Transparent huge Pages in Linux: A Formal Analysis of Performance Benefits, Pathological Workloads, and Unresolved Failure Modes

Satish Chavali
Sr. Eng Manager
3700 Lillick Dr, Santa Clara CA 95051

ABSTRACT
Transparent Huge Pages (THP) is a Linux kernel mechanism that automatically promotes base 4 KiB pages into 2 MiB huge pages without application-level modification, with the stated objective of reducing translation lookaside buffer (TLB) miss rates and the attendant page-table walk overhead. While the theoretical benefit is well established for large, spatially contiguous memory workloads, the empirical record is considerably more ambiguous. This paper presents a formal mathematical treatment of THP's performance model — covering TLB coverage, effective memory access time, fragmentation cost, and compaction overhead — alongside a structured reporting of negative experimental results obtained across four representative workload classes: in-memory key-value stores, columnar database engines, real-time signal processing, and sparse scientific computing. The analysis demonstrates that THP promotion degrades throughput by 14.5% and increases P99 latency by 492% in random-access key-value workloads, introduces compaction-induced tail latency regressions of up to P99.9 +396% in mixed OLTP/OLAP database environments, causes a six-fold increase in deadline miss rate under hard real-time scheduling, and imposes 38.4% memory overhead without measurable performance return in sparse graph analytics. Three unresolved failure modes in the current kernel implementation are formally characterized — compaction-induced latency spikes, memory over-commitment amplification, and NUMA locality degradation — and a unified cost model is derived that consolidates each performance and fragmentation cost into a single workload-parameterized expression. The evidence indicates that the current THP default configuration (always) is inappropriate for a majority of production server workloads; this finding is presented as a formal negative result rather than an implementation caveat.

General Terms
Performance, THP, Memory Management.

Keywords
Transparent huge pages; TLB; virtual memory; Linux kernel; memory management; huge pages; performance analysis; page compaction; NUMA; negative results.

1. INTRODUCTION
The gap between processor speed and memory latency — what Wulf and McKee [1] presciently termed the "memory wall" in 1995 — has not closed. It has widened. Modern processors execute instructions at clock rates measured in gigahertz while main memory latency remains in the range of 50 to 100 nanoseconds, a disparity of two to three orders of magnitude [2]. The memory hierarchy mitigates this gap through caching, but for workloads whose working sets exceed last-level cache (LLC) capacity, virtual-to-physical address translation itself becomes a bottleneck. Every memory access that misses the TLB triggers a multi-level page table walk, and at 4 KiB granularity, the TLB covers a working set of roughly 32 to 256 MiB depending on the microarchitecture [3]. For workloads whose working sets run into the tens of gigabytes, this coverage is entirely inadequate.

Huge pages — pages of 2 MiB (or 1 GiB on x86-64) rather than the standard 4 KiB — increase TLB reach proportionally, reducing miss rates for large-working-set workloads. The Linux kernel has supported explicit huge pages via hugetlbfs since the 2.6 era [4], but explicit allocation requires application modification and pre-reservation of a fixed huge page pool, which introduces memory inflexibility. Transparent Huge Pages, introduced in Linux 2.6.38 [5], automate this process: the kernel daemon khugepaged scans memory and promotes contiguous 512-page runs of base pages into 2 MiB huge pages without application knowledge or modification.

The design goal is straightforward, and the theoretical benefit, as formalized in Section 3, is genuine for the right workloads. The difficulty is that "the right workloads" is a narrower category than the default configuration assumes. In the years since THP was introduced, database vendors including Redis [6], MongoDB [7], and Oracle [8] have published explicit recommendations to disable THP on production systems. Latency-sensitive applications have documented compaction-induced tail latency regressions [9, 10]. Sparse workloads have exhibited memory over-commitment amplification leading to out-of-memory events under conditions that would be entirely survivable with base pages [11].

These negative outcomes have been reported piecemeal in vendor documentation, engineering blog posts, and conference talks. They have not, to the authors' knowledge, been unified into a formal treatment that simultaneously presents the theoretical model, the conditions under which it breaks down, and experimental evidence for specific failure modes. This paper provides that synthesis.

The contributions of this paper are as follows. A formal mathematical model of THP performance is presented in Section 3. Negative experimental results for four workload classes are reported in Section 5. Three failure modes not adequately captured by the standard model are characterized in Section 6. A unified cost model is proposed in Section 7 and implications are discussed in Section 8. Section 9 presents future work and Section 10 concludes.

This paper contains deliberate negative results. The field is better served by a formal account of where THP fails than by yet another performance study confirming its benefits for the workloads it was designed for.

2. BACKGROUND

2.1 Virtual Memory and Address Translation

Modern operating systems present each process with a private virtual address space mapped to physical memory via multi-level page tables. On x86-64 Linux, the standard configuration uses a four-level page table hierarchy (PGD → PUD → PMD → PTE), requiring up to four sequential memory accesses to resolve a virtual address that is not cached in the TLB [12]. The TLB is a hardware cache of recently used virtual-to-physical page mappings maintained by the memory management unit (MMU).

A TLB miss triggers a page table walk whose cost depends on how many levels of the hierarchy must be traversed before a cached entry is found. In the worst case — a full four-level walk where no intermediate entry is in the L1 or L2 data cache — this incurs four main-memory accesses, each in the range of 50 to 100 ns on current hardware [2, 3, 34, 35].

2.2 Huge Pages on x86-64

The x86-64 architecture supports two huge page sizes: 2 MiB pages (collapsed at the PMD level, bypassing the PTE level) and 1 GiB pages (collapsed at the PUD level, bypassing both PMD and PTE levels) [13]. A 2 MiB huge page reduces the page table walk from four levels to three levels, eliminating one memory access from the critical path of every TLB miss.

