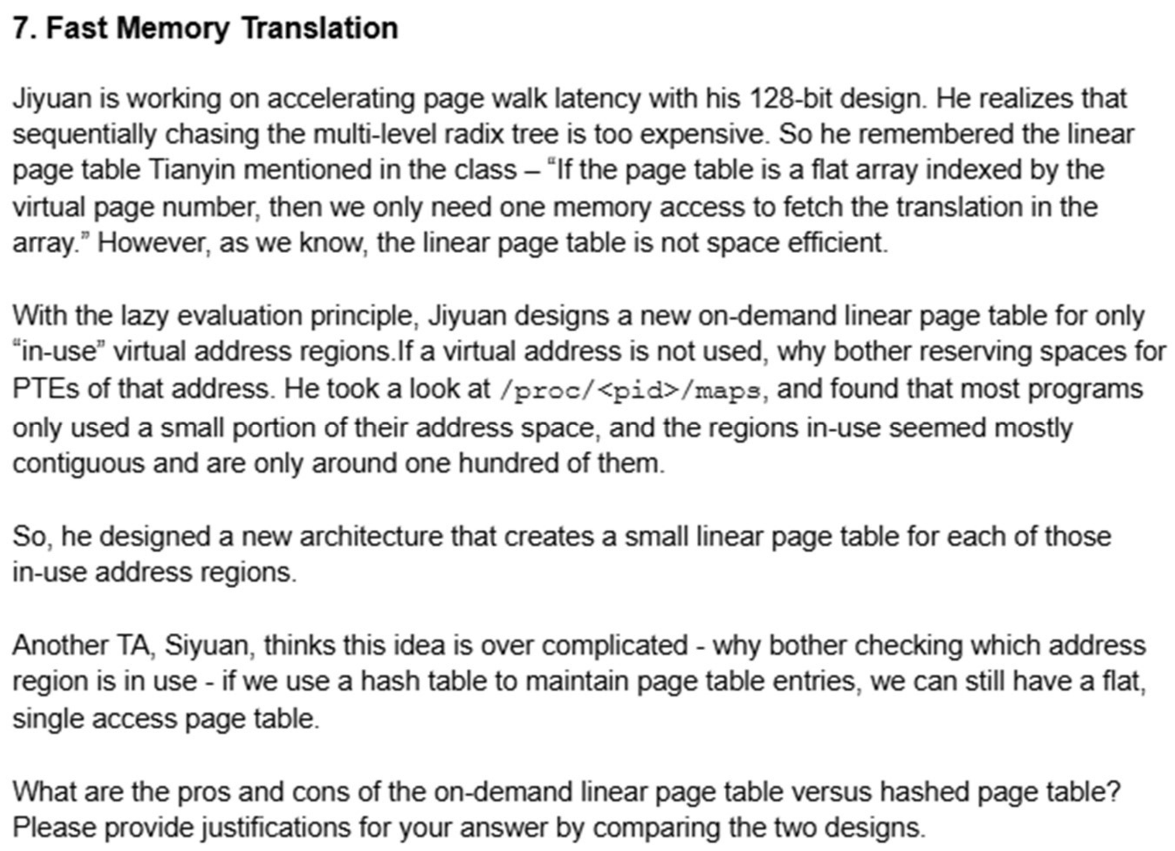
Direct Memory Translation for Virtualized Clouds

Jiyuan Zhang, Weiwei Jia, Siyuan Chai,
Peizhe Liu, Jongyul Kim, and Tianyin Xu

Top Secret 🤫



7. Fast Memory Translation

Jiyuan is working on accelerating page walk latency with his 128-bit design. He realizes that sequentially chasing the multi-level radix tree is too expensive. So he remembered the linear page table Tianyin mentioned in the class – “If the page table is a flat array indexed by the virtual page number, then we only need one memory access to fetch the translation in the array.” However, as we know, the linear page table is not space efficient.

