Virtual Address Translation via Learned Page Table Indexes

Artemiy Margaritov[†] Dmitrii Ustiugov[‡] Edouard Bugnion[‡] Boris Grot[†]

[†]University of Edinburgh [‡]EPFL

Abstract

Address translation is an established performance bottleneck [4] in workloads operating on large datasets due to frequent TLB misses and subsequent page table walks that often require multiple memory accesses to resolve. Inspired by recent research at Google on *Learned Index Structures* [14], we propose to accelerate address translation by introducing a new translation mechanism based on learned models using neural networks. We argue that existing software-based learned models are unable to outperform the traditional address translation mechanisms due to their high inference time, pointing toward the need for hardware-accelerated learned models. With a challenging goal to microarchitect a hardware-friendly learned page table index, we discuss a number of machine learning and systems trade-offs, and suggest future directions.

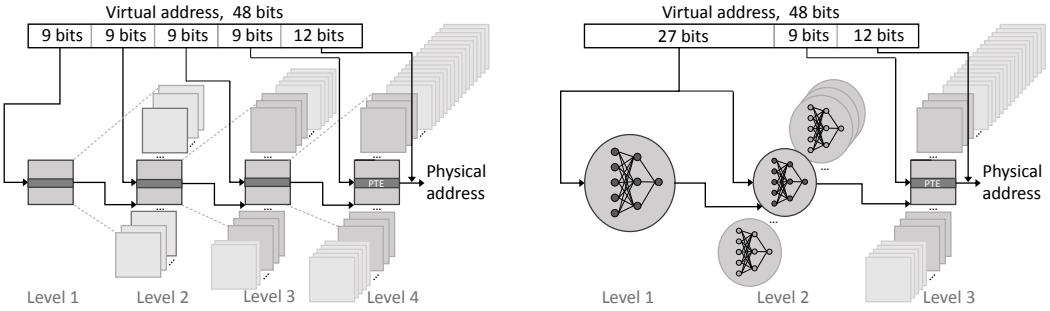
1 Introduction

Massive in-memory datasets are a staple feature of many server applications, including databases, key-value stores, and data analytics frameworks. The large – and rapidly growing – data footprints, coupled with irregular access patterns, in many of these workloads result in frequent TLB misses that require a walk of the operating system's radix tree-based page table. During the walk, the levels of the radix page table (which are memory resident) must be traversed one by one (see Fig. 1a), incurring high latency overhead. Modern processors include several hardware features to accelerate page table walks, including hardware walkers, multi-level TLBs and translation caches. Despite these features, recent studies show that up to 50% of performance in big-data server workloads can be lost to address translation [4]. The performance cost of address translation is destined to increase in the future, as larger memory capacities enabled by emerging memory technologies (e.g., Intel's 3D XPoint) will necessitate the addition of yet another (fifth) level in the radix page table that must be visited on each page table walk. Indeed, the industry has already started preparing for the eventual transition to five-level page tables [1].

Recent research proposals seeking to ameliorate the high cost of address translation tend to fall into one of two categories: incremental improvements to existing designs and disruptive changes to the virtual memory system. Incremental approaches include aggressive coalescing of Page Table Entries (PTEs) within TLBs [17, 18] and support for variable page sizes [7, 15, 19]. These techniques are fundamentally limited by coalescing opportunities exposed by the application and OS, as well as the capacity of the physical TLB structures. The disruptive proposals include the use of segment-based virtual memory [4, 11] and application-specific address translation [2]. While attractive from a performance perspective, these proposals require a radical re-engineering of the virtual memory subsystem at both OS and hardware levels, which presents a difficult path to adoption.

The challenge for future virtual memory systems is to enable high-performance address translation for terabyte-scale datasets without disrupting existing system stacks. Toward that goal, we suggest a new approach to accelerate radix page table walks through the use of *learned models*. Our approach is inspired by recent work from Google on Learned Index Structures [14], which shows that a lean

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.



(a) Conventional radix tree

(b) Learned index

Figure 1: Page table walk mechanisms

learned model can effectively replace a B-Tree for indexing a sorted key range. Our insight is that a radix page table is conceptually similar to a B-Tree index in that it consists of several indexing levels which progressively narrow down the region in which the target key – in this case, a PTE – is located. We hypothesize that a learned model can replace the page table walk, hence dramatically lowering the page walk latency. Fig. 1b shows the proposed approach.