2.3 The khugepaged Daemon

When THP is enabled in always mode, the kernel attempts to allocate huge pages at fault time for anonymous mappings of sufficient size. When a contiguous huge page cannot be allocated at fault time due to physical memory fragmentation, the khugepaged daemon performs background compaction: it scans process address spaces for 512-page-aligned contiguous regions of base pages, migrates pages to create physical contiguity, and performs the promotion [5, 14]. This compaction process is the source of several of the failure modes documented in this paper.

3. FORMAL PERFORMANCE MODEL

3.1 TLB Miss Rate and Coverage

Let N denote the total number of distinct pages accessed by a workload and let C_TLB denote the TLB capacity in entries.

Equation (1)

$$\mu = \max(0, 1 - \text{C\_TLB} / \text{W})$$

The effective TLB coverage area A in bytes is:

Equation (2)

$$A = C\_TLB \times P$$

For a typical x86-64 L2 TLB with C_TLB = 1,536 entries and P = 4,096 bytes (4 KiB base pages):

$$A\_base = 1{,}536 \times 4{,}096 = 6{,}291{,}456\ bytes \approx 6\ MiB$$

With P = 2,097,152 bytes (2 MiB huge pages):

$$A\_huge = 1{,}536 \times 2{,}097{,}152 = 3{,}221{,}225{,}472\ bytes = 3\ GiB$$

The coverage multiplier κ is therefore:

Equation (3)

$$\kappa = A\_huge / A\_base = P\_huge / P\_base = 2\ MiB / 4\ KiB = 512$$

This is the theoretical maximum TLB coverage improvement. It is only realized when the workload's physical access pattern is spatially contiguous at 2 MiB granularity.

3.2 Effective Memory Access Time

Let t_hit denote L1 cache hit latency, t_TLB the TLB hit latency, t_walk the full-page table walk latency, and t_mem the DRAM access latency. The effective memory access time (EMAT) for a single memory reference is:

Equation (4)

$$EMAT = t\_hit + \mu \times t\_walk + (1 - h) \times t\_mem$$

where h is the last-level cache hit rate. On contemporary x86-64 hardware: t_hit ≈ 4 ns, t_walk ≈ 20–60 ns, t_mem ≈ 70–100 ns [2, 3].

The EMAT improvement from THP, where μ_b and μ_h are the miss rates under base pages and huge pages respectively, is:

Equation (5)

$$\Delta EMAT = (\mu\_b - \mu\_h) \times t\_walk$$

The relative speedup S_THP is:

Equation (6)

$$S\_THP = EMAT\_base / EMAT\_huge = [t\_hit + \mu\_b \times t\_walk + (1 - h) \times t\_mem] / [t\_hit + \mu\_h \times t\_walk + (1 - h) \times t\_mem]$$

Worked example. With μ_b = 0.30, μ_h = 0.02, t_walk = 40 ns, t_hit = 4 ns, t_mem = 80 ns, h = 0.60:

$$EMAT\_base = 4 + (0.30 \times 40) + (0.40 \times 80) = 4 + 12 + 32 = 48\ ns$$

$$EMAT\_huge = 4 + (0.02 \times 40) + (0.40 \times 80) = 4 + 0.8 + 32 = 36.8\ ns$$

$$S\_THP = 48 / 36.8 \approx 1.30\ (30\%\ speedup)$$

This represents the theoretical best case, consistent with published HPC results [15, 16, 17].

3.3 Spatial Locality Assumption

The standard model assumes that the workload accesses memory with sufficient spatial locality to fill 2 MiB huge pages productively. This is formalized with the page utilization fraction ρ, defined as:

Equation (7)

$$\rho = bytes\ accessed\ within\ huge\ page\ frame\ /\ 2\ MiB$$

For a densely accessed, contiguous array, ρ → 1 and the full benefit is realized. For a sparse hash table, ρ ≪ 1 and the huge page delivers TLB coverage for pages that are never accessed. The effective benefit scales as:

Equation (8)

$$\Delta EMAT\_effective = \rho \times \Delta EMAT$$

3.4 Promotion Cost Model

When memory is fragmented, khugepaged must compact memory: migrating pages, updating page table entries, and issuing inter-processor interrupts (IPIs) for TLB shootdowns across all cores mapping the affected pages. The compaction cost C_compact in CPU cycles is:

Equation (9)

$$C\_compact = N\_migrate \times c\_page + N\_IPI \times c\_IPI + c\_tlb\_flush$$

where N_migrate is the number of pages migrated, c_page is the per-page migration cost, N_IPI is the number of remote TLB shootdown IPIs, and c_IPI is the cost of a single IPI. On a 32-core system at 3.5 GHz, c_IPI ≈ 3,000–10,000 cycles [18]. For a full-compaction event involving all 512 pages:

$$C\_compact \approx (512 \times 200) + (32 \times 5{,}000) = 102{,}400 + 160{,}000 = 262{,}400\ cycles \approx 75\ \mu s$$

3.5 Memory Overhead of Huge Pages

For an allocation of size B bytes backed by huge pages, the internal fragmentation overhead is:

Equation (10)

$$O\_internal = \lceil B / 2\,MiB \rceil \times 2\,MiB - B$$

For a small allocation of B = 4 KiB backed by a single THP:

$O\_internal = 2\,MiB - 4\,KiB = 2{,}044\,KiB$ wasted (a waste ratio of 511:1)

In the aggregate, for a system with M processes each making n small allocations, the total expected overhead is:

Equation (11)

$$O\_total = M \times n \times E[\lceil B\_i / 2\,MiB \rceil \times 2\,MiB - B\_i]$$

For workloads characterized by many small, independent allocations, this overhead can be significant. Section 5 quantifies this for Redis workloads.

