Colored Huge Pages: A Hardware-Software Approach for Enhanced Isolation and Performance

Georgios-Alexandros Kostas
University of Athens, Greece
alexk@di.uoa.gr

Dimitris Gizopoulos
University of Athens, Greece
dgizop@di.uoa.gr

Vasileios Karakostas
University of Athens, Greece
vkarakos@di.uoa.gr

***Abstract*—Multicore CPUs typically share the Last-Level Cache (LLC) across cores, leading to interference between co-executing workloads with significant performance and security implications. Page coloring has emerged as an effective software mechanism for LLC partitioning. Simultaneously, virtual memory enables fundamental abstractions, but incurs increasing performance overheads due to address translation. Huge pages alleviate this issue by expanding TLB reach, thereby reducing TLB misses and the associated costly page table walks. However, these two techniques are considered mutually exclusive, since huge pages span all LLC sets, precluding coloring.**

In this paper we introduce Colored Huge Pages (CHP), a hardware-software co-design that enables the simultaneous use of page coloring and huge pages. By distributing the physical frames of a virtually contiguous huge page across physical memory in a predictable strided pattern, our design allows coloring of the individual pages while preserving the TLB reach and translation efficiency of conventional huge pages. On the software side, we modify the OS allocator to construct colored huge pages by extracting appropriately colored pages from larger physical blocks and caching leftover mappings for future use. On the hardware side, we extend the L2 TLB to efficiently translate these mappings by leveraging their regular structure. We implement our approach in a recent Linux kernel and evaluate it using memory intensive workloads. CHP mitigates LLC contention and address translation overheads, improving performance by 33.7% compared to using 4 KB pages without LLC partitioning, requiring only minimal OS and architectural modifications. Contrary to prior approaches, our proposal maintains comparable effectiveness under fragmentation, avoids inducing additional cache misses, and incurs only negligible page fault overhead relative to Transparent Huge Pages (THP).

***Index Terms*—Page Coloring, Huge Pages, Cache Contention, Last-Level Cache, Virtual Memory**

I. Introduction

Modern multiprocessing systems share several microarchitectural components, such as the Last Level Cache (LLC), across multiple cores, making them vulnerable to contention and interference. This can lead to significant performance degradation, reduced predictability, and increased security risks. *Page coloring* [17], [32] is a memory management technique that allows the operating system (OS) to control which portions of shared hardware resources a process can access by carefully managing the allocation of its physical pages, in order to improve performance isolation and enhance system security.

Simultaneously, virtual memory has become a fundamental abstraction in modern computing systems, enabling process isolation and memory protection by decoupling the virtual address space of different programs from the underlying physical memory layout. To realize this abstraction, virtual addresses are mapped to physical addresses via multi-level page tables managed by the operating system, while hardware support transparently walks the page tables and caches mappings in the Translation Lookaside Buffer (TLB). These address translation mechanisms, however, incur significant performance overheads, requiring costly page table walks on every TLB miss, especially in modern memory-intensive workloads. *Huge pages* [11], [21] are often employed to reduce the frequency of TLB misses by expanding TLB reach, and effectively reducing address translation costs.

Page coloring and huge pages are traditionally considered incompatible, because huge pages occupy all available colors, leaving no address bits that influence cache indexing under OS control [8], [9]. Prior work aiming to combine the two techniques suffers either from significant contiguity constraints on the OS, limiting address translation effectiveness under memory fragmentation [8], or restricts every huge page to a specific color, unnecessarily increasing conflict cache misses and harming performance [9]. In addition, those prior studies did not present in detail, nor evaluate, the necessary OS support for combining page coloring with huge pages. Our goal in this paper is to enable the simultaneous use of huge pages and page coloring for LLC partitioning, while retaining the performance advantages and surpassing the limitations of prior approaches through a practical design.

We propose *Colored Huge Pages* (CHP): a hardware-software co-design that enables the simultaneous use of page coloring and huge pages by breaking virtually contiguous huge pages into multiple colored physically non-contiguous base pages arranged in a predictable, strided pattern. Through lightweight microarchitectural enhancements and OS support, the system enables cache partitioning, while retaining the TLB reach benefits of traditional huge pages by translating colored huge pages efficiently, as if they were physically contiguous. Our design includes a custom allocator, integrated within the Linux kernel, that constructs these strided colored huge pages, along with lightweight modifications in the L2 TLB, Page Table, and Page Table Walker that exploit the regular mapping structure to accelerate address translation.

