CBMM: Financial Advice for ~~Hedge Fund~~ Kernel Memory Managers

Abstract

First-party datacenter workloads present new challenges to kernel memory management (MM), which allocates and maps memory and must balance competing performance concerns in an increasingly complex environment. In a datacenter, performance must be both good *and* consistent to satisfy service-level objectives. Unfortunately, current MM designs often exhibit inconsistent, opaque behavior that is difficult to reproduce, decipher, or fix stemming from (1) a lack of high-quality information for policymaking, (2) the cost-unawareness of many current MM policies, and (3) opaque and distributed policy implementations that are hard to debug. For example, the Linux huge page implementation is distributed across 8 files and can lead to page fault latencies in the 100s of ms.

In search of a MM design that has consistent behavior, we designed Cost-Benefit MM (CBMM), which uses empirically based cost-benefit models and pre-aggregated profiling information to make MM policy decisions. In CBMM, policy decisions follow the guiding principle that *userspace benefits must outweigh userspace costs*. This approach balances the performance benefits obtained by a kernel policy against the cost of applying it. CBMM has competitive performance with Linux and HawkEye, a recent research system, for all the workloads we ran, and in the presence of fragmentation, CBMM is 36% faster than Linux on average. Meanwhile, CBMM nearly always has better tail latency than Linux or HawkEye, particularly on fragmented systems. It reduces the cost of the most expensive soft page faults by 2-3 orders of magnitude for most of our workloads, and reduces the frequency of 10-1000μs-long faults by around 2 orders of magnitude for multiple workloads.

1 Introduction

Datacenter workloads present new challenges to kernel memory management (MM). MM encompasses a large collection of kernel mechanisms and policies to allocate and map physical memory. Cumulatively, they comprise a complex set of tradeoffs that, when poorly navigated, lead to poor performance or unexpected behavior. For example, we found that for some workloads on Linux, a soft page fault lasting 25ms occurs every 100ms. This drastic tail latency is due to memory compaction or reclamation when attempting to allocate a huge page – a misnavigated tradeoff. Many applications would violate response latency objectives if one request per 100ms takes 25ms due to a page fault. As a result, Redis, MongoDB, and others advise users to disable Linux's Transparent Huge Page (THP) feature [2, 3, 4, 7, 48]. Table 1 lists other examples of MM policies and their potential pathologies.

The hardware and software in modern datacenters differ vastly from those in use when MM techniques were first designed. Service providers run diverse first-party software, often concurrently on the same machine [1, 6, 26, 41, 43, 44]. Increased memory capacities allow more workloads to run but bring challenges too: huge page management becomes more critical due to increased reliance on TLB performance, but memory fragmentation and huge page management overheads also increase with memory capacity [35]. Datacenters also prioritize *tail* latency as a key service-level metric, in addition to median latency and throughput [18]. Datacenter behavior must be consistent, i.e., low variance, without compromising performance metrics to satisfy service-level objectives and efficiency goals.

Unfortunately, current MM designs often fall short of modern computing needs by exhibiting inconsistent, opaque behavior that is difficult to reproduce, decipher, or fix. These issues come from three key limitations.

First, kernel MM must predict workload behavior in an information-poor environment. Current MM designs rely on online measurements, particularly page table access/dirty bits and the frequency and location of page faults. Unfortunately, this information is expensive to collect and low bandwidth. For example, Google uses access bits to detect idle memory [14], but other work finds them insufficient to predict TLB miss overheads accurately [37], even though they can cost up to 11% of CPU cycles to collect [14]. Other data collection mechanisms induce additional page faults [15, 23].

Policy	Goal	Pathology
Huge Page Allocation	Reduce TLB misses and page faults	Bloat memory usage if not all memory is used; increase page fault latency if compaction is required
Copy-on-Write or Demand Paging	Reduce unnecessary memory bloat and initialization time	Increase page fault latency post-initialization if the page is written/accessed
Eager Paging [28]	Move page fault latency to allocation time, saving time later	Bloat memory usage if not all memory is used
Background Compaction	Reduce memory fragmentation and huge page fault latency	Increase CPU overhead
Background Zeroing	Reduce page fault latency	Increase CPU overhead
Idle Page Reclamation [31, 46]	Improve memory utilization	Increase overhead to fault reclaimed pages back in; increase CPU overhead to choose pages to reclaim

Table 1: Different MM policies and their goals and pathologies

Recent work uses performance counters in kernelspace [37], but currently available counters are hardware-thread-oriented and do not provided the detailed spatial information useful for most MM policies.

Second, current MM designs often ignore the cost of various MM operations, leading to inappropriate policy decisions. For example, Linux allocates a huge page when a memory region is first touched; however, we find that allocating and zeroing a huge page costs 10^6 cycles in the best case. Thus, promoting a page that averts $\leq 10^6$ cycles worth of TLB misses and page faults actually *regresses* performance, but the kernel does not account for this cost.

Third, current MM designs are implemented as disjointly acting policies distributed throughout the kernel that are hard to debug. As a result, users and developers observe erratic slowdowns but have no indication what causes them or how to address them. Instead, they often resort to coarse-grain solutions that miss opportunities, such as disabling huge pages [2, 3, 4, 7, 48]. By distributing and obscuring policy-implementing code, current kernel MM implementations make it difficult for both kernel and userspace developers to decipher system behavior. For example, code implementing Linux's huge page policies is scattered across more than eight files (and numerous functions), mixed with unrelated code. This opaque system implementation and its consequent opaque behavior is a primary obstacle to improving kernel MM performance, consistency, and debuggability.

In search of a MM design that has consistent behavior, we designed Cost-Benefit MM (CBMM). CBMM reflects that all kernel MM operations have a cost and a benefit to userspace, and it estimates them using empirically based cost-benefit models to guide MM policy decisions. By explicitly modeling cost and benefit, CBMM is more cost-aware than current designs, so it makes fewer pathologically bad policy choices. Also, CBMM augments online statistics with offline-aggregated profiles to improve the quality of information available to the kernel. CBMM simplifies policy debugging and enables incremental performance improvement by centralizing models in a new kernel component: the *estimator*. To understand and fix anomalies, one need only understand the model inputs to determine the cause of a policy decision.

Our prototype implements models for huge page promotion, asynchronous page prezeroing, and eager paging [28], based on an in-depth analysis of huge page behavior and soft page faults. At runtime, they may make use of in-built empirically based assumptions (e.g., about average TLB miss latency), online information (e.g., the current number of free pages), or offline-aggregated profile information (e.g., fine-grained information about huge page benefits). We focus on *first-party* datacenter workloads – software run by service providers in their own datacenters – as they are highly controlled and relatively stable over time, allowing better profiling and modeling [1, 6, 9, 11, 26, 41, 43, 44].

CBMM improves system consistency; it nearly always has better tail latency than Linux or HawkEye, particularly on fragmented systems. It reduces the cost of the most expensive soft page faults by 2-3 orders of magnitude for most of our workloads, and reduces the frequency of 10-1000μs-long faults by around 2 orders of magnitude for multiple workloads. Meanwhile, it has competitive performance with Linux and HawkEye, a recent research system [37], for all the workloads we ran, and in the presence of fragmentation, CBMM is up to 36% faster than Linux on average – all while using no more huge pages than Linux or HawkEye in most cases.

2 Motivation: Evaluating Current Behavior

To quantify the extent of these challenges and inform our design, we do an in-depth analysis of two important kernel MM code paths, huge page management and page fault handling. Our experimental setup is described in Section 5.1.

