Enabling Huge Pages for Real-World Executables

Nadav Amit
Technion — Israel Institute of Technology
Haifa, Israel
namit@cs.technion.ac.il

Abstract

While huge pages can dramatically reduce address translation overhead, their use for executable code remains limited by structural barriers in binary formats, page cache management, and loader mechanisms. Existing solutions copy code into anonymous memory, abandoning file-backed semantics and breaking cross-process sharing, memory reclamation, and debugging tools. Emerging kernel support for file-backed huge pages remains insufficient without relinking and modifying loaders, impractical for deployed and closed-source software.

We present Hugifier, a userspace solution that enables huge page mappings for existing executables while preserving file-backed memory semantics—executable memory sharing, paging, and debugging all continue to work correctly. Hugifier combines a binary transformation that aligns code segments to huge page boundaries without disassembly or relinking, with a runtime component that ensures huge page mappings on current kernels. Evaluation shows over 90% iTLB miss reduction and up to 11.3% speedup—exceeding libhugetlbfs by up to 5.3% and Linux's current RO-THP by 7.3%; against a stronger "aligned" baseline modeling future kernel/loader support without segment separation, the binary transformation still adds 5.5%.

CCS Concepts: • **Software and its engineering → Virtual memory**; *Memory management*; *Software performance*.

Keywords: huge pages, instruction TLB, ELF, binary transformation, dynamic loader, page cache, file-backed memory

ACM Reference Format:
Nadav Amit. 2026. Enabling Huge Pages for Real-World Executables. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on Memory Management (ISMM '26), June 16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3814942.3816136

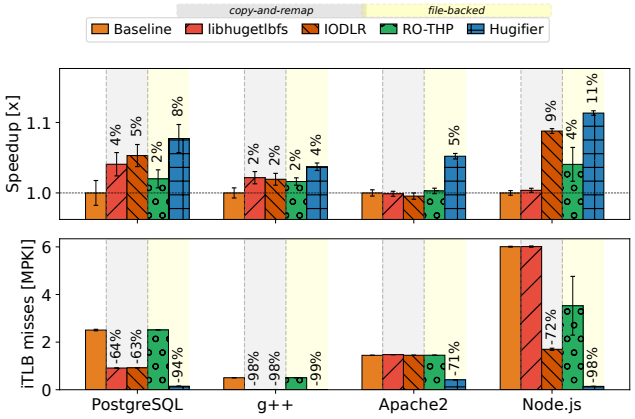

Figure 1. x86 performance: Hugifier achieves up to 11.3% speedup over baseline across server workloads, outperforming copy-and-remap solutions (libhugetlbfs, IODLR) without their limitations. RO-THP (experimental kernel huge page support) provides limited benefit for unaligned executables. Detailed results and methodology appear in Section 4.

1 Introduction

Despite huge pages being supported in commodity servers since the 1990s [47], their use for executable code has been limited by practical challenges. The instruction TLB (iTLB), which translates every code fetch, has an outsized performance impact: misses stall the front-end and cannot be hidden by out-of-order execution [11, 38, 53], yet huge pages remain largely unused for executable code.

Enabling huge pages for executable code requires solving two distinct prerequisites: first, binaries must be properly aligned to huge page boundaries, and second, the OS must actually map them using huge pages. Because the kernel maps executables directly from the page cache, the on-disk binary layout dictates how code is cached and whether huge pages can form—making binary format a memory management concern. Since addressing one requirement without the other provides no benefit, it appears neither is commonly solved. Executables are not built with huge page alignment, and operating systems lack reliable mechanisms to map executable code with huge pages.

While relinking can produce aligned executables, it is often impractical: it requires rebuilding the entire software stack including all shared libraries, many organizations rely on proprietary components [52], and binary format transitions like position-independent executable (PIE) adoption took decades. Today's workarounds are instead copy-and-remap approaches such as libhugetlbfs [34, 36] and IODLR [27], which abandon file-backed memory entirely.

ISMM '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2720-7/2026/06
https://doi.org/10.1145/3814942.3816136

Abandoning file-backed semantics has severe consequences: the OS cannot share executable memory between processes, debugging and profiling tools fail [54], and the kernel can no longer reclaim code pages without swap.

Even with emerging kernel support for file-backed huge pages [51], existing executables see limited benefits: code segments rarely align to 2 MB boundaries, leaving significant portions on small pages. Experimental kernel enhancements [35] and hardware approaches like ARM64's ContPTE attempt to work within these constraints, but fall short of the full benefits that 2 MB huge pages can provide—leaving significant performance unrealized.

Hugifier takes a pure userspace approach that requires neither kernel modifications nor relinking, making it immediately applicable to existing binaries. Our key insight is that we can shift code and data segments forward together while preserving their relative offsets, avoiding the disassembly and intermediate representation lifting required by binary rewriting tools like BOLT [41] and Egalito [56]. By maintaining the Executable and Linkable Format (ELF) layout and directly manipulating segment addresses, we sidestep the heuristic-based analysis that causes these tools to fail on diverse production binaries. Combined with our runtime loader component that ensures proper huge page mapping, we preserve file-backed memory—enabling the OS to share code between processes while maintaining debugging and profiling capabilities.

As shown in Figure 1, Hugifier achieves up to 11.3% performance improvement over unmodified binaries on x86 systems, with iTLB miss reductions exceeding 90% in most cases. Hugifier outperforms two classes of existing solutions, each with a different baseline: copy-and-remap tools by up to 5.3% without suffering their shortcomings, and the current readonly file THP (RO-THP) kernel support in Linux by up to 7.3%. Furthermore, even compared to a stronger "aligned" baseline that simulates future kernel and loader support for offset alignment and runtime mapping but not segment separation, Hugifier's binary transformation still adds up to 5.5% (Section 4). These gains preserve all system functionality—code sharing, debugging, and memory management all continue to work correctly—at the cost of a modest reduction in address space layout randomization (ASLR) entropy (Section 3.4).

This paper makes the following contributions:

- We design and implement Hugifier, a system that enables huge page mappings for existing executables while preserving file-backed memory semantics. Its binary transformation aligns code segments to huge page boundaries without relinking or disassembly, applicable to legacy and closed-source binaries.
- We develop a lightweight runtime component that ensures huge page mappings on current Linux kernels without requiring loader modifications.
- We demonstrate up to 11.3% performance improvement on production binaries while preserving cross-process code sharing, memory reclamation, and debugging capabilities. Even with future loader support for file-backed huge pages, Hugifier's binary transformation provides up to 5.5% additional improvement.

2 Motivation

As memory sizes and application working sets continue to grow, Translation Lookaside Buffer (TLB) sizes have not kept pace, creating significant bottlenecks in address translation [7]. Virtual memory requires traversing multiple page table levels—reaching up to five levels in recent Intel processors [14] and potentially 35 steps in virtualized environments [8]. Huge pages mitigate these translation overheads by allowing each TLB entry to cover a larger memory range, reducing the number of TLB misses and accelerating page walks by eliminating traversal levels.

OSes provide two primary mechanisms for huge page support: static reservation using dedicated interfaces like hugetlbfs in Linux [20, 32, 39], or dynamic promotion through transparent huge pages (THP) [4]. THP offers better adaptability and recent improvements have enhanced its performance and memory utilization [33, 42, 60], while also reducing the overhead of huge-page mapping changes in page tables [57]. However, these advances have primarily targeted anonymous memory rather than file-backed memory [10], leaving file-backed executables largely unaddressed—except for just-in-time (JIT) compiled code, which already benefits from THP since it resides in anonymous pages [27].

Historically, hardware constraints justified limited huge page support for code segments. For instance, Intel's Skylake processors contained only 8 L1 iTLB entries for huge pages, making code huge pages often counterproductive due to thrashing [59]. However, modern processors have largely eliminated these hardware limitations, theoretically enabling broader adoption of huge pages for code segments. With modern CPUs efficiently supporting huge page entries in the iTLB, leveraging this capability becomes crucial—recent studies show over 10% of CPU cycles can stall due to address translation overhead in executable code [58]. The impact is particularly significant because instruction TLB (iTLB) misses typically incur higher penalties than data TLB (dTLB) misses: iTLB misses block instruction fetch entirely, stalling the pipeline, while dTLB misses can often be hidden by out-of-order execution.

Currently, using huge pages for code segments faces two structural limitations, which we term *offset alignment* and *segment separation*. Offset alignment requires strict alignment between file offsets and virtual addresses for huge page mappings; the kernel enforces this to simplify page cache management [55]. Segment separation arises because architectural constraints dictate that huge pages can only

map complete 2 MB regions with uniform permissions, so code segments smaller than 2 MB or those sharing a huge page boundary with data segments cannot use huge pages.[1]

Offset alignment can usually be addressed by adjusting alignment metadata in existing binaries, but segment separation requires restructuring the binary layout—inserting padding between code and data segments shifts their relative positions, invalidating cross-segment references that binary rewriters struggle to reliably identify and update.