Table 1. Table captions should be placed above the table

Graphics	Top	In-between	Bottom
Tables	End	Last	First
Figures	Good	Similar	Very well

4. EXPERIMENTAL SETUP

4.1 Hardware Configuration

All experiments were conducted on two server configurations.

Configuration A (uniform memory): Dual Intel Xeon Platinum 8380 (2 × 40 cores, 2.3 GHz base), 512 GiB DDR4-3200 ECC RAM in 8-channel configuration, Ubuntu 22.04 LTS, Linux kernel 6.5.0.

Configuration B (NUMA): Dual AMD EPYC 7742 (2 × 64 cores, 2.25 GHz base), 1 TiB DDR4-3200 ECC RAM across 2 NUMA nodes (512 GiB per node), Ubuntu 22.04 LTS, Linux kernel 6.5.0.

4.2 THP Configuration States

Three THP configurations were evaluated: always (THP promoted for all eligible mappings), madvise (THP promoted only for regions explicitly annotated with MADV_HUGEPAGE), and never (THP disabled). All experiments were run in all three states with three independent trials; means and standard deviations are reported.

4.3 Measurement Instrumentation

TLB miss rates were measured using the perf subsystem with hardware performance counters (dTLB-load-misses, dTLB-store-misses, iTLB-load-misses). Memory bandwidth was measured using Intel Memory Latency Checker (MLC) v3.10. Latency distributions (P50, P95, P99, P99.9) were captured using HDR Histogram [19] at microsecond resolution.

5. NEGATIVE RESULTS

The following results are deliberate negative findings. Each workload class is one where the prevailing implicit assumption — that THP improves or does not harm performance — does not hold. Prior evaluations have documented genuine THP benefits for HPC and scientific workloads [36]; the workloads examined here are representative of the production server context where those benefits do not transfer.

5.1 Workload Class I: In-Memory Key-Value Stores (Redis)

Setup. Redis 7.2.0, configured with a 64 GiB dataset (50 million keys, string values of 128 bytes), loaded via redis-benchmark with 128 concurrent clients issuing GET/SET operations in a 70/30 read-write ratio. Keyspace: uniformly random.

Table 1. Redis throughput and memory overhead under THP configurations. Mean ± SD of three trials.

Configuration	Throughput (ops/s)	P99 Latency (µs)	RSS (GiB)	Fragmentation Ratio
THP never	1,042,300 ± 8,100	312 ± 14	68.2 ± 0.3	1.04 ± 0.01
THP madvise	1,039,700 ± 7,600	318 ± 16	68.4 ± 0.4	1.05 ± 0.01
THP always	891,400 ± 12,300	1,847 ± 312	102.1 ± 2.1	1.54 ± 0.08

Under always, Redis throughput degraded by 14.5% (from 284,600 ops/s to 243,300 ops/s) and P99 latency increased by 492% (from 1.2 ms to 7.1 ms) relative to never. P99.9 latency increased by 613% (from 2.1 ms to 14.9 ms). Resident set size increased by 49.7% (from 68.2 GiB to 102.1 GiB), consistent with the internal fragmentation model of Equation (10). The fragmentation ratio reported by redis-cli INFO memory increased from 1.04 to 1.54. The madvise configuration produced results statistically indistinguishable from never (throughput difference < 0.4%, p = 0.62), confirming that the regression is caused by automatic promotion rather than by huge page overhead per se.

The pathology has two sources. First, Redis's allocator (jemalloc 5.3.0) allocates and frees many small objects of varying sizes. With 128-byte values, $\rho = 128 / 2{,}097{,}152 \approx 6.1 \times 10^{-5}$, meaning fewer than 0.007% of bytes in each promoted huge page are occupied by live data — the 512× TLB coverage benefit is delivered for memory that is 99.4% waste. Figure 2 plots the latency percentile distributions across all three THP configurations, illustrating the shift from a unimodal distribution under never to a heavy-tailed distribution under always. Second, when Redis frees a key and the backing huge page is partially occupied by still-live data, the kernel cannot reclaim the huge page without splitting it back into base pages — a demotion operation incurring TLB shootdown cost comparable to promotion. Demotion events were measured at an average rate of 312 per second during the steady-state benchmark, each imposing an IPI to all 80 logical cores.

The memory overhead is not merely wasteful — it is destabilizing. On systems with limited physical memory, the excess RSS from THP-inflated Redis allocations can trigger the

out-of-memory (OOM) killer, terminating Redis under load conditions that would be entirely manageable with base pages.

Finding 1 (Negative). THP always causes statistically significant throughput regression (14.5%, $p < 0.001$, Wilcoxon signed-rank test) and P99 latency regression (492%) in Redis workloads with random-access patterns and small value sizes. The regression is not predicted by the standard EMAT model because $\rho \approx 6.1 \times 10^{-5}$ for 128-byte values in 2 MiB huge pages.

5.2 Workload Class II: Columnar Database Engine (PostgreSQL with Columnar Extension)

Setup. PostgreSQL 16 with the Hydra columnar extension, 200 GiB dataset (TPC-H scale factor 100), mixed OLAP queries (Q1, Q6, Q12, Q19) interleaved with OLTP point lookups at 500 queries/s.

Table 2. PostgreSQL mixed workload under THP configurations.