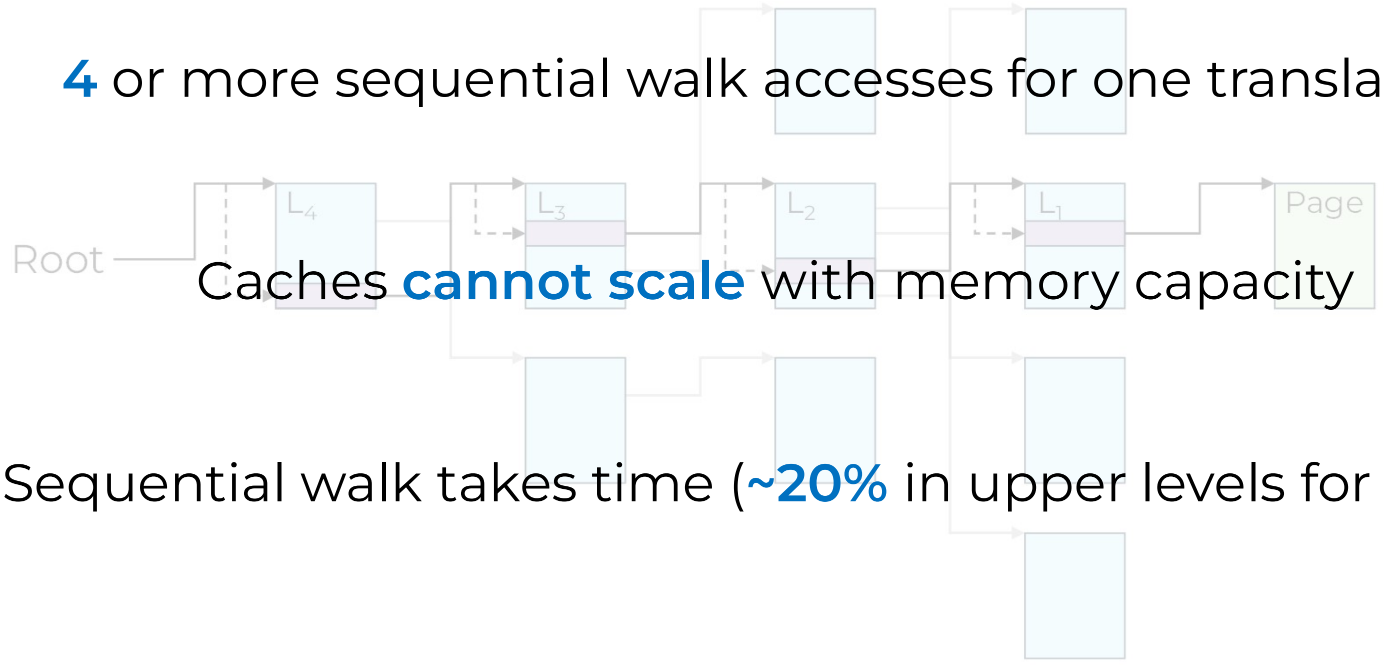
With the lazy evaluation principle, Jiyuan designs a new on-demand linear page table for only “in-use” virtual address regions.If a virtual address is not used, why bother reserving spaces for PTEs of that address. He took a look at `/proc/<pid>/maps`, and found that most programs only used a small portion of their address space, and the regions in-use seemed mostly contiguous and are only around one hundred of them.

So, he designed a new architecture that creates a small linear page table for each of those in-use address regions.

Another TA, Siyuan, thinks this idea is over complicated - why bother checking which address region is in use - if we use a hash table to maintain page table entries, we can still have a flat, single access page table.

What are the pros and cons of the on-demand linear page table versus hashed page table? Please provide justifications for your answer by comparing the two designs.

Translation Overhead is High



4 or more sequential walk accesses for one translation

Caches **cannot scale** with memory capacity

Sequential walk takes time (**~20%** in upper levels for Redis)

