Speculative Offset Address Translation

Chloe Alverti, Vasileios Karakostas
{xalverti,vkarakos}@cslab.ece.ntua.gr
National Technical University of Athens

1 Introduction

Research Problem. Virtualization provides enhanced security, consolidation, and efficient resource management. However, the technique itself can harm performance with memory virtualization considered as one of the most dominant factors [9,10,19]. Virtualization introduces an additional layer of memory abstraction that magnifies address translation overhead. To improve performance, industry has followed a holistic approach for huge pages [1, 2]. Processor vendors have improved Translation Lookaside Buffer (TLB) reach by substantially increasing its size for huge pages. To exploit that hardware support, OS-/hypervisor vendors have enhanced the memory management algorithms to map memory with huge pages.

Motivation. Despite their prominent efficiency [14], huge pages alone still fail to eliminate the address translation performance overhead for modern applications with large working sets and irregular access patterns. For example, in our experiments we observe up to 35% of performance overhead with virtualization due to TLB misses. The root cause is that huge pages are yet another form of *fixed size* mappings, providing limited TLB reach. As applications needs grow and address spaces are also about to expand [4], we expect memory virtualization overheads to increase further even in the presence of huge pages, following the fate of 4KB pages.

Our goal is to provide an address translation scheme that reduces the memory virtualization overhead. Prior works that increase TLB reach use contiguous virtual-to-physical mappings [2, 5, 7, 9, 12, 15, 16, 17, 20] to perform address translation through some form of *[base,limit,offset,permission]* representation. Instead, we propose decoupling offsets from strict virtual-to-physical contiguous address boundaries and use broader *offset mappings* to predict a missing address translation. Our proposed translation scheme is an opportunistic mechanism for hiding TLB misses and could accompany mechanisms that increase TLB reach.

2 Background and Related Work

Virtual memory simplifies programming and enables resource management. Architectural support primarily relies on TLBs that cache address translations. However, TLBs often fail to cover the large working sets of applications, exposing virtual memory as a performance bottleneck in modern systems.

Virtualization is an abstraction technique that allows a hypervisor to create and present virtual platforms to multiple guest virtual machines, managing the host physical resources. Virtualizing memory adds an extra layer of indirection between applications and physical memory. Three different address spaces exist: (i) the guest virtual address space (GVA), (ii) the guest physical address space (GPA), and (iii) the host physical address space (HPA), imposing 2D address translation requirements. The guest operating system (OS) manages the guest page table (gPT) that holds GVA→GPA mappings for each process. Nested paging is the major hardware-assisted technique for memory virtualization. The guest OS controls gPT and the hypervisor maintains separate nested page tables (nPT) holding GPA→HPA mappings. Special 2D hardware page walkers walk gPT and nPT in a nested way to retrieve GVA→GPA→HPA translations. Nesting multiplies TLB miss latency, requiring up to 24 memory references [6].

Huge pages reduce the cost and the number of TLB misses. Guest and nested huge pages reduce the first and second dimension of a 2D walk. A huge page that spans both dimensions is cached on a single TLB entry, increasing TLB reach. Modern system software uses huge 2M pages transparently on a best effort basis [3]. The enhanced hardware support in modern processors has boosted the technique's efficiency, establishing its presence for next generation processors. However, huge pages still fail to eliminate TLB misses.

3 Speculative Offset Address Translation

We propose SpOT, a hardware/software co-designed address translation scheme that combines *offset mappings* with speculative execution. We define an offset mapping as a group of virtual pages that are mapped in physical memory using the same *offset*. Offset mappings are of unlimited size and can be contiguous or discontinuous (Figure 1a). These two properties qualify offset mappings as an ideal substrate for enhancing TLB performance through a best-effort approach. SpOT introduces software and hardware support to enhance the creation of offset mappings and to leverage their existence by speculative address translation. On the software side, we extend the system memory management to enable and enhance offset contiguity. SpOT does not require storing offset mappings in memory. Instead it completely decouples them from virtual address boundaries and relies on modest hardware support to track dynamically active offsets and predict address translation on TLB misses.