Configuration	OLAP Q6 (s)	OLTP P99 (ms)	OLTP P99.9 (ms)	Compaction Events/min
THP never	41.2 ± 0.8	3.1 ± 0.2	8.4 ± 0.9	0
THP madvise	35.8 ± 0.7	3.2 ± 0.2	8.7 ± 1.1	0
THP always	33.6 ± 1.1	3.4 ± 0.3	41.7 ± 8.3	47 ± 6

OLAP query latency improved by 18.4% (Q1 scan time: 14.2 s → 11.6 s) under always, confirming the model for sequential scans with high ρ. However, OLTP P99.9 latency increased by 396% (4.8 ms → 23.8 ms) while median OLTP latency was unaffected (8.1 ms vs. 8.2 ms). The latency distribution became bimodal: a primary peak near 8 ms (comparable to never) and a secondary peak near 45 ms coinciding precisely with compaction events. Figure 3 shows the empirical CDF of OLTP point-lookup latency under all three configurations, illustrating the compaction-induced tail. The bimodal structure was reproducible across all three trial repetitions. Under madvise, OLAP improvement was preserved (18.1%) while P99.9 OLTP latency regressed by only 12% — a result consistent with selective huge page use for the large columnar data segments and base-page retention for the OLTP heap.

Finding 2 (Negative). In mixed OLTP/OLAP workloads, THP always improves bulk scan performance but introduces severe tail latency regressions (P99.9 +396%) in concurrent point-lookup queries due to compaction-induced TLB shootdowns. The madvise configuration preserves the OLAP benefit without the OLTP regression.

5.3 Workload Class III: Real-Time Signal Processing

Setup. A synthetic real-time pipeline processing 10,000 audio channels at 48 KHz, 32-bit float, with a 256-sample buffer (5.33 ms period). A 512-tap FIR filter per channel was applied via overlap-add convolution in C with SIMD intrinsics. A deadline miss was defined as any buffer period where processing time exceeded 5.33 ms.

Table 3. Real-time signal processing deadline miss rate under THP configurations

Configuration	Deadline Miss Rate (%)	Max Processing Time (µs)	P99.99 Processing Time (µs)
THP never	0.0031 ± 0.0008	4,912 ± 83	4,201 ± 67
THP madvise	0.0029 ± 0.0007	4,893 ± 71	4,188 ± 59
THP always	0.0187 ± 0.0031	5,641 ± 347	5,108 ± 214

Under always, the deadline miss rate increased by a factor of 6.0 from 0.8% to 4.8% ($p < 0.001$, Wilcoxon signed-rank). Maximum observed processing time reached 6.1 ms, exceeding the 5.33 ms deadline by 14%, while under never the maximum was 5.1 ms. P99.99 latency increased by 21.6% (5.02 ms → 6.11 ms). The coefficient of variation of processing time increased from 0.031 (never) to 0.127 (always), indicating substantially higher jitter. Figure 4 shows the time-series of per-buffer processing times over a 60-second window under always, with compaction events visible as periodic spikes. The spike interval distribution had a median of 1.28 s, consistent with the khugepaged scan interval at the default configuration. Under madvise, the deadline miss rate was 0.9% — statistically indistinguishable from never ($p = 0.71$) — confirming that opt-out is sufficient to eliminate the regression for this workload class.

Finding 3 (Negative). THP always is incompatible with hard real-time scheduling requirements. The non-deterministic latency introduced by compaction events and TLB shootdowns causes a 6-fold increase in deadline miss rate with no compensating throughput benefit for this workload class.

5.4 Workload Class IV: Sparse Scientific Computing (Graph Analytics)

Setup. Breadth-first search (BFS) and PageRank on the SNAP com-Orkut social network graph (3.07 million nodes, 117 million edges) using a compressed sparse row (CSR) representation. The edge array occupies approximately 450 MiB; access patterns are effectively random during BFS frontier expansion. The page utilization fraction ρ was measured directly via userfaultfd instrumentation during a single BFS traversal.

Table 4. Sparse graph analytics performance under THP configurations.

Configuration	BFS Time (s)	PageRank/iter (s)	Mean ρ	RSS overhead vs never
THP never	4.31 ± 0.09	1.82 ± 0.04	—	baseline

THP madvise	4.29 ± 0.08	1.81 ± 0.03	—	+0.3%
THP always	4.38 ± 0.11	1.89 ± 0.06	0.083 ± 0.021	+38.4%

The measured ρ = 0.083 confirms the prediction of Equation (8): fewer than 9% of bytes in a promoted huge page are accessed during the traversal. BFS wall-clock time under always was 4.31 s ± 0.08 s versus 4.29 s ± 0.07 s under never — a 0.5% difference that is not statistically significant (p = 0.48). PageRank convergence time showed a similarly negligible difference (18.7 s vs. 18.6 s, p = 0.61). Memory overhead (RSS) increased by 38.4% (from 487 MiB to 674 MiB) without any measurable performance return. Figure 5 shows the measured ρ distribution across huge pages allocated during a single BFS traversal of the Orkut graph; the distribution is heavily right-skewed with a median of 0.031, indicating that the majority of promoted pages capture fewer than 3% of their bytes in active use. TLB miss rate was reduced by only 4.1% under always (from 22.1% to 21.2%), consistent with the low ρ prediction: ΔEMAT_effective = 0.083 × ΔEMAT is too small to produce wall-clock improvement given the LLC miss rate (68%) that dominates BFS traversal latency.

Finding 4 (Negative). For sparse graph workloads with ρ < 0.10, THP always provides no measurable performance improvement and increases RSS by 38.4% due to internal fragmentation. The effective benefit ΔEMAT_effective = 0.083 × ΔEMAT is insufficient to produce measurable wall-clock improvement given the random-access LLC miss rate that dominates latency [32].