Our Work

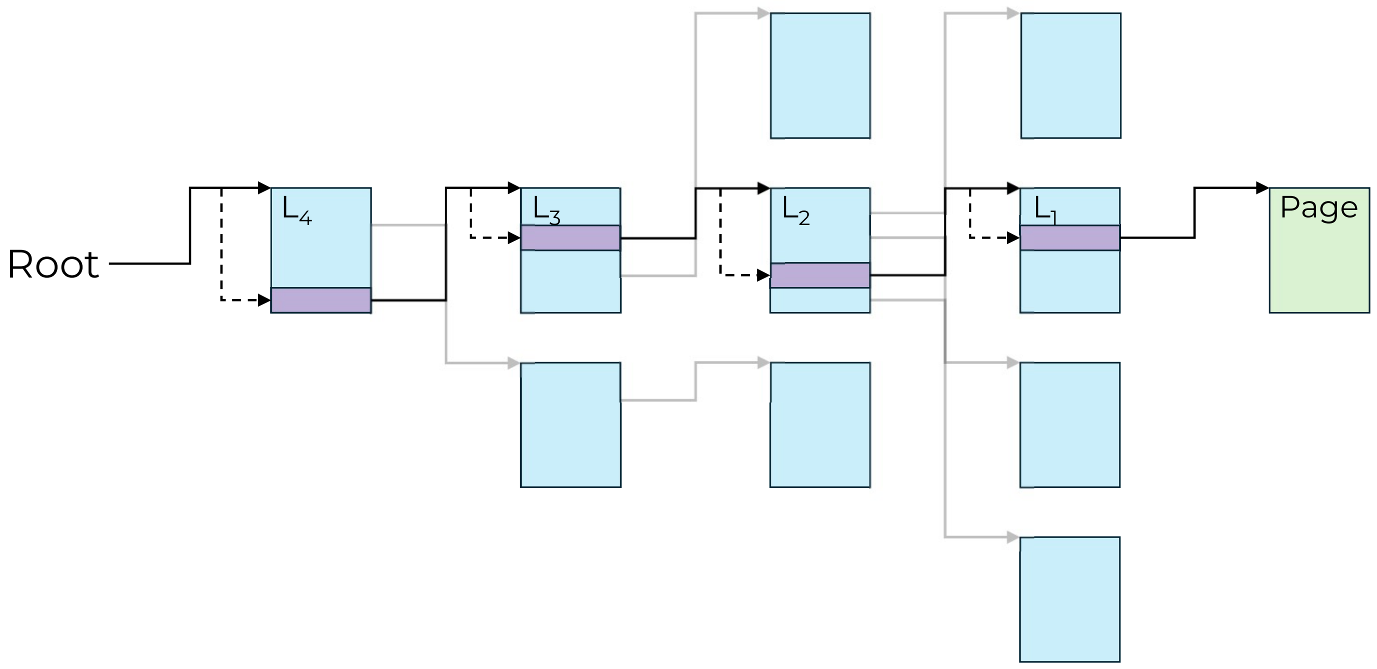
Problem: 4 sequential walk accesses

Our Work

Problem: **4** sequential walk accesses

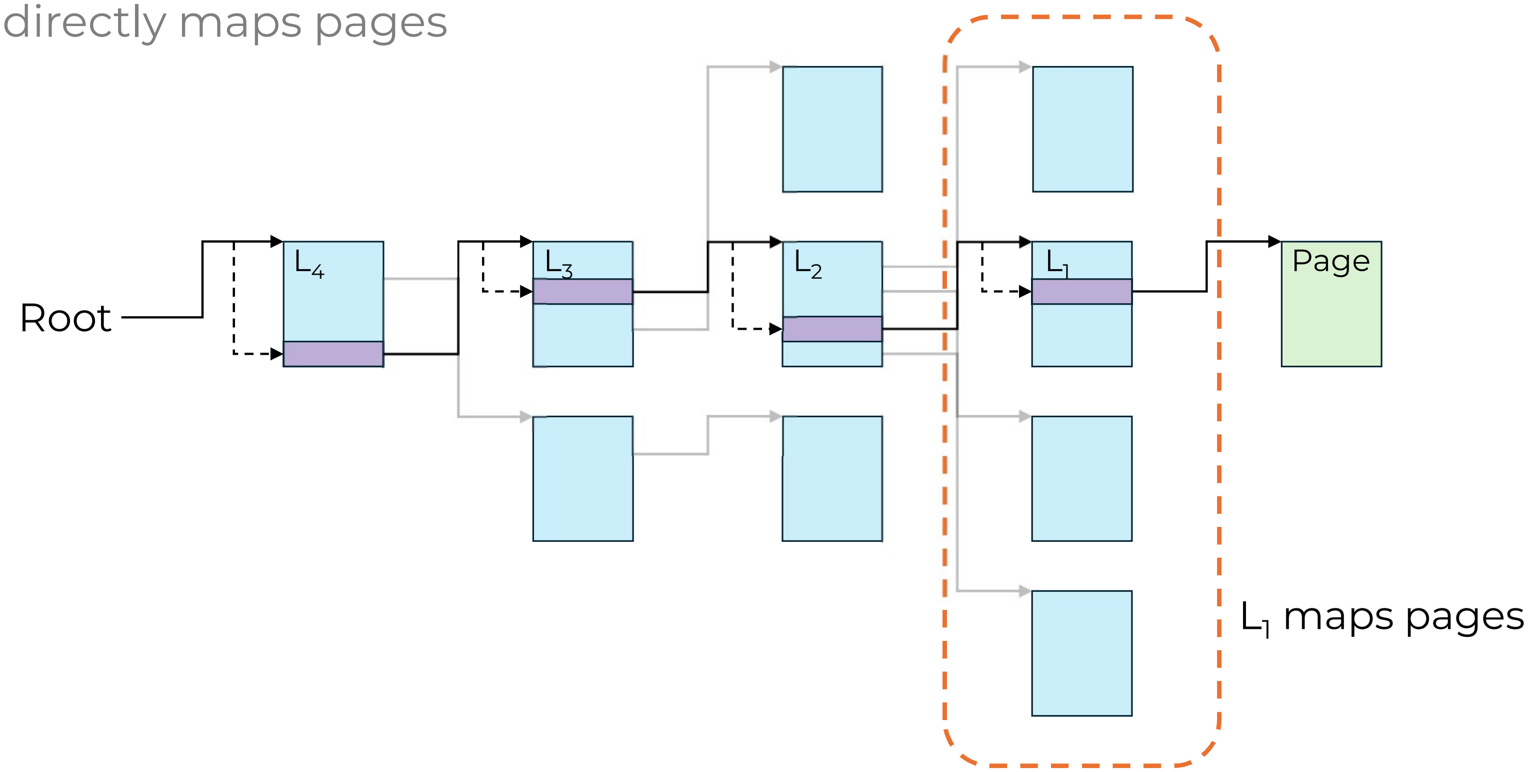
Solution: **Directly fetching** the last-level PTEs