2.1 Measuring Huge Page Benefits

It is well known that huge pages speed up many workloads, but nobody has quantified the impact of workload behavior on the amount of speedup it receives from huge pages. Thus, we measure the fine-grained benefit of huge pages as described in Section 4.1. Figure 1 shows the results[1]. For each workload, we divide the address space into 100 equally sized ranges, excluding unmapped regions, and repeatedly run the workload

[1] Space prohibits showing all workloads, but we will publish them online.

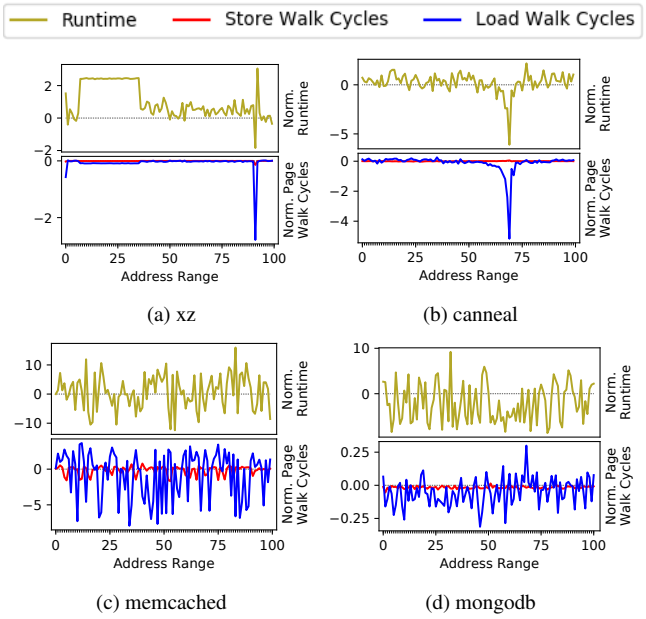

Figure 1: Normalized runtime and percentage of usermode cycles spent in page walks for each address range compared to no huge pages (lower is better). Note the varying y-axes.

backing one range at a time with huge pages. Each point on the x-axis represents one range, such that the x-axis represents the virtual address space. The top y-axis shows the normalized performance compared to no huge pages. The bottom y-axis shows the normalized percentage of time spent in usermode page walks (i.e., TLB misses) for loads and stores.

The impact of huge pages varies extensively between workloads. `xz` and `canneal` primarily see improvements in *load* page walk cycles from backing particular regions of the address space corresponding to hot data structures. `memcached` and `mongodb` produce noisy results because of randomness in the workload. The magnitude of impact ranges from about 0.25% in `mongodb` to almost 7% in `canneal`.

Another benefit of huge page usage is fewer page faults. We found that they have only a minor contribution to performance (e.g., less than 1.2% of execution time for `canneal`).

Also, the relationship between runtime improvement and reduction in page walk cycles is not straightforward. For all workloads, runtime improvement is loosely correlated with either load or store page walk cycles. Strong effects on either load or store page walk cycles tended to be reflected in runtime, as seen in `xz` and `canneal`, but the magnitude of that effect varies. Small changes in page walk cycles often have no apparent effect on runtime.

Discussion Huge page impact varies greatly by workload, including the type and location of impacted memory accesses and the magnitude of impact. Additionally, the relationship between page walk cycles and runtime is complex, illustrating the challenge of huge page management given the limited, low-quality information available to the kernel at runtime

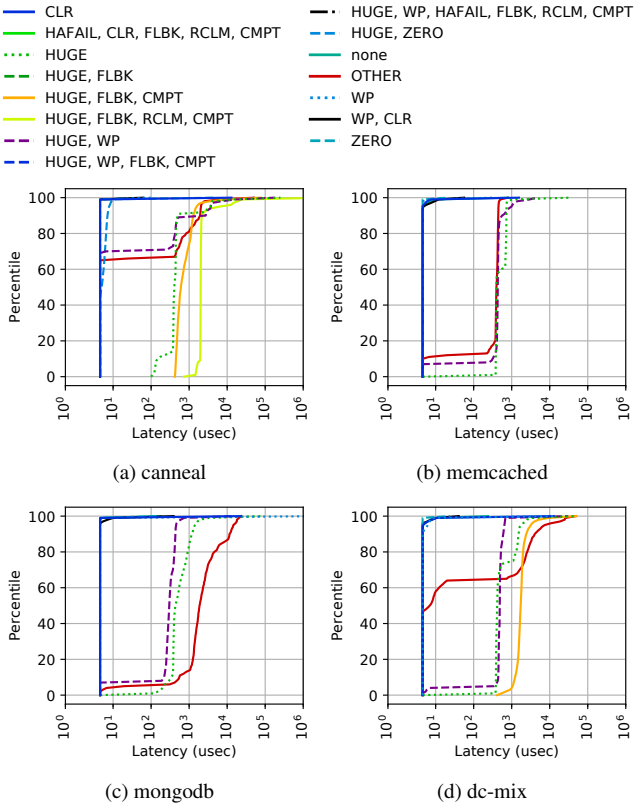

Figure 2: CDF of Linux soft page fault latency by type of page fault. Not all page fault types occur in all graphs.

Flag	Description
`CLR`	Memory was zeroed (usually during allocation).
`CMPT`	Allocator used memory compaction.
`FLBK`	Allocator used a fallback path during the page fault.
`HAFAIL`	Attempted to allocate a huge page and failed.
`HUGE`	A huge page was mapped.
`PREZ`	Allocator allocated a prezeroed page (CBMM only).
`RCLM`	Allocator used direct reclamation.
`WP`	Page fault due to write to a write-protected page.
`ZERO`	A (shared) zero page was mapped.

Table 2: Subset of bitflags for page fault tracing.

such as CPU performance counters and page referenced bits. For example, `dc-mix` (not depicted) benefits from backing individual regions with huge pages, but when THP is turned on, it sees a net regression in performance due to the overhead of compaction. CBMM aims to mitigate this problem by supplying the kernel with higher-quality information.

2.2 Soft Page Fault Latency Breakdown

We instrument Linux's page fault handler to trace sources of page fault latency. Page fault tracing allows us to characterize system-wide costs, such as the cost of zeroing memory. We identified a set of events that occur during page faults and associate each with a bitflag (Table 2). Our instrumentation records the total time of the page fault, the time to allocate

memory, and the time to clear/copy memory contents.

We record the flags and timing of all events longer than 10^4 cycles, and a count of shorter events, allowing us to compute the proportion of all page faults with each set of flags. We exclude hard page faults from our results, as they incorporate other kernel subsystems (e.g., block I/O, file systems). Our tracing records the total time to handle a page fault, but on x86 the handler can be interrupted in favor of another task, which inflates the latency of the page fault. This is rare in most workloads except `mongodb`, which uses a userspace asynchronous I/O framework and thread pool; even though a page fault handler may be descheduled for a while, userspace requests continue to make progress because of userspace threading.

Figure 2 shows the soft page fault latency breakdown for multiple workloads[2]. For each distinct set of flags, the CDF of page faults with those flags is plotted. Note that the x-axis uses a log scale. The plot includes samples lower than the threshold by treating them as if they all took 10^4 cycles (in reality, most are faster than that). The figure shows results on a freshly booted, unfragmented system, which represents best-case performance; we also recorded results on fragmented system, and found them significantly worse for all workloads.

