Advances in Operating System Design

Autumn 2023

Group 6

Paging

Term Project Final Report

Group Members and Paper Links:

- 20CS30016, Divyansh Vijayvergia
  EPK: Scalable and Efficient Memory Protection Keys, ATC 2022
  https://www.usenix.org/system/files/atc22-gu-jinyu.pdf

- 20CS10039, Nikhil Saraswat
  Memory-Efficient Hashed Page Tables, HPCA 2023
  https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10071061

- 20CS30003, Amit Kumar
  Reducing Minor Page Fault Overheads through Enhanced Page-Walker, TACO 2022
  https://dl.acm.org/doi/pdf/10.1145/3547142

- 20CS10007, Anand Manojkumar Parikh
  Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism, ASPLOS 2020
  https://dl.acm.org/doi/pdf/10.1145/3373376.3378493

EPK: Scalable and Efficient Memory Protection Keys, ATC 2022

This paper discusses the concept of Extended Protected Keys (EPKs), an extension of Memory Protection Keys (MPKs), which are an Intel feature designed to significantly enhance the isolation, security, and performance of the software. One of the limitations of MPK is that memory can be partitioned into only 16 memory domains, but the proposed EPK solution leverages the existing hardware

Introduction:

The introduction of Intel MPK revolutionized intra-process memory isolation, offering high efficiency with up to 16 memory domains within an application. However, the limit of 16 domains posed significant usability challenges. To address this, several solutions have been proposed but these approaches suffer from overhead or feasibility issues. In this paper, EPK is introduced, a mechanism that efficiently extends MPK's domain capabilities to 7,680 domains using fast EPT-switching.

EPK combines MPK and fast EPT-switching, leveraging both hardware features to achieve extensive memory domain expansion. The new system manages to abstract memory domains and EPTs, offering an easy-to-use library for applications while facilitating domain switching across multiple EPTs for a single thread seamlessly. In server and persistent memory applications, EPK demonstrates overheads of around 5%, significantly outperforming the state-of-the-art solution, libmpk.

Hardware background

MPK supports dividing a process's memory into memory domains with distinct access permissions, providing highly efficient intra-process memory isolation. This division is made possible by utilising four previously unused bits in the page table entry, allowing for a total of 16 potential protection keys serving as domain IDs. The access permissions for these memory domains are stored in a dedicated 32-bit per-CPU User Access Register (PKRU), which can be modified by issuing the WRPKRU instruction from the user space. Since PKRU is a CPU register, it inherently operates on a per-thread basis, potentially giving each thread a different set of protections from every other thread.
Other hardware terms and their respective definitions are given below

- **MPK in the VM:** The applicability of MPK within a virtual machine, still tagging protection keys in page tables.
- **Extended Page Table (EPT) and VMFUNC:** Hardware virtualization technology using EPT for memory virtualization and VMFUNC for managing EPT pointers and switching.
- **Virtualization Exceptions (VE):** Allowing EPT violations to trigger exceptions without VMExits, giving control to the hypervisor for configuration.

This technical overview sets the groundwork for understanding the EPK system's integration with existing hardware features and its approach to extending memory domain capabilities.

Motivation:

There are several methods for memory isolation such as

Software Fault Isolation (SFI):

SFI is a security technique that improves memory isolation by instrumenting and restricting memory accesses within an application. It involves adding checks and restrictions in the

application's code to prevent unauthorized memory access and potential security vulnerabilities.

However, SFI can introduce non-negligible runtime performance overhead. This means that the additional checks and restrictions can slow down the execution of the application. SFI is also often less flexible in terms of fine-grained control, making it challenging to implement complex and dynamic security policies.

Using the MMU:

The MMU (Memory Management Unit) is a hardware component in modern processors responsible for virtual memory management. It maps virtual addresses used by software to physical addresses in physical memory. It offers different memory partitions of a process using different page tables or extended page tables managed by the MMU. By isolating memory at the page granularity, it's possible to use the MMU to check and enforce memory access restrictions. This approach is typically more efficient and flexible than SFI. With this method, each page can have its own access rights, and the MMU can handle the access checks, providing strong memory isolation without significant performance overhead.

However, constructing different memory domains with page tables is not free. Switching between different domains requires changing the page table through specialized hardware instructions. In an experiment, the page-table bases approach posed 70% performance overhead while the EPT bases solution which uses VMFUNC instruction to switch between EPTs poses 12% overhead. In the same experiment, the MPK-based solution only adds about 3% overhead, demonstrating the efficiency of MPK.

A major problem with MPK is that it only offers 16 memory domains which limits its usability. Recent efforts seek to address these constraints, aiming for scalable domain solutions. Although libmpk offers the illusion of multiple domains using virtual keys, the overhead from key eviction impacts performance significantly. Switching domains with libmpk becomes more costly as the number of domains increases. The trade-off is evident when the domain size grows, resulting in higher switch costs due to TLB flushes and page table entry updates.

Challenges:

The proposed EPK solution involves increasing the number of memory domains offered by combining two already existing features which are Memory Protection Keys (MPKs) and fast Extended Page Table (EPT) switching. Combining these two features poses several challenges which are discussed below

1. Unified Abstraction: Integrating two distinct hardware features while presenting a unified abstraction to applications poses the first challenge. EPK aims to maintain the existing memory domain and domain-switching abstractions inherited from MPK while concealing the complexities of EPTs from applications. This involves intricately managing domain mappings in multiple EPTs and developing a user-friendly library to offer accessible APIs.