From Conventional Radix Table



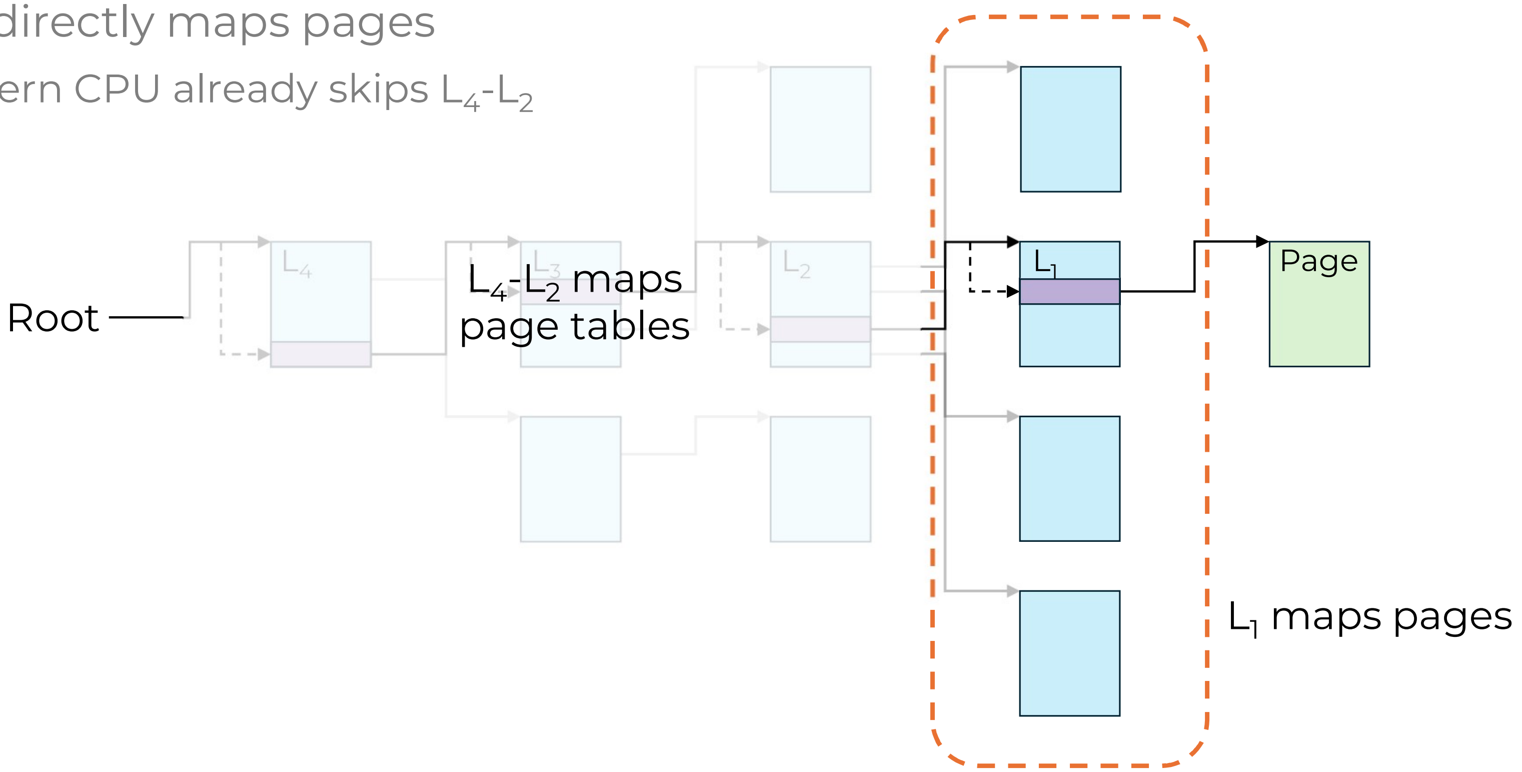
Only L_1 Directly Maps 4 KiB Pages

- Only L_1 directly maps pages



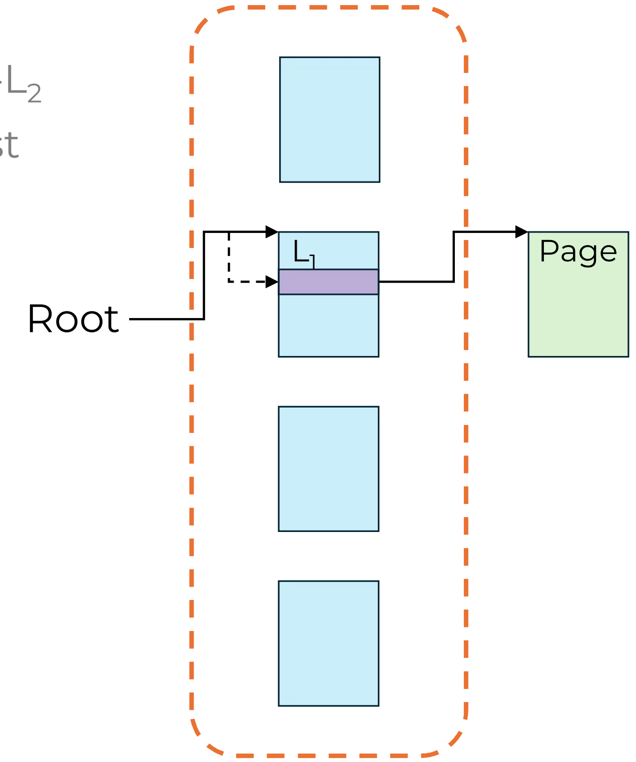
L_4-L_2 are Skippable

- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2



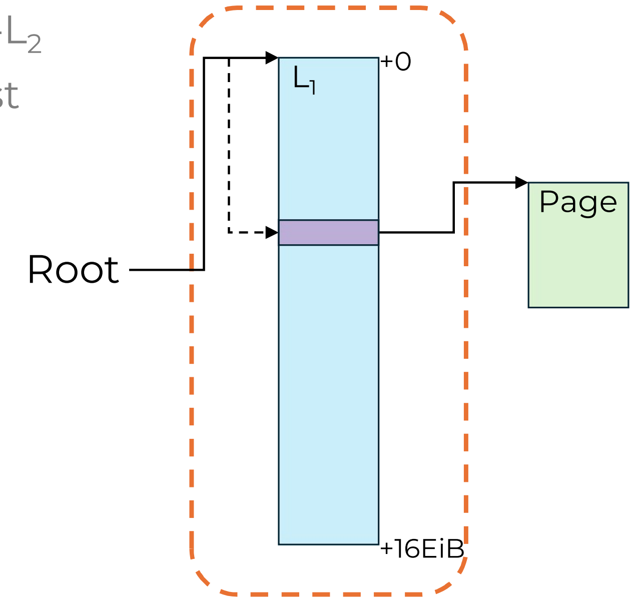
Directly Fetching L_1 PTE

- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost



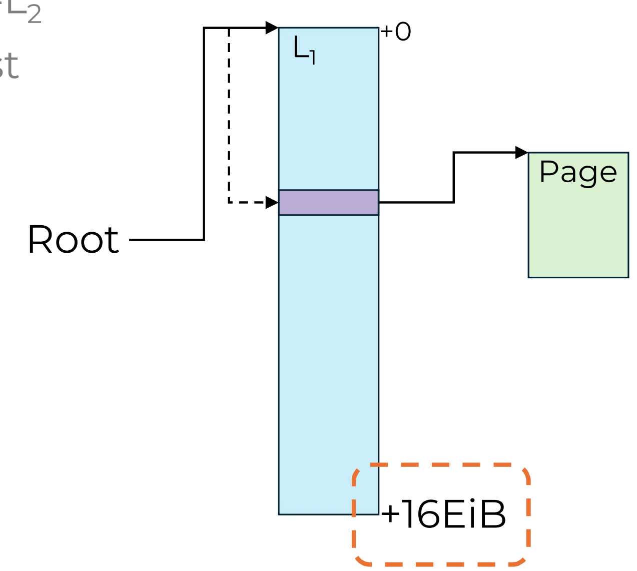
Physically Contiguous L_1 for Direct Indexing

- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost
- Use physically contiguous L_1