The results indicate three challenges current MM designs face. First, applications trigger a wide variety of kernel behaviors. Each of the 15 flag-sets of Figure 2 is a different combination of code paths. Second, different paths have very different latencies but are relatively consistent across workloads. For example, even in this best case, a huge page consistently takes hundreds of microseconds to be allocated (`HUGE` in the figure) due to zeroing overhead. Third, many pathological code paths execute that do not benefit applications. Most notably, a huge page allocation may invoke a fallback path (`FLBK`), which transitively invokes compaction (`CMPT`) or reclamation (`RCLM`). Worse, the fallback may fail (`HAFAIL`), resulting in a base page allocation after all. In `canneal` (Figure 2a) and `dc-mix` (Figure 2d), these fallback paths can take *dozens or hundreds of milliseconds*. In contrast, an Amazon search for "DRAM" completes in only 900ms from our office.

Discussion Linux's fallback algorithms are severely cost-unaware and make system behavior inconsistent: invoking compaction or reclamation almost certainly outweighs any benefits of using a huge page. Also, the high cost of zeroing suggests that memory prezeroing (Section 4.2) may be a useful optimization to make huge pages more useful. Currently, if an average TLB miss costs around 30 cycles, then a huge page must avert over 33,000 TLB misses to pay for itself. These results highlight the need for cost-aware MM policies.

3 Cost-Benefit Memory Management

To mitigate the above challenges, we created the Cost-Benefit Memory Manager (CBMM). CBMM has several goals:

- Improve kernel MM behavioral consistency,
- Match existing systems' performance,
- Improve the debuggability of policy decisions,
- Allow incremental improvement of individual policies.

Our key insight is that all MM decisions incur a cost against and provide a benefit to userspace. For example, huge page promotion averts TLB misses but may require zeroing or compacting memory. In CBMM, policy decisions follow the guiding principle that *userspace benefits must outweigh userspace costs*. By applying this principle uniformly, CBMM significantly improves consistency over Linux and HawkEye [37], while matching their performance. We design models for three important kernel MM policies: huge page promotion, asynchronous page prezeroing, and eager paging [28].

CBMM introduces a new component, the *estimator*, to the kernel. It estimates the cost and benefit of a given MM operation whenever a policy decision is needed. If $cost < benefit$, the kernel decides to execute the operation.

The estimator makes estimates based on empirically derived cost and benefit models. Models can optionally use live metrics and/or pre-aggregated profiling information. Such pre-aggregated information can mitigate the lack of high-quality online information. Meanwhile, CBMM explicitly estimates MM operation costs, improving cost-awareness.

In current MM implementations, policy decisions are scattered across the kernel, making it difficult to coordinate their actions and difficult to debug anomalous behavior. In contrast, CBMM invokes the estimator at decision points, which predicts the cost and benefit of taking an action. This centralizes decision making and explicitly marks policy decisions points. It also makes coordination between policies easier.

A key requirement of CBMM is that the system behavior can be modeled and/or profiled. This requirement holds for many first-party datacenter workloads, which often run with high redundancy for long amounts of time [1, 6, 9, 11, 26, 41, 43, 44], giving ample opportunity to observe and instrument a workload before applying policies to them.

3.1 The Estimator

In CBMM, the MM subsystem invokes the estimator at places in the code where policy decisions need to be made. We call these places in the code *decision points*. It uses models to estimate the cost and benefit of a particular MM operation and returns the estimates to the decision point, which executes the operation if $cost < benefit$.

When a decision point invokes the estimator, it passes information to the estimator about the type and parameters of the operations. For example, the decision point would pass the address to consider promoting or a number of pages to attempt to prezero. The estimator acts as a black box that returns a cost and benefit estimate for the given MM operation and parameters. In CBMM, costs and benefits are computed

[2] Space prohibits showing all workloads, but we will publish them online.

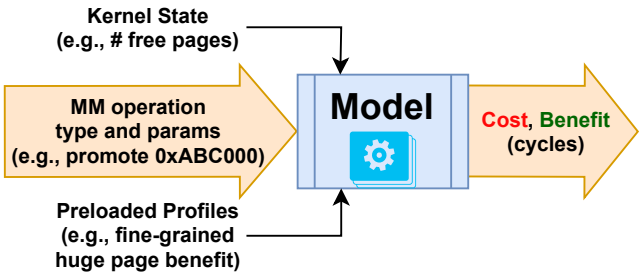

Figure 3: CBMM model inputs and outputs.

in units of time saved or lost by userspace, which usually corresponds closely to user objectives. In particular, CBMM uses the *rate* of time saving/loss over some horizon, as many datacenter workloads run continuously.

3.2 Cost and Benefit Models

Internally, the estimator comprises a collection of cost models and benefit models for different MM operations. Each model is built out of simpler submodels that estimate one cost and/or benefit well; the submodel results are added to produce the overall result. This allows reuse of submodels for different decision points, simplifying implementation and leading to more consistent behavior across decision points. For example, our huge page cost-benefit models were useful in both the page fault handler and `khugepaged`, the background promotion daemon, and our model for estimating the cost of running a daemon could be used for multiple daemons in the future.

Concretely, models manifest as C code in the estimator (in the kernel); in Listings 1 - 3 (discussed further in Section 4), we show the models in our prototype of CBMM. Each (sub)model is a self-contained black box that takes information from the decision point, combines it with information from the ambient kernel state and *preloaded profiles* – files loaded into the kernel that supply information about application-specific behavior – and outputs an estimate, as shown in Figure 3. The additional input from the kernel state and preloaded profiles allows the models to be more context-aware and to make use of higher-quality information about workload behavior.

Performance Debugging Unlike current heuristics, CBMM isolates MM policies to specific cost and benefit models; their inputs and outputs can be observed, and they can be improved in a single place, making performance anomalies easier to debug in CBMM than Linux. In addition, CBMM makes it easier to account for second-order/delayed effects (e.g., allocating now may produce memory pressure later), which current Linux heuristics typically ignore because such effects are complicated and policies are distributed throughout the kernel.

A central idea behind CBMM's debuggability is the ability to observe and control the inputs to models. Thus, while in principle, models can make use of any kernel or hardware state, models should use only state that has an intuitive interpretation, such as the number of free pages, rather than internal implementation metrics that have non-obvious meanings. For example, our huge page promotion model takes into account whether any prezeroed huge pages are available and uses a profile to determine the worth of promoting a page. In contrast, internal implementation metrics (e.g., the number of pages in a list, the "swapiness" of a page, etc) often give limited information about the origin of their values and how to cause them to change, making bug fixing difficult.

Model Development in CBMM can be done iteratively by beginning with a simple model and refining it as needed. For example, Listing 2 shows our asynchronous prezeroing model. Initially, we only accounted for the zeroing time of the daemon, but we found that this led to high lock contention on the allocator, so we refined the model to account for contention.

In designing our models, we found that benefits tend to be application-specific, whereas costs tend to be system-specific. For example, each application tends to benefit differently from huge pages, but the cost to allocate a huge page is application-independent and depends more on the state of the system allocator. As a result, our benefit models tend to use preloaded profiles, whereas our cost models tend to query kernel state.

Models necessarily make assumptions to simplify implementation and to make their execution cheaper than the actual MM operations. We based our assumptions on our empirical measurements, unlike many existing heuristics, which rely on intuitive simplicity or common-case optimization. For example, unlike Linux, CBMM does not blindly assume huge pages improve performance; rather, it incorporates the cost of promotion as measured by our experimental analysis and uses empirically derived profiles to estimate the benefit of promoting a particular memory region. Notably, CBMM improves system behavior even with imprecise profiles, as we will show in Section 5.5, making it practical to start with a simple model and refine it over time.

