019.

[10] Intel Corporation. ZoneSort module. "https://github.com/oslab-swrc/flsched/blob/main/knc/linux/drivers/zonesort/zonesort_module.c", 2017.

[11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGPLAN Notices*, 48(4):381–394, 2013.

[12] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[13] Intel. How Does the DRAM Caching Work in Memory Mode Using Intel® Optane™ Persistent Memory? "https://www.intel.com/content/www/us/en/support/articles/000055901/memory-and-storage/intel-optane-persistent-memory.html", 2021.

[14] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 521–534, 2017.

[15] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.

[16] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[17] Amar Phanishayee, David G Andersen, Himabindu Pucha, Anna Povzner, and Wendy Belluomini. Flex-kv: Enabling high-performance and flexible kv systems. In *Proceedings of the 2012 workshop on Management of big data systems*, pages 19–24, 2012.

[18] Moinuddin K Qureshi and Gabe H Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 235–246. IEEE, 2012.

[19] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem - artifact. "https://sysartifacts.github.io/sosp2021/results.html", 2021.

[20] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.

[21] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors. *ACM SIGOPS Operating Systems Review*, 41(3):47–58, 2007.

[22] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[23] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. Tmo: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 609–621, 2022.

[24] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[25] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered

memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

[26] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.

[27] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ACM Sigplan Notices*, 45(3):129–142, 2010.

[28] Daniluk Łukasz. mm: Add cache coloring mechanism. "https://lkml.org/lkml/2017/8/23/195", 2017.