Our preliminary results demonstrate that even a straight-forward application of the published learned indexes [14] can achieve excellent accuracy of PTE location prediction. However, the large inference time of the software-based models diminishes the benefit of eliding the page table walk. Hence, to enable fast and flexible learned page table indexes, we identify a key challenge to be that of architecting a *hardware-friendly* learned indexing scheme, and anticipate the solution to necessitate a number of machine learning and systems trade-offs and optimizations. In the following sections, we present our proposed design in §2, quantify the opportunity on modern servers, and study the inference time of the learned page table indexer for a number of implementations in §3, and discuss future directions towards fast and scalable indexer design in §4.

2 Toward learned page table indexes

Today's radix page tables use a 4-level organization, in which the last level has the PTEs containing the actual translation (i.e., the physical address corresponding to the virtual address) and metadata including the permission bits. While it seems tempting to learn the actual translations from virtual to physical space, there are two critical problems to contend with. First, there is no clear relationship between virtual and physical page addresses; while addresses of virtual pages obviously follow a sorted order, the corresponding physical pages might be scattered over the memory space, hence presenting a challenge for learning. While pages could be reordered in memory to facilitate learning by imposing some sort of an ordering, such a reordering might interfere with memory management, particularly in a virtualized environment. The second problem is more insidious and has to do with the fact that an output of a learned translation is speculative. In case the predicted translation is wrong, a process would effectively be allowed to access physical memory not allocated to it – a major security headache. Even if the translation is subsequently validated via a full page table walk and rolled back if found incorrect, the recently exposed Meltdown [16] and Spectre [13] vulnerabilities clearly indicate how dangerous such aggressive speculation can be.

Due to the challenges associated with directly learning virtual to physical translations, we suggest a more pragmatic strategy of using a learned model to predict the address of the PTE. The predicted PTE, which contains the actual translation, will be loaded similarly to how it is done in today's systems and validated through a tag check. A failed tag check (i.e., incorrect prediction) will trigger a conventional page table walk. As Fig. 1b shows, the proposed learned approach enables bypassing of all but the last level in the radix page table; thus, as the depth of the radix page table expands to five levels in the near future, the benefits of the learned approach will amplify.

The Learned Index framework [14] can be directly applied to the proposed idea, with the virtual page address acting as the key, and the address of the PTE as the index produced by the model. However, despite excellent accuracy of the Learned Index, its hierarchical structure and model's computational complexity largely negate the potential benefits due to the high inference time when

compared to a radix tree traversal. In practice, a range of consecutive PTEs is clustered on an OS page, so only the page address needs to be predicted, and the target PTE can be located within the page using a simple offset calculation (see Fig. 1b). This simple insight enables a nearly two-order-of-magnitude reduction in the number of addresses that need to be learned, thus enabling smaller, less computationally intensive, models.

We observe that there exists an opportunity to further reduce model cost and lower the inference time by lowering PTE location prediction accuracy. To compensate for the reduced accuracy, instead of outputting a single PTE location, the model can produce a range of possible locations. The corresponding PTE entries can all be fetched in parallel, so the latency need not suffer as compared to a single-PTE-per-prediction design; however, fetching multiple PTEs does require additional memory bandwidth. Fortunately, prior characterizations [9, 10] of datacenter services has shown that memory bandwidth is generally underutilized, hence affording a slight increase in its usage.

In the rest of the section, we discuss various aspects of integration of learned page table indexes into modern systems.

Model Training: Training can be a time-consuming task; however, several observations make it tractable in our context. First, the structure of the neural network will be fixed at processor design time, which means that only the weights need to be determined at application runtime. Secondly, server applications typically stay up for a long time and their memory footprint is stable; thus, training time can be easily amortized over the lifetime of the application. Moreover, training does not need to happen immediately at application start-up; instead, it can occur as a background task or whenever the load on the server is idle. Until a neural network is trained, conventional radix page table walk can be used to resolve TLB misses. Finally, we note that big-memory server applications tend to allocate memory at start-up and then manage it internally [4]; hence, the frequency of re-retraining should be low.

System Software Support: The neural indexer learns to approximate a monotonic function, as described by the prior work [14], which requires larger virtual addresses to correspond to PTEs located at larger physical addresses. Similarly to the memory compaction daemon (i.e., Transparent Hugepage Support [3]) already present in the Linux kernel, we propose an add-on daemon that will reorder the memory pages containing the radix page table to guarantee monotonicity. Note that application's memory pages do not need to be reordered. Because the footprint of the page table is small and the footprint of the application memory is expected to be stable, the page table ordering cost is low and is incurred at application start-up and rarely thereafter.

Software Compatibility: A particular advantage of the neural indexer is that it preserves the existing virtual memory abstraction and page table organization.