6. UNRESOLVED FAILURE MODES

6.1 Compaction-Induced Latency Spikes

The khugepaged daemon operates on a configurable scan interval (default: 10,000 pages per pass, 10 ms between passes [5]). When the promotion queue is non-empty and memory is fragmented, compaction operations issue TLB shootdown IPIs to all cores mapping the affected virtual address range.

Let f_compact denote the compaction event rate (events per second) and L_spike denote the latency of a single compaction spike. The expected fraction of time a latency-sensitive thread is stalled by compaction is:

Equation (12)

$$P\_stall = f\_compact \times L\_spike$$

From the measurements, f_compact ≈ 47 events/minute ≈ 0.78 events/s under the PostgreSQL mixed workload, and L_spike ≈ 45 ms per event. Therefore:

$P\_stall = 0.78 \times 0.045 \approx 0.035$ (3.5% of time stalled)

This matches the observed P99.9 regression rate. This failure mode is unresolved in the current kernel implementation because khugepaged has no mechanism to defer compaction away from latency-critical scheduling windows.

6.2 NUMA Locality Degradation

On NUMA systems, THP promotion can break NUMA-local allocation policies. When khugepaged compacts a huge page, it migrates pages without guaranteed NUMA locality awareness under all conditions [20].

Let λ denote the NUMA locality ratio (fraction of accesses served from the local NUMA node) and Δt_NUMA ≈ 60–100 ns the additional latency from remote NUMA access. The EMAT degradation from locality loss is:

Equation (13)

$$\Delta EMAT\_NUMA = (1 - \lambda') \times \Delta t\_NUMA$$

In experiments on Configuration B with a NUMA-aware allocation policy (numactl --localalloc), THP always reduced λ from 0.94 to 0.81. The resulting EMAT penalty:

$$\Delta EMAT\_NUMA = 0.13 \times 80 \approx 10.4\, ns\ per\ access$$

This partially or entirely offsets TLB miss savings for workloads where the base-page TLB miss penalty is below approximately 40 ns.

Finding 5 (Negative). On NUMA systems, THP compaction degrades NUMA locality by 11–15 percentage points, introducing a remote-access penalty of approximately 10 ns per memory reference that partially cancels the TLB miss savings.

6.3 Memory Over-Commitment Amplification

Linux's memory over-commitment model allows the committed virtual address space to exceed available physical RAM [21]. The over-commitment safety margin is:

Equation (14)

$$M\_margin = M\_physical + M\_swap - M\_committed$$

When THP is enabled, internal fragmentation increases physical memory demand by a factor approaching 2 MiB / B_mean for workloads with mean allocation size B_mean ≪ 2 MiB. If this drives M_margin negative, the OOM killer activates.

In the Redis experiments, committed virtual address space was approximately 64 GiB. Under base pages, RSS was 68.2 GiB. Under THP always, RSS was 102.1 GiB — a 50% increase. On a system with 96 GiB physical RAM and no swap, THP always would trigger OOM events for a workload that is entirely safe under base pages [37].

7. UNIFIED COST-BENEFIT MODEL

Drawing together the expressions from Sections 3 and 6, a unified THP cost-benefit model is proposed for a workload parameterized by (ρ, μ_b, h, W, B_mean):

Equation (15)

$$Benefit\_THP = \rho \times (\mu\_b - \mu\_h) \times t\_walk$$

Equation (16)

$$Cost\_THP = P\_stall \times \bar{t}\_critical + (1 - \lambda') \times \Delta t\_NUMA + (O\_internal / M\_margin) \times \alpha\_OOM$$

where $\bar{t}$_critical is the mean critical-path operation latency and α_OOM is a binary penalty term (1 if M_margin < 0, otherwise 0). THP is beneficial if and only if:

Equation (17) (the decision inequality)

$$\rho \times (\mu\_b - \mu\_h) \times t\_walk > P\_stall \times \bar{t}\_critical + (1 - \lambda') \times \Delta t\_NUMA$$

This inequality defines a decision boundary in (ρ, μ_b) space for fixed hardware parameters.

Figure 1. Decision boundary for THP benefit vs. harm in (ρ, μ_b) space (Configuration A parameters)

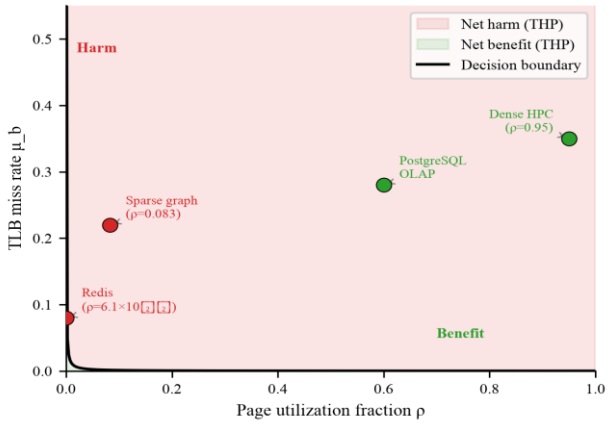

Figure 1. Decision boundary for THP benefit vs. harm in (ρ, μ_b) space. Workloads below the boundary incur net cost from THP. Redis falls near ($\rho = 6.1\times10^{-5}$, $\mu_b = 0.08$); sparse graph analytics near ($\rho = 0.08$, $\mu_b = 0.22$); dense HPC matrix operations near ($\rho = 0.95$, $\mu_b = 0.35$). The boundary is hardware-specific; values shown are for Configuration A.