Beyond these structural constraints, a third limitation we term *runtime mapping* governs how executables are brought into the page cache and mapped in the page tables—as huge pages or small ones. Even with proper offset alignment and segment separation, the kernel requires specific access patterns during page faults to form huge page mappings—patterns that code execution rarely exhibits. As we detail in Section 3.2, these runtime behaviors cause code to be mapped using base pages in virtually all cases.

The root cause lies in how the page cache manages file data. When an executable page fault occurs, the kernel reads file data into the page cache as individual 4 KB folios. Forming a huge page mapping requires either allocating a 2 MB folio upfront—which the kernel only does when sequential readahead detects a suitable access pattern—or collapsing existing 4 KB folios into a large folio after the fact. Code execution generates sparse, non-sequential page faults that do not trigger readahead, and once small folios populate the cache, they block subsequent huge page formation for that region. This creates a catch-22: the pages that would benefit most from huge mappings are the least likely to receive them.

While we focus on Linux, Windows and macOS face the same structural challenges with even more restrictive limitations (privileged-only large pages on Windows [39], silent downgrades of 2 MB superpage requests on macOS [18]). Existing solutions for enabling huge pages for executable code either require fundamental compromises or fail to provide complete functionality, as we detail next.

2.1 Copy-and-Remap Solutions

Copy-and-remap solutions sidestep offset alignment and runtime mapping limitations by abandoning file-backed memory entirely—copying code into anonymous or reserved huge pages that the kernel readily provides. However, they cannot overcome segment separation: code and data segments retain their original layout, so only code that falls within huge page boundaries benefits. As we show in Section 4, not requiring offset alignment slightly alleviates this limitation compared to file-backed approaches.

[1]While x86-64 supports 1 GB pages at the page table level, common microarchitectures lack 1 GB iTLB entries [58], making them unsuitable for executable code.

Since relinking system binaries is often impractical, these solutions represent the most common workaround for enabling huge pages. They copy code into a different type of memory that is then remapped to the original virtual addresses, effectively mischaracterizing the memory type. Despite these tools existing for years, they appear to see little adoption—we uncovered three implementation bugs that prevented libhugetlbfs from working in common cases [3], suggesting minimal real-world usage.

Copy-and-remap approaches suffer from various limitations stemming from their mischaracterization of memory type. Libhugetlbfs uses pre-allocated huge pages, preventing crucial OS services such as NUMA migration [1] and making memory reclamation impossible under pressure. While libhugetlbfs enables sharing for code it remaps, it cannot properly reclaim memory when no longer needed, leading to inefficient resource utilization over time. Similarly, IODLR mischaracterizes the memory as anonymous, preventing efficient process memory sharing and requiring swap-out instead of discard during memory pressure—particularly problematic when swap space is unavailable.

Beyond memory management issues, these approaches break application binary interface (ABI) compatibility: memory accounting (cgroups), file descriptor tracking (lsof), file advisory mechanisms (fadvise), and resource limits (ulimit) all behave unexpectedly when file-backed memory is replaced with anonymous memory. Furthermore, hugetlbfs requires memory to be reserved in advance, making it unavailable for other purposes.

Perhaps most critically, these approaches break debugging and profiling tools that rely on the file-to-memory mapping for symbolic information. IODLR users persistently report missing symbols in perf output [28], and the only workaround—perf-map [54]—is specific to perf and requires manual intervention. Other essential tools like gdb, vtune, crash dump analysis, and production monitoring remain broken without similar tool-specific solutions.

2.2 Kernel-Level Approaches

Rather than working around OS limitations in userspace, several efforts have attempted to modify the kernel itself to enable huge pages for executables. These approaches primarily address runtime mapping by implementing special-case logic in the kernel's memory management subsystem, though some also tackle offset alignment or segment separation.

Read-Only File THP. The experimental `CONFIG_READ_ONLY_THP_FOR_FS` option [35] (hereafter, readonly file THP or RO-THP) aims to enable transparent huge pages for read-only file-backed memory—the read-only restriction arises because the kernel lacks mechanisms to handle copy-on-write faults and writeback for huge file-backed pages. A

few distributions, notably Fedora, have recently enabled this feature by default.

However, the mechanism has significant operational drawbacks. Pages are initially read into the page cache as 4 KB pages and then asynchronously copied into huge pages if conditions permit. Applications requiring synchronous promotion must explicitly invoke `MADV_COLLAPSE`, which occasionally fails on file-backed text pages [22, 31]. Moreover, RO-THP only addresses runtime mapping; without proper offset alignment and segment separation, existing binaries see no benefit even when the feature is enabled. The feature has also shown stability issues [6] and security vulnerabilities [40]. Kernel developers widely consider it a preliminary hack [23, 55] that interacts poorly with other filesystem mechanisms. While these implementation issues may be resolved over time, the fundamental limitation remains: without binary transformation for segment separation, RO-THP cannot form huge pages for code regardless of implementation maturity.

Other Kernel Modifications. Beyond readonly file THP, other kernel modification proposals have been rejected or face significant limitations. Alibaba Cloud Linux developed Hugetext [2] to address some of readonly file THP's shortcomings, adding kernel-level handling of offset alignment. However, even with these kernel modifications, Alibaba's documentation acknowledges that executables must still be relinked for proper segment separation, recommending users modify linker scripts to force alignment of code and data segments [2]. Concerns about maintainability and architectural cleanliness led to the patches' rejection by upstream Linux maintainers [24]. Taking a different approach, Zhou et al. [58] propose kernel modifications to handle both offset alignment and segment separation transparently, but their solution often maps non-code data as executable, creating additional return-oriented programming (ROP) gadgets that undermine modern hardware control-flow integrity protections.

Native Large Folio Support. Native large folio support for Linux filesystems [25] is intended to eventually supersede readonly file THP by properly integrating huge pages into the filesystem layer. However, this support remains incomplete: filesystems cannot collapse or merge existing 4 KB pages into large folios after initial page faults, and not all filesystems support large folios. Critically, without relinking executables for proper segment separation, native large folio support provides no benefit for executable code—the kernel cannot allocate large folios for segments that share huge page boundaries regardless of filesystem capabilities.

2.3 Relinking-Based Approaches

Relinking addresses both offset alignment and segment separation through linker options: the maximum page size controls offset alignment, while the common page size controls segment separation by adding padding between segments with different permissions. However, relinking alone does not address runtime mapping—even properly aligned executables are unlikely to achieve huge page mappings without loader support.

One might ask why executables are not built with huge page alignment by default. Relinking with huge page alignment pads *every* segment boundary to 2 MB, including data segments that derive no performance benefit from huge pages. Relocation read-only (RELRO) further doubles this data-segment cost by splitting the writable region into two segments with distinct post-relocation permissions [50], each requiring its own alignment padding. Combined with concerns about embedded systems [21], security policies that prohibit sharing pages between code and data [49], and the fact that huge pages for file-backed memory are not enabled by default in most OSes, relinking remains unattractive as a general solution.

The scale of this challenge is substantial: enterprise surveys reveal that 55% of organizations rely on proprietary software [52], while 62% of IT teams continue to depend on legacy systems [46]. Binary format changes propagate slowly: the a.out to ELF migration took nearly three decades [12], and PIE adoption progressed gradually despite clear security benefits, with major distributions only enabling it by default in the mid-2010s. Even organizations willing to maintain their own relinked binaries face significant costs: Chromium requires over 90 minutes to build [30], and database servers need coordinated toolchains and continuous rebuilds for each security advisory.

2.4 Binary Transformation Approaches

Beyond relinking from source, another approach involves transforming already-compiled binaries to enable huge page support. These tools operate on existing executables without requiring source code access.

Post-link Optimization. Tools like BOLT [41], Egalito [56], and RetroWrite [17] can theoretically realign segments, but they require disassembly and heuristic identification of cross-section references. Our experiments show that BOLT crashes on critical system libraries like `libc.so.6`, and that Redis occasionally crashes after BOLT transformation (see Section 5 for details).

Simple Alignment Tools. Tools like `objcopy` can modify section alignment in ELF binaries, addressing offset alignment but not segment separation. However, fixing offset alignment alone still requires loader support to actually map huge pages. As we show in Section 4.2, even with both offset alignment and loader support, without proper segment separation significant code remains outside huge page boundaries, limiting the performance benefit.