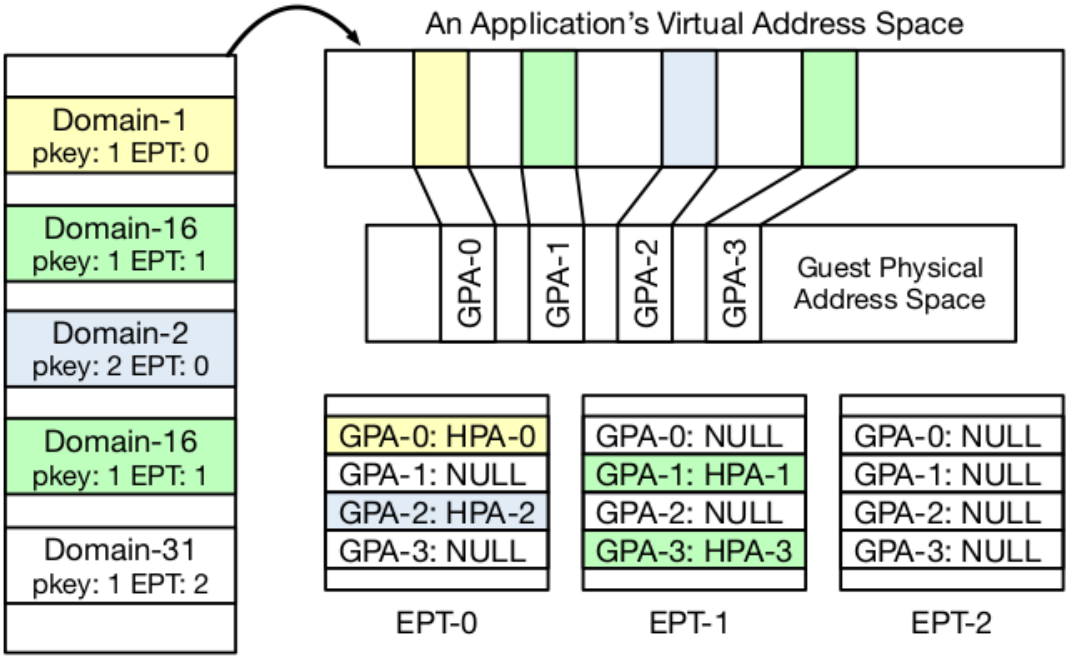
2. Simultaneous Thread Access Across Different EPTs: Another challenge is enabling one thread to access memory domains across different EPTs, similar to the original MPK's capability of accessing multiple domains simultaneously. EPK utilizes the virtualization exception (VE) hardware feature to seamlessly switch EPTs for a thread transparently in the event of a domain access causing EPT violations.

Implementation:

-> The proposed EPK solution addresses both of these issues by capitalizing on the support of the existing hardware architecture to expand the capabilities of MPK. It takes advantage of a hardware feature related to fast Extended Page Table switching (EPT), primarily used for mapping Guest Physical Addresses (GPAs) to Host Physical Addresses (HPAs). This feature is facilitated through the use of the VMFUNC instruction, a crucial element in virtualization that streamlines EPT switching by enabling the loading of one of up to 512 configured EPTs managed by the hypervisor It should be noted that no TLB flushing is required while executing VMFUNC which provides significant performance enhancement.

-> It can be observed that switching domains within the same EPT requires a simple WRPKRU instruction (as required in MPK) but switching between domains in different domains requires an additional VMFUNC instruction for EPT switching. Since both these two instructions are non-privileged, the domain switches are efficiently finished in user mode.MPK and VMFUNC share similarities in decoupling privilege-mode management and non-privilege-mode fast switching.

-> The key idea behind EPK involves reusing the same MPK protection keys in different extended page tables (EPT). Thus, with 512 EPTs, EPK can support up to 7,680 domains. Simply put, now the (extended) protection key of a domain will no longer be just the MPK value but a tuple of EPT index (0-511) and MPK value(1-15). So inherently, to use the functionality of EPTs, EPK requires an application to run within a Virtual Machine (VM)



-> EPK needs to ensure seamless domain switching between domains in the same EPT and across different EPTs while hiding underlying details from the application. For this, EPK uses another existing hardware feature called Virtualization Exception (VE) to switch between EPTs when a domain access causes an EPT violation without causing expensive VM exits.

-> EPK provides intuitive APIs for applications, very similar to MPKs with differences in their underlying implementation. Some interesting optimization can be observed here, for example, while allocating domains, the EPK implementation tries to allocate domains in the same EPT. The idea is based on the heuristic that the domains may have affinity and are likely to be traversed together which will save comparatively expensive VMFUNC.

Experiments and results:

1) Case study: protecting server applications:

The EPK is compared against several other alternatives which include

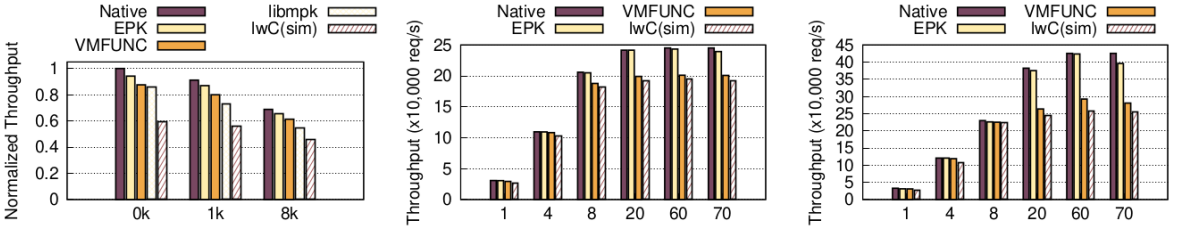
- Vanilla implementation - without isolation
- SFI - Software-based extension of MPKs
- lwC - Different domains in different page tables (at most 1 per page table)
- VMFUNC - Different domains in different Extended page tables (at most 1 per EPT)
- Libmpk - a previous state-of-art approach which uses time-division multiplexing which requires modifying page table entries and TLB flushing

Micro-service benchmark - Multiple domain creation and switching costs

Libmpk's performance significantly deteriorates beyond 15 domains, escalating further with larger domain sizes due to increased page table updates during key eviction, considerably impacting performance compared to EPK. The VMFUNC-only solution faces increased switch costs as the domain count exceeds 3, primarily due to decreased TLB hit rates with more involved EPTs. EPK demonstrates superior efficiency, leveraging WRPKRU for most switches and outperforming libmpk and the VMFUNC-only solution in various domain count and size scenarios. However, in high domain count scenarios, EPK's performance may converge with the VMFUNC-only solution, particularly in instances necessitating random switches.