3.3 Preloaded Profiles

Different applications respond differently to MM policies, and kernels currently lack high-quality information with which to predict workload behavior. Preloaded profiles are files loaded into the kernel when starting a process (e.g., by a cluster manager) to provide models with information about a process's behavior. They allow the estimator to benefit from offline processing for particular policy decisions. In contrast, prior methodologies resort to measuring inaccurate and expensive proxy statistics such as page fault counts or page access bits.

In CBMM, preloaded profiles specifically provide *spatial, per-process information*; that is, they provide information about regions of a single address space at arbitrary granularity as small as a 4KB page. For example, a profile may specify per-region reduction in page walk cycles from use

of huge pages, or a bit indicating whether a page is likely to be touched or not. Models can query this information when making cost and benefit estimates. For example, to estimate the benefit of using a huge page, a model may incorporate the number of averted page walk cycles, or to determine whether to eagerly allocate memory or use copy-on-write, a model may incorporate information about the likelihood of memory accesses. This structure for preloaded profiles, while simple, is quite useful because many MM policies make *spatial* decisions, such as whether/how to map/unmap/remap a memory region. However, CBMM's design is flexible enough to admit future extensions to profiles. For example, it may be desirable to account for phases of workload activity or to apply profiles at different granularities, such as per-thread or system-wide.

4 Implementation

We implement CBMM based on Linux 5.5.8 for three kernel MM policies: huge page management, asynchronous prezeroing, and eager paging [28]. We implement the estimator and its models, along with related debugging interfaces, code for parsing profiles, and other infrastructure in 1159 lines of C in the kernel in a new and self-contained file. Additionally, we add 87 lines of instrumentation throughout the page fault handler and page allocator for page fault tracing (see Section 2.2). We add 10 calls to the estimator throughout the MM subsystem; each is self-contained and consists of about 10 lines of code to initialize a struct, make a function call, and respond to estimates. Asynchronous prezeroing is implemented in a kernel module from HawkEye. We modify the module to run in a kernel thread and to query the estimator before running. Our version of the module is 196 lines of C. We will open-source CBMM after review.

4.1 Huge Page Management

Background A primary challenge of huge page management is determining which memory regions to back with huge pages: the kernel must determine which memory regions would see enough benefit from huge pages. Linux's THP aggressively tries to promote on the first page fault to the huge page, potentially leading to memory bloat and increased page fault latency. FreeBSD waits for a specific percentage of the huge page to be touched before promoting. Various research systems use a mix of page access bits, performance counters, LRU lists, and trial-and-error [30, 37, 47] with mixed success.

Model Listing 1 shows CBMM's model for huge page promotion. It is used in both the page fault handler and khugepaged to decided whether to promote a page. We built this model based on our analysis of huge page promotion overheads. It makes a number of simplifying assumptions when estimating both the cost and benefit, most notably that the cost is dominated by the allocation and zeroing time and that compaction and reclamation have a large fixed cost. We choose to ignore other costs in our model, such as caching, mapping changes, or potential memory bloat, but CBMM allows models to be iteratively improved over time.

```
void hpage_promo_model(u64 addr, mm_cost_delta *cost)
{   // COST. Simplify using assumptions.
    // - Alloc is free if have free zeroed pages.
    // - Alloc cost is zeroing if have free unzeroed.
    // - Alloc cost is 2^32 if need to free mem.
    // - We don't care what node it is on.
    // - Constant prep costs (zeroing or copying), ~100us

    // `have_free_hpage` checks the allocator free list.
    enum free_hpage_status fhps = have_free_hpage();
    u64 alloc_cost = fhps > fhps_none ? 0 : (1ul << 32);
    u64 prep_cost = fhps > fhps_free ? 0 : 100 * 2000;
    cost->cost = alloc_cost + prep_cost;

    // BENEFIT = averted TLB miss cycles from profile.
    cost->benefit = compute_hpage_benefit(addr);       }
```

Listing 1: CBMM huge page cost-benefit model.

Profiling Our methodology generates for each workload a mapping from virtual memory regions (i.e., ranges of virtual addresses) to the number of averted usermode page walk cycles when the region is backed by huge pages. We modify the Linux kernel to give precise control over promotions. We then repeatedly run a given workload varying the set of promoted pages. We additionally run the workload with no huge pages as a baseline. We use hardware performance counters to measure the number of TLB misses, the number of cycles spent in pages walks, and the overall cycle count for kernelspace and userspace execution. We then take the difference in overhead and overall runtime between any given set of pages and the baseline. The size of the sets of promoted pages can be varied to tradeoff profiling time with precision. Our prototype uses the offset into allocation zones instead of virtual addresses, so that profiles tolerate Address Space Layout Randomization.

Broadly, we found that our workloads could be categorized as *high-processing* or *low-processing*. *High-processing* workloads, such as `xz`, `canneal`, or `mcf`, heavily process their input data to produce internal data structures; their memory access patterns are driven by computation over these data structures. *Low-processing* workloads, such as `memcached`, `mongodb`, and `dc-mix` (see Section 5.1), often do little more than data storage and retrieval, so their access patterns are driven by client request patterns. We found that we can reliably distinguish between high- and low-processing workloads using the skewness [3] of the distribution of averted page walk cycles. High-processing workloads often have a small number of highly-impactful memory regions, so they have a high positive skew ($skew > 2$ seems to work empirically). When generating profiles for low-processing workloads, we assigned all regions a benefit equal to the mean benefit measured empirically. For high-processing workloads, we assigned each region the benefit it individually demonstrated.

At runtime, we can supply a profile to the kernel in the form of a CSV file that lists virtual address ranges and their benefit

[3] *skewness* is a statistical measure of distribution asymmetry.

from huge pages. Our implementation aims to demonstrate the potential of our approach while remaining simple to implement. We do not attempt to account for phases in workload behavior, but our design is amenable to such an upgrade in the future by repeating the profiling process at multiple points during the workload's execution. We assume the workload size is stable but can handle other input changes; in Section 5, we use randomized inputs for most workloads.

4.2 Asynchronous Prezeroing

Background We examine *asynchronous prezeroing* as a means of improving the latency of large physical memory allocations. Asynchronous prezeroing clears free pages using a background daemon to save time during a page fault when it would slow down userspace programs. Our analysis indicates that prezeroing would reduce the cost of a huge page by almost two orders of magnitude.

Prezeroing has fallen out of favor because the primary cost of zeroing 4KB pages is cache misses, but prezeroing pages leaves them cold when users access them, so latency is merely shifted to userspace [8, 45]. Recently, Panwar et al. find prezeroing is beneficial for huge pages and use non-temporal store instructions to avoid cache pollution [37]. However, their approach requires hand-tuning to avoid excessive CPU usage or lock contention on the page allocator. CBMM adapts their prezeroing implementation with a model to determine when and how long to run, eliminating the need for hand-tuning.

Model Listing 2 shows CBMM's model for running the asynchronous prezeroing daemon. The model makes numerous assumptions; most importantly, it assumes that CPU time is free unless taken away from userspace (i.e., the system is not idle) and that the chief costs of prezeroing are the execution time of zeroing and contention on the allocator lock, rather than cache pollution. This matches our own analysis and observations while working on CBMM. The chief benefit of prezeroing is to move zeroing overhead out of the critical path of huge page promotion. For simplicity, we assume a constant processor frequency over short time windows, even though the frequency varies.