Real-World Closed-Source Layouts. Surveying widely deployed closed-source applications reveals three layout classes that block huge pages: *modern 4-LOAD* binaries (e.g., MegaSync, WebEx) where code and rodata occupy distinct LOAD segments but neither is 2 MB-aligned; *lld-linked* binaries (e.g., Sublime Text, Slack) whose executable LOAD segment has $p_{\text{vaddr}} \bmod 2\text{MB} \neq p_{\text{offset}} \bmod 2\text{MB}$, ruling out file-backed huge pages even under RO-THP; and *legacy 2-LOAD* binaries (e.g., libjvm.so) that pack code and rodata into one RX segment. Hugifier's segment-end alignment addresses all three classes without source code or relinking.

2.5 Hardware-Based Approaches

TLB Compression. Some architectures provide alternative approaches through hardware features that compress contiguous page table entries into single TLB entries, effectively creating larger mappings without full huge pages. ARM64 supports contiguous page table entries (ContPTE), which allow the kernel to map 64 KB regions using contiguous 4 KB page table entries that the hardware can coalesce into a single TLB entry [5]. RISC-V provides a similar mechanism through its Svnapot extension [44]. While ARM64 executables are naturally aligned to 64 KB boundaries in their ELF headers, ContPTE mappings rarely materialize for executable code due to runtime mapping limitations. Roberts' recent kernel patch [45] addresses this by extending the readahead mechanism to prefetch 64 KB folios for executable memory, facing challenges analogous to those we address for 2 MB huge pages—any pre-existing 4 KB pages in the cache prevent contiguous mapping formation.

While TLB compression offers a compromise between 4 KB and 2 MB pages, it has limitations: many CPUs (including x86) lack hardware support for coalescing contiguous PTEs into a single TLB entry, restricting this approach to specific architectures like ARM. Additionally, the performance benefits are more modest compared to full 2 MB huge pages, as demonstrated in our evaluation (Section 4.4).

3 Design

Our design enables huge page support for executable code without requiring modifications to the OS or loader, and without relinking the binary while maintaining code sharing between processes, debugging capabilities, and security properties. The binary transformation addresses both offset alignment and segment separation, while a runtime component overcomes runtime mapping limitations. Together with automated transformation of library dependencies and their debug information, this provides a complete solution.

Our approach assumes the following about target binaries:

- Position-independent code (PIC/PIE), as used by 95% of executables on a standard Ubuntu installation.

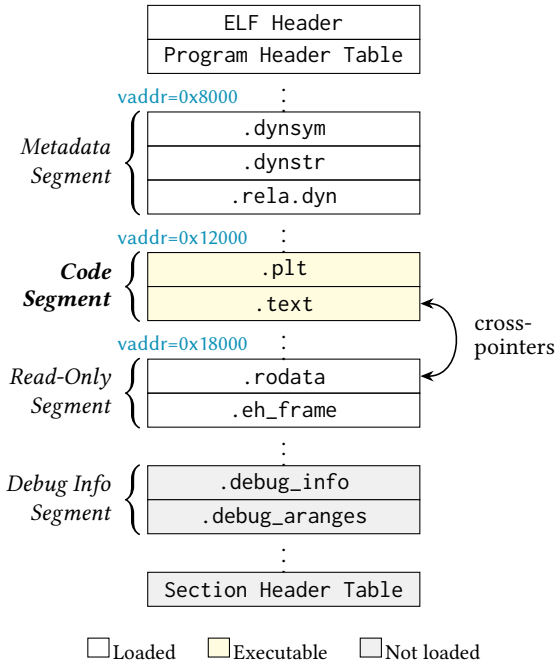

Figure 2. Typical layout of an ELF binary. File offsets typically are not aligned to huge pages. Most segments require distinct permissions and hold cross-references, preventing independent realignment.

- A single contiguous executable segment per binary—the standard output of `gcc` and `clang`; `-ffunction-sections` creates multiple compiler sections, but linkers merge them into one loadable segment.
- No interleaving of data within the code segment.

No source code, special compiler flags, or debug symbols are required. The transformation operates on standard ELF binaries as distributed by package managers. We discuss limitations and edge cases in Section 3.4.

3.1 Code-Oblivious ELF Realignment

Achieving segment separation requires realigning code segments to huge page boundaries within existing binaries. Figure 2 illustrates the challenge: ELF organizes programs into segments with distinct permissions—metadata, code, read-only data—where cross-segment references (e.g., between `.text` and `.rodata`) use relative offsets established during linking. While Position Independent Code (PIC) allows relocating the entire binary as a unit, these relative offsets prevent moving segments independently.

We consider using binary rewriting tools that lift code to intermediate representations, but their reliance on heuristic reference identification makes them unreliable on diverse production binaries (Section 5). Instead, we avoid disassembly entirely.

Uniform Segment Shifting. Our key insight is that we can shift code and data segments forward together while preserving their relative distances, avoiding disassembly or intermediate representation reconstruction entirely. The first loaded segment—containing dynamic symbol tables, string tables, and relocation records—remains at its original location, allowing us to change its relative position to the rest of the executable without affecting runtime behavior.

Figure 3 illustrates this transformation on `libc.so`. In the original layout (a), the 1.6 MB code segment starts at 0x28000, misaligned with any 2 MB boundary. After transformation (b), we shift it to 0x278000 so that it ends at 0x400000, placing the entire code within the 2 MB–4 MB huge page region. The first loaded segment stays in place; only the code and subsequent segments shift forward. Since cross-segment references use relative offsets, shifting all segments by the same amount preserves program correctness—provided we update absolute addresses in ELF metadata and adjust file offsets affected by the shift.

This approach offers three key advantages. First, unlike kernel-level approaches that map non-code data as executable to achieve alignment [58], our transformation preserves the original permission boundaries: the first segment stays read-only, and no additional executable regions are created. Second, position-independent executables already embed all relocation information needed for this repositioning—the same information the dynamic loader uses at runtime. Unlike tools such as BOLT that require binaries compiled with pre-linkage relocation data for reliable transformation, our approach works with standard PIC binaries. Finally, unlike copy-and-remap approaches such as IODLR and libhugetlbfs that copy code to anonymous memory—losing file-backed benefits and only partially mapping code with huge pages due to alignment constraints (Section 4.2)—our approach enables complete huge page coverage while preserving file-backed mappings.

ELF Transformation. The transformation involves two phases: inserting padding to achieve alignment, and updating ELF metadata to restore binary correctness (Figure 3). The first phase is straightforward; the second requires careful updates to several interrelated data structures.

For alignment, our solution analyzes the ELF structure to identify the executable segment and calculates new addresses and file offsets that align the segment end with a 2 MB huge page boundary. We update the executable segment's virtual address (`p_vaddr`) and file offset (`p_offset`) in the program header, addressing segment separation. To handle offset alignment, we set the program header alignment field (`p_align`) to the huge page size (2 MB) for the first loadable segment, enabling the loader to maintain this alignment during process initialization or shared library loading. The executable segment's memory and file sizes (`p_memsz` and `p_filesz`) are rounded up to the huge page boundary. We

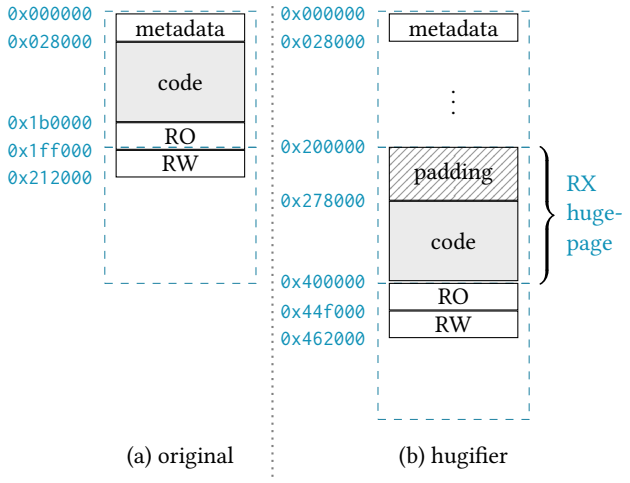

Figure 3. ELF transformation of `libc.so`: (a) original layout with unaligned code segment; (b) after Hugifier inserts padding to align the code segment to a 2 MB huge page boundary. Hatched region is filled with trap instructions.

add padding filled with breakpoint instructions (`int3` on x86, `brk #0` on ARM) between segments to prevent any segment from sharing huge pages with other segments, preserving the W⊕X protection model.