3.1 Software Support for Offset Contiguity

We propose enhancing the system allocator with *offset contiguity awareness (OCA)*. Prior research typically employs preallocation [5,9,12] to create large contiguous mappings. However, preallocation is not always sufficient to extract all available contiguity in the system, particularly when applications allocate and release memory dynamically. In addition, preallocation restricts agile memory management and introduces memory bloat. OCA enhances the creation of offset mappings, releasing from preallocation need.

Background on Memory Management. Each physical memory frame is represented in the OS by the `page struct` holding status metadata. Buddy allocator is the core mecha-

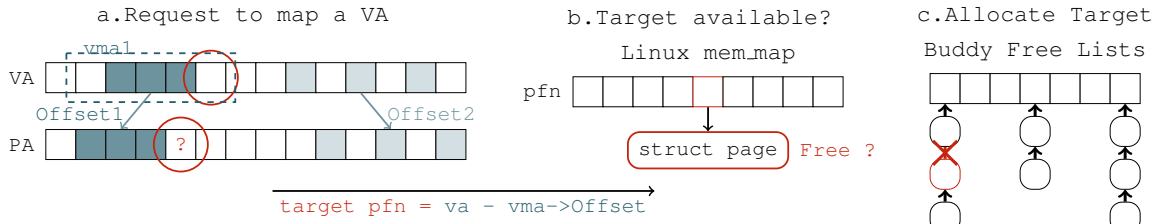


Figure 1: The OCA mechanism in native execution.

	Average	Min	Max
Default	0.22	0.10	155
OCA	0.19	0.09	158

Table 1: Latency (us) of allocation with OCA.

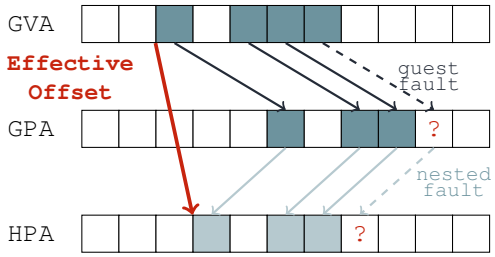


Figure 2: OCA combined with pure nested paging creates effective end-to-end offset mappings.

nism for memory management. It maintains [0, MAX_ORDER] lists, each populated by free aligned blocks of 2^{order} pages. This power-of-two logic supports fast memory coalescing and simplifies the management of free blocks. Memory allocation requests are served by order 0 (4K pages) or order 9 (2M pages [2]). If the lists are empty, larger blocks are split recursively. When memory is freed, the buddy allocator coalesces properly aligned free neighboring blocks (buddies) of the same order recursively, to control external fragmentation.

Contiguous populated virtual memory areas (VMA) in a process's address space are represented in the OS by the `vma struct`. By default the system maps each area using random pages from the buddy allocator. Page allocation occurs when virtual pages are accessed for the first time (demand paging). This random allocation inhibits the creation of contiguous mappings.

OCA mechanism. We propose a set of simple extensions to map virtual memory areas to contiguous physical regions using the same offset on a best effort basis. Figure 1 depicts the basic steps of the mechanism. OCA tracks the *offset* (i.e., $vaddr - paddr$) of the first page allocation for a VMA and stores it as minimal metadata extension in the corresponding structure. On a future neighboring page allocation in the same VMA, OCA tries to allocate the page starting at $paddr = vaddr - vma{\rightarrow}offset$ to extend the current offset mapping of the region.

OCA examines the availability of the target page relying completely on existing structure page metadata. It retrieves a handle to the target page's structure using the system memory map (`mem_map`) that is indexed by page physical address. Dedicated attributes (`_mapcount`, `_count`) indicate if the target is already in use or if it is part of a free list along with its current order (`private`). To locate a target page that is part of a larger block, OCA examines recursively the status of the higher order blocks that could contain that page using buddy address computation. To control external fragmentation, OCA conditionally constrains the recursion depth.