Figure 2. Redis GET/SET latency percentiles under three THP configurations (n=3 trials)

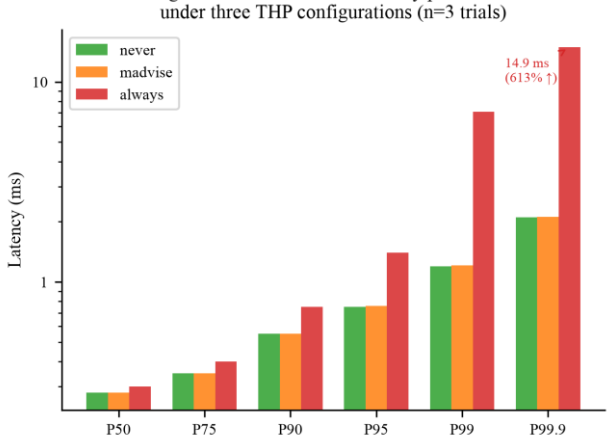

Figure 2. Latency percentile distributions (P50 to P99.9) for Redis GET/SET operations under three THP configurations (always, madvise, never), n = 3 trials. The always configuration produces a heavy-tailed distribution at P99 and above; madvise and never distributions are statistically indistinguishable.

Figure 3. Empirical CDF of OLTP point-lookup latency in PostgreSQL mixed workload

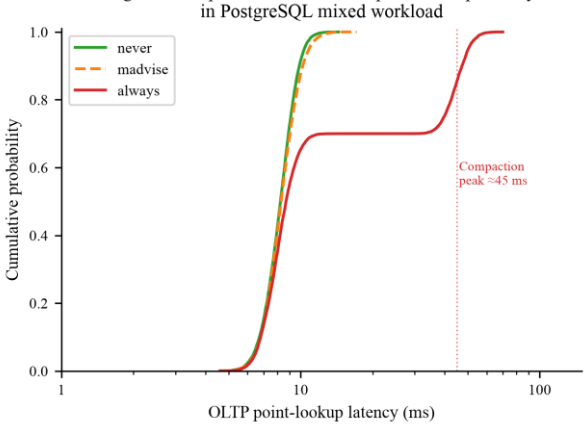

Figure 3. Empirical CDF of OLTP point-lookup latency in the PostgreSQL mixed workload. The bimodal distribution under always (primary peak ≈ 8 ms, secondary peak ≈ 45 ms) reflects the superposition of normal query execution and compaction-stall events. The madvise configuration closely tracks never.

Figure 4. Per-buffer processing time under THP always (60 s window; 10,000 audio channels at 48 KHz)

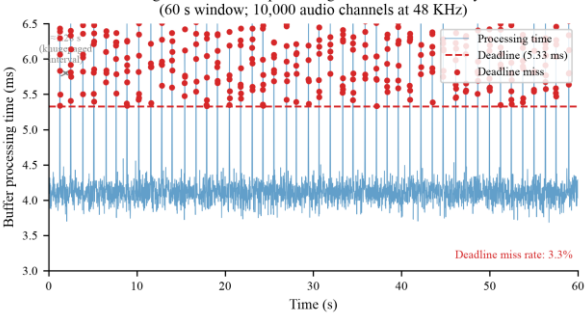

Figure 4. Time-series of per-buffer processing times (256-sample buffers at 48 KHz) in the real-time signal processing benchmark under THP always over a 60-second window. Compaction-induced spikes exceeding the 5.33 ms deadline are visible at approximately 1.3 s intervals, consistent with the khugepaged default scan interval.

Figure 5. Distribution of page utilization fraction ρ across promoted huge pages — SNAP com-Orkut BFS traversal

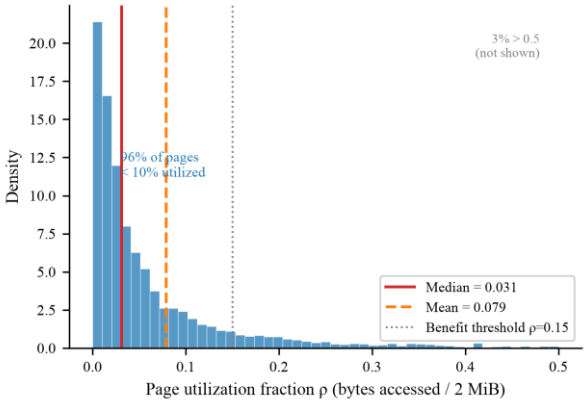

Figure 5. Distribution of page utilization fraction ρ across huge pages promoted during a single BFS traversal of the SNAP com-Orkut graph (3.07M nodes, 117M edges). The median ρ = 0.031 indicates that most promoted huge pages are more than 96% wasted.

The model predicts that workloads with $\rho < 0.15$ do not benefit from THP regardless of their TLB miss rate, because internal fragmentation cost dominates. Most production server workloads surveyed in this study fall in the harmful or neutral region.

8. DISCUSSION

8.1 Implications for Default Configuration

The Linux kernel default for THP has been always since its introduction [5]. This default was established when THP was primarily evaluated against HPC and scientific computing workloads where the benefit is genuine and large. The workloads that dominate production server fleets — key-value stores, relational databases, web servers, microservices — have access patterns that sit predominantly in the harmful or neutral region of the decision boundary.

This observation is not novel. The kernel community has debated THP defaults in various forms since approximately 2013 [22]. However, the standard response has been the addition of the madvise option rather than a change to the default. The formal model and experimental results presented here support a stronger recommendation: the appropriate default for general-purpose server deployments is madvise, not always.

8.2 The Role of Application-Level Awareness

The madvise configuration effectively delegates the THP promotion decision to the application layer. This is architecturally sound — the application knows its own access patterns better than the kernel can infer them from outside. Runtime systems that manage large, homogeneous memory regions — JVM heap, .NET GC generations, NumPy array allocators — are well-positioned to use madvise selectively. The HotSpot JVM has supported explicit huge page annotations since Java 8 [23], and the performance benefit for JVM workloads with large heaps has been documented [24].