Macro-benchmark tests using NGINX and Memcached servers

In these tests, EPK demonstrates lower overhead compared to other solutions in NGINX and Memcached. For NGINX, EPK's overhead ranges from 4.3% to 5.8%, outperforming the VMFUNC-only approach. In Memcached, EPK shows minimal overhead (up to 2.9%) for GET operations with fewer than 60 clients. VMFUNC causes higher overhead due to TLB misses. Overall, EPK maintains a relatively lower overhead, especially in scenarios with moderate domain counts.

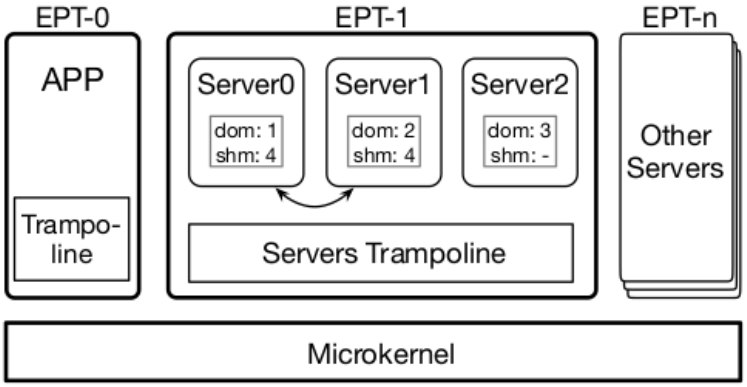


(a) NGINX. X-axis: each request size. (b) Memcached SET. X-axis: the number of users. (c) Memcached GET. X-axis: the number of users

2) Case study: Boosting IPCs in microkernels

Microkernel, as opposed to a traditional monolithic kernel, has only minimal functionality in the kernel while all other functionalities such as file system, network stack, etc are provided as a user space program. The major drawback of this kernel is an increase in latency due to the use of IPCs to communicate with different kernel modules in user space.
There are several attempts to minimize this latency and this paper explores EPK-based HyBridge, a novel approach that enhances IPC (Inter-Process Communication) efficiency for microkernels, addressing the limitations of existing IPC designs. By leveraging EPK's ability to efficiently construct numerous isolated memory domains and facilitate rapid domain switches at the user level, HyBridge is proposed as an improvement over UnderBridge

(previous state-of-the-art model), overcoming its limitations. It allows multiple system servers to operate within separate memory domains, ensuring the isolation similar to separate processes while enabling optimized IPC. The solution enhances IPC communication: cross-server communication happens between system servers within the same EPT, while Application-to-server IPCs necessitate EPT switching, employing a trampoline for the switch and ensuring secure data exchange. To enhance security, HyBridge implements measures to prevent malicious server activities, employing binary scanning to eliminate specific instructions and employing a token-based authentication mechanism for IPC invocations, ensuring authentication and preventing unauthorized IPCs. The overall design improves IPC efficiency in microkernels, maintaining robust security measures and enhancing performance.



Conclusion

This paper introduces EPK, a cutting-edge approach that cleverly combines MPK with hardware virtualization features to solve the problem of a limited number of memory domains possible due to the existing MPK technology. Its primary goal is to establish seamless and efficient memory isolation within a single process. Through practical case studies, the paper demonstrates the diverse ways EPK can be applied, highlighting its potential in various usage scenarios.

Memory-Efficient Hashed Page Tables, HPCA 2023

I. Introduction:

The conventional implementation of radix tree page tables faces inherent challenges, notably in terms of memory overhead and contiguity constraints. Radix trees often demand substantial contiguous physical memory, hindering their scalability and adaptability to dynamic workloads. This limitation becomes more pronounced as modern computing environments grapple with diverse applications and evolving memory requirements. In addition to contiguous memory issues, the conventional approach encounters inefficiencies related to dynamic resizing operations, particularly when addressing varying workload demands.

Issues with Radix Trees:

- **Contiguous Memory Constraints:** Radix trees exhibit a propensity for large, contiguous memory allocations, posing challenges in scenarios where such allocations are impractical or inefficient.
- **Dynamic Resizing Overheads:** The conventional radix tree implementation struggles with dynamic resizing operations, impacting both memory utilization and system performance, especially in scenarios where workloads vary over time.

Addressing Challenges with HPCA:

In response to these challenges, this paper introduces Memory Efficient Hashed Page Tables (ME-HPTs), a novel approach designed to mitigate the issues associated with radix trees. ME-HPTs leverage efficient hashing techniques, introducing innovations like the L2P table, dynamically changing chunk sizes, and in-place and per-way resizing. This innovative approach not only reduces contiguous memory requirements but also enhances the adaptability and performance of page tables under dynamic workloads. ME-HPTs, introduced in the context of HPCA, offer a promising solution to the limitations of traditional radix tree implementations, paving the way for more efficient and scalable memory management in contemporary computing environments.

II. Techniques Overview:

- **L2P Table:** The L2P table introduces a level of indirection to efficiently manage page table entries. For implementation, each entry in the L2P table corresponds to a chunk of contiguous physical memory and contains pointers to the corresponding Hashed Page Table entries. This indirection allows for dynamic resizing without the need to move the entire page table. The L2P table is consulted during memory access, minimizing the impact of resizing operations on performance.

- **Dynamically-changing Chunk Sizes:** ME-HPTs dynamically adjust chunk sizes based on memory requirements. Implementation involves monitoring HPT occupancy, and when it surpasses a threshold, resizing occurs. The algorithm dynamically selects between 8KB and 1MB chunk sizes. Smaller chunk sizes are preferred to reduce wasted memory, aligning allocations more closely with actual

needs. This adaptability ensures efficient memory utilization while accommodating diverse application demands.

- **In-place HPT Resizing:** In-place resizing minimizes data movement during HPT expansion. As new entries are added, the resizing algorithm reorganizes existing entries within the current memory chunk. This approach avoids the need for copying entire chunks, reducing the time and resources traditionally associated with resizing operations. In-place resizing is triggered based on HPT occupancy and contributes significantly to maintaining memory contiguity.