To the best of our knowledge, this is the first work to propose novel hardware and system-level extensions to significantly relax the strict memory contiguity requirements imposed on the OS allocator for the construction of colored huge pages, which become impractical in the presence of memory fragmentation

978-3-9826741-1-7/DATE26/© 2026 EDAA

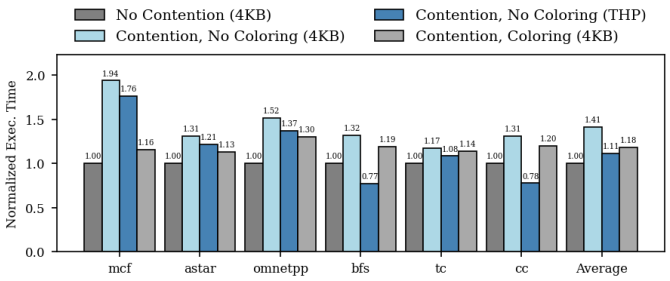


Fig. 1: Normalized execution time under contention with 4 KB pages, THP, and page coloring, compared to no contention.

under realistic system conditions. Furthermore, this is the first work to design, implement, and evaluate the operating system support required to integrate page coloring with huge pages.

We prototype CHP in Linux kernel v6.6 and evaluate our solution using a set of memory-intensive workloads and an analytical performance model. CHP enables the simultaneous use of page coloring and huge pages, improving performance by 33.7% - compared to 17.4% with 4 KB coloring and 21% with Transparent Huge Pages (THP). CHP maintains robustness comparable to THP under memory fragmentation and delivers lower cumulative page fault overhead than both default paging and regular 4 KB coloring. Thanks to its flexibility, our design achieves higher effectiveness under memory fragmentation compared to [8] and outperforms [9] by up to 13%.

In summary, the key contributions of this paper are:

- We quantify the impact of cache interference and address translation and show that combining page coloring with huge pages can simultaneously reduce LLC interference and mitigate TLB pressure.
- We propose Colored Huge Pages (CHP): a complete hardware-software co-design that enables a huge page to be mapped to individually non-contiguous colored base pages while preserving translation efficiency.
- We comprehensively evaluate the combined benefits of CHP under a variety of system conditions.

II. Motivation

A. Page Coloring & Huge Pages

Page coloring has emerged as a practical technique for software-based partitioning of shared hardware resources [6], [15], [17], [22]–[26], [28], [34], [36], [38]–[41]. In this work, we focus on page coloring for LLC partitioning. Page coloring is orthogonal to hardware-based way cache partitioning techniques [2], [3], [13], [14]. Those approaches only enable coarser-grained partitions, since the number of ways in modern caches is severely restricted by latency, energy consumption, and die area tradeoffs. As a result, the number of partitions does not scale well with the increasing core counts of contemporary systems. Unlike way-based partitioning, page coloring applies set-partitioning, enabling finer-grained partitioning than is otherwise achievable through either technique in isolation [35].

TABLE I: Comparison of huge page coloring approaches.

Approach	Fragmentation Robustness	No Extra Cache Conflicts	HW/SW Compatibility	OS Design & Evaluation
Scattered [8]	×	✓	✓	×
SWAP [9]	✓	×	×	×
CHP	✓	✓	✓	✓

Moreover, it remains the only viable mechanism for controlling the allocation of resources that lack dedicated hardware partitioning support.

Independently, virtual memory and address translation through the TLB has become a major performance bottleneck in memory-intensive modern workloads. Huge pages [11], [21] can help mitigate this issue by mapping a significantly larger portion of a process's address space with a single translation entry, i.e., 2 MB vs 4 KB on x86 systems. This increases TLB reach and reduces the frequency of TLB misses, which would otherwise trigger expensive page table walks. Furthermore, huge pages shorten the walk by skipping the last level in the page table. Overall, huge pages can significantly improve application performance.

B. Potential Performance Improvement

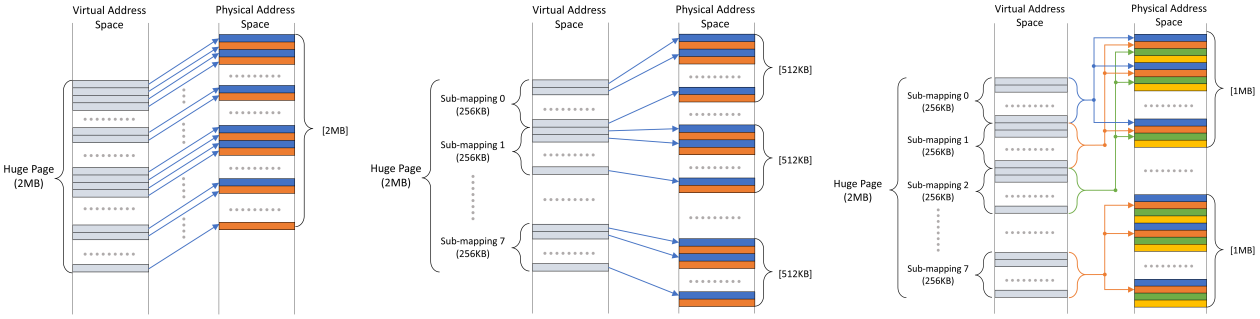
Page coloring and huge pages are incompatible, because the large page offset in a huge page usually consumes all the address bits, leaving none of the cache-indexing bits under operating system control [8], [9]. However, combining the two techniques would achieve significant performance benefits by reducing cache contention and the number of page table walks.