  - Can be direct indexed



32 PiB Memory Space Consumption?

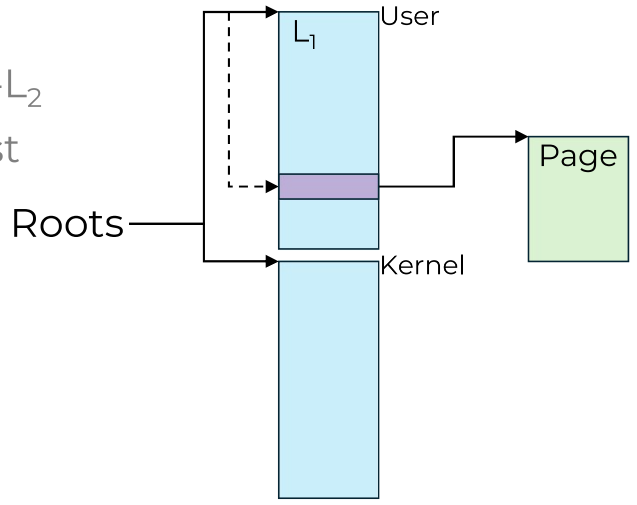
- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost
- Use physically contiguous L_1
  - Can be direct indexed
  - Huge memory consumption

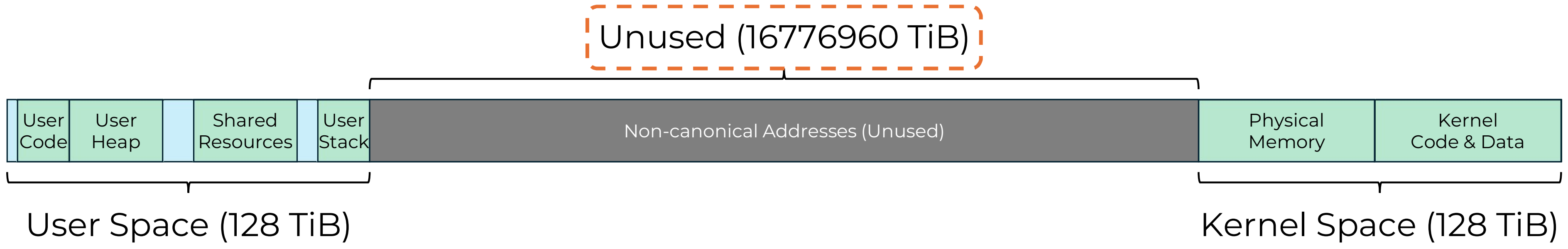


16 EiB ÷ 4 KiB/Page × 8 Bytes/PTE = **32 PiB/Addr. Space**

Not All Addresses are Mappable

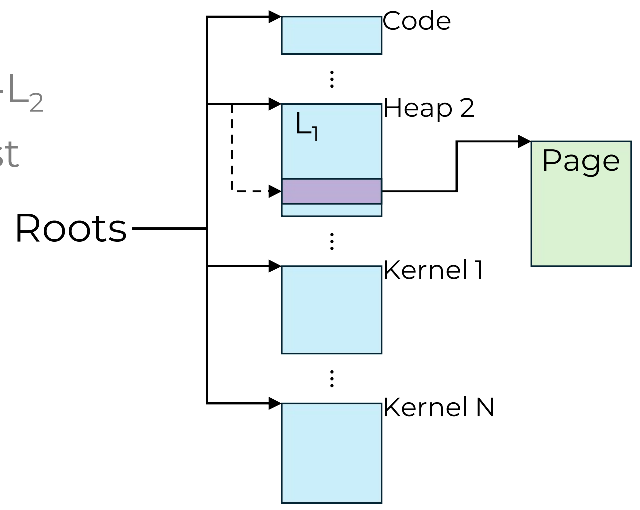
- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost
- Use physically contiguous L_1
  - Can be direct indexed
  - Huge memory consumption
- Do not map memory holes