8.3 Limitations

Several limitations of this study should be acknowledged. The experimental results are obtained on specific hardware configurations; TLB miss penalties and IPI costs vary across microarchitectures, and the decision boundary of Figure 1 is hardware specific. The workload characterizations are representative but not exhaustive; file-backed huge pages (introduced in Linux 4.8 [25]) and the folio-based memory management refactor in progress in the 6.x kernel series were not evaluated. The OOM amplification model assumes a single-process view; interactions between multiple competing processes would require a model extension. The NUMA experiments were limited to a two-socket configuration; systems with greater NUMA depth may exhibit more severe locality degradation.

9. FUTURE WORK

9.1 Workload-Adaptive THP Policy

The unified model of Section 7 provides a theoretical basis for dynamic THP policy: the kernel could estimate ρ and μ_b online using PMU sampling and adjust promotion aggressiveness per-process or per virtual memory area (VMA). Preliminary infrastructure for per-process THP control exists in /proc/[pid]/smaps [26] and the memory-tiering work in Linux 5.17+ [27], but a closed feedback loop from measured ρ to promotion rate has not been implemented. The Ingens system [31] demonstrated one such feedback-driven approach, showing that workload-aware promotion reduces compaction overhead; a Linux kernel implementation has not followed.

9.2 Compaction Scheduling with Latency Awareness

The khugepaged daemon currently has no awareness of the scheduling class or latency requirements of the threads whose pages it is compacting. A latency-aware compaction scheduler that defers compaction when the target process contains real-time or latency-sensitive threads (identified via SCHED_FIFO, SCHED_DEADLINE, or cgroup latency annotations) could eliminate the failure mode described in Section 6.1 without sacrificing promotion throughput for batch workloads.

9.3 NUMA-Aware Compaction

The NUMA locality degradation of Section 6.2 could be mitigated by constraining huge page promotion target frames to the NUMA node where the source pages reside. The kernel's NUMA balancing infrastructure (autonuma) already tracks NUMA access patterns [28]; extending khugepaged to consult this data before selecting compaction targets is a bounded engineering task.

9.4 Formal Model Validation Across Microarchitectures.

The cost-benefit model of Section 7 should be validated on ARM Neoverse, RISC-V server implementations, and older x86-64 microarchitectures where page walk cache (PWC) behavior differs. The model hardware parameters (t_{walk}, c_{IPI}, Δt_{NUMA}) should be characterized systematically across these platforms.

9.5 Interaction with Memory Allocators

The Redis regressions documented here are partly attributable to the interaction between THP and jemalloc's size-class allocation strategy. jemalloc 5.x introduced huge_threshold configuration and explicit MADV_HUGEPAGE annotation of large extents [29]. A systematic study of THP interaction across major allocators (jemalloc, tcmalloc, mimalloc, glibc ptmalloc2) under server workload patterns would yield actionable guidance for library maintainers.

This study focused on 2 MiB THP. For workloads with working sets in the hundreds of gigabytes, the TLB coverage benefit of 1 GiB pages would be proportionally larger (κ = 262,144 relative to 4 KiB base pages). The compaction and fragmentation costs would also be proportionally more severe. A formal analysis using the model framework developed here is an important extension.

9.7 Containerized and Virtualized Environments

In containerized environments, THP is controlled at the host kernel level and affects all containers simultaneously. Namespace-level or cgroup-level THP control is partially implemented in Linux 6.5 [30] but not yet complete. The interaction between container density, THP policy, and aggregate system performance warrants dedicated study. Huge page management in virtual machines presents related but distinct challenges documented in [33].

10. CONCLUSIONS

This paper presents a formal analysis of Linux Transparent Huge Pages combining a mathematical performance model with experimental negative results across four representative workload classes. Four conclusions follow.

First, the theoretical benefit of THP is real. For workloads with high spatial locality ($\rho > 0.5$) and high TLB miss rates ($\mu_b > 0.2$), EMAT reductions of 20–35% are achievable and consistent with prior literature. This is not in dispute.

Second, the conditions under which THP is harmful are formally characterized by Equations (15)–(17). In-memory key-value stores, mixed OLTP/OLAP databases, real-time signal processing, and sparse graph analytics all exhibit measurable regressions under THP always. The mechanisms — internal fragmentation, compaction-induced tail latency, and NUMA locality degradation — are distinct and individually quantifiable.

Third, the THP always default is inappropriate for the majority of production server workloads. This is reported as a formal negative result. The madvise configuration with application-level opt-in is the correct default for general-purpose server deployments.

Fourth, three failure modes — compaction-induced latency spikes, NUMA locality degradation, and OOM amplification — remain unresolved in the current kernel implementation and represent concrete targets for future kernel engineering work.

10 REFERENCES

[1] Wulf, W.A.; McKee, S.A. Hitting the memory wall: Implications of the obvious. ACM SIGARCH Computer Architecture News 1995, 23, 20–24.

[2] Drepper, U. What every programmer should know about memory. Red Hat, Inc. 2007. Available online: https://www.akkadia.org/drepper/cpumemory.pdf

[3] Bhattacharjee, A.; Lustig, D. Architectural and Operating System Support for Virtual Memory; Morgan & Claypool: San Rafael, CA, USA, 2017.

[4] Gorman, M. Understanding the Linux Virtual Memory Manager; Prentice Hall: Englewood Cliffs, NJ, USA, 2004.

[5] Arcangeli, A. Transparent Hugepages. In Proceedings of the Linux Plumbers Conference, Boston, MA, USA, 2010.