To motivate our approach, we study the performance impact of cache contention, huge pages, and page coloring. Section IV-A describes the evaluation methodology. Fig. 1 shows the normalized execution time relative to the baseline (no contention, 4 KB pages). Cache contention degrades performance by up to 1.94× (mcf) and 1.41× on average. Page coloring [39] alleviates this interference, reducing the average overhead to 1.18×. THP [11] provide additional performance gains, particularly in bfs and cc, where the execution time under contention improves to 0.77× and 0.78× of the baseline. These results indicate that combining huge pages with cache isolation via page coloring can deliver substantially greater performance improvements than either technique in isolation.

C. Prior Approaches

Only two prior studies have investigated the integration of page coloring with huge pages. Table I summarizes how CHP compares to the previous approaches in several key aspects.

Scattered Superpages [8] maps a virtual huge page across multiple physical huge pages. Similar to CHP, that approach requires expanding the TLB to store additional color information per huge page. However, Scattered Superpages are constructed using a contiguous set of *multiple huge physical pages*. That memory management approach limits the resilience and effectiveness of Scattered Superpages under realistic fragmentation conditions (Section IV-B3). Motivated by this limitation, we introduce novel hardware and OS mechanisms that allow each colored huge page to consist of multiple independent submappings, relaxing contiguity requirements.



(a) Regular Huge Page (no coloring) (b) Colored Huge Page with 1 out of 2 colors (c) Colored Huge Page with 3 out of 4 colors

Fig. 2: The virtual-to-physical page mapping for regular huge pages (a) and colored huge pages in a system with 2 (b) and 4 colors (c). Contrary to regular huge pages, CHP allows the use of page coloring. Each sub-mapping can begin independently of the others to relax the requirements for vast physical memory contiguity and maintain effectiveness under fragmentation.

Another prior work [9] integrates huge pages with page coloring by swapping bits from the huge page number with LLC index bits prior to cache access, creating a *pseudo-physical address space*. However, that design restricts each huge page to a single color, increasing conflict misses and degrading performance in some benchmarks (Section IV-B4). Moreover, this hardware indirection hides real physical memory from the OS, which may complicate the compatibility with existing software and hardware components (e.g., drivers and DMA engines) that require real physically contiguous memory. Supporting both real and pseudo-physical spaces significantly complicates OS management. In contrast, CHP avoids inducing additional cache conflicts by distributing portions of huge pages to multiple colors, and allows the OS to still manage the real physical address space, providing full compatibility with existing mechanisms.

Finally, neither of the two prior approaches design, implement, and evaluate the operating system support required to integrate page coloring with huge pages.

III. Colored Huge Pages

The *key idea* of our approach is that a huge page can remain contiguous in virtual memory, but get broken down into multiple individual non-contiguous physical frames, allocated with a regular pattern. With lightweight microarchitectural modifications, the hardware exploits this allocation pattern to perform address translation as if the pages were physically contiguous, preserving the TLB reach and performance benefits of huge pages. We propose *Colored Huge Pages* (CHP): a complete system design that spans the hardware-software boundary, combining the necessary microarchitectural modifications with operating system support. We assume and maintain compatibility with x86 translation mechanisms in this paper, but the principles can be trivially extended to other ISAs.

A. Overview

CHP targets large anonymous Virtual Memory Areas (VMAs) of memory-intensive processes. Page faults within these VMAs are intercepted and handled by a custom page allocation system, which backs each faulting 2 MB region with carefully constructed mappings. These mappings consist of base pages allocated at a constant stride so that they share the same color(s). Pages with colors differing from the target one are effectively skipped, resulting in mappings composed exclusively of pages with the desired color(s), as illustrated in Fig. 2(b). The regularity of these mappings allows the offset of i-th page from the start of the mapping to be computed with inexpensive bitwise operations, enabling accelerated address translation with minimal microarchitectural modifications, even though the individual pages are not physically contiguous.

To construct a colored huge page, our custom allocator first identifies a sufficiently large contiguous memory region, from which base pages corresponding to the desired colors are selectively extracted. On its own, such an approach is impractical due to the memory fragmentation exhibited in long-running systems [37]. To alleviate these strict contiguity requirements, we propose a novel approach that, with minimal additional hardware support and negligible overhead, allows each 2 MB strided mapping to be divided into 8 independent sub-mappings. Each of these sub-mappings maintains the previously described strided structure, but can begin independently of the others, significantly reducing the demand for physically contiguous memory regions, while also improving color allocation flexibility when multiple colors are permitted for a process. An example of this subdivision is illustrated in Fig. 2(c). CHP falls back to existing mechanisms if a strictly compliant region cannot be allocated, and operates completely transparently to the applications in a best-effort manner.