- **Per-way Resizing:** ME-HPTs employ per-way resizing to tailor memory allocation for individual ways within the HPT. This involves adjusting the size of each way independently, accommodating the varying needs of different application scenarios. Implementation details include tracking the occupancy of each way and dynamically resizing based on way-specific thresholds. Per-way resizing contributes to a fine-grained allocation strategy, further enhancing memory efficiency.

These implementation details highlight the practical aspects of ME-HPTs' techniques, showcasing their adaptability and efficiency in real-world scenarios. The combination of these techniques forms a cohesive strategy for addressing memory challenges associated with Hashed Page Tables.

III. Experimental Methodology:

- **Modeled Architectures:** The experimental methodology involves employing full-system cycle-level simulations to model an 8-core server architecture with 64 GB of main memory. The baseline architecture is based on Elastic Cuckoo Page Tables (ECPTs), and ME-HPT enhancements are introduced for comparison. The architectures are evaluated under different scenarios, including systems with only 4KB pages and those with multiple page sizes enabled by Transparent Huge Pages (THP) in the Linux kernel. Additionally, a system with state-of-the-art radix-tree page tables is modeled for a comprehensive comparison.

- **Modeling Infrastructure:** Integration of Simics, SST framework, and DRAMSim2 memory simulator provides a robust modeling infrastructure. Simics intercepts instructions and virtual memory operations on the fly, facilitating detailed back-end cycle-level simulations for ME-HPTs. The Intel SAE on Simics is employed for OS instrumentation. ME-HPT hardware components are meticulously modeled using SST. Real system measurements on a highly fragmented system with a Fragmented Memory Fragmentation Index (FMFI) of 0.7 are utilized for allocation overhead evaluation.

IV. Evaluation:

- **Memory Contiguity Savings:** ME-HPTs demonstrate significant success in reducing the maximum size of contiguous memory allocation required, showcasing an average reduction of 92%. This is particularly noteworthy for memory-intensive applications

like GUPS and SysBench, where contiguous memory requirements decreased from 64MB to 1 MB.

- **Application Performance:** The performance evaluation reveals an average speedup of 8.9% for ME-HPTs compared to ECPTs. The lower memory allocation overhead and reduced data movement contribute to improved overall system performance.

- **Memory Savings:** ME-HPTs exhibit substantial memory savings, averaging 43% without THP and 41% with THP compared to the ECPT baseline. In-place resizing and per-way resizing contribute to these savings, emphasizing the efficiency of the proposed techniques.

- **ME-HPT Characterization:** The paper provides an insightful characterization of ME-HPTs, including resizing operations, the effectiveness of per-way resizing, data movement reduction, the number of L2P table entries used, and benefits for small-size applications. These analyses contribute to a comprehensive understanding of ME-HPTs' behavior and performance.

V. Application to Other Problems:

The paper discusses the potential applicability of ME-HPT techniques beyond Hashed Page Tables. The hashing optimizations are considered applicable to various hash table designs, including directories for cache coherence, memory indices for databases and file systems, and key-value stores.

VI. Broad Contributions and Significance:

The Memory Efficient Hashed Page Tables (ME-HPTs) mark a paradigm shift in system architecture and OS research, presenting a comprehensive solution to the persistent challenges associated with conventional radix tree implementations.

- **Holistic Approach to Memory Management:** ME-HPTs offer a holistic approach to memory management, transcending the limitations of contiguous physical memory allocation. The integration of the L2P table, dynamic chunk sizing, and innovative resizing techniques collectively address issues of memory efficiency, fragmentation, and system responsiveness.

- **Relevance in Contemporary Server Environments:** In the broader context of OS research, ME-HPTs stand as a crucial advancement for contemporary server environments dealing with diverse and dynamic workloads. The adaptability and scalability introduced by ME-HPTs align seamlessly with the demands of modern computing, providing a robust framework for efficient memory utilization.

- **Practical Solutions and Future Exploration:** The paper not only provides practical solutions to existing challenges but also serves as a catalyst for future exploration in optimizing memory management within the intricate landscape of modern operating

systems. ME-HPTs set a precedent for innovation in memory-centric research, opening avenues for continued refinement and enhancement.

- **The promise of Enhanced System Performance:** ME-HPTs carry the promise of significantly enhancing overall system performance. By addressing fundamental issues in memory allocation and resizing, ME-HPTs pave the way for a more responsive and adaptive memory management system, crucial for the evolving needs of contemporary computing.

- **Pivotal for Future OS Evolution:** The contributions made by ME-HPTs are pivotal for the future evolution of operating systems. As computing environments continue to diversify and workloads become more dynamic, ME-HPTs provide a foundational framework that can adapt and scale, ensuring that memory management remains a cornerstone of system efficiency and performance.

VI. Conclusion:

ME-HPTs present an effective solution to the contiguous memory challenge faced by Hashed Page Tables. The combination of L2P table, dynamically changing chunk sizes, in-place resizing, and per-way resizing results in substantial improvements in memory utilization and overall system performance. The paper's comprehensive evaluation and characterization contribute to establishing ME-HPTs as a promising advancement in page table design, with potential applications in diverse computing domains.

Reducing Minor Page Fault Overheads through Enhanced Page Walker

The problem:

The paper highlights a problem where computer systems experience slowdowns due to "minor page faults." These minor page faults can make programs run much slower, sometimes by as much as 29%. The paper discusses how this problem affects real-world applications, with a focus on a popular one called Function-as-a-Service (FaaS). This study examines how minor page faults affect FaaS applications, leading to slower response times and increased user costs. Minor faults become more frequent with higher demand, particularly during when new tasks are initiated. They found that when more people use FaaS at the same time, problem gets worse, and it ends up costing users more money.

Research Challenges:

1. Handling Major Page Faults: While the proposed approach effectively addresses minor page faults, extending its capabilities to handle major page faults could present a notable research challenge. This entails ensuring efficient fault handling for scenarios where entire datasets may not fit in memory.