Also, this model exemplifies CBMM's iterative approach to building models. We started with a model that only accounted for CPU time and potential huge page allocations. As we ran experiments, we discovered the lock contention and improved the model to account for it by adding the lines labeled as `COST of lock contention` in Listing 2, resolving the performance issue. The entire process took less than a day of debugging, measurement, and implementation.

4.3 Eager paging

Background Eager paging allocates physical memory upon user request, rather than lazily on a page fault (the default) [28]. It enables large contiguous physical memory allocations, which are easier to back with huge pages and enable useful hardware optimizations [28, 36, 39, 42, 47]. However, a drawback to eager paging is memory bloat if the workload does not use all the allocated memory [28]. Preloaded profiles unlock this optimization while avoiding memory bloat.

```
void prezeroing_model(mm_action *action,
                      mm_cost_delta *cost)
{   // COST of the runtime itself... Assume:
    // - Don't care about NUMA nodes.
    // - Zeroing costs ~10^6 cycles.
    __kernel_ulong_t cpu_load = get_avenrun();
    int ncpus = num_online_cpus();
    const u64 HPAGE_ZERO_COST = 100000;

    // ncpus > cpu load average => idle cpu, free to run.
    if (ncpus > cpu_load) {
        cost->cost = 0;
    } else {
        cost->cost = HPAGE_ZERO_COST * action->prezero_n;
    }

    // COST of lock contention. Assume:
    // - Cost of lock acquisition = ~150cyc, do it 2x.
    // - Lock is unheld for ~1ms/horizon => free locking
    const u64 UNHELD = FREQ_MHZ * 1000; // cycles
    const u64 crit_sect_cost = 150 * 2; // cycles
    const u64 nfree = UNHELD / crit_sect_cost;
    cost->cost += (action->prezero_n > nfree
                   ? action->prezero_n - nfree : 0) *
                    critical_section_cost;

    // BENEFIT. Assume past usage predicts future usage.
    u64 recent_used = mm_estimated_prezeroed_used();
    cost->benefit = min(action->prezero_n, recent_used)
                    * HPAGE_ZERO_COST;                }
```

Listing 2: CBMM async prezeroing cost-benefit model

Model Listing 3 shows CBMM's model for eager paging, which is invoked by `mmap` or `brk` system calls. It uses a preloaded profile to determine which subregions will be touched and assumes that the model has perfect knowledge, allowing it to ignore the cost of potential bloat. If more than one page is being eagerly allocated, we create opportunity for contiguous allocation.

```
void eager_paging_model(vm_area_struct *mmap_region,
                        mm_cost_delta *cost)
    // ASSUME: past usage predicts future; use profile.
    // COST: time to create new page.
    const u64 PF_NEW_PAGE = FREQ_MHZ * 10; // cycles
    struct range *ranges = prev_touched(mmap_region);
    cost->cost = len(ranges) * PF_NEW_PAGE;

    // BENEFIT: time to create new page, coalesced faults
    const u64 PF_CS = 300; // cycles
    cost->benefit = len(ranges) * PF_NEW_PAGE
                  + (len(ranges) - 1) * PF_CS;          }
```

Listing 3: CBMM eager paging cost-benefit model

Profiling We profile eager paging behavior by periodically reading the `/proc/<pid>/pagemap` file while the workload is running. This file contains information about memory mappings for the given process and allows us to detect which virtual memory regions have been faulted in. Pages that were faulted in during the execution are noted in the profile, and the model assumes they will be faulted in again in the future.

5 Evaluation

CBMM seeks to improve consistency while matching or exceeding the performance and efficiency of existing systems. We evaluate CBMM along multiple axes. First, we evaluate the page fault latency of CBMM to understand its consistency compared to Linux and HawkEye. Second, we measure the end-to-end performance of CBMM. Third, we look at the efficiency of CBMM's use of huge pages. Finally, we evaluate the generality of our approach by looking at the sensitivity of performance to profile changes.

5.1 Methodology

Table 3 describes our workloads. They represent a variety of software behaviors and exercise the kernel in different ways. `mongodb`, `memcached`, and `dc-mix` are memory-intensive workloads common in datacenters. `mongodb` and `memcached` are data stores, and `mongodb` is I/O heavy and makes use of the page cache. `dc-mix` induces memory pressure and tries to simulate a real system in which a server, device driver, and batch job are using system resources. We drive the data stores in these workloads using YCSB [16] with different read-write ratios to increase the variety of MM behavior. `mcf`, `xz`, and `canneal` are computational workloads. We scale up the inputs of `xz` and `canneal` to use more memory. In all experiments with server applications (e.g., memcached, redis, mongodb), we run the client program on the same machine as the server, so as not to measure network effects. We run each workload with its default number of threads and pin all workloads to one NUMA node to reduce variation caused by NUMA effects. To reduce noise, we run each experiment 5 times and report the median results. For all workloads except `mcf` and `xz`, the input is randomized and changes between executions of the workload. For `xz`, we use the native input to generate a profile and use a custom input when evaluating performance.

All experiments run on CloudLab [40] c220g5 machines with two Intel Xeon Silver 4114 (10C/20T, 2.2 GHz, Skylake 2017), 192GB 2666MHz DDR4 ECC DRAM, and a 480GB SAS SSD. We set the CPU scaling governor to `performance`. *Unless otherwise noted, we do not tune our systems at all*; the results represent CBMM's "out of the box" behavior.

We replace the system allocator with `jemalloc`, which is better in a datacenter setting and is used by Facebook [22]. All experiments run on CentOS 7.8.2003 with the relevant kernel. We disable Meltdown and Spectre [29, 33] mitigations, which cause severe performance degradation. We use unmodified Linux 5.5.8 with Transparent Huge Pages enabled as our baseline. We configure CBMM similarly to Linux but we preload a profile of huge page benefits and eager paging, as derived in Sections 4.1 and 4.3. We also compare against HawkEye [37], a state-of-the-art research huge page management system based on Linux 4.3. We configure HawkEye as in its paper, including its prezeroing daemon. We ran experiments against stock Linux 4.3 and found that it is within 15% of Linux 5.5.8 on average (see Figure 5). To measure page fault latency in HawkEye, which runs on a different kernel without our instrumentation, we use eBPF to instrument the `handle_mm_fault` function, which represents the main portion of the page fault handler. We found that `canneal` crashes with a segfault on HawkEye when the system is unfragmented, so we omit that experiment from results.

Fragmentation We run each workload on a freshly rebooted system and on a preconditioned system. Preconditioning aims to simulate a long-running datacenter environment by inducing external fragmentation, which hinders large physical memory allocations, such as huge pages.

We had difficulty identifying a reproducible fragmentation methodology. We attempted to reuse techniques from prior work [37, 47, 49] and also made several attempts at our own methodologies with little success; on Linux, deferred freeing of physical memory and kernel daemons such as kcompactd and kswapd cause variable results. Also, each methodology preconditions machines in a different way, none of which is obviously more realistic than the others.

For our evaluation, we choose a simple methodology derived from prior work [17, 47, 49]. We enable `CONFIG_SLAB_FREELIST_RANDOM` and `CONFIG_SHUFFLE_PAGE_ALLOCATOR` when compiling the kernel and add a sysfs file that triggers shuffling of the kernel physical memory free lists. To precondition the system, we reboot and then trigger free list shuffling. Then we run a program that allocates all system memory (with `mmap(MAP_POPULATE)`) and frees all but the first page of each 2MB region before sleeping for the duration of the workload. This methodology is simple and yields comparable results to other methodologies. We measure the Free Memory Fragmentation Index [24, 30, 37, 49] after preconditioning but before the workload begins. On CBMM and HawkEye, preconditioning consistently leaves around 183GB of free memory with 99% fragmentation. On Linux, half of runs experience similar results, but in the other half, deferred page freeing causes $<$ 2GB of memory to be considered free, making it difficult to measure fragmentation. More research is needed on fragmentation and how to study it.