Scalability: Because the performance overheads of address translation are likely to increase in the future with the emergence of 5-level page tables and terabyte-scale in-memory datasets, we anticipate that the benefits of the proposed neural page table indexer will be even larger than in today's systems.

3 Quantifying the opportunity

3.1 Radix tree

To define the performance requirements for the proposed neural page table indexer design, we study the performance and locality characteristics of the conventional radix tree traversals. Contrary to the B-tree structures discussed in [14], the radix tree data structure is optimized for low latency traversals thanks to combining high fan-out intermediate nodes, up to 512 in x86 CPUs, with offset-based indexing that minimizes the number of memory accesses per walk. To further decrease radix tree traversal time, modern CPUs allow caching of the intermediate tree nodes in the on-chip cache hierarchy. In practice, radix tree traversal time highly depends on the effectiveness of the capacity-limited CPU caches that are severely pressured by the cloud applications' growing datasets [4, 9]. For modern cloud applications with large in-memory datasets, fast private L1 and L2 caches are too small to contain Level 3 and 4 of the radix tree (Fig. 1a), whereas large LLC capacity is typically shared among many workloads running simultaneously on a modern many-core server. Since many cloud workloads are highly memory intensive and generate high pressure on the on-chip caches [9, 10], the lower levels of the radix tree get thrashed out from the cache hierarchy to slow main memory, resulting in long-latency page table walks.

To quantify the performance headroom, we develop a synthetic benchmark that applies significant TLB pressure on a modern Intel Xeon E5-2630 v4 server, featuring Ubuntu 14.04, and collect the values of the corresponding performance counters. The benchmark serially accesses a large linked list up to 100GB in size, and triggers a page walk on every pointer access. To mimic a real world deployment of a latency-critical application [22], we study the benchmark in a multi-tenant environment, i.e., colocated with a multi-threaded streaming workload, with Transparent Huge Pages mechanisms disabled to avoid the associated memory management delays [8]. In various colocation scenarios, we find that for each data access generated by the application, the page walk mechanism causes 1-2 LLC accesses and 1-2 main memory accesses that together result in 211-397 CPU cycles of latency per walk.

3.2 Software-based learned indexes

To quantify the potential of learning the page table index, we use a neural model similar to the one proposed by Kraska et al. [14] for range indexing to perform address translation on a 20GB Memcached deployment. Empirically, we find the following model architecture to work well: a two-level hierarchy of models with a single model in the first level, and 32 models in the second level. The models in both levels have the same organization featuring 27 input neurons, 32 neurons in a single hidden layer, and a single output neuron. We train the overall model using a complete Linux page table dump for the target application. To simplify the quantitative comparison of a number of models and the radix tree baseline below, we focus on estimating the latency from the beginning of the page walk till the point when the PTE location is determined, either by the radix tree lookup or by inference in the learned model, and excluding the subsequent fetch of the PTE location from memory. In the baseline radix page table walk, this latency (i.e., excluding the actual PTE fetch) is 22-208 CPU cycles.

In terms of accuracy, the results are encouraging: the model is able to exactly pinpoint the target PTE page with $\sim$99.9% accuracy for all virtual addresses in the page dump. However, even with an aggressive software implementation, the model's large inference time fails to outperform the radix tree baseline. We estimate the inference time of the model that runs on a CPU with a vector unit, similar to Intel AVX-512 [5, 6], that allows execution of up to 16 floating point operations in parallel, each floating point operation takes four cycles, and each floating point unit has a throughput of one operation per cycle. We find that the model takes approximately 340 CPU cycles to calculate the location of a PTE. Thus, although the neural model is able to accurately pinpoint the target PTE location, naively applying the published learned index design is insufficient to outperform a conventional page walk and motivates the need for a lower-latency learned index architecture.

One promising insight is that the number of unique PTE pages that must be indexed, even for a 100GB dataset, is a few orders of magnitude smaller than the size of the key space studied in [14]. The small key space implies that a smaller, single-level model might be sufficient for page table indexing, which could help lower indexing latency. As discussed in §2, it may also be possible to trade-off prediction accuracy for model complexity, thus predicting a range of PTE locations via a simpler, lower-latency model. Our preliminary results show that using a single-level model that predicts 16 candidate locations, instead of one location, reduces the model size by 75$\times$. However, even this simpler model still requires approximately 120 cycles to produce the target PTE location, which significantly limits its benefit. Thus, we anticipate that further inference time reduction should come from using a *microarchitectural* learned page table indexer.