At this point, the binary is broken: segment addresses have changed but ELF metadata still references the old locations, and relocation entries point to incorrect offsets. We restore correctness by updating program headers to reflect new segment addresses and offsets, adjusting section headers to maintain correct placement, and updating the entry point. Dynamic section entries containing memory addresses such as `DT_PLTGOT`, `DT_INIT`, and `DT_FINI` are updated to reference their new locations. Symbol table entries in both `.dynsym` and `.symtab` sections are adjusted to reflect new addresses.

Relocations are central to our approach: they are the mechanism by which PIC binaries inform the dynamic loader where to apply fixups when relocating segments at load time. Since these are the same fixups the loader applies at runtime, we can update them to reflect the new segment locations without any disassembly, though the updates themselves require care. Each relocation's offset (`r_offset`) indicates where to apply the fix and must be updated to point into the shifted segments. The target value—the absolute address being relocated—also becomes stale, but how we update it depends on the encoding. For RELA-format relocations, the target is stored in an explicit addend field (`r_addend`), which we adjust directly. For REL-format relocations, no such field exists; the absolute address is embedded inline at the target location in the binary, so we apply the shift delta to the value at that file offset. RELR relocations use a compact bitmap

representation and require special processing to update both direct and bitmap-encoded entries.

Beyond relocations, other structures embed absolute addresses that must be updated after the shift. For executables containing User Statically-Defined Tracing (USDT) probes, we update address references in the `.note.stapsdt` section. Similarly, exception handling tables and GNU hash tables contain segment-relative addresses that we adjust to match the new layout.

Debug Information Handling. DWARF (Debugging With Attributed Record Format) debug information enables source-level debugging and profiling by maintaining mappings between code addresses and source locations. When we shift code segments to new addresses during binary transformation, all these address references become invalid—debuggers would show incorrect source lines, breakpoints would fail, and profilers would misattribute performance data. Our approach systematically updates all address references in the DWARF structures while preserving semantic relationships and format-specific encoding.

DWARF includes several interrelated structures that must be updated coherently. We parse and modify several debug sections[2]. We identify base addresses in DWARF data—absolute addresses that serve as reference points for offsets—and update them to reflect the new segment locations.

Crucially, we perform all DWARF modifications in-place, avoiding cascading changes that would require rebuilding entire debug sections. We use libdwarf to traverse debugging information entries (DIEs) but perform updates directly rather than reconstructing the debug information, modifying address attributes (`DW_FORM_addr`) and location expressions containing `DW_OP_addr`. The line-to-address mapping—essential for debuggers to correlate source code with binary instructions—requires updating `DW_LNE_set_address` opcodes while preserving state machine semantics.

Following common practice, we modify the executable's build-id by flipping a single bit so that it differs from the original. Many distributions store debug symbols in separate files identified by build-id; without this change, tools would mistakenly load the original's debug information rather than the updated version matching our transformed binary.

3.2 Runtime Huge Page Formation

Even with properly aligned executables, achieving huge page mappings requires cooperation between the dynamic loader and the kernel—cooperation that does not exist in practice. Dynamic loaders (such as `ld.so` in Linux) map executable segments into memory and perform relocations, but do not advise the kernel through hints like `madvise` when loading code segments. In most kernel configurations, such hints are required to enable huge page formation.

[2]Including `.debug_info`, `.debug_aranges`, `.debug_line`, `.debug_loclists`, and `.debug_rnglists`.

Even if the loader provided such hints, kernels impose additional constraints on when huge pages can form for file-backed memory—constraints designed for data access patterns rather than code execution. If these constraints are not satisfied, huge pages either fail to materialize entirely, or—with features like Linux's readonly file THP—form only asynchronously. While the loader could use `MADV_COLLAPSE` to request promotion, this may fail for various internal reasons [22, 31].

Proactive Huge Page Population. We overcome these runtime mapping limitations through two principles: proactive huge page population of both page cache and page tables before execution begins, and visibility into mapping state to verify success. Unlike RO-THP, which reads pages at 4 KB granularity and later attempts to copy them into huge pages, our approach triggers huge page allocation directly—code arrives in the page cache as large folios from the start, with no subsequent copying or promotion. We explicitly read executable content in 2 MB-aligned sequential accesses to trigger this allocation, then use OS facilities to inspect how content is mapped in both the page cache and page tables, allowing us to detect and remedy cases where small pages were allocated instead.

Runtime Components. We implement these principles using two components: a dispatcher process that prefetches statically-known dependencies before execution, and an `LD_AUDIT` library that handles dynamically-loaded libraries (via `dlopen`) at runtime. The dispatcher walks the dependency tree (similar to `lddtree`), prefetching each library into 2 MB-aligned page cache entries before continuing the scan—unlike standard dependency resolution which would fragment the cache with non-sequential accesses. After prefetching all dependencies, the dispatcher executes the target process. Our `LD_AUDIT` library then intercepts library loading events (`la_objopen`) to apply the same prefetching logic to any libraries loaded dynamically during execution.

The prefetching process works as follows. For each executable segment, we first query the page cache state using the `cachestat` system call. If uncached, we prefetch it with sequential 2 MB accesses to trigger the kernel's huge page readahead. If already cached with small pages, we must evict and reload: `FADV_DONTNEED` attempts to remove pages from the cache, though it may not always succeed. After eviction, we remap with `MAP_FIXED` and fault each 2 MB boundary to establish huge page mappings. We use `MADV_HUGEPAGE` to mark these regions for huge page promotion, which is preferred over system-wide "always" policies that promote indiscriminately. Finally, we verify success by checking `/proc/self/smaps` to confirm huge page mappings were actually established.

The runtime relies only on stable, documented interfaces (`madvise`, `MAP_FIXED`, `cachestat`, `/proc/self/smaps`) and on the documented THP behavior of allocating large folios

on sequential readahead; we avoid `MADV_COLLAPSE` because of the file-backed promotion failures noted above.

3.3 Implementation

Our implementation comprises about 4,200 lines of code: a binary converter (2,448 lines of C) that handles ELF manipulation, a loader override library (1,527 lines), and Python and Bash helper scripts. It runs on standard Linux kernels (tested on 6.14) with only the THP support enabled by virtually all distributions.

3.4 Limitations

Our approach requires position-independent code (PIC/PIE), though in practice 95% of executables on our Ubuntu installation already meet this requirement. Additionally, it may fail with unconventionally structured executables where data interleaves with code segments, though we have not encountered such cases.

Memory Overhead. Binary transformation pads each transformed executable by up to 4 MB. For PostgreSQL (65 dependencies), padding all binaries larger than 4 KB inflates the total by 42.4 MB (364%); a 1 MB threshold reduces this to 2.5 MB (21%). Apache2 (203 modules, never loaded simultaneously) is the extreme case: 176.2 MB (2,215%) without a threshold, 0.9 MB (11%) with one.

This overhead does translate to memory consumption in the page cache, but crucially does not grow with the number of concurrent process invocations—executables are shared across all processes. Additionally, padding regions contain only trap instructions and are never loaded into CPU caches, minimizing performance impact. As we show in Section 4.3, selective application with the 1 MB threshold can actually improve performance by avoiding microarchitectural conflicts.

Security Considerations. Our approach using 2 MB huge pages reduces ASLR entropy by 9 bits compared to 4 KB pages. Linux provides 28 bits of entropy by default for executable mappings on 64-bit systems [37], already a compromise from the theoretical maximum to limit page table overhead—and distributions like Ubuntu have increased this to 32 bits precisely to accommodate PMD-aligned mappings [16]. Moreover, ASLR's effectiveness has been repeatedly challenged through brute-force attacks [9] and side-channel attacks [19]. Given these fundamental vulnerabilities, the 9-bit reduction from 2 MB pages represents a marginal change to an already imperfect defense. Our 2 MB approach balances security and performance while preserving all other PIC/PIE protections.

Signed Binaries. Our transformation invalidates existing binary signatures: Windows executables would fail Authenticode validation [15] and macOS binaries would be rejected by Gatekeeper [26]. On systems with mandatory code signing, binaries must be re-signed after transformation. Our transformation is predictable and auditable, suitable for deployment pipelines that can incorporate re-signing where required.

Applicability. In environments where memory is limited and the CPU architecture provides limited TLB benefit from huge pages (such as embedded systems or older processors with few huge page iTLB entries), the memory overhead introduced by our transformation may outweigh performance gains. Similarly, as discussed in Section 4.3, applications with many small libraries may experience microarchitectural conflicts when all are 2 MB-aligned. These environments are better served by selective application.

4 Evaluation

Our evaluation addresses three key questions:

1. How does Hugifier compare to existing solutions?
2. How important is binary transformation compared to ideal loader support?
3. Should huge pages always be applied to all executables?