2. Integration with Network Controllers: Exploring seamless integration with proposals suggesting page fault support for network controllers (e.g., avoiding page pinning for DMA requests) poses a challenge. This integration could enhance fault handling latency, requiring careful consideration of hardware and software coordination.

3. Coordinated Memory Management: In the context of large pages and TLB optimizations, developing a memory management strategy that minimizes kernel involvement in promoting and demoting huge pages and handles memory compaction efficiently is a challenge. The goal is to reduce overhead for various workloads, especially those involving small object allocations.

4. Compatibility with Alternative Memory Management Systems: Ensuring compatibility and effective collaboration with alternative memory management systems, such as Midgard, which employs VMAs for memory management, presents a challenge. The focus is on achieving improved TLB efficiency while acknowledging the persistence of page fault overhead in such systems.

5. Mitigating Context Switch Overheads: Addressing context switch overheads related to page faults, as proposed by Alam et al., poses a challenge. The goal is to find a balance between reducing context switches and ensuring compatibility with diverse workloads, especially in virtualized environments.

Core Solution Idea:

The core solution idea involves the implementation of a Minor Fault Offload Engine (MFOE) to address the performance impact of minor page faults in computer systems. MFOE leverages optimized hardware page fault handling, introducing system calls like MFOE_enable to activate pre-allocated physical frames per core. Pre-allocation tables, adopting a lockless ring buffer architecture, facilitate efficient page fault handling, and a per-core allocation policy minimizes Non-Uniform Memory Access (NUMA) inefficiencies. MFOE's systematic approach extends to software handling, ensuring compatibility with user-space applications, and integrates post-page fault processing, enhancing system

efficiency by periodically replenishing pre-allocated pages. The solution targets a reduction in overall page fault-related latency, optimizing performance for real-world applications.

Design Overview :

The design choices for enhanced page fault handling using MFOE(Minor Fault Offload Engine): This approach is discussed in context of Intel x86-64 architecture and Linux OS.

1. Page Fault Handling Breakdown: Page fault handling is divided into three parts: pre-fault tasks, the critical path for fault handling, and post-fault tasks. The design of MFOE focuses on offloading pre- and post-fault tasks to reduce fault handling latency and optimising the critical path in hardware. Pre-fault and post-fault functions are moved to a background thread, eliminating their impact on the fault handling process.

2. Page Pre-allocation: Allocating pages usually involves obtaining a pointer to a struct page, which can be easily converted to the physical page frame number. By saving these frame numbers in a specific format and making them accessible to hardware, we can use pre-allocated pages to handle page faults. Advantages: Avoids blocking user applications, Eliminates the need for zeroing entire pages in the critical path, enhancing efficiency.

3. Legal Virtual Address Space Indication: It focuses on page fault offloading mechanism on user space, specifically minor page faults caused by functions like malloc and mmap. To inform hardware about the validity of faulting addresses, researchers introduced an "MFOEable" (Minor Fault Offload Enable) bit in the page table path, repurposing an unused PTE bit. When applications use malloc or mmap, a background thread is created by the kernel before returning from the system call. This thread constructs page table paths in the background for newly created virtual address regions, setting the MFOEable bit to indicate valid addresses. If the hardware page walker doesn't find this bit in the PTE, it triggers a kernel fault handling exception.

4. MFOE-Minor Fault Offload Engine: MFOE incorporates a pre-allocation page table featuring entries with pre-allocated page frame numbers and MFOEable bits set at the leaf entries for legal addresses. Here's how it works: When a faulting address is accessed, causing a TLB miss, the hardware page walker checks the MFOEable bit. If it's set (indicating a legal address) and the present bit is not set (implying no allocated page), MFOE takes over. It selects an entry from the pre-allocation page table, updates the PTE, sets necessary flag bits, creates a TLB entry, and proceeds with the faulting instruction, bypassing kernel software. Otherwise, the hardware page table walker triggers the typical kernel page fault exception to handle the fault.

5. Post-Page Fault Handling: These include incrementing the mm object's counter(by calling inc_mm_counter function), establishing a reverse map for the physical page(through the page_add_new_anon_rmap function), adding the page to the LRU list for future swapping (through the lru_cache_add_active_or_unevictable function), and managing cgroup-based memory usage(by calling the mem_cgroup_commit_charge function). These tasks can be deferred and executed in the background if the program is ongoing. Moreover, program termination may require cleaning up unused physical pages, though this may not be necessary in a real-world scenario if MFOE is the default page fault handling method for all programs.

The implementation of MFOE:

1. MFOE Enable/Disable System Calls: MFOE_enable activates optimized page fault handling and requires specifying pre-allocated frames per core. Pre-allocation tables are created per core but populated later by a kernel thread for efficiency. A "mfoe" flag in mm_struct indicates if MFOE is enabled for a process.

2. Pre-Allocation Table: The pre-allocation table utilizes a lockless ring buffer architecture. The kernel pre-allocates pages by updating the head index and setting the page frame number in an available entry. A per-core allocation policy is used, preferring local node pages and falling back to remote node pages when local pages are exhausted. This policy minimizes NUMA-related inefficiencies.

3. Pre-Page Fault Software Handling: The kernel ensures MFOE compatibility by setting the MFOEable bit, certain flags like VM_MFOE and other bits like the legality of the address, read/write permissions, and TGID in page tables when enabled for processes. This is achieved without affecting swap computation. A background thread is used to minimize user-space blocking, making it efficient for large programs with extensive virtual memory.

4. Hardware Page Fault Handling: MFOE handles page faults as follows: It's activated after a TLB miss and the last-level PTE. It checks PTE bits - if both the present and MFOEable bits are unset, it triggers a page fault. If the present bit is unset but the MFOEable bit is set, it proceeds. MFOE retrieves the pre-allocation table's starting address, reads the entry, and checks its valid bit. If valid, MFOE proceeds to handle the page fault; if not, it triggers a page fault. MFOE updates the pre-allocation entry with TGID, VA fields, and PFN increments the tail index, and updates the PTE and TLB to avoid future page walks.