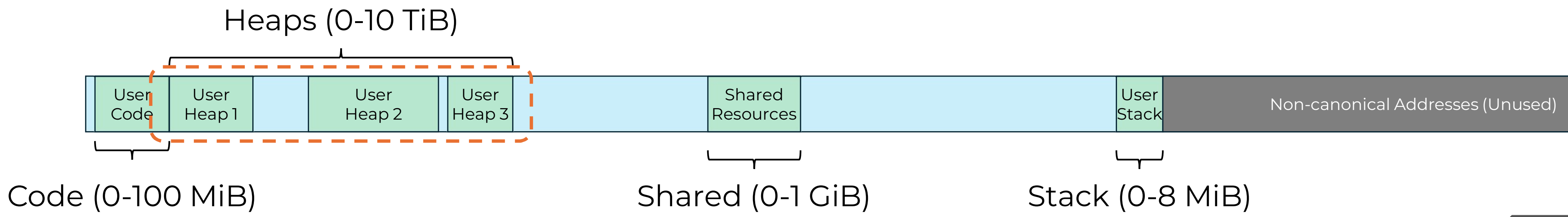




Not All Addresses are Mappable or Used

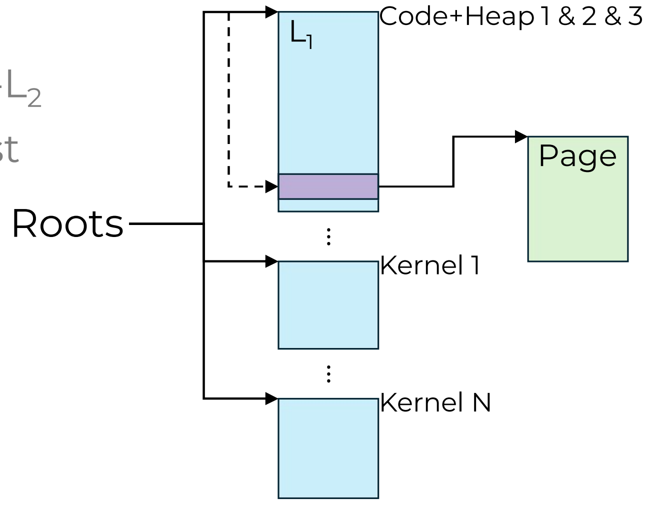
- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost
- Use physically contiguous L_1
  - Can be direct indexed
  - Huge memory consumption
- Do not map memory holes

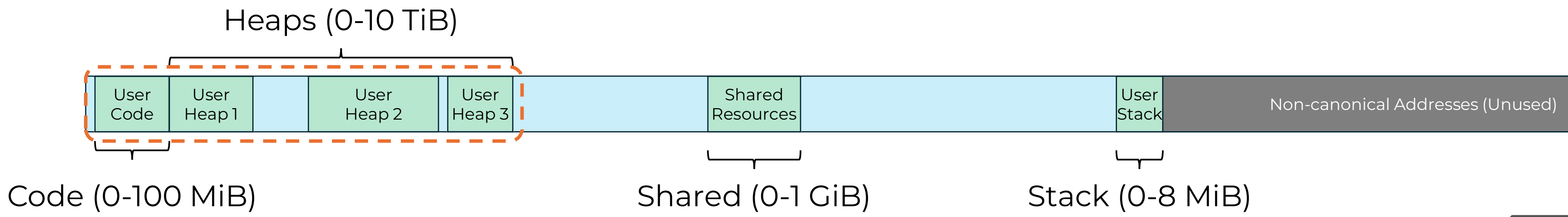




Cluster For Less Roots

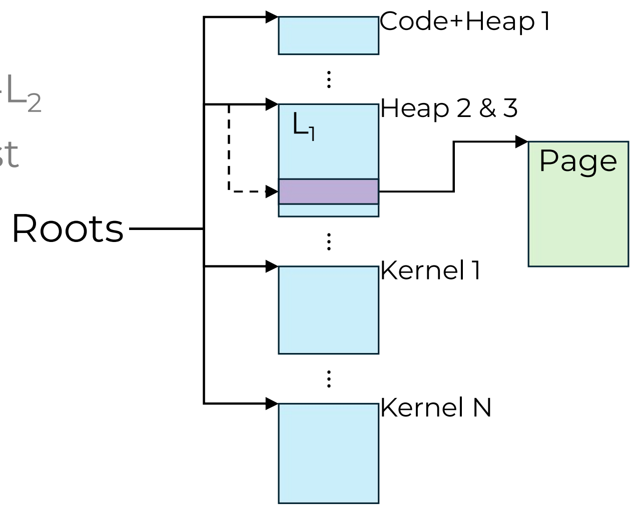
- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost
- Use physically contiguous L_1
  - Can be direct indexed
  - Huge memory consumption
- Do not map memory holes
  - Cluster and split for manageability