Experimental Setup. We evaluate PostgreSQL (pgbench TPC-B), MySQL (sysbench OLTP), Apache2 (autocannon with PHP), Node.js (autocannon with cluster module), and compilers g++ and clang++ (tramp3d-v4.cpp from LLVM test suite). Experiments run on x86 (Intel Sapphire Rapids with 256 L1 iTLB entries for 4 KB pages, 32 for 2 MB) and ARM (AWS Graviton3 with 48 iTLB entries). Database and web server benchmarks utilize 8 cores with workload generators on a separate NUMA node; compiler benchmarks run single-threaded. Each configuration runs 5 iterations. We compare six configurations: baseline, libhugetlbfs [34], IODLR [27], "aligned" (using our loader without binary transformation), full hugifier, and selective hugifier (hug. 1 MB+). For Node.js we use IODLR rather than its built-in large-pages option, since IODLR also maps shared libraries. Hugifier transforms all shared-library dependencies of our applications that meet the Section 3 prerequisites. Dispatcher startup adds about 25 ms on our heaviest dependency tree (Java/Cassandra), small relative to runtime for all but the shortest-lived processes.

Across all configurations, performance improvements correlate positively with iTLB miss reduction (Figure 4; r=0.93 on x86, r=0.88 on ARM64), confirming that huge page optimization drives these gains. The correlations are not perfect for several reasons: our x86 measurements capture only retired instruction iTLB misses, excluding speculative fills; huge pages also reduce L2 TLB misses and page walk latency not reflected in L1 iTLB counters; and application performance depends on additional factors including I/O latency and branch prediction. Nevertheless, the consistent positive

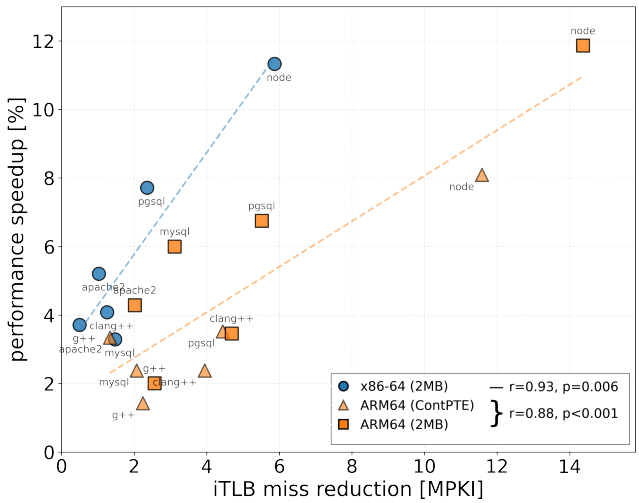

Figure 4. Correlation between iTLB MPKI reduction and performance speedup across workloads.

correlation validates that iTLB miss reduction—not incidental factors—underlies the observed speedups. Data TLB miss rates remain essentially unchanged across Hugifier's configurations, further confirming that the gains stem from code segment huge pages.

4.1 How Does Hugifier Compare to Alternative Solutions?

Figure 5 compares baseline (leftmost), two copy-and-remap solutions (libhugetlbfs and IODLR), readonly file THP kernel support (RO-THP), and Hugifier's configurations (rightmost). We focus on x86 as these solutions are architecture-agnostic with identical behavior across platforms.

Copy-and-Remap Solutions. Libhugetlbfs provides inconsistent gains as it is limited to main executables only, while IODLR handles libraries but remaps file-backed memory as anonymous. Both break system functionality: libhugetlbfs prevents memory reclamation and NUMA migration, while IODLR prevents process memory sharing.

Hugifier achieves equal or better performance than copy-and-remap solutions across our benchmarks. Apache2 exemplifies where copy-and-remap approaches fail: libhugetlbfs and IODLR provide virtually no benefit as they cannot handle Apache2's numerous dynamic shared objects (DSOs) smaller than 2 MB—Apache2 loads over 50 libraries under 1 MB. In contrast, Hugifier's ability to transform and map these small libraries delivers 5.2% performance improvement.

While libhugetlbfs's limitation to main executables impacts most applications, clang++ exemplifies this: it shows no benefit from libhugetlbfs despite iTLB pressure, as its functionality resides in LLVM shared libraries that libhugetlbfs cannot handle. Hugifier's library support ensures consistent improvements across diverse architectures without requiring pre-allocated huge page pools or breaking functionality.

Readonly File THP Kernel Support. We evaluate RO-THP by rebuilding the Ubuntu kernel with `CONFIG_READ_ONLY_THP_FOR_FS` enabled. Since using a dynamic loader hook to synchronously promote code to huge pages proved unreliable [22], we configured THP to promote all eligible memory system-wide.

The results are underwhelming. PostgreSQL shows 0% iTLB reduction—essentially no benefit for code—while its 2% performance gain likely reflects THP promoting anonymous data pages. MySQL suffers -2% performance degradation despite 1% iTLB reduction; since global THP forces 2 MB-aligned virtual address placement, the resulting address layout changes may introduce microarchitectural conflicts in MySQL's hot paths. Only Node.js shows meaningful iTLB improvement (41% reduction, 4.1% speedup), but with high variance across runs—likely from non-deterministic interaction between THP promotion and JIT-compiled code. These results illustrate the risks of enabling THP globally rather than selectively [48]: workloads respond unpredictably, and without binary transformation for segment separation, code segments see little benefit regardless.

4.2 How Important Is Binary Transformation?

As our RO-THP evaluation showed, current loaders lack reliable mechanisms to trigger huge page mappings for executables. But if proper kernel and loader support existed—solving the runtime mapping problem—to what extent would binary transformation still matter? To answer this, we test an "aligned" configuration: executables and libraries larger than 2 MB have their ELF header alignment modified via objcopy without adjusting file offsets, then loaded using our dispatcher to ensure 2 MB-aligned mapping. This represents the best-case scenario for any future solution that addresses offset alignment and runtime mapping but not segment separation.

Figure 5 shows mixed results. MySQL achieves similar performance with both approaches—its large executable (>23 MB) naturally achieves good huge page coverage even with simple alignment. However, PostgreSQL, Node.js, and Apache2 demonstrate where Hugifier's full transformation excels: PostgreSQL achieves 6.9% versus "aligned"'s 4.4%, Node.js shows 11.2% versus 8.3%, and Apache2 gains 5.2% versus virtually no improvement with alignment alone.

The "aligned" configuration occasionally performs slightly worse than IODLR. This occurs because typical ELF layouts place a small read-only segment before the executable segment. When forced to 2 MB alignment, this first segment wastes the initial portion of the huge page region, leaving less executable code on huge pages. In contrast, IODLR copies only executable code to anonymous memory, where ASLR may position it more favorably within huge page boundaries.