5.2 System Behavioral Consistency

Figure 4 shows tail latency on each kernel without fragmentation (when latency should be lowest); note the log x- and y-scales. To account for differences in the frequency of page faults due to differing MM decisions, we show the average interval between events, rather than the percentile on the y-axis.

Unlike Linux (Figure 2d), CBMM rarely attempts an expensive fallback path (e.g., compaction or reclamation) during huge page promotion, even under fragmentation; allocation failures usually result in the allocation of base pages. CBMM often experiences more page faults than Linux or HawkEye, but as Figure 4 shows, CBMM still sees a lower rate of long

Workload	Description	Input	Peak Mem
xz	data compression [5]	profiling: native input, eval: custom scaled up input	150GB
mcf	combinatorial optimization, scheduling [5]	native input	3GB
canneal	simulated annealing, chip routing [12]	custom input, randomly generated each time	150GB
mongodb	KV store	YCSB driver [16], 75%W-25%R	150GB
memcached	in-memory KV store	YCSB driver [16], 1%W-99%R	150GB
dc-mix	redis (KV store), memhog (microbench., creates fragmentation), metis (in-memory map-reduce) [27]	redis: YCSB driver [16], 50%W-50%R; memhog: N/A; metis: built-in	165GB

Table 3: Description of Workloads – their behavior, inputs, and peak memory usage.

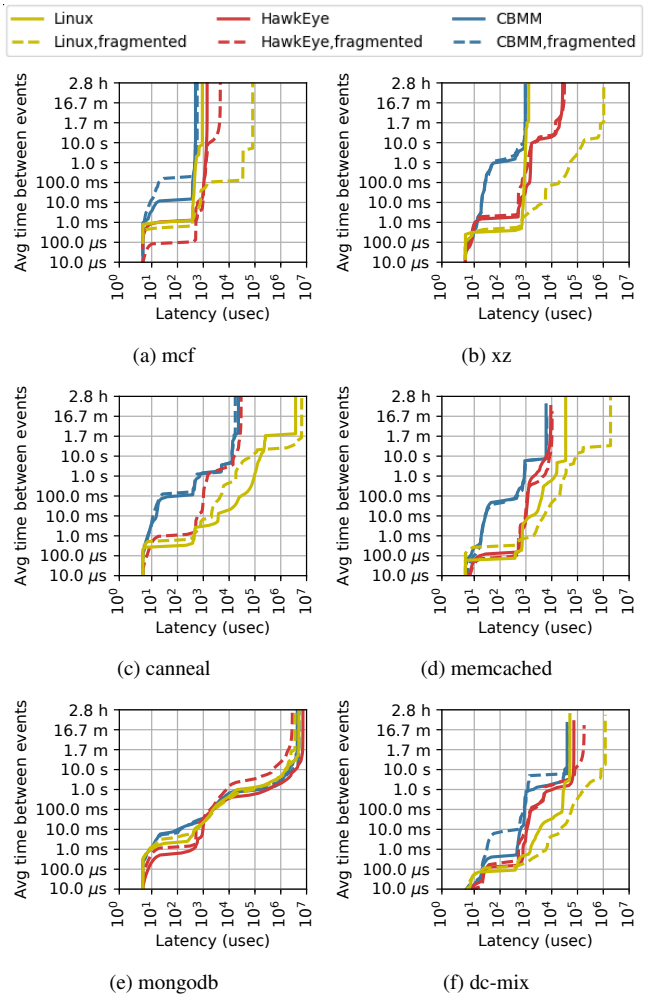

Figure 4: CDF of soft page fault tail latency on each system, weighted by page fault rate (e.g., on CBMM, `canneal` experiences a 100μs page fault every 100ms on average).

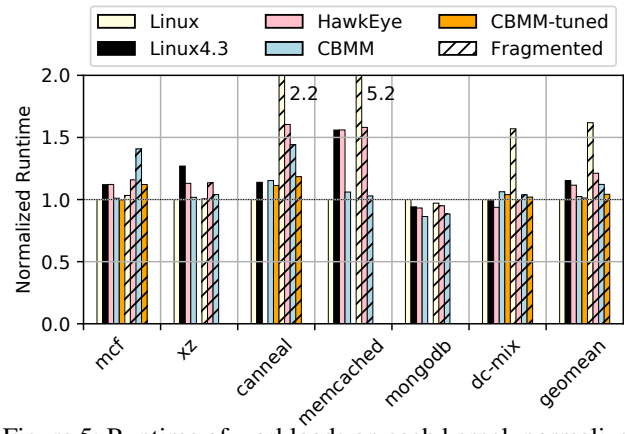

Figure 5: Runtime of workloads on each kernel, normalized to Linux with THP without fragmentation (lower is better).

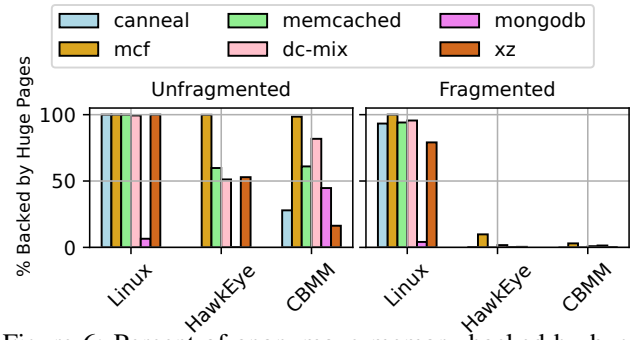

Figure 6: Percent of anonymous memory backed by huge pages on CBMM, HawkEye, and Linux with THP.

page faults than they do because its cost-awareness leads to fewer pathological cases, falling back to 4KB pages instead.

Even without fragmentation, CBMM always matches or improves on the tail latency of Linux and HawkEye, often by wide margins. In `xz`, `canneal`, and `memcached`, CBMM reduces the frequency of page faults taking 10-1000μs by around two orders of magnitude compared to Linux or HawkEye. In `canneal`, CBMM reduces the frequency of (or eliminates) all page faults slower than 10μs by two or more orders of magnitude compared to Linux. In `memcached`, page faults taking over 1ms are reduced by nearly an order of magnitude, while in `dc-mix`, they are reduced in frequency from nearly constant in Linux to every 10s or longer in CBMM. `mongodb` uses a userspace asynchronous I/O framework, as previously discussed, so its page fault latencies are dominated by context switches and other userspace threads; thus, our improvements are not visible in the figure. However, the figure does show that CBMM does not regress page fault latency, and as we will see in the next section, CBMM achieves significantly better performance than Linux or HawkEye for this workload.

Under fragmentation, CBMM usually achieves even larger tail latency improvements, particularly compared to Linux. For all workloads except `mongodb`, CBMM reduces the frequency of all page faults taking $\geq$ 500μs by 1-3 orders of magnitude compared to Linux and up to one order of magni-

tude compared to HawkEye. `mongodb` performs similarly to the unfragmented case, as discussed above.