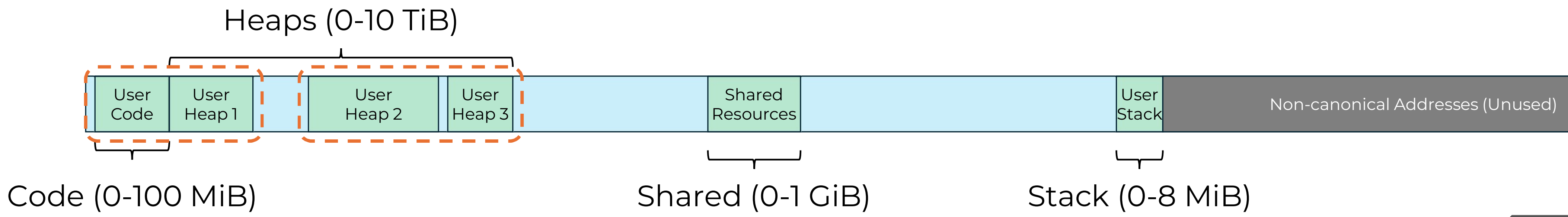




Split When Low Contiguity

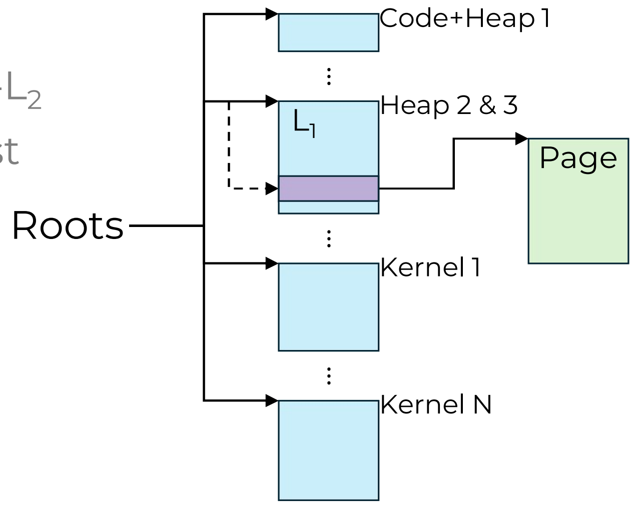
- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost
- Use physically contiguous L_1
  - Can be direct indexed
  - Huge memory consumption
- Do not map memory holes
  - Cluster and split for manageability





16 Roots are Enough

- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost
- Use physically contiguous L_1
  - Can be direct indexed
  - Huge memory consumption
- Do not map memory holes
  - Cluster and split for manageability
  - 16 contig. L_1s can map 99% of memory

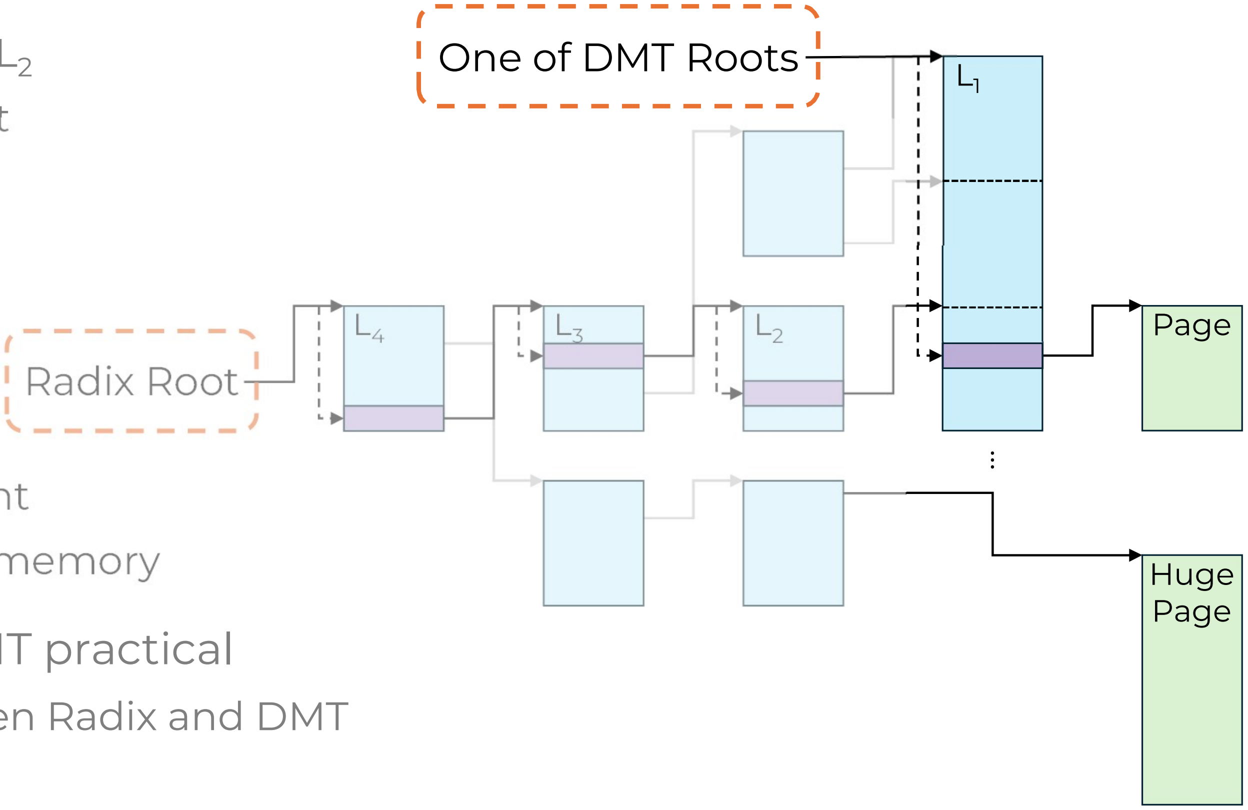


Only **a handful of** Virtual Memory Areas (VMA) are of significant size and are frequently accessed.

We can further **cluster** VMAs with small bubbles (<2% waste) to cover **99%** of total working set.