4 Future directions

We believe that an efficient hardware implementation of a learned page table indexer is possible. We envision a neural network-based indexer at each CPU core, optimized for three metrics of interest: latency, silicon area (comprised of logic for the compute and storage for the weights), and accuracy. Toward that goal, we consider the following techniques:

1. **Reduced precision** of weights and activations to reduce both compute and storage cost.
2. **Quantization of neuron activations** limits the number of possible outputs a neuron can produce. With quantized activations, multiplication operations in the neural model can take on a very limited number of input values, which makes it possible to replace the multiplication operation with much faster table lookup operations [20]. Because the tables are likely to increase the storage requirements of a neural model, there is a trade-off that needs to be studied between computational cost and memory storage requirements when quantization of neuron activations is applied.

3. **Binarization of weights and activations of the predictors** can help avoid complex multiplication operations by replacing them with much simpler boolean operators like XNOR [12]. To the best of our knowledge, the prior work in this space targets classification tasks, whereas our goal is recall of precise values. Thus, low accuracy is a potential concern with this approach.
4. **StrassenNets neural models** [21] cast matrix multiplications as two-layer sum-of-products networks that allow imposing a budget on the number of multiplication operations, thus enabling a potential hardware cost reduction without sacrificing accuracy.

In general, we note that low-latency hardware implementations of neural nets are an emerging area, with little work done to-date. This is an exciting space to study in order to understand the efficacy of existing neural network optimization techniques, which have been developed for and evaluated on software models. Learned page table indexing is a promising way to accelerate address translation, but it requires hardware support to be attractive performance-wise. This makes learned page table indexing an excellent vehicle for bringing together machine learning research and hardware design.

5 Conclusion

To continue improving user experience, many of the modern datacenter services strive for larger memory capacities that place significant pressure on the virtual memory subsystem. Established many decades ago, current software and hardware virtual memory mechanisms are reaching their limits, motivating us to look for their alternatives. With the advent of machine learning, we see a clear opportunity for building a fast learned page walk accelerator whose performance could scale well to accommodate growing application datasets. A key requirement for such an accelerator is low inference time that cannot be attained through previously proposed software-based models. In pursuit of a low-latency learned page table indexer, we argue for a hardware-based design and identify several systems and machine learning trade-offs and optimizations that could help realize it.

Acknowledgements

The authors thank Priyank Faldu, Vasilis Gavrielatos, Adrien Ghosn, Reiley Jeyapaul, Marios Kogias, James Larus, George Prekas, Amna Shahab, Urmish Thakker, as well as the members of EPFL's DCSL group for their valuable feedback and discussions of the work. This work was supported by the EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh and ARM PhD Scholarship Program.

References

[1] 5-Level paging and 5-level EPT. White Paper 335252-002, Intel, May 2017.

[2] H. Alam, T. Zhang, M. Erez, and Y. Etsion. Do-It-Yourself virtual memory translation. ISCA, 2017.

[3] A. Arcangeli. Transparent Hugepage Support. *KVMForum*, 2010.

[4] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient virtual memory for big memory servers. ISCA, 2013.

[5] Capabilities of Intel AVX-512 in Intel Xeon Scalable Processors (Skylake). Available at colfaxresearch.com/skl-avx512. Visited in October, 2018.

[6] Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures. Available at uops.info. Visited in October, 2018.

[7] G. Cox and A. Bhattacharjee. Efficient address translation for architectures with multiple page sizes. ASPLOS, 2017.

[8] Disabling Transparent HugePages. Available at docs.oracle.com/en/database/oracle/oracle-database/12.2/ladbi/disabling-transparent-hugepages.htmlguid-02e9147d-d565-4af8-b12a-8e6e9f74beea. Visited in October, 2018.

[9] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. ASPLOS, 2012.

[10] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. ISCA, 2015.

[11] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. Redundant memory mappings for fast access to large memories. ISCA, 2015.

[12] M. Kim and P. Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.

[13] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. S&P'19, 2019.

[14] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. SIGMOD, 2018.

[15] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and efficient huge page management with Ingens. OSDI, 2016.

[16] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. USENIX Security Symposium, 2018.

[17] C. H. Park, T. Heo, J. Jeong, and J. Huh. Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented Mmmory allocations. ISCA, 2017.

[18] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee. CoLT: Coalesced large-reach TLBs. MICRO, 2012.

[19] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? MICRO, 2015.

[20] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing. LookNN: Neural network with no multiplication. DATE, 2017.

[21] M. Tschannen, A. Khanna, and A. Anandkumar. StrassenNets: Deep learning with a multiplication budget. *arXiv preprint arXiv:1712.03942*, 2017.

[22] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. EuroSys, 2015.