5. Post-Page Fault Software Handling: We've added post-page fault processing in the Linux kernel with a 2ms delay. For each pre-allocation table, it checks the used bit and processes if set. It extracts TGID, VA, and PFN, executes functions, clears fields, allocates a new page, and updates the PFN and index.

Evaluation:

1. They ran a microbenchmark where they simulated a program that accesses memory. MFOE improved average fault handling latency by 33x and reduced tail latency by 51x compared to the current systems.

2. They also tested a software-only version of MFOE and found that it improved average fault handling latency by 1.59x and reduced tail latency by 1.82x compared to the baseline.

3. In a multi-threaded microbenchmark, They found that MFOE reached a 90% hit rate even at a high fault rate of one fault every 21,000 cycles.

4. MFOE achieved an average performance gain of 6.6% across all workloads.

5. Notably, memcached achieved a high hit rate of 97%, resulting in a 6.2% runtime improvement, effectively eliminating the overhead due to minor faults.

6. gcc had a 68% hit rate and achieved a substantial 25% performance gain.

To understand the impact of the number of pages allocated during pre-allocation and the background thread refresh interval, researchers studied four benchmarks with different

MFOE configurations. They found that performance gains decrease as the background thread refresh interval increases. The MFOE hit rate increases with a larger pre-allocation width. The benefits of these factors depend on the application's nature. Applications with low hit rates still saw performance improvements, suggesting that MFOE miss penalties were not substantial compared to the overall gains. In summary, MFOE provided varying degrees of performance improvement depending on the application and MFOE configuration.

Interesting Observations:

1. MFOE significantly improved fault handling latency and reduced tail latency, demonstrating its effectiveness in minimizing the impact of minor page faults on system performance.

2. The integration of a background thread for pre- and post-page fault tasks allows for efficient handling without blocking user applications, contributing to enhanced overall system responsiveness.

3. MFOE demonstrated notable performance gains across various workloads, with substantial improvements in hit rates and runtime for applications like memcached and gcc.

4. The study provided insights into the impact of pre-allocation width and background thread refresh interval on MFOE's performance, highlighting the importance of configuration parameters.

Comments:

The paper's innovative approach in offloading minor page fault handling to hardware through MFOE addresses a critical performance bottleneck in computer systems. By seamlessly integrating with the Linux kernel and leveraging hardware optimizations, MFOE demonstrates the potential for substantial improvements in system efficiency. The extensive evaluation across different benchmarks and configurations strengthens the paper's contributions, providing valuable insights for OS researchers and developers. Overall, MFOE presents a promising solution to enhance the responsiveness and efficiency of modern operating systems.

Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism, ASPLOS 2020

Introduction:
Growing memory needs of modern applications has led to a massive increase in memory sizes in computing hardware. Virtual memory translation in such large systems has become a performance bottleneck and traditional translation techniques like multi-level radix page tables or global inverted hashed page tables are not scalable anymore. This work introduces a novel virtual memory translation technique using purely software-level and algorithmic changes that is scalable in large memory systems. Specifically, it supports qualities such as dynamic page-table resizing, efficient hash-collision resolution, process private page tables, multiple sizes of page tables and page sharing between processes.

Before discussing the proposed architecture, we explore the flaws of current techniques.

1. Multi-Level Radix Page Tables:
Most current architectures implement a multi-level radix tree as a page table. Assuming a 48-bit virtual address, regular 4KB page size and byte-addressable memory, the lowest 12 bits are used as an offset into the physical page. To get the address of the physical page, we divide the remaining 36 bits into 4 9-bit slices, starting from the highest level page table, we use these slices as an index to get the base address of the next level page table. In case of a TLB miss, such a page walk will need 4 sequential memory accesses + 1 to fetch data in the worst case. To improve upon this, we can use larger page sizes, for instance, 2 levels of 2 MB page sizes (virtual address bits divided as 18,18,12) or a 1 GB page followed by a 4KB page (virtual address bits divided as 27,9,12) reducing the sequential access to 2 sequential access + 1 to fetch the data in the worst case. In addition, the MMU has page walk caches (PWCs) that store recently accessed page table translation entries, so in case of a data miss, we can look for the lowest level table entry cached (if any at all) and continue the walk from there.
Drawbacks:
This technique, though currently widely used, is not scalable for very large memories. The paper has given examples where such a translation can take 20-50% of total execution time and the page walks can take 20-40% of main memory accesses. Increasing the size of caches for faster access, increasing page table page sizes beyond 1GB or increasing the number of levels of page table from 4 to 5 are possible, but not at all feasible options.

2. Hashed Page Tables:
As an alternative, we can use hashed page tables, where the entire virtual address is hashed using some hash function, and use that as an offset to find the corresponding physical address. This raises many issues.
Drawbacks:
There is loss of spatial locality, since hashing scatters the entries of consecutive virtual addresses all over the table. Also, with entry, we need to associate a hash tag (the virtual address which was hashed to get the offset) which increases the page table entry sizes. The paper has suggested techniques to solve the above 2 issues. But the major issue of hash collision remains unresolved. Chaining and open addressing can be used to resolve collisions, but they again require sequential memory accesses to find the desired value. The paper has statistics which show that despite of using expensive cryptographic hashing techniques (e.g. BLAKE), only 50% of page table entries have at most 1 virtual address mapped to them (if we use a hashing space of 1.5 times the total number of translations). Also, since such page tables cannot be resized efficiently, the system has to stall and wait for all entries to be copied to a larger table and then continue with execution of user code.

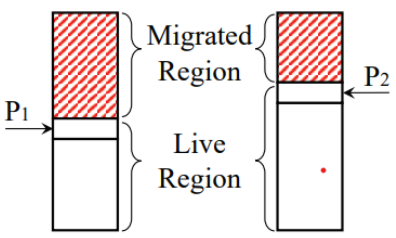
3. Global Inverted Page Tables:
Global inverted page tables are also possible, where all processes share a single page table. This page table can be allocated enough space at once to not need resizing.
Drawbacks:
This has several drawbacks. Firstly, since each page table entry corresponds to a <pid,virtual address> and the offset is the physical page address, processes cannot share pages (since there can be at most 1 entry at an offset). Secondly, when a process is deleted, we need to perform a linear scan of the entire page table to invalidate all entries of that pid. We can also not support multiple page sizes without an extra level of indirection.
Due to the failure/non-scalability of the above ideas, we now introduce Cuckoo Hashing.