Summary CBMM improves tail latency compared to Linux or HawkEye. For multiple workloads, CBMM reduces the frequency of slow page faults by one or more orders of magnitude, especially under fragmentation.

5.3 End-to-End Performance

The major goal of CBMM is to improve consistency and the debuggability of MM-related performance issues without degrading performance. Figure 5 shows the performance of each kernel with and without fragmentation. All results are normalized to Linux *without* fragmentation. On average, *without* fragmentation, CBMM has performance comparable to Linux and better than HawkEye. On average, *with* fragmentation, *CBMM is 7% and 30% faster than HawkEye and Linux*; in fact, it is only 10% slower than without fragmentation. With minimal tuning, on average, CBMM is *14% and 36% faster than HawkEye and Linux* under fragmentation.

Without fragmentation, CBMM matches or exceeds the performance of Linux or HawkEye for all workloads except `canneal`. For `canneal`, CBMM is 15% slower than Linux or HawkEye because our profiles underestimate the benefit of huge pages. For `mongodb`, CBMM is 14% faster than Linux and 7% faster than HawkEye because CBMM uses significantly more huge pages than either of them.

With fragmentation, CBMM outperforms Linux and/or HawkEye for all workloads except `mcf`. `mcf` uses too little memory to induce memory pressure; thus, CBMM overestimates the cost of huge pages and uses significantly fewer huge pages than Linux. In all other workloads, CBMM outperforms at least one of Linux or HawkEye, often by wide margins. In `mongodb`, `dc-mix`, `canneal`, and `memcached`, CBMM outperforms Linux by 10%, 34%, 35% and 80%, respectively, because its cost models allow it to adapt to a fragmented context, reflecting CBMM's focus on consistent behavior. Notably, this includes all of our datacenter workloads.

To demonstrate CBMM's benefit to performance debugging, we tune the performance of `mcf`, `canneal`, and `dc-mix` beyond the above results. In `mcf` and `dc-mix`, CBMM underestimates the benefit of huge pages, so we adjust the benefit upward in the respective profiles. We found that `canneal` exhibits a strong tradeoff between performance and page fault tail latency. As `canneal` is a non-interactive computational workload, we optimize for end-to-end performance by adjusting the profile to more aggressively allocate huge pages for the most import memory regions. After tuning, with fragmentation, `dc-mix` and `mcf` run 2% and 20% faster than without tuning but have no regression in tail latencies. `canneal` runs 17% faster than without tuning (45% faster than Linux) at the expense of some degradation in tail page fault latencies. In total, the tuning effort took less than a week, most of which was spent waiting for workloads to run.

Summary CBMM's has competitive performance with Linux/THP and HawkEye *and* better tail latency and more interpretable behavior. In most cases, CBMM matches or exceeds Linux's performance. Under fragmentation, CBMM often performs vastly better than Linux or HawkEye because of its focus on consistent behavior. Also, CBMM is easily debuggable and tunable by adjusting profiles and/or models.

5.4 Efficiency

Figure 6 shows the percentage of anonymous memory used by each workload that is backed by a huge page in CBMM, HawkEye, and Linux (with THP) with and without fragmentation preconditioning. Allocating huge pages to memory regions that do not need them wastes contiguous memory and promotion overheads and possibly bloats memory usage.

Generally, preloaded profiles drive CBMM's huge page usage, while HawkEye and Linux are more indiscriminate with huge page promotion. Usually, Linux attempts to use more huge pages than CBMM or HawkEye, often backing almost all memory with huge pages. HawkEye seems to use huge pages more efficiently than Linux, often achieving similar performance with much fewer huge pages. For most workloads, Linux still attempts to use huge pages under fragmentation, whereas CBMM and HawkEye do not, leading to significantly better tail latencies, and often better performance.

For `xz`, CBMM's profile allows it to promote only a small but important part of the address space, so it matches Linux's performance (and outperforms HawkEye) while using almost 80% fewer huge pages. For `mongodb`, CBMM outperforms Linux and HawkEye by using more huge pages in the absence of fragmentation and fewer in its presence, exemplifying CBMM's cost-awareness.

Summary Despite having the most consistent behavior and sometimes better performance, CBMM often uses significantly fewer huge pages than Linux or HawkEye. By being cost- and context-aware, CBMM is more targeted in its use of huge pages, though in some cases, our profiles underestimate the benefit of huge pages.

5.5 Generality

CBMM has benefits even when a profile is highly imprecise, primarily by avoiding the pathological behavior of Linux. We compare three versions of profiles: `standard` is as in Section 4.1, `perapp` assigns all memory regions in the workload the average benefit of enabling THP for the workload, and `shared` is shared between all workloads and assigns all memory regions the mean benefit of the `perapp` profiles.

In all cases, *CBMM with the simpler profiles outperformed Linux* with fragmentation, and in most cases, the performance differences between the three profiles are within 5%. The `perapp` and `shared` profiles outperforming the `standard` profiles slightly in some workloads. One exception is `mcf` in the

fragmented case, where both the `perapp` and `shared` profiles outperform the `standard` profile by 20%, similar to the tuned profile described above in Section 5.3 and for the same reason. There the `perapp` and `shared` profiles have minor regressions in page fault tail latencies compared to the `standard` profiles. The more general profiles have a larger impact on tail latency, where results are still better than Linux but not as good as the standard profile.

Summary More precise profiles improve CBMM's performance and tail latency, but imprecise profiles still have good results.

5.6 CBMM Models

We evaluate the contribution of each model in Section 4 by evaluating three configurations of CBMM: `huge` enables only the huge page promotion model, `async` additionally enables prezeroing, and `standard` enables all three models.

Each policy provides benefits at different times and in different ways. Huge pages alone (`huge`) captures most of the performance benefit of CBMM, performing on average the same as Linux without fragmentation, and 30% faster than Linux with fragmentation. Adding asynchronous pre-zeroing (`async`) reduces performance slightly on an unfragmented system, because there are ample free pages available so the prezeroing is wasted work. With fragmentation, though, prezeroing on average improves performance by 3.5% by making more huge pages available for use. In particular, with prezeroing `mongodb` performance improves by 17% from the extra huge pages. With or without fragmentation, `async` decreases page fault tail latency because huge pages become cheaper.

Eager paging does not provide a direct performance benefit, but enables larger contiguous allocations where hardware supports it [28, 36, 39, 42, 47]. Indeed, by comparing `standard` and `async`, we see that eager paging causes $< 5\%$ slowdown in almost all cases. To evaluate how well CBMM can make large allocations, we compare the number of eagerly allocated regions and peak memory usage of CBMM with eager paging against Linux with `MAP_POPULATE`, the `mmap` flag that eagerly maps a memory region. In all workloads, regardless of fragmentation, CBMM uses eager paging for nearly the entire working set of the workload and has $< 1\%$ memory bloat. Moreover, in all workloads except `dc-mix`, CBMM does not significantly increase the number of contiguous mapped memory ranges compared to `mmap`, making it possible to map the eager regions with contiguous memory. Meanwhile, thanks to its use of profiles, CBMM uses between 3%-48% less memory than `MAP_POPULATE` would use.

Summary CBMM's huge page model provides significant tail latency (and often performance) improvements. Asynchronous prezeroing enables more huge page usage under fragmentation, but has a modest cost on unfragmented systems. Eager paging has a modest performance cost but enables more contiguous memory allocation.