B. Hardware Modifications

1) Modified L2 TLB: Our design leaves the L1 TLB path intact and only extends the L2 TLB. By integrating CHP into the L2 TLB, we preserve the reduced page table walk frequency of huge pages while avoiding added complexity or latency in the L1 TLB's critical path. CHP augments each L2 TLB entry with the starting Physical Frame Numbers (PFNs) of all eight sub-mappings, widening it to include seven additional PFNs. Tag, permission and access bits are shared across all sub-regions. A dedicated bit per L2 TLB entry indicates whether it refers to a conventional (huge) page translation or a colored

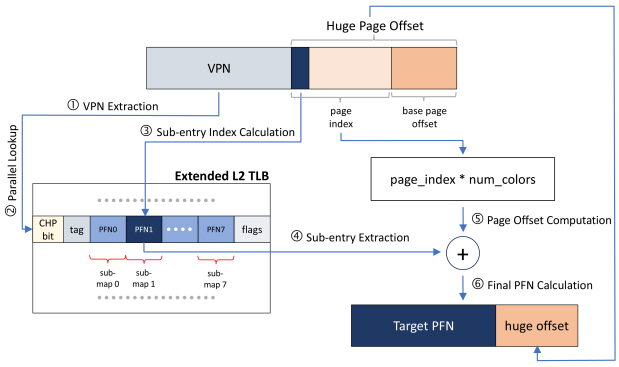


Fig. 3: Hardware-assisted address translation for colored huge pages. *Page index* refers to the offset of a 4 KB base page within a 2 MB huge page.

huge page. The extended L2 TLB is looked up after an L1 TLB miss. Upon an L2 TLB hit for a colored huge page, the corresponding 4 KB entry is constructed and cached into the L1 TLB to accelerate subsequent accesses. Importantly, L2 TLB indexing remains unaffected, as the number of entries is unchanged.

2) Address Translation: Fig. 3 illustrates how the proposed translation mechanism for the augmented L2 TLB is realized in hardware to achieve fast address translation for Colored Huge Pages, assuming 4 KB and 2 MB base and huge page sizes:

Step 1 - VPN Extraction: the lower 21 offset bits of the incoming virtual address are discarded to extract the Virtual (Huge) Page Number (VPN).

Step 2 - Parallel TLB Lookup: the VPN is looked up in the enhanced L2 TLB as normally. A dedicated per-entry *CHP bit* identifies TLB hits corresponding to colored huge pages.

Step 3 - Sub-Entry Index Calculation: upon a colored huge page hit, the three most significant bits of the 2 MB page offset (bits 18–20) are isolated to determine which sub-mapping corresponds to the current address.

Step 4 - Sub-Entry Extraction: the selected sub-entry is extracted based on the computed index. Page access permissions are validated at this stage and the base PFN of the relevant sub-mapping is obtained.

Step 5 - Page Offset Computation: the offset of the target page compared to the start of the corresponding sub-mapping is calculated by multiplying its page index with the constant number of available colors. This operation can be performed using a bitwise shift.

Step 6 - Final Target PFN Calculation: the final PFN of the target page is derived by bitwise *OR*-ing the offset from Step 5 to the base PFN from Step 4.

Steps 1 and 2 mirror conventional huge page translation implementations, while Steps 3 and 5 can execute in parallel with TLB Lookup (Step 2), effectively hiding their latency. Our approach introduces only minimal additional overhead in the critical path of the L2 TLB, i.e., some minor multiplexer delay in Step 4 and a single bitwise *OR* operation in Step 6, while saving L2 TLB misses that trigger costly page table walks.

3) Modified Page Table & Page Table Walk: Upon an L2 TLB miss within a colored huge page, 8 PFNs - corresponding to the 8 individual PTEs of the sub-mappings - must be fetched into the L2 TLB. Fortunately, this incurs no additional cost in the page table walk path, as all 8 PTEs lie within a single cache line and can be retrieved with a single memory reference [33].

We modify the Page Table Walker (PTW) so that upon reaching a PMD entry with the *Colored-Huge-Page Bit* set (that is managed by the OS), it fetches 8 PTEs from the address indicated by the PMD into the augmented L2 TLB. The PTW identifies the sub-entry for the faulting address, fetching and using it immediately while filling the rest in the background.

C. Operating System Support

In CHP, the OS must allocate, manage, and install mappings in the proposed strided format. We prototype our design on Linux Kernel v6.6, introducing a dedicated colored huge page allocator built on top of the Linux buddy allocator [10]. This approach preserves compatibility with existing memory management while enabling color-aware huge page allocation.