Cuckoo Hashing:
Cuckoo hashing resolves collision problems by allowing an element to have multiple possible hashing locations while guaranteeing that it stays in at most one of these at any given time and can be moved among these locations. Let's look at a Cuckoo Hash Table that internally consists of 2 hash tables (or ways) T_1 and T_2, each with their own distinct hash functions H_1 and H_2. An element x is present either at $T_1[H_1(x)]$ or $T_2[H_2(x)]$. This lookup can be done parallely and hence takes only constant time. A deletion first looks up an element and if found, deletes it. An insertion does the following: First perform a lookup to check if it already exists. If it does not, then choose an empty slot out of $T_1[H_1(x)]$ or $T_2[H_2(x)]$ if available. If both slots are filled, choose any one randomly, let's say $T_1[H_1(x)]$ and remove the element y which occupied this slot. Insert it at $T_2[H_2(y)]$. If $T_2[H_2(y)]$ was is in turn occupied by some z, remove z and put it in $T_1[H_1(z)]$ and so on and so forth until either an empty element is found or a certain threshold of evacuations are exhausted, in which case declare a failure. Statistical results show that the average number insertion attempts in a 2-ary table with 50% occupancy is ~2. For 4-ary and 8-ary tables, we can insert in average of 2 attempts upto 70% and 80% occupancy.

Resizing Hash Tables:
Despite faster lookups, inserts and deletes, the problem of dynamic resizing still persists. We cannot resize a hash table fully at once because of the time it takes to make a complete copy. So, it is gradually resized. Once the occupancy crosses a particular threshold, we allocate a new table but do not copy all entries at once. Each insertion only takes place in the new table. Each lookup must now parallelly look into both the tables (so still takes constant time) and in each lookup, we also shift one random element from the old table into the new one and once the old table empties, it is de-allocated. In case we resize a d-ary cuckoo hash table, we make a copy of it with d ways. So now a lookup needs 2 x d parallel accesses, which is not very feasible. Also, half these lookups are in the old hash table, so making these accesses pollutes the cache severely by fetching useless entries.For reducing the degree of parallelism to d parallel lookups and not polluting the cache, the paper introduces Elastic Cuckoo Hashing. At any point of time, the Elastic Cuckoo Hash Table has 2 d-ary tables, old (with d ways T_i) and new (with d ways T_i').

Elastic Cuckoo Hashing:

Elastic Cuckoo Hashing works differently than regular Cuckoo Hashing. Each old way T_i of the old table has a corresponding Rehashing Pointer P_i. Entries above P_i are in the migrated region, i.e. they have been shifted to the corresponding new way of the new table T_i'. And the entries below P_i are still inside the old way T_i of the old table.



This will help us to resize the hash table dynamically in an efficient manner as demonstrated:

1. <u>Rehash</u>:
A rehash operation is called from other operations which follow. A rehash operation removes the element pointed to by the pointer P_i of old way T_i, and inserts it in new way T_i' at $T_i'[H_i'(x)]$. The insert operation again checks if that spot is occupied by some element y. If so, we choose some new way T_j' (j != i) and insert it that new way. This process continues till an empty spot is found. P_i is incremented by 1.

2. <u>Look-up</u>:
Every location of the element x is computed in the old way T_i as $H_i(x)$. If this value is < P_i, then the element has already been migrated into the new table, so we search in location $T_i'[H_i'(x)]$ in the new way T_i', else we search in $T_i[H_i(x)]$ of the old way T_i. Since we now look in either the old or new way, we need only d parallel look-ups instead of 2 x d. Also, since the migrated entries are never accessed, cache pollution by useless memory is avoided.

3. <u>Insert</u>:
We pick a random way T_i of the old table and compute $H_i(x)$. If this value is < P_i, then we attempt to insert x into location $T_i'[H_i'(x)]$, else we attempt to insert it into $T_i[H_i(x)]$. In case an element y is present in the location at which we attempt, we choose another way (old T_j or new T_j' depending on where we added the initial element) and continue the process recursively till an empty spot is found or the threshold number of attempts are exhausted upon which, we declare failure. After this, we perform one Rehash Operation which helps in gradually shifting the values from the old table to the new one.

4. <u>Delete</u>:
A delete first performs a look-up and if found, clears that entry.

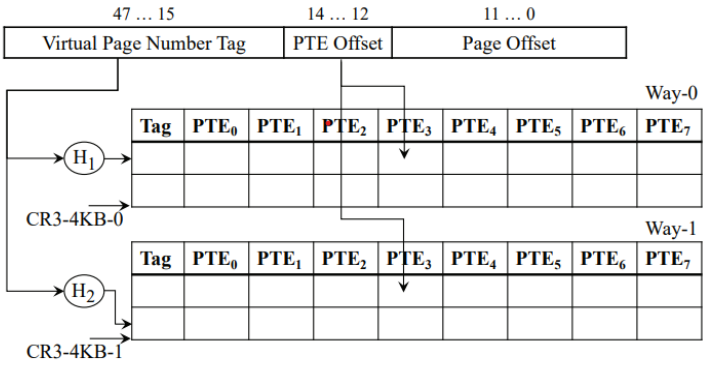
5. <u>Resize</u>:
We do not start with 2 copies of d-ary tables. We start with only the old table. If the occupancy of the old table crosses a bound of some threshold t, we allocate a new d-ary table with each way T_i' having size k times that of any old way T_i. It has been shown that k > (t+1/t) is a good choice of k. Not too large to waste space and not too small to trigger re-allocation of the new table itself before the old table is emptied (rehashed) gradually.