When allocating a page with OCA there are three scenarios. In case the target physical page is free and of the requested order, OCA allocates it. In case the target page is free but part of a higher order block, OCA splits it using the default buddy mechanisms and allocates the target page. In case the target page is occupied, OCA falls back to system's default random allocation routine. After a number of unsuccessful allocation attempts using the same offset, the $vma \rightarrow offset$ is updated based on the last random mapping. Allowing multiple unsuccessful attempts can counter-intuitively boost contiguity since a few occupied pages do not negate an offset's contiguity potential. In fact, the neighbors of an occupied page are likely to be free due to buddy allocator's splitting nature. However, failed attempts may introduce small discontinuities or "holes". Fortunately, our proposed hardware support can deal with such discontinuities. The threshold of allocation attempts could change dynamically based on VMA size and OCA success statistics. In our current implementation we statically set it to 2048 base pages.

OCA is very simple to implement, avoids costly data structure searches and remains extremely lightweight in terms of latency (Table 1). OCA provides contiguity across allocation requests and is independent to their order. Therefore it preserves demand paging and supports both base and huge pages. Its effectiveness relies primarily on the locality of memory references. Applications tend to access regions of their address space in bursts, triggering successive allocation requests. This behavior combined with the coalescing nature of buddy allocator occasionally leads to contiguous or clustered page mappings even in the default system [7, 16, 17]. OCA controls and enhances this effect by seeking offset contiguity explicitly.

Virtualized execution. Nested paging involves the memory management of both guest OS and hypervisor in the 2D mappings creation ($GVA{\rightarrow}GPA{\rightarrow}HPA$), imposing additional challenges in contiguity extraction. Applying OCA independently in each dimension boosts the creation of end-to-end offset mappings successfully, releasing from the necessity of coordination between the two software components. Figure 2 provides an example of OCA nested paging.

Interaction with preallocation. Eager paging [12] is a flexible preallocation technique that targets the creation of contiguous

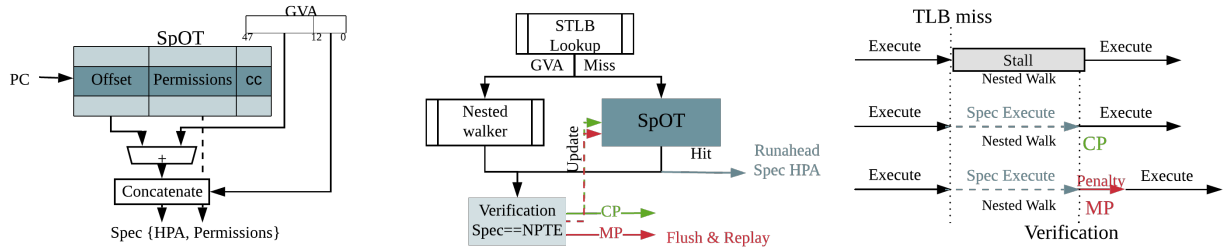


Figure 3: Speculative Address Translation based on Offset (SpOT).

	Mem. (GB)	Nested THP Ranges	Nested THP Offsets	Demand THP CA Ranges	Demand THP CA Offsets	Eager THP CA Ranges	Eager THP CA Offsets
mummer	0.47	1351	1252	1150	1030	95	80
deepsjeng	7.05	1083	288	25	12	52	41
Liblinear	30.53	3767	2131	31	22	20	19
PageRank	82.07	11077	10169	97	65	28	28
XSBench	128.23	3129	1136	123	62	14	10
BT-E	174.46	27273	19102	141	102	51	45
average	-	2795	1672	96	65	41	37

Table 2: Number of contiguous mappings (ranges) and unique offsets to map 99% of application's total memory footprint (first column) with nested paging using (i) demand THP, (ii) demand THP OCA, and (iii) eager THP OCA.

mappings. It allocates physical memory at application request time (malloc()) by requesting high order blocks from the buddy allocator. OCA is orthogonal to eager paging and can boost further its efficiency, as it extends mappings across eager requests and exploits unaligned contiguity spread across buddy allocator's blocks.