6 Related Work

Performance consistency at scale is a well-known problem [18] afflicting, among other systems, cluster computations [19] and distributed caching [10]. Redundancy is a common workaround [19]. MittOS uses deadline-aware kernel APIs to improve tail latency [25]. 0sim approaches the problem of scaling memory capacity, finding that existing algorithms may not scale well [35]. Like MittOS and 0sim, we seek to fix consistency issues rather than work around them.

Kwon et al. observe that current huge page support is "a hodge-podge of best-effort algorithms and spot fixes" [30]. They and others identify concerns, but devise more *ad hoc* algorithms to address them [13, 30, 37, 38, 47]. CBMM tames the increasing complexity of MM policy decisions by consolidating it in one place and reducing anomalous behavior.

VMware ESX Server explores MM techniques based on economic models by quantifying the value of idle memory and "taxing" processes for it [46]. Google and Facebook have idle memory tracking systems, too [14]. Google uses their idle tracking to centrally and empirically coordinate content migration to far-memory tiers (e.g., compressed memory) [31]. Google also profiles the lifetime of allocations to decrease memory fragmentation [34]. These approaches inspired our work; they use empirical measurements and MM-wide guiding principles to make MM decisions. Our work extends and generalizes this idea, rather than proposing a solution that applies to a particular MM decision, such as far-memory management. Sriraman et al. take a step in this direction by comprehensively profiling Facebook workloads and using the profiles to guide coarse-grained boot-time system tuning [41].

There is much prior work on asynchronous prezeroing of pages [8, 20, 21, 32, 37, 45]. Recent work observes that larger page sizes and non-temporal store instructions make prezeroing useful again [37]. We demonstrate the usefulness of our approach by quantifying zeroing costs and the prezeroing implementation, and integrating them into our prototype.

7 Conclusion

Modern computing needs are placing new demands on kernel MM. To meet these demands, kernel MM must begin to prioritize behavioral consistency and debuggability. We propose CBMM, a MM system that uses cost-benefit analysis to make policy decisions. Despite using relatively simple models in its cost-benefit estimation, CBMM's principled approach to MM allows matching the performance of existing systems while also improving system behavioral consistency. CBMM paves a way for kernel MM behavior to become less opaque, unlocking further performance and optimizations in the future... and making prezeroing great again!

References

[1] Borg Cluster Traces from Google. https://github.com/google/cluster-data.

[2] Database Installation Guide. https://docs.oracle.com/cd/E11882_01/install.112/e47689/pre_install.htm#LADBI1152.

[3] Disable Transparent Huge Pages (THP). https://docs.mongodb.com/manual/tutorial/transparent-huge-pages.

[4] Disabling Transparent Huge Pages (THP). https://docs.couchbase.com/server/current/install/thp-disable.html.

[5] SPEC CPU 2017 Benchmark Suite. https://www.spec.org/cpu2017/.

[6] Microsoft Azure Traces. https://github.com/Azure/AzurePublicDataset.

[7] Redis Latency Problems Troubleshooting. https://redis.io/topics/latency.

[8] Remove PG_ZERO and zeroidle (page-zeroing) entirely. https://news.ycombinator.com/item?id=12227874, August 2016.

[9] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Morgan & Claypool Publishers, 2013.

[10] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Mor Harchol-Balter, and Siddhartha Sen. RobinHood: Tail Latency-aware Caching – Dynamic Reallocating from Cache-rich to Cache-poor. In *Proceedings of the Thirteenth Conference on Operating Systems Design and Implementation*, OSDI'18, October 2018.

[11] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016.

[12] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[13] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an Experiment in Operating System Structure and State Management. In *Proceedings of the Fourteenth Symposium on Operating Systems Design and Implementation*, OSDI '20, 2020.

[14] Shakeel Butt, Suren Baghdasaryan, and Yu Zhao. Finding more DRAM. https://www.linuxplumbersconf.org/event/4/contributions/282/, September 2019.

[15] Richard W. Carr and John L. Hennessy. WSCLOCK – a Simple and Effective Algorithm for Virtual Memory Management. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, December 1981.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the First ACM Symposium on Cloud Computing*, SoCC '10, 2010.

[17] Dan Williams. Randomize free memory. https://lwn.net/Articles/767614/.

[18] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2), February 2013.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the Sixth Conference on Symposium on Operating Systems Design & Implementation*, OSDI'04, December 2004.

[20] Cort Dougan, Paul Mackerras, and Victor Yodaiken. Optimizing the idle task and other MMU tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, February 1999.

[21] Lars Eggert, Alan Cox, Cort Dougan, and Matt Dillon. Clearing Pages in the Idle Loop. https://www.mail-archive.com/freebsd-hackers@freebsd.org/msg13993.html, July 2000.

[22] Jason Evans. Scalable memory allocation using jemalloc, January 2011.

[23] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. BadgerTrap: a Tool to Instrument x86-64 TLB Misses. *ACM SIGARCH Computer Architecture News*, 42(2), 2014.

[24] Mel Gorman and Andy Whitcroft. The What, the Why and the Where to of Anti-fragmentation. In *Proceedings of the Linux Symposium*, volume 1, January 2006.

[25] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the Twenty-Sixth Symposium on Operating Systems Principles*, SOSP '17, 2017.

[26] John Wilkes. More Google Cluster Data. http://ai.googleblog.com/2011/11/more-google-cluster-data.html.

[27] Frans Kaashoek, Robert Morris, and Yandong Mao. Optimizing MapReduce for Multicore Architectures. Technical Report MIT-CSAIL-TR-2010-020, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 2010.

[28] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the Forty-Second Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[29] Paul Kocher, Jann Horn, Anders Fogh, and Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Fortieth IEEE Symposium on Security and Privacy*, S&P, 2019.

[30] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the Twelfth Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.

[31] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, 2019.

[32] Christopher Lameter. Increase page fault rate by prezeroing V1 [0/3]: Overview. https://lkml.org/lkml/2004/12/21/142, December 2004.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Twenty-Seventh USENIX Security Symposium*, USENIX Security, 2018.

[34] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[35] Mark Mansi and Michael M. Swift. 0sim: Preparing System Software for a World with Terabyte-scale Memories. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020.

[36] Michal Nazarewicz. A deep dive into CMA. https://lwn.net/Articles/486301/.

[37] Ashish Panwar, Sorav Bansal, and K. Gopinath. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, 2019.

[38] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, 2018.

[39] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: can you have it both ways? In *Proceedings of the Forty-Eighth International Symposium on Microarchitecture*, MICRO-48, December 2015.

[40] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *;login:*, 39(6), December 2014.

[41] A. Sriraman, A. Dhanotia, and T. F. Wenisch. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *Proceedings of the Forty-Sixth Annual International Symposium on Computer Architecture*, ISCA '19, 2019.

[42] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. *ACM SIGPLAN Notices*, 29(11), November 1994.

[43] Huangshi Tian, Yunchuan Zheng, and Wei Wang. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, 2019.

[44] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the Next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[45] Linus Torvalds. Page zeroing strategy. https://yarchive.net/comp/linux/page_zeroing_strategy.html, December 2000.

[46] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, OSDI'02, 2002.

[47] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation Ranger: Operating System Support for Contiguity-aware TLBs. In *Proceedings of the Forty-Sixth International Symposium on Computer Architecture*, ISCA, 2019.

[48] Wenbo Zhang. Why We Disable Linux's THP Feature for Databases. https://pingcap.com/blog/why-we-disable-linux-thp-feature-for-databases, December 2020.

[49] Weixi Zhu, Alan L. Cox, and Scott Rixner. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *2020 USENIX Annual Technical Conference*, ATC '20, 2020.