CHP operates transparently to applications and can be selectively enabled for specific processes through procfs. CHP intercepts anonymous page faults within large VMAs and attempts to populate the faulting 2 MB region with a colored huge page. Constructing such a page requires locating eight independent 256 KB strided sub-mappings, with all pages in each sub-mapping sharing the same color. To allocate each sub-mapping, CHP requests from the buddy allocator a sufficiently large (256 KB $\times$ number of system colors) contiguous block from which it extracts the desired strided mappings.

If successful, the required individual pages are carved out from the contiguous block and installed into the process's address space by crafting the appropriate Page Table Entries (PTEs). To enable prototyping on existing hardware, we currently register and install each 4 KB page individually. An actual implementation would install only one PTE per sub-mapping (8 total). The OS also sets the special *Colored-Huge-Page Bit* at the PMD level to signal to the PTW that this region is backed by a colored huge page.

CHP additionally introduces an *Allocator Cache* for caching the left-over physical pages from the allocated block for future use. Each sub-mapping request first attempts to get serviced by the Allocator Cache, before allocating pages from a new contiguous physical block obtained through the buddy allocator. Additionally, strided mappings can be reclaimed from the process's memory and returned to the Allocator Cache when a VMA is unmapped. The CHP allocator operates in a best-effort manner, falling back to PALLOC's [39] 4 KB page coloring implementation if the desired strided mapping can not be constructed. Contrary to SWAP [9], CHP ensures fair color distribution by assigning each sub-mapping a different color from the process's allowed set in a round-robin manner.

IV. Evaluation

A. Methodology

1) System Configuration & Benchmarks: We evaluate CHP on an 8-core Intel i7-10700 machine with 16 GB RAM (Ta-

TABLE II: System Configuration.

CPU Model	Intel Core i7-10700 CPU @ 2.90GHz, 8 cores
Memory	16 GB, DDR4 3200MHz
L1 Data/Instr. Cache	32 KB 8-way set associative (per core)
L2 Cache	256 KB 4-way set associative (per core)
L3 Cache	16 MB 16-way set associative (shared)
L1 Data TLB	4 KB / 2 MB / 1 GB pages: 4-way, 64/32/4 entries
L2 TLB	4 KB / 2 MB pages: 6-way, 1536 entries
L2 / L3 Coloring Bits	4 (12-15)
Used / Supported Colors	8 / 16

TABLE III: Evaluation Workload Suite.

Application	Benchmark Suite	Memory Footprint	Num. Colors
mcf	SPEC 2006	1.7 GB	5
astar	SPEC 2006	339 MB	5
omnetpp	SPEC 2006	167 MB	6
bfs / cc	GAPBS	4.5 GB	5
tc	GAPBS	2.5 GB	4

ble II), prototyped in Linux Kernel v6.6. We use benchmarks from GAPBS [5] and SPEC 2006 [30] (Table III). Cache contention is introduced via *stress-ng* [31] with 4 *cache* stressors. Cache isolation for the stressor is enforced using regular page coloring [39]. Each benchmark is assigned the minimum number of colors that keeps its runtime within 4% with respect to using the entire LLC when running alone. The remaining colors are assigned to the stressor. We pin applications to separate cores and present average results from 3 runs.

2) Performance Projection Model: Our evaluation focuses on performance and relies on an analytical model driven by hardware performance counters to estimate the benefits of CHP. While simulation could model the proposed translation hardware support, it would be prohibitively slow for practical evaluation [1], [4], [16]. We report runtimes under four scenarios: (i) contention with 4 KB pages (baseline), (ii) THP without coloring, (iii) 4 KB page coloring and (iv) projected runtime with CHP (huge pages + coloring).

Using Linux *perf*, we collect total execution cycles (T) and cycles spent in page table walks (C), assuming these translation-related cycles are not overlapped with useful work or stalls from LLC misses. We decompose all reported runtimes (R) into translation (O_{trans}) and cache (O_{cache}) components, such that the total runtime is given by: $R = O_{trans} + O_{cache}$. All runtimes are normalized to the 4 KB contention baseline. For each execution scenario S, we compute the normalized translation component as $O^S_{trans} = \frac{C_S}{T_{contention\text{-}4K}}$ and the cache component as $O^S_{cache} = \frac{T_S - C_S}{T_{contention\text{-}4K}}$.