Results. Table 2 shows the number of contiguous mappings (ranges) and unique offsets that are needed to map 99% of applications' memory with (i) demand nested THP, (ii) demand nested THP OCA, and (iii) eager nested THP OCA. We observe that default nested paging with THP results in a high number of contiguous and offset mappings. With nested THP OCA, the number of both mappings reduces significantly. Applying eager allocations reduces the number further but to a less extent. Taking also into account the benefits of preserving demand paging, we assume demand nested THP OCA for the rest of the paper. Finally, these results show that using offset instead of contiguous mappings would requires less translation entries for all software management schemes.

3.2 Hardware Support for Offset Prediction

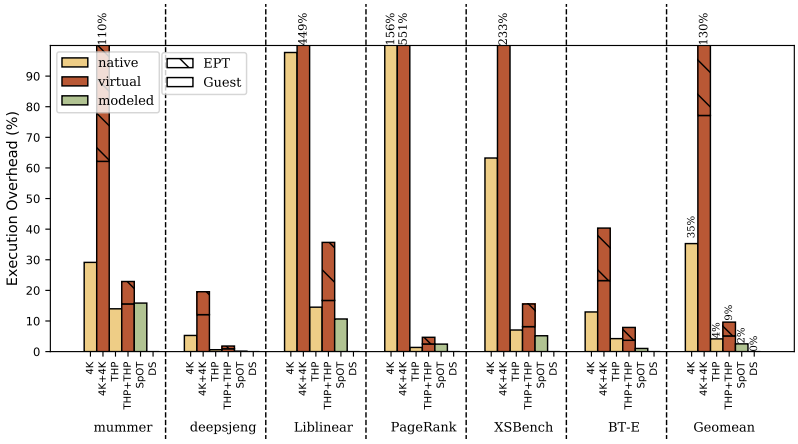
SpOT targets to reduce TLB miss overhead by hiding it with speculative execution. On a TLB miss event, the proposed mechanism speculatively selects an *offset* from GVA to HPA and predicts the host physical translation while a verification page walk happens in the background. SpOT's hardware support primarily consists of a simple *prediction table* that holds *[offset,permission]* values. Figure 3 presents SpOT's hardware support along with its basic functionality.

Prediction table. We use program counter (PC) for indexing and tag matching in the prediction table, since only a few instructions are typically responsible for the majority of TLB misses. SpOT associates a PC that triggered a TLB miss with the offset of the missing address translation. Next time the same PC triggers a TLB miss, SpOT will use the previous offset to predict the missing address translation. SpOT derives its effectiveness from instructions locality and the coarse representation of offset mappings (Section 3.1). Note that recent work [22] proposed a similar mechanism to predict a few bits of physical address, targeting L1 data cache capacity. SpOT predicts the entire physical addresses targeting virtual memory overhead.

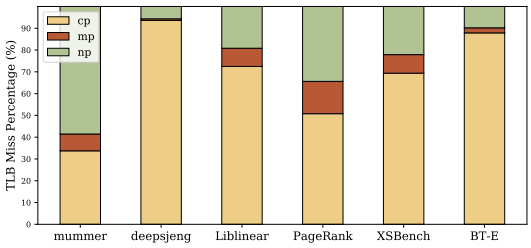
How it works. We consider speculation for last level TLB misses that trigger page walks. A TLB miss triggers both the default nested page walk and a lookup in the prediction table. SpOT uses the PC of the instruction that missed in the TLB, subtracts the selected *offset* from the GVA that triggered the miss, and predicts a *spec HPA* $= GVA - offset$, together with the permission rights. The CPU execution proceeds with the *spec HPA*, entering speculative execution-mode, and the verification page walk continues in the background. When the walk completes there are two scenarios: (i) speculation was correct and SpOT managed to hide the page walk cost, and (ii) speculation was incorrect and SpOT must trigger a pipeline flush and instruction replay. Flush is necessary because following instructions may have consumed incorrect data, similarly to branch mispredictions. The prediction table is updated after each speculation event, as explained next.