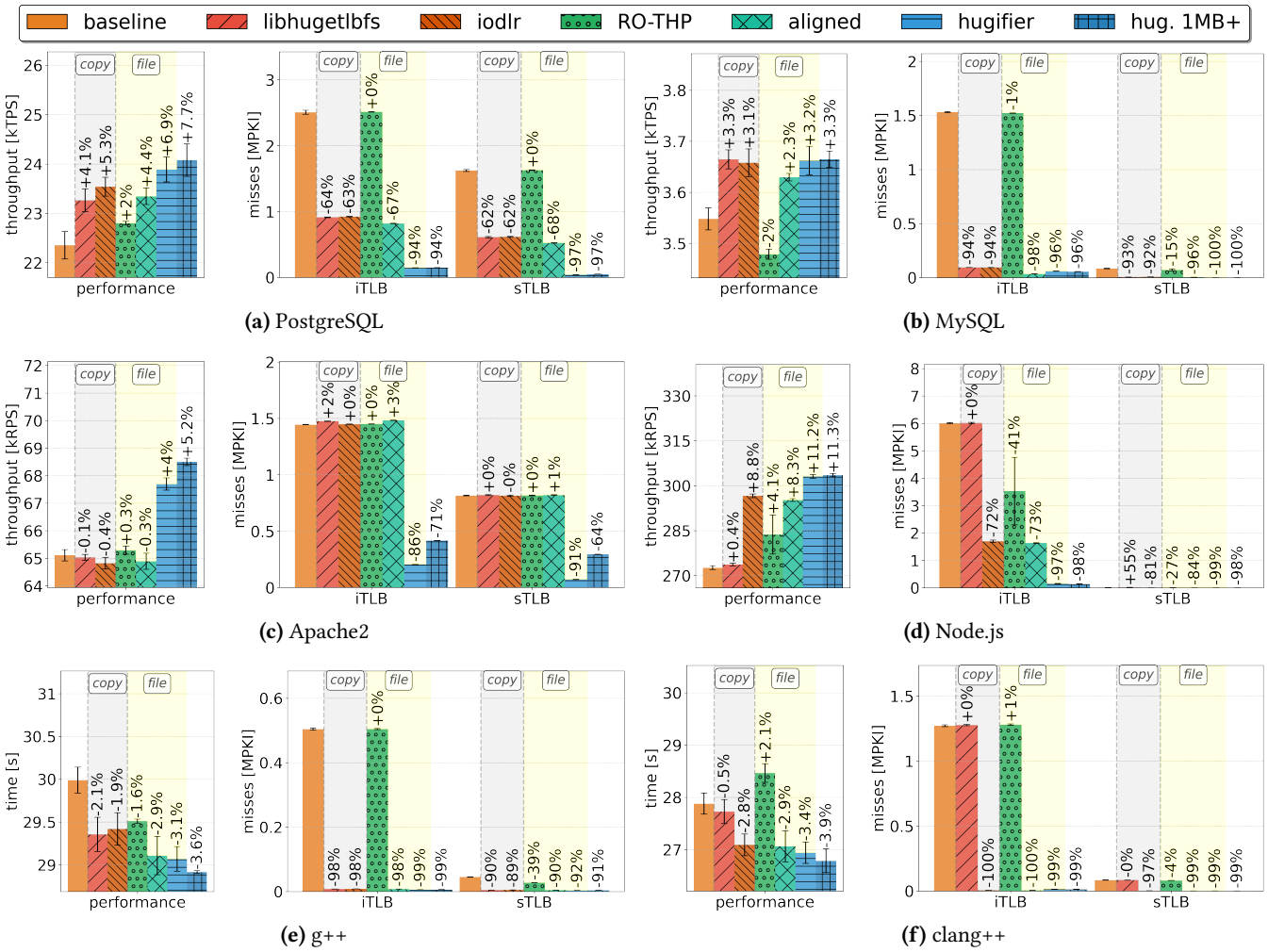

Figure 5. x86 benchmark performance and TLB Misses Per Kilo Instructions (MPKI) comparing baseline, copy-and-remap solutions (copy), and file-backed approaches (file) including RO-THP kernel support and Hugifier.

This highlights a fundamental limitation: offset alignment ensures the segment starts at a huge page boundary but cannot ensure code fills complete huge pages. PostgreSQL's 6 MB executable demonstrates this—without segment separation, only 4 MB can be mapped with huge pages, and since PostgreSQL forks a process per connection, this overhead multiplies.

The benefits are also evident on ARM64 (Figure 6). Despite ARM64 executables being naturally aligned to 64 KB boundaries, alignment without padding is insufficient. Apache2 gains an additional 1.8 percentage points with full transformation, Node.js gains 4 percentage points, and MySQL shows 2.4 percentage points improvement. In half our workloads, padding provides considerable benefits by ensuring optimal huge page coverage.

4.3 Should All Executables Use Huge Pages?

Given that each transformed library requires up to 4 MB additional space, one might apply huge pages selectively to conserve resources. We evaluated a "hug. 1 MB+" configuration that applies Hugifier only to executables and libraries over 1 MB, expecting to trade performance for reduced overhead.

Surprisingly, selective application can improve performance in certain cases. While most workloads show identical performance with selective versus full coverage, Apache2 demonstrates notable differences. On x86, selective achieves 5.2% performance versus 4% with full coverage—a considerable benefit. The effect is even more pronounced on ARM64 (Figure 6): selective delivers 4.3% improvement while full Hugifier causes a 0.5% degradation despite 80% iTLB miss reduction.

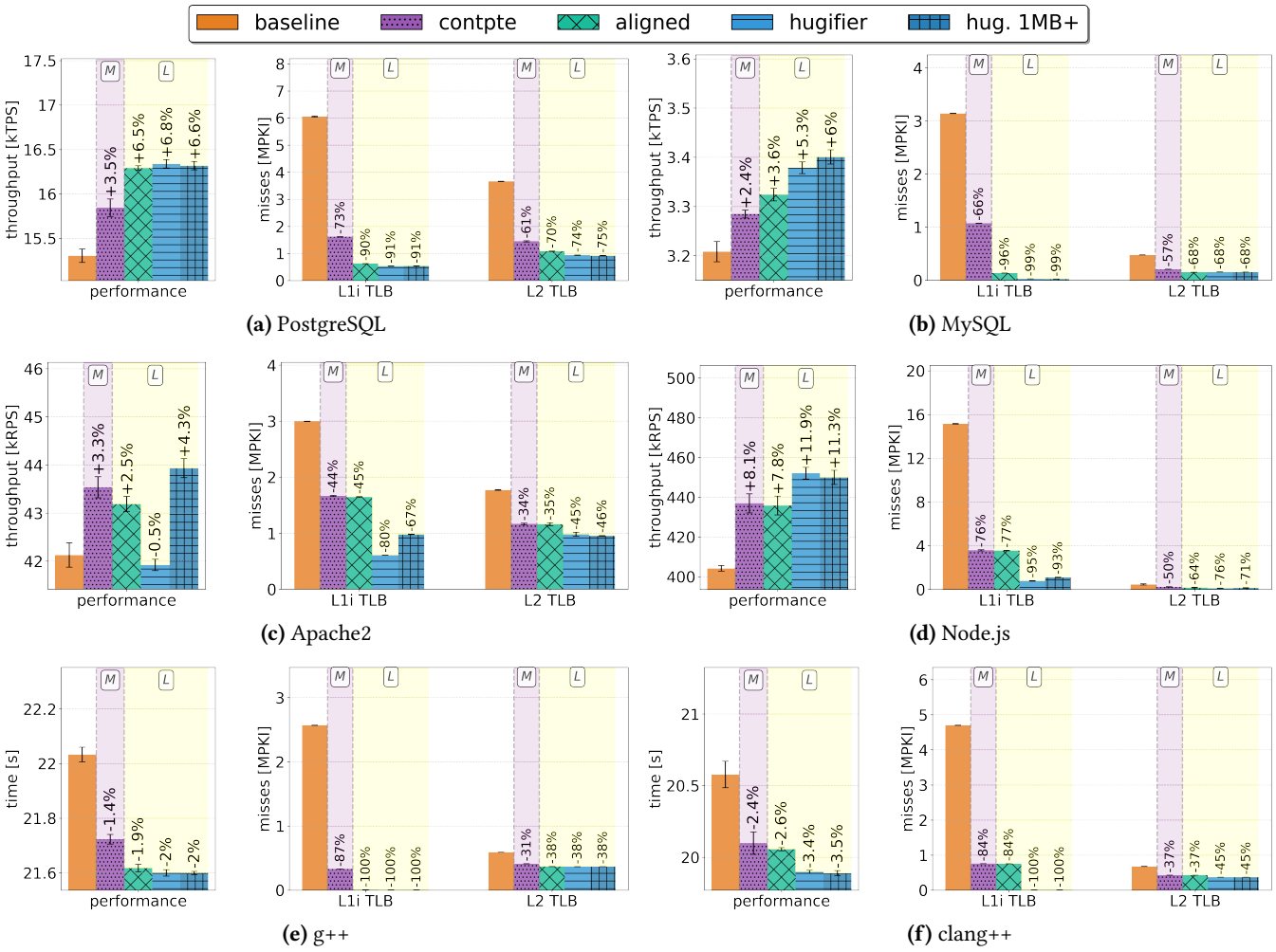

Figure 6. ARM benchmark performance and TLB Misses Per Kilo Instructions (MPKI). "M" indicates configurations using 64 KB pages via contiguous page table entries, while "L" indicates Hugifier's configurations using 2 MB large pages.

We attribute this performance difference to microarchitectural conflicts when small libraries align to identical 2 MB boundaries. One instance we observe is increased branch target buffer (BTB) conflicts—the BTB predicts branch targets based on instruction addresses. On Intel Sapphire Rapids processors, our hypothesis is supported by a 69% increase in BTB misses (from 0.6 to 1.1 per kilo instructions) when using full huge page coverage. We also measured L1i, L1d, L2, and LLC miss rates and branch-prediction accuracy across all configurations; the BTB conflict reported here was the only consistent adverse signal.

Apache2 exemplifies why selective huge page application matters. With over 50 dynamic shared objects (DSOs) under 1 MB, forcing all these small libraries to 2 MB alignment can cause branches from different libraries to map to identical predictor indices.

Focusing on executables and libraries larger than 1 MB achieves the same—and sometimes better—performance, even with higher iTLB misses. The optimal threshold varies with processor microarchitecture and workload, but 1 MB provides a reasonable default.