To estimate CHP performance, we assume its translation cost equals that of contention-THP. To evaluate CHP's cache impact, we adopt a pessimistic model that likely underestimates its benefits, as it ignores reductions in cache pressure from fewer TLB misses. We model the effect of coloring-induced isolation in the execution time by scaling the contention-THP cache component by the cache speedup of 4 KB coloring. The overall CHP runtime estimate (R_{CHP}) is given by (1).

$$R_{CHP} = \underbrace{O^{cont\text{-}THP}_{trans}}_{O^{CHP}_{trans}} + \underbrace{O^{cont\text{-}THP}_{cache} \cdot \frac{O^{color\text{-}4K}_{cache}}{O^{cont\text{-}4K}_{cache}}}_{O^{CHP}_{cache}} \quad (1)$$

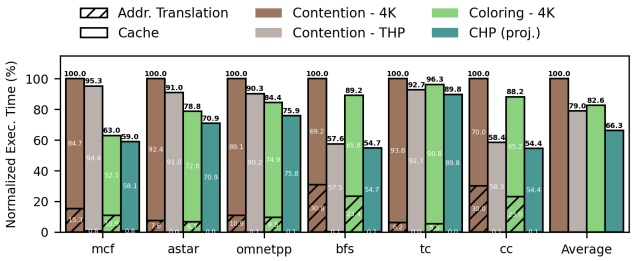


Fig. 4: Normalized runtime breakdown showing projected performance gains from CHP, compared to regular 4 KB pages, THP and 4 KB page coloring, all under contention.

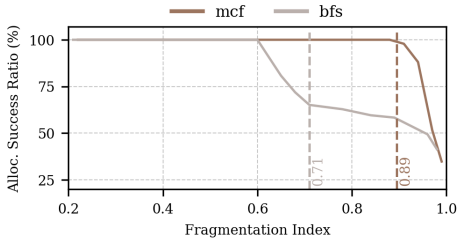


Fig. 5: CHP allocation success ratio vs. fragmentation index.

B. Results

1) Performance: Fig. 4 presents the runtime decomposition for all benchmarks under the four scenarios mentioned in Section IV-A2, with all runtimes normalized to scenario (i), i.e., the 4 KB contention baseline. Each bar decomposes total runtime into address translation and cache components.

When used individually, THP and page coloring reduce runtime to 79% (as low as 57.6%) and 82.6% (as low as 63%) of the baseline, respectively. Even assuming no additional cache benefits from reduced TLB misses and page table walks beyond what coloring with 4 KB pages provides, our approach consistently outperforms both THP and 4 KB coloring, reducing average runtime to 66.3% of the baseline (i.e., improving performance by 33.7%) and reaching as low as 54.4% for cc.

2) Fragmentation Sensitivity: We evaluate the success ratio of CHP allocations under varying memory fragmentation levels. Fragmentation is quantified by the *fragmentation index* (F_i), adapted from [12], which denotes the fraction of physical memory that cannot be backed by the currently available huge pages. We focus on mcf and bfs, whose memory footprints correspond to roughly 11% and 29% of the system's physical memory. Therefore, assuming regular huge pages (e.g., THP), the critical fragmentation threshold, beyond which allocation success begins to degrade, is expected at $F_i \approx 0.89$ for mcf and $F_i \approx 0.71$ for bfs.

As shown in Fig. 5, our solution is resilient under low to moderate fragmentation. For both benchmarks, success ratios degrade before the critical fragmentation point, as much as 10% earlier for bfs, indicating CHP's slightly higher fragmentation sensitivity relative to THP. Nonetheless, allocations remain near-perfect up to moderately high fragmentation ($F_i = 0.6$

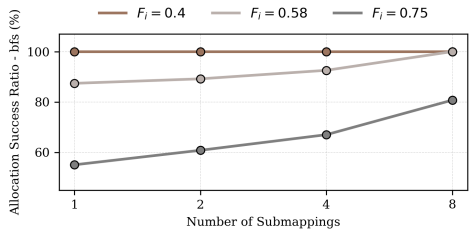


Fig. 6: Allocation success ratio for bfs under varying fragmentation levels and sub-mapping counts.

even for the larger workload), demonstrating reasonable robustness relative to THP under realistic fragmentation conditions.

3) Number of Sub-mappings: Fig. 6 shows allocation success ratios for 1, 2, 4, and 8 sub-mappings across different fragmentation levels. Using a single sub-mapping corresponds to Scattered Superpages [8], where the strided mapping spans *strictly contiguous* physical huge pages. We use bfs for this analysis due to its large 4.5 GB footprint.

As fragmentation approaches the critical threshold ($F_i = 0.58$), using 8 sub-mappings preserves a 100% allocation success ratio, whereas fewer sub-mappings degrade allocation effectiveness. This effect becomes more pronounced at higher fragmentation levels ($F_i = 0.75$), where reducing the number of sub-mappings leads to even more substantial drops in successful allocations. These results demonstrate that the finer sub-mapping granularity enabled by our design significantly improves resilience to fragmentation compared to Scattered Superpages under realistic memory conditions.