Increasing accuracy. Mis-speculation penalty can restrict the SpOT's effectiveness and even affect overall performance. To increase its accuracy we introduce a 2-bit confidence counter (cc) for each prediction entry. In case the same or different offset for a PC is verified, the confidence counter increases or decreases accordingly. For every TLB miss, SpOT uses the confidence counter to control whether that offset should be used for prediction.

Increasing coverage. If the prediction table is always updated with the latest offset after the verification page walk, then offsets that exhibit low prediction potential might thrash the prediction table. To address that, the OS marks a bit in each page table entry (PTE) that belongs to an offset mapping. The prediction table gets updated only by entries with that bit set. OS marking is simple: OCA sets that PTE bit at allocation time (section 3.1) by examining if a number of the new entry's neighbours share the same *offset*. In virtualized 2D execution, this support is included in both gPT and nPT, so that SpOT is updated only if both entries have the bit set.



(a) Performance



(b) SpOT accuracy

Figure 4: (a) Virtual memory execution overheads due to data TLB misses that trigger page walks. (b) Percentage of TLB misses that SpOT made (i) correct predictions (cp), (ii) mispredictions (mp), and (iii) no predictions (np).

Performance Model	
Ideal execution time	$T_{ideal} = T_{2M} - C_{2M}$
Avg. page walk cost	$AvgC_{4K,2M} = C_{4K,2M}/M_{4K,2M}$
Meas. page walk overhead	$Over_{4K,2M} = C_{4K,2M}/T_{ideal}$
SpOT overhead	$Over_{SpOT} = ((NP_{SIM} * AvgC_{2M}) + MP_{SIM} * (AvgC_{2M} + MP_{penalty}))/T_{ideal}$
T: Total execution cycles	$M_{4K,2M}$: page walks with 4K/2M
C: Cycles spent in page walks	NP_{SIM}: Simulated no predictions
MP_{SIM}: Simulated mispredictions	$MP_{penalty}$: 20 cycles

Table 3: Performance model based on hardware performance counters and hardware emulation through BadgerTrap.

Hiding the verification page walk cost. With SpOT the cost of the verification page walk cost can be entirely or partially hidden with useful work in case of correct prediction. Useful work can happen by prefetching data using the speculative address translation, overlapping the page walk cost with the data fetch cost [11, 18]. In case the processor allows also aggressive speculative execution, it can execute instructions that depend on the missing translation/data, increasing further performance opportunity.

Security considerations. Speculation has been identified as source for security vulnerabilities through side-channel attacks in memory. Fortunately, several mechanisms have been proposed to hide the speculation effects in the memory system [13, 21]. We assume the existence of such mechanisms to ensure secure execution. Note that such mechanisms are necessary regardless of employing SpOT.

4 Results and Contributions

Methodology. We prototype OCA in Linux kernel v3.15.5 for anonymous and copy-on-write page faults (4K and THP), khugepaged allocations and page cache allocations through `readahead()`. We run our workloads natively and virtualized using qemu-kvm v2.1.2. We use a system based on Intel Xeon CPU E5-2630 v4 cores (Broadwell microarchitecture) at 2.2GHz, equipped with 256 GB memory. Each core has a private TLB hierarchy: the L1 TLB has 64 and 32 entries for 4K and 2M pages, the L2 STLB has 1536 entries for both 4K/2M pages. We collect statistics from performance counters with *perf* (CPU cycles, TLB misses, page walk cycles) to quantify virtual memory overhead. We emulate SpOT with BadgerTrap [8], instrumenting the L2 STLB misses of our real system as applications run. To estimate the performance impact of SpOT we use a simple linear performance model [5, 7, 12, 18] as shown in Table 3. For SpOT we assume that: (i) correct speculations hide the entire TLB miss cost, (ii) decisions to not apply speculation expose the entire TLB miss cost, and (iii) mis-speculations add extra 20 cycles for flushing the pipeline [18] on top of the TLB miss cost. We use a diverse set of TLB intensive benchmarks from the domains of graph analytics, high-performance computing, and machine learning.