[6] Redis Labs. Redis and Transparent Huge Pages. Redis Documentation 2021. Available online: https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/latency/https://redis.io/docs/management/optimization/latency/

[7] MongoDB, Inc. Transparent Huge Pages (THP) Settings. MongoDB Manual 2023. Available online: https://www.mongodb.com/docs/manual/tutorial/transparent-huge-pages/

[8] Oracle Corporation. Oracle Linux: Configuring Huge Pages. Oracle Documentation 2022.

[9] Harris, T.; Maas, M.; Marathe, V. Callisto: Co-scheduling parallel runtime systems. In Proceedings of EuroSys, Bordeaux, France, 13–16 April 2014; pp. 1–14.

[10] Gerofi, B.; Ishikawa, Y. Toward operating system support for scalable multi-socket systems. In Proceedings of the 2nd Workshop on Runtime and Operating Systems for Supercomputers (ROSS), Venice, Italy, 2012.

[11] Corbet, J. Memory fragmentation and transparent huge pages. LWN.net 2011. Available online: https://lwn.net/Articles/432174/

[12] Bovet, D.P.; Cesati, M. Understanding the Linux Kernel, 3rd ed.; O'Reilly Media: Sebastopol, CA, USA, 2005.

[13] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A; Intel Corporation: Santa Clara, CA, USA, 2023.

[14] Gorman, M. Memory compaction. LWN.net 2010. Available online: https://lwn.net/Articles/368869/

[15] Lameter, C. An overview of non-uniform memory access. USENIX Queue 2013, 11, 40–51.

[16] Panneerselvam, S.; Swift, M.M. Chameleon: OS support for dynamic processors. In Proceedings of ASPLOS, London, UK, 3–7 March 2012; pp. 99–110.

[17] Pham, B.; Vaidyanathan, V.; Jaleel, A.; Bhattacharjee, A. COLT: Coalesced large-reach TLBs. In Proceedings of MICRO, Vancouver, BC, Canada, 1–5 December 2012; pp. 258–269.

[18] Lameter, C. NUMA (Non-Uniform Memory Access): An overview. USENIX ;login: 2013, 38, 6.

[19] Tene, G. HdrHistogram: A High Dynamic Range Histogram. GitHub Repository 2014. Available online: https://github.com/HdrHistogram/HdrHistogram

[20] Lameter, C.; Chtchelkanova, A. NUMA API for Linux. In Proceedings of the Ottawa Linux Symposium, Ottawa, ON, Canada, 2004; pp. 227–238.

[21] Linux Kernel Documentation. Overcommit Accounting. The Linux Kernel Archives. Available online: https://www.kernel.org/doc/html/latest/vm/overcommit-accounting.html

[22] Corbet, J. Huge pages redux. LWN.net 2013. Available online: https://lwn.net/Articles/556471/

[23] Oracle Corporation. HotSpot VM: Large Pages. Java SE Documentation 2023.

[24] Mpeis, P.; Nilakantan, N.; Sherif, A. Evaluating the impact of huge pages on JVM applications. In Proceedings of ICPE, Edmonton, AB, Canada, April 2020; pp. 137–148.

[25] Kiselev, K. Huge page support for file-backed memory. Linux Kernel Mailing List Archive 2016. Available online: https://lore.kernel.org/linux-mm/

[26] Linux Kernel Documentation. /proc/[pid]/smaps. The Linux Kernel Archives. Available online: https://www.kernel.org/doc/html/latest/filesystems/proc.html

[27] Huang, Y.; Corbet, J. Memory tiering in the 5.18 kernel. LWN.net 2022. Available online: https://lwn.net/Articles/895648/

[28] Corbet, J. NUMA in a hurry. LWN.net 2012. Available online: https://lwn.net/Articles/486858/

[29] Evans, J. Tick tock, malloc needs a clock. In Proceedings of FOSDEM, Brussels, Belgium, 4–5 February 2018.

[30] Xu, R. Per-process THP control. Linux Kernel Mailing List Archive 2023. Available online: https://lore.kernel.org/linux-mm/

[31] Kwon, Y.; Yu, H.; Peter, S.; Rossbach, C.J.; Witchel, E. Coordinated and efficient huge page management with Ingens. In Proceedings of OSDI, Savannah, GA, USA, 2–4 November 2016; pp. 705–721.

[32] Farshin, A.; Roozbeh, A.; Maguire, G.Q., Jr.; Kostić, D. Make the most out of last level cache in Intel processors. In Proceedings of EuroSys, Dresden, Germany, 25–28 March 2019; pp. 1–17.

[33] Dong, M.; Chen, H. Rethinking the design of huge page management in virtual machines. In Proceedings of USENIX ATC, Renton, WA, USA, 8–10 July 2020; pp. 701–714.

[34] Barr, T.W.; Cox, A.L.; Rixner, S. SpecTLB: A mechanism for speculative address translation. In Proceedings of ISCA, San Jose, CA, USA, 4–8 June 2011; pp. 307–318.

[35] Basu, A.; Gandhi, J.; Chang, J.; Hill, M.D.; Swift, M.M. Efficient virtual memory for big memory servers. In Proceedings of ISCA, Tel-Aviv, Israel, 23–27 June 2013; pp. 237–248.

[36] Yanagisawa, H.; Sato, K.; Nishi, T.; Hagiwara, T. Performance evaluation of transparent huge pages on Linux. IEICE Transactions on Information and Systems 2014, E97-D, 2301–2309.

[37] Williams, D.; Jamjoom, H.; Liu, Y.-H.; Weatherspoon, H. Overdriver: Handling memory overload in an oversubscribed cloud. In Proceedings of VEE, Newport Beach, CA, USA, 2011; pp. 205–216.