4) Comparison to SWAP: We modify our implementation to emulate the mappings created by SWAP, which contrary to CHP restricts an entire huge page to a single cache color, and measure the performance difference compared to CHP. For the *GAPBS* workloads, we notice only negligible differences between the two approaches in terms of both LLC misses and execution time. However, SWAP increases LLC misses for mcf by 9.4% and 12% when using 5 and all 8 colors, leading to performance degradations of 11.8% and 13.1% respectively. This analysis demonstrates that the flexibility provided by CHP can significantly outperform SWAP in some cases, effectively offsetting its modest additional overheads.

5) Overhead Analysis: Table IV reports average and tail (99th percentile) page fault latency, fault counts, and cumulative fault time, measured with *bpftrace* [7], for bfs. CHP introduces overhead to the fault path compared to 4 KB pages, PALLOC and THP. Unlike THP, which installs a huge page via a single PMD update, our implementation must register and install each 4 KB page individually to allow prototyping on existing hardware. Nonetheless, CHP reduces fault frequency to THP levels, lowering cumulative fault time well below both 4 KB paging and PALLOC. Ultimately, the few hundred milliseconds of page fault overhead ($< 0.2\%$ of bfs's runtime) are negligible relative to workload duration (hundreds of seconds) and outweighed by the benefits of reduced cache contention.

In terms of area, CHP expands the data part of each L2 TLB entry by $\sim$8x to store the additional PFNs. However, the tag and permission bits, along with the number of entries remain unchanged, so no extra hardware is required for indexing. This minimal hardware overhead is outweighed by the increased TLB reach (by 512$\times$) enabled by each augmented entry.

TABLE IV: Page fault latency metrics (in us) for bfs.

Scenario	Avg. Lat.	Tail Lat.	Page Faults	Total Fault Time
4 KB	0.53	0.62	1,179,711	627,469
PALLOC [39]	1.47	2.62	1,179,709	1,737,566
THP	2.02	3.29	2,878	5,803
CHP	160.7	277.15	2,876	461,523

V. Related Work

Sections II and IV compared CHP qualitatively and quantitatively with the most closely related work [8], [9]. Here we discuss other prior page coloring work for cache partitioning. Page coloring was first proposed by Taylor et al. [32] to improve TLB hit rate, while Kessler et al. [17] later proposed page placement algorithms for cache partitioning. Since then, two main software approaches have emerged. In static page coloring [15], [17], [22], [24], [26], [28], a process is allocated a portion of the cache statically during its execution. In contrast, dynamic coloring [6], [22], [24], [36], [38], [40] adapts to time-varying cache access patterns and allocation needs by relying on page recoloring support. In addition, previous efforts [15], [18]–[20], [27]–[29], [36] have also applied page coloring in cloud environments to improve inter-VM isolation. Finally, Wang et al. [35] combined page coloring with hardware way partitioning to support fine-grained cache partitions.

VI. Conclusions

Modern systems can benefit significantly from the simultaneous use of page coloring and huge pages to improve both cache isolation and translation efficiency. In this paper we proposed CHP, a custom hardware-software co-design that enables virtually contiguous huge pages to be distributed across individually colored physical frames without compromising address translation performance. We designed and implemented operating system support in the Linux kernel and proposed lightweight microarchitectural enhancements to support fast translation of these mappings, while simultaneously relaxing the contiguity constraints on the OS memory allocator. Our evaluation results showed that CHP successfully manages to combine the benefits of both page coloring and huge pages, providing improved performance with respect to prior work.

Acknowledgment

This work was supported by the framework of H.F.R.I call "3rd Call for H.F.R.I.'s Research Projects to Support Faculty Members & Researchers" (H.F.R.I. Project Number: 26508 – COMPLETEVM), and by the European Union's Horizon Europe research and innovation programme under grant agreement No 101093062 (Vitamin-V). Views and opinions expressed are, however, those of the authors only and do not necessarily reflect those of the European Union or the HaDEA. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] C. Alverti, S. Psomadakis, V. Karakostas, J. Gandhi, K. Nikas, G. Goumas, and N. Koziris, "Enhancing and Exploiting Contiguity for Fast Memory Virtualization," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020.

[2] "AMD64 Technology PlatformQuality of Service Extensions," https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_03_PUB.pdf.

[3] "Arm DynamIQ Shared Unit Technical Reference Manual," https://developer.arm.com/documentation/100453/0401/L3-cache/L3-cache-partitioning.

[4] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, p. 237–248.

[5] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv preprint arXiv:1508.03619*, 2015.

[6] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen, "Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches," in *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, 1994, pp. 158–170.

[7] "bpftrace: High-level Tracing Language for Linux eBPF," https://github.com/bpftrace/bpftrace.

[8] L. Chen, Y. Wang, Z. Cui, Y. Huang, Y. Bao, and M. Chen, "Scattered Superpage: A Case for Bridging the Gap between Superpage and Page Coloring," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 177–184.