Results. Figure 4a presents performance overheads due to data TLB misses for native (yellow bar) and virtualized (red bar) execution of the real system, and for SpOT emulated system (green bar) under virtualized execution. Note that overhead values that are higher than 100% appear because the performance counters are unable to capture overlapped page walk cycles and the baseline is the ideal native execution with 2M pages.

We observe that the address translation overhead is exceptionally high in the presence of 4K pages. Huge pages reduce the translation overhead, but they do not eliminate it despite the facts that: (i) latest generation processors have substantially increased substantial TLB capacity for huge pages, and (ii) huge pages cover more than 99% of all applications memory footprint except for mummer (64%). Virtualization amplifies the address translation overhead which can grow up to 35% (9% geomean), with almost half of the cost attributed to nested page walks.

We evaluate SpOT with a 32-entry prediction table as an opportunistic mechanism to improve the performance of huge pages. SpOT reduces significantly the performance overhead of huge pages (2% geomean). The performance improves for

all applications, but to a less extent for mummer and liblinear. These applications experience a high number of TLB misses with irregular access pattern that stress SpOT. Note that SpOT with eager nested paging (not shown) reduces further the performance overhead of huge pages at 1% (geomean), because preallocation assists the creation of larger offset mappings (Table 2). To understand better the performance of SpOT, Figure 4b breaks down the percentage of TLB misses for which SpOT predicted correctly, mis-predicted, and did not predict at all, per application. We observe that the percentage of correct predictions can be over 95%, while the percentage of mis-speculations is never more than 14%.

We compare SpOT's performance with direct segments [5,9] that map a large contiguous region of a workload's virtual memory to a single segment using *[base, limit, offset, permissions]* representation. Figure 4a shows that direct segments (DS bar) eliminate the cost of TLB misses. Despite their efficiency, direct segments are rigid as the technique mostly omits paging and requires segment creation when booting the VM. SpOT preserves the benefits of demand paging and achieves performance close to that of direct segments.

Overall, the high quality of offset mappings that OCA generates together with the prediction table and its confidence mechanisms allow SpOT to successfully predict the address translation for the majority of TLB misses while keeping the mis-predictions rare.

References

[1] libhugetlbfs(7) - Linux man page, 2006.

[2] Transparent Huge Pages in 2.6.38. `http://lwn.net/Articles/423584/`, 2011.

[3] Transparent Hugepage Support. `https://www.kernel.org/doc/Documentation/vm/transhuge.txt`, 2011.

[4] 5-level paging and 5-level ept white paper. Technical report, Intel, 2017.

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *ISCA 2013*, 2013.

[6] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *ASPLOS 2008*, 2008.

[7] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. In *ASPLOS 2017*, 2017.

[8] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *SIGARCH Comput. Archit. News*, 42(2):20–23, September 2014.

[9] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *MICRO 2014*, 2014.

[10] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile paging: Exceeding the best of nested and shadow paging. In *ISCA 2016*, 2016.

[11] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing memory in heterogeneous systems. In *ASPLOS 2018*, 2018.

[12] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. In *ISCA 2015*, 2015.

[13] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. *CoRR*, abs/1806.05179, 2018.

[14] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *VEE 2016*, 2016.

[15] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations. In *ISCA 2017*, 2017.

[16] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing tlb reach by exploiting clustering in page translations. In *HPCA 2014*, 2014.

[17] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. Colt: Coalesced large-reach tlbs. In *MICRO 2012*, 2012.

[18] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *MICRO 2015*, 2015.

[19] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb. In *ISCA 2017*, 2017.

[20] Madhusudhan Talluri and Mark D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *ASPLOS 1994*, 1994.

[21] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *MICRO 2018*, 2018.

[22] Tianhao Zheng, Haishan Zhu, and Mattan Erez. SIPT: Speculatively Indexed, Physically Tagged Caches. In *HPCA 2018*, 2018.