Now, let us use Elastic Cuckoo Hash Tables to build Elastic Cuckoo Page Tables (ECPTs)

<u>Elastic Cuckoo Page Tables</u>:
An ECPT is basically a d-ary Elastic Cuckoo Hashed Table. Some slice of the virtual address is used as the hashed index into this page table to retrieve a page table entry (PTE). We assume that the process's virtual address can translate to a physical address that can possibly lie in a page size of 1GB, 2MB or 4KB. So, each process has 3 separate ECPTs, one for each page size. For spatial locality of nearby virtual addresses, we cluster 8 consecutive virtual address translations into a single page table entry, i.e. we assign the same unique hash index to 8 consecutive translations (a.k.a. virtual page number or VPN) We divide our virtual address as follows for each possible page size:

1. 1GB Page size: Each 1GB physical page contains 2^{30} bytes (30 bit page offset). We need 3 bits for unclustering (3 bit PTE offset). The remaining 15 bits are the VPN tag. Hence, VA[47:33] = VPN Tag, VA[32:30] = PTE Offset, VA[29:0] = Page Offset
2. 2MB Page size: Each 2MB physical page contains 2^{21} bytes (21 bit page offset). We need 3 bits for unclustering (3 bit PTE offset). The remaining 24 bits are the VPN tag. Hence, VA[47:24] = VPN Tag, VA[23:21] = PTE Offset, VA[20:0] = Page Offset
3. 4KB Page size: Each 4KB physical page contains 2^{12} bytes (12 bit page offset). We need 3 bits for unclustering (3 bit PTE offset). The remaining 33 bits are the VPN tag. Hence, VA[47:15] = VPN Tage, VA[14:12] = PTE Offset, VA[11:0] = Page Offset

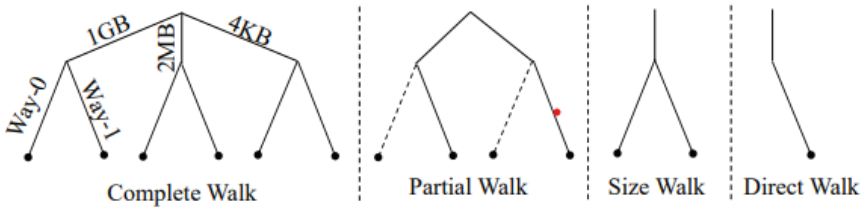
This example shows a 2-ary Elastic Cuckoo Page Table translation scheme for 4KB target pages. The paper has also demonstrated how we can tweak the currently used bits to fit each page table entry into a 64 byte cache line. Since it can be done for 4KB pages with maximum VPN tag size, it can also be done for page sizes 2MB and 1GB



Now, with this design, we can exploit 2 levels of parallelism. We can access all different ECPTs (for different target page sizes) in parallel and inside each ECPT, access all d ways in parallel. So, if S x d parallel look-ups (S = no. of different page sizes supported, 3 in our case) are allowed, we can fetch the physical address in constant time.

However, S x d is a very high degree of parallelism, and now we see how to reduce this.

<u>Cuckoo Walk Tables</u>:
To reduce this degree of parallelism, the OS constructs and maintains Cuckoo Walk Tables (CWTs) as it makes changes to the actual ECPTs. These tables efficiently encode information as to which directions to explore and the hardware automatically reads these tables to make pruning decisions while performing a page walk.



There are 4 types of Walk patterns that are possible:
1. Complete Walk: When the CWTs hold no information, the hardware needs to explore all ways of all ECPTs (of different sized page tables).
2. Partial Walk: When the CWTs show that the requested physical page is not of a particular size, the hardware potentially explores all ways of all other ECPTs except that size.
3. Size Walk: When the CWTs show that the requested physical page is of particular size, the hardware potentially explores all ways of that size.
4. Direct Walk: When the CWTs show both the page size and the way of the ECPT, only a single access is sufficient to access the memory location.

Of course, the OS maintains 1 CWT per ECPT (one for each supported page size), and the hardware makes sequential access to these CWTs in decreasing order of page size, since they contain increasingly finer grained information. Interestingly enough, the CWTs themselves are organized as Cuckoo Walk Tables.
Let's see an example of a 2MB page size ECPT's CWT Entry. Each entry of a 2MB ECPT contains 8 contiguous virtual memory translations (clustering as discussed before) hence a single 2MB ECPT entry contains information about 16MB of physical space. Each entry in the CWT contains information about 64 such sections (so it is indexed by 18 bits VA[47:30])

and hence contains information about 1GB of physical memory. Each entry is of 4 bits which denote the following information:
1. Bit 0: Is 2MB page size present in this section? (0 if not, else 1)
2. Bit 1: Is 4KB page size present in this section? (0 if not, else 1)
3. Bit 2, Bit 3: If Bit 0 is set (i.e. 2MB page size present), then which way to check (assuming we have at most 4 ways in each ECPT) [obviously valid only if Bit 0 is set]
The CWT entry for a virtual address tells the hardware where to find the requested page.

Page Size Present?		Which way of 2MB ECPT contains the translation	How to find the translation
2 MB	4 KB		
0	0	X	Page Fault (neither size found)
0	1	X	Look into 4KB CWT or a size walk over 4KB ECPT
1	0	i	Direct Walk to way i of 2MB ECPT
1	1	i	Direct Walk to way i of 2MB ECPT and size walk over 4KB ECPT (or look into 4KB CWT)

Now, since the CWTs themselves lie in memory, the paper has also shown how these entries can be cached into 64 byte cache lines in the form of CWCs (Cuckoo Walk Caches).

We can start using the above logic from the highest CWC (for 1GB pages) and work our way down till we find the target page. The paper also contains the exact final flowchart depicting the algorithm for fetching physical memory from the virtual address using these components.

Evaluation:
The authors evaluated the performance of the ECPT architecture on a server with 8 cores and 64GB main memory. The baseline model is the traditional x86-64 4-level radix tree. The proposed architecture resulted in an application speedup of 3%-28% with a mean of 11%. It also reduced the MMU overhead by 34%-41%. Also, the meticulously designed caches have a hit rate of 87.7%-99.9%. Overall, the address translation overhead (considering all steps) was reduced by 41% over the baseline.