[9] Z. Cui, L. Chen, Y. Bao, and M. Chen, "A Swap-based Cache Set Index Scheme to Leverage both Superpage and Page Coloring Optimizations," in *51st Annual Design Automation Conference (DAC)*, 2014, p. 1–6.

[10] M. Gorman, "Understanding The Linux Virtual Memory Manager," https://www.kernel.org/doc/gorman/html/understand/.

[11] M. Gorman and A. Arcangeli, "Transparent Hugepages in the Linux Kernel," https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[12] M. Gorman and A. Whitcroft, "The What, The Why and the Where To of Anti-Fragmentation," in *Ottawa Linux Symposium*, vol. 1. Citeseer, 2006, pp. 369–384.

[13] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 657–668.

[14] "Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family," https://www.intel.com/content/www/us/en/developer/articles/technical/-introduction-to-cache-allocation-technology.html.

[15] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li, "A Simple Cache Partitioning Approach in a Virtualized Environment," in *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2009, pp. 519–524.

[16] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant Memory Mappings for Fast Access to Large Memories," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, 2015, p. 66–78.

[17] R. E. Kessler and M. D. Hill, "Page Placement Algorithms for Large Real-Indexed Caches," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 338–359, 1992.

[18] D. Kim, H. Kim, N. S. Kim, and J. Huh, "vCache: Architectural Support for Transparent and Isolated Virtual LLCs in Virtualized Environments," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 623–634.

[19] J. Kim, J. Kim, D. Ahn, and Y. I. Eom, "Page Coloring Synchronization for Improving Cache Performance in Virtualization Environment," in *Computational Science and Its Applications-ICCSA 2011: International Conference, Santander, Spain, June 20-23, 2011. Proceedings, Part III 11*. Springer, 2011, pp. 495–505.

[20] H. Li, T. Lu, Y. Liu, and M. Chen, "Make Page Coloring more Efficient on Slice-Based Three-Level Cache," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2019, pp. 310–317.

[21] "libhugetlbfs(7) - Linux man page," https://linux.die.net/man/7/libhugetlbfs.

[22] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 2008, pp. 367–378.

[23] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 367–375.

[24] W. L. Lynch, B. K. Bray, and M. J. Flynn, "The Effect of Page Allocation on Caches," *ACM SIGMICRO Newsletter*, pp. 222–225, 1992.

[25] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 374–385.

[26] S. Perarnau, M. Tchiboukdjian, and G. Huard, "Controlling cache utilization of hpc applications," in *Proceedings of the International Conference on Supercomputing*, 2011, p. 295–304.

[27] A. Scolari, D. B. Bartolini, and M. D. Santambrogio, "A Software Cache Partitioning System for Hash-Based Caches," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[28] A. Scolari, F. Sironi, D. B. Bartolini, D. Sciuto, and M. D. Santambrogio, "Coloring the Cloud for Predictable Performance," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, 2013.

[29] X. Shang, W. Jia, J. Shan, X. Ding, and C. Borcea, "Reestablishing Page Placement Mechanisms for Nested Virtualization," *IEEE Transactions on Cloud Computing*, vol. 11, no. 3, pp. 3239–3250, 2023.

[30] Standard Performance Evaluation Corporation, "SPEC CPU2006 Benchmark," http://www.spec.org/cpu2006/.

[31] "Stress-ng: a tool to load and stress a computer system," https://manpages.org/stress-ng.

[32] G. Taylor, P. Davies, and M. Farmwald, "The TLB Slice - A Low-Cost High-Speed Address Translation Mechanism," in *Proceedings of the 17th annual international symposium on Computer Architecture*, 1990.

[33] G. Vavouliotis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jiménez, and M. Casas, "Exploiting Page Table Locality for Agile TLB Prefetching," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 85–98.

[34] S. Volos, C. Fournet, J. Hofmann, B. Köpf, and O. Oleksenko, "Principled Microarchitectural Isolation on Cloud CPUs," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 183–197.

[35] X. Wang, S. Chen, J. Setter, and J. F. Martínez, "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 121–132.

[36] X. Wang, X. Wen, Y. Li, Y. Luo, X. Li, and Z. Wang, "A Dynamic Cache Partitioning Mechanism under Virtualization Environment," in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2012, pp. 1907–1911.

[37] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Translation Ranger: Operating System Support for Contiguity-Aware TLBs," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[38] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A Dynamic Cache Partitioning System Using Page Coloring," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014, p. 381–392.

[39] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 155–166.

[40] X. Zhang, S. Dwarkadas, and K. Shen, "Towards Practical Page Coloring-based Multi-core Cache Management," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 89–102.

[41] Y. Zhong, D. S. Berger, C. Waldspurger, R. Wee, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury *et al.*, "Managing Memory Tiers with CXL in Virtualized Environments," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 37–56.