4.4 Comparison with TLB Compression

On ARM64, hardware TLB compression through contiguous page table entries (ContPTE) allows multiple contiguous PTEs to be compressed into single TLB entries. While this hardware support has existed, it rarely materialized for executables because the kernel lacked mechanisms to allocate contiguous folios during executable page faults. Recent kernel developments introduced multi-size transparent huge pages (mTHP) support [51] to leverage this capability. Roberts' work, presented at the 2025 Linux Summit [13], extends readahead to prefetch 64 KB folios on executable page faults—similar to our dispatcher. ARM64 executables align to 64 KB naturally, making them compatible with TLB compression.

We evaluate Hugifier against ContPTE on our ARM testbed running Linux 6.16-rc7 with Roberts' patch applied (recently upstreamed). Figure 6 shows ContPTE provides meaningful improvements over baseline. However, Hugifier consistently outperforms ContPTE, with particularly striking advantages in MySQL (3.6 percentage points better) and PostgreSQL (3.2 percentage points better). While 64 KB folios offer partial benefits, 2 MB huge pages deliver substantially greater performance improvements—ARM faces 2.8–3.1× higher iTLB pressure than x86.

mTHP has inherent limitations: 64 KB folios deliver lower performance than 2 MB huge pages, and TLB compression is architecture-specific—Intel CPUs lack hardware support for compressing contiguous 4 KB PTEs. The approaches are complementary: mTHP reduces the importance of mapping small executables with huge pages on ARM64, while Hugifier delivers maximum performance across all architectures.

5 Related Work

Approaches to reducing address translation overhead for executable code span hardware optimizations, kernel modifications, runtime techniques, and binary transformation.

Hardware Translation Optimization. Basu et al. [7] propose direct segments to bypass page walks entirely for contiguous virtual address ranges. ARM's ContPTE [45] and RISC-V's Svnapot [44] coalesce contiguous page table entries into single TLB entries, offering a compromise between 4 KB and 2 MB pages—as we show in Section 4.4, full 2 MB huge pages provide substantially greater benefit.

OS Huge Page Management. Ingens [33] coordinates huge page promotion with memory pressure to avoid bloat, Hawkeye [42] adds fine-grained utilization tracking, and Quicksilver [60] balances incremental and eager promotion to reduce latency spikes. These policies improve huge page utilization for anonymous memory but do not address file-backed executable code.

Copy-and-Remap Solutions. The most common approach copies code into huge pages at runtime. Libhugetlbfs [34] uses pre-allocated hugetlbfs pages, while IODLR [27] copies into anonymous memory. Both abandon file-backed semantics, breaking cross-process code sharing, memory reclamation under pressure, and debugging tools [28, 54], trading OS memory management correctness for huge page mappings.

Load-Time Relayout. Prelink [29] pre-computes relocation results into on-disk binaries, reducing dynamic linking overhead but requiring re-prelinking on library updates. iFed [43] goes further, concatenating same-type sections from all libraries into contiguous huge page regions at load time. While iFed achieves significant iTLB reductions, it incurs 4–6× loading overhead, requires recompilation with `-emit-relocs`, and breaks cross-process library sharing since each process produces a unique layout.

Binary Transformation. BOLT [41] can enable huge page support as a side effect of post-link optimization, but relies on heuristic reconstruction of cross-section references—our experiments show it crashes on critical libraries and production binaries. Egalito [56] and RetroWrite [17] lift binaries to intermediate representations for security transformations; while their IR-based approach could in principle be extended to relayout segments for huge page alignment, both rely on heuristic reference identification that limits reliability: Egalito reports only 60% success on Debian packages, while RetroWrite cannot handle C++ due to exception handling complexity.

Kernel Modifications. Linux's readonly file THP [35] is a preliminary mechanism for file-backed huge pages that suffers stability and security issues [6, 40]. Alibaba's Hugetext [2] extends this but still requires relinking for segment separation. Native large folio support [25] is intended as a replacement but remains incomplete and cannot help executables lacking proper segment separation regardless of filesystem capabilities. Zhou et al. [58] handle alignment transparently but map non-code data as executable, undermining control-flow integrity protections.

6 Conclusion

Kernel support for executable huge pages is emerging, but full benefits remain years away, requiring coordinated changes across kernels, loaders, and build systems. Hugifier bridges this gap today, delivering up to 11.3% speedup on production workloads while preserving file-backed semantics—cross-process code sharing, memory reclamation, and debugging all continue to function correctly. Our results show 2 MB huge pages consistently outperform ARM's ContPTE, and that selective application to larger executables balances performance with resource efficiency.

Whether readonly file THP eventually provides a complete kernel solution is open; either way, segment separation remains necessary and the installed base of binaries will not be relinked overnight. The same structural challenges apply to Windows PE, where closed-source software dominates.

Availability

Source code is at: https://github.com/anadav/hugifier.

Acknowledgments

This research was supported by the Israel Science Foundation (grant No. 3310/25).

References

[1] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)* 52, 4 (April 2017), 631–644. doi:10.1145/3093336.3037706

[2] Alibaba Cloud. 2025. Hugetext. https://www.alibabacloud.com/help/en/alinux/user-guide/hugetext. accessed: 2025-05-08.

[3] Nadav Amit. 2025. elflink.c: Fix HUGETLB_FORCE_ELFMAP partial segment remapping. https://github.com/libhugetlbfs/libhugetlbfs/commit/166588f8015fcf01a12230b31db584391132673e. libhugetlbfs GitHub commit.

[4] Andrea Arcangeli. 2011. Transparent Hugepage Support. Linux Foundation Collaboration Summit, https://events.static.linuxfound.org/slides/2011/lfcs/lfcs2011_hpc_arcangeli.pdf. Accessed: 2024-12-31.

[5] Arm Limited. 2025. *Arm Neoverse V1 Core Technical Reference Manual.* Revision: r1p2, Document ID: 101427_0102_04_en.

[6] Vlastimil Babka. 2022. read() data corruption with CONFIG_READ_ONLY_THP_FOR_FS=y. Linux Kernel Mailing List, https://lore.kernel.org/all/df3b5d1c-a36b-2c73-3e27-99e74983de3a@suse.cz/.

[7] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. 2013. Efficient virtual memory for big memory servers. *ACM/IEEE International Symposium on Computer Architecture (ISCA)* 41, 3 (2013), 237–248. doi:10.1145/2485922.2485943

[8] Ravi Bhargava, Ben Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 26–35. doi:10.1145/1346281.1346286

[9] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *IEEE Symposium on Security and Privacy (SP)*. 227–242. doi:10.1109/SP.2014.22

[10] Neil Brown. 2016. Transparent huge pages in the page cache. LWN.net, https://lwn.net/Articles/686690/. Accessed: 2023-11-02.

[11] Dimitrios Chasapis, Georgios Vavouliotis, Daniel A Jiménez, and Marc Casas. 2025. Instruction-Aware Cooperative TLB and Cache Replacement Policies. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 619–636. doi:10.1145/3669940.3707247

[12] Jonathan Corbet. 2022. Removing a.out support. https://lwn.net/Articles/888741/. Accessed: 2025-12-02.

[13] Jonathan Corbet. 2025. Using large folios for text areas. https://lwn.net/Articles/1016416/.

[14] Intel Corporation. 2017. 5-Level Paging and 5-Level EPT. https://cdrdv2-public.intel.com/671442/5-level-paging-white-paper.pdf. Document 335252-002, Revision 1.1.

[15] Microsoft Corporation. 2021. Using SignTool to Verify a File Signature. https://learn.microsoft.com/windows/win32/seccrypto/using-signtool-to-verify-a-file-signature. Accessed 8 May 2025.

[16] Thadeu Lima de Souza Cascardo. 2023. UBUNTU: SAUCE: Maximize kernel ASLR entropy. https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/noble/commit/?h=master-next&id=760c2b1fa1f5e95be1117bc7b80afb8441d4b002. Ubuntu kernel commit noting ASLR entropy loss from PMD-aligned mappings.

[17] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *IEEE Symposium on Security and Privacy (SP)*. 1497–1511. doi:10.1109/SP40000.2020.00009

[18] Stack Overflow discussion. 2017. OS X `mmap` with `VM_FLAGS_SUPERPAGE_SIZE_2MB`. https://stackoverflow.com/questions/47129067. Accessed 8 May 2025.

[19] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *IEEE/ACM International Symposium on Microarchitecture*. 1–13. doi:10.1109/MICRO.2016.7783743

[20] Unix & Linux Forum. 2006. mmap(2) [osx man page]. https://www.unix.com/man-page/osx/2/mmap. Accessed: 2024-12-31.

[21] FreeBSD Bugzilla. 2019. Bug 237676: LLD Filesize and default option suggestions. https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=237676. Accessed: 2025-08-18.

[22] Shivank Garg. 2025. madvise(MADV_COLLAPSE) fails with EINVAL on dirty file-backed text pages. Linux Kernel Mailing List, https://lore.kernel.org/all/4e26fe5e-7374-467c-a333-9dd48f85d7cc@amd.com/.

[23] Christoph Hellwig. 2021. Re: [PATCH 0/3] mm, thp: introduce a new sysfs interface to facilitate file THP for .text. https://lore.kernel.org/all/YWPwjTEfeFFrJttQ@infradead.org/. Linux kernel mailing list.

[24] Christoph Hellwig. 2021. Re: [PATCH 0/3] mm, thp: introduce a new sysfs interface to facilitate file THP for .text. https://lore.kernel.org/all/YWPwjTEfeFFrJttQ@infradead.org/.

[25] David Hildenbrand. 2025. Re: [PATCH v11 08/18] KVM: guest_memfd: Allow host to map guest_memfd pages. https://lore.kernel.org/all/6cf86edb-1e7e-4b44-93d0-f03f9523c24a@redhat.com/. Linux kernel mailing list.

[26] Apple Inc. 2024. Resolving Common Notarization Issues. https://developer.apple.com/documentation/security/resolving-common-notarization-issues. Accessed 8 May 2025.

[27] Intel Corporation. 2024. Runtime Performance Optimization Blueprint: Intel Architecture Optimization with Large Code Pages. https://www.intel.com/content/dam/develop/external/us/en/documents/runtimeperformanceoptimizationblueprint-largecodepages-q1update.pdf. Accessed: 2024-09-16.

[28] IODLR Contributors. 2019. perf output will not have symbols #9. https://github.com/intel/iodlr/issues/9. GitHub Issue, last activity: 2023.

[29] Jakub Jelinek. 2004. *Prelink*. Technical Report. Red Hat, Inc. https://people.redhat.com/jakub/prelink.pdf

[30] Solomon Kinard. 2021. Chromium Build Time on 2021 MacBook Pro M1. https://medium.com/@solomonkinard/chromium-build-time-on-2021-macbook-pro-m1-bcdc0a8b10c8. Accessed 8 May 2025.

[31] Avi Kivity. 2024. Possible regression with file madvise(MADV_COLLAPSE). Linux Kernel Mailing List, https://lore.kernel.org/all/8ac28fb858a2394cc72c3dc5924f1fd031fc6fe0.camel@scylladb.com/.

[32] Mike Kravetz. 2017. Hugetlbfs Reservation. https://www.kernel.org/doc/html/v5.1/vm/hugetlbfs_reserv.html. Accessed: 2024-11-17.

[33] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with Ingens. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*. 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[34] Linux Programmer's Manual. 2024. libhugetlbfs - Library to support large pages on Linux. https://linux.die.net/man/7/libhugetlbfs. Accessed: 2024-09-16.

[35] Song Liu. 2019. Enable THP for text section of non-shmem files. Linux Kernel Mailing List, https://lore.kernel.org/all/20190613052151.3782835-1-songliubraving@fb.com/.

[36] H.J. Lu, Kshitij Doshi, Rohit Seth, and Jantz Tran. 2006. Using Hugetlbfs for Mapping Application Text Regions. In *Ottawa Linux Symposium (OLS)*. 83–90. https://www.kernel.org/doc/ols/2006/ols2006v2-pages-83-90.pdf

[37] Hector Marco-Gisbert and Ismael Ripoll Ripoll. 2019. Address Space Layout Randomization Next Generation. *Applied Sciences* 9, 14 (2019). doi:10.3390/app9142928

[38] Timothy Merrifield and H. Reza Taheri. 2016. Performance Implications of Extended Page Tables on Virtualized x86 Processors. In *ACM/USENIX International Conference on Virtual Execution Environments (VEE)*. 25–35. doi:10.1145/2892242.2892258

[39] Microsoft. 2021. Large-Page Support. https://learn.microsoft.com/en-us/windows/win32/memory/large-page-support. Accessed: 2024-12-31.

[40] National Institute of Standards and Technology. 2024. CVE-2024-50066 Details. https://nvd.nist.gov/vuln/detail/CVE-2024-50066. Accessed: 2024-11-28.

[41] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. doi:10.1109/CGO.2019.8661201

[42] Ashish Panwar, Sorav Bansal, and K Gopinath. 2019. Hawkeye: Efficient fine-grained OS support for huge pages. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 347–360. doi:10.1145/3297858.3304064

[43] Yuxin Ren, Kang Zhou, Jianhai Luan, Yunfeng Ye, Shiyuan Hu, Xu Wu, Wenqin Zheng, Wenfeng Zhang, and Xinwei Hu. 2022. From Dynamic Loading to Extensible Transformation: An Infrastructure for Dynamic Library Transformation. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*. 603–617.

[44] RISC-V International. 2021. Svnapot Extension. RISC-V ISA Manual, Version 1.0.0, https://riscv-software-src.github.io/riscv-unified-db/manual/html/isa/isa_20240411/exts/Svnapot.html.

[45] Ryan Roberts. 2025. [PATCH v5 0/5] Readahead tweaks for larger folios. Linux Kernel Mailing List, https://lore.kernel.org/all/20250609092729.274960-1-ryan.roberts@arm.com/.

[46] Saritasa. 2025. Legacy Software Modernization in 2025: Survey of 500 U.S. IT Pros. https://www.saritasa.com/insights/legacy-software-modernization-in-2025-survey-of-500-u-s-it-pros. Accessed: 2025-12-03.

[47] Tom Shanley. 1998. *Pentium Pro and Pentium II System Architecture*. Addison-Wesley Professional. 439 pages.

[48] Yafang Shao. 2025. [PATCH v8 mm-new 00/12] mm, bpf: BPF based THP order selection. Linux Kernel Mailing List, https://lore.kernel.org/all/20250926093343.1000-1-laoar.shao@gmail.com/. Accessed: 2025-12-11.

[49] Fangrui Song. 2019. [ELF] Add `-z separate-code` and pad the last page of PT_LOAD. https://reviews.llvm.org/D64903. Accessed 8 May 2025.

[50] Fangrui Song. 2023. Exploring the section layout in linker output. https://maskray.me/blog/2023-12-17-exploring-the-section-layout-in-linker-output. Accessed 3 April 2026.

[51] The kernel development community. 2025. Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html.

[52] TuxCare. 2024. Enterprise Linux & Open-Source Landscape Report 2024. https://tuxcare.com/wp-content/uploads/2024/02/Enterprise-LinuxOpen-Source-Landscape-Report-2024.pdf. Accessed: 2025-12-03.

[53] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. 2021. Morrigan: A composite instruction TLB prefetcher. In *IEEE/ACM International Symposium on Microarchitecture*. 1138–1153. doi:10.1145/3466752.3480049

[54] Steven Wang. 2024. Add a tool for generating perf map file. https://github.com/LeiW000/iodlr-intel/commit/4d2aabb804673a9e8416d96e7f304a7ab8255429. Accessed: 2024-12-30.

[55] Matthew Wilcox. 2023. Re: [PATCH] mm: remove VM_EXEC requirement for THP eligibility. https://lore.kernel.org/all/ZYSXH58aQpI1SLr2@casper.infradead.org/. Linux kernel mailing list, describing CONFIG_READ_ONLY_THP_FOR_FS as "a preliminary hack".

[56] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-agnostic binary recompilation. In *ACM Architectural Support for Programming Languages & Operating Systems (ASPLOS)*. 133–147. doi:10.1145/3373376.3378470

[57] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*. 540–555. doi:10.1145/3447786.3456258

[58] Yufeng Zhou, Alan L Cox, Sandhya Dwarkadas, and Xiaowan Dong. 2023. The Impact of Page Size and Microarchitecture on Instruction Address Translation Overhead. *ACM Transactions on Architecture and Code Optimization (TACO)* 20, 3 (2023), 1–25. doi:10.1145/3600089

[59] Yufeng Zhou, Xiaowan Dong, Alan L Cox, and Sandhya Dwarkadas. 2019. On the impact of instruction address translation overhead. In *IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS)*. 106–116. doi:10.1109/ISPASS.2019.00018

[60] Weixi Zhu, Alan L Cox, and Scott Rixner. 2020. A comprehensive analysis of superpage management mechanisms and policies. In *USENIX Annual Technical Conference (ATC)*. 829–842. https://www.usenix.org/conference/atc20/presentation/zhu-weixi