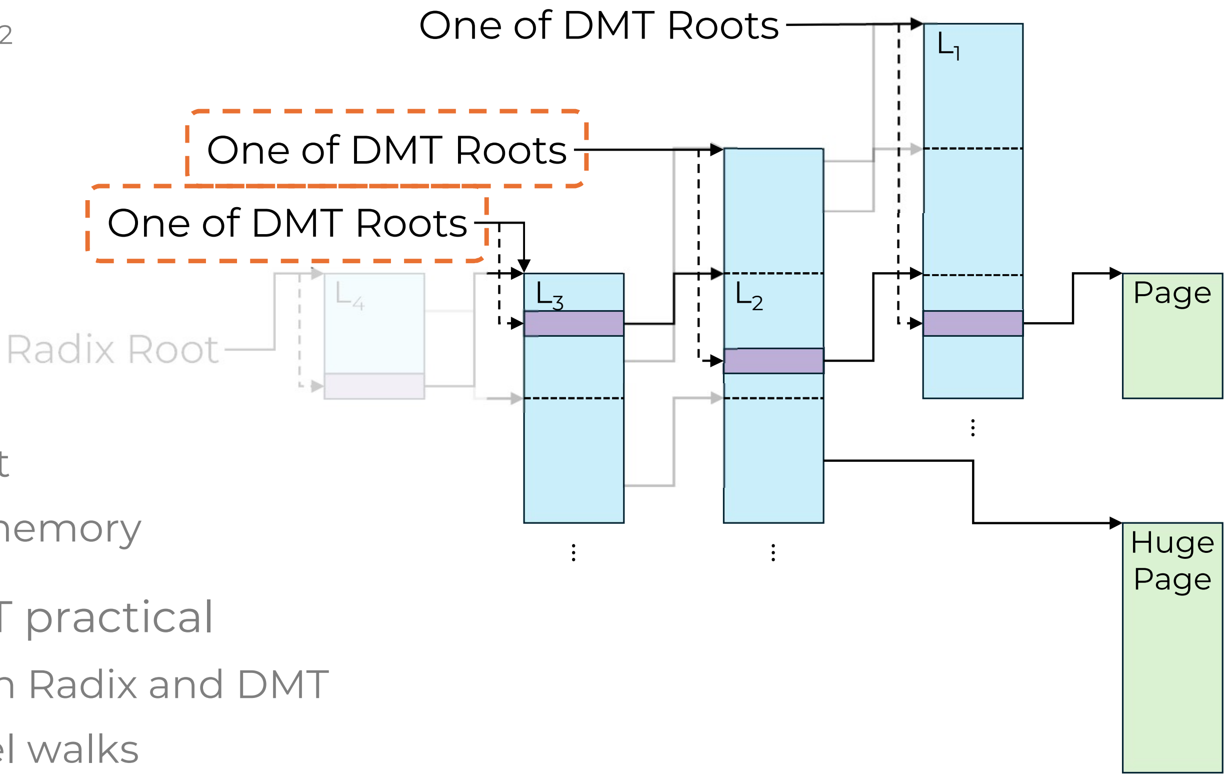
DMT is Radix Compatible

- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost
- Use physically contiguous L_1
  - Can be direct indexed
  - Huge memory consumption
- Do not map memory holes
  - Only a few VMAs are significant
  - 16 contig. L_1s can map 99% of memory
- Radix-compatibility makes DMT practical
  - Can seamlessly switch between Radix and DMT

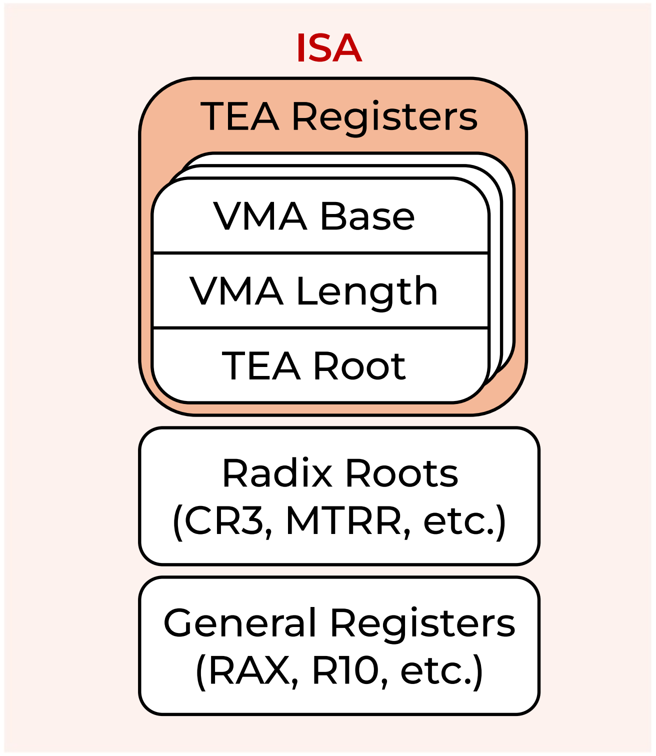


DMT Supports Huge Pages

- Only L_1 directly maps pages
  - Modern CPU already skips L_4-L_2
  - Directly fetch L_1 to reduce cost
- Use physically contiguous L_1
  - Can be direct indexed
  - Huge memory consumption
- Do not map memory holes
  - Only a few VMAs are significant
  - 16 contig. L_1s can map 99% of memory
- Radix-compatibility makes DMT practical
  - Can seamlessly switch between Radix and DMT
  - Supports huge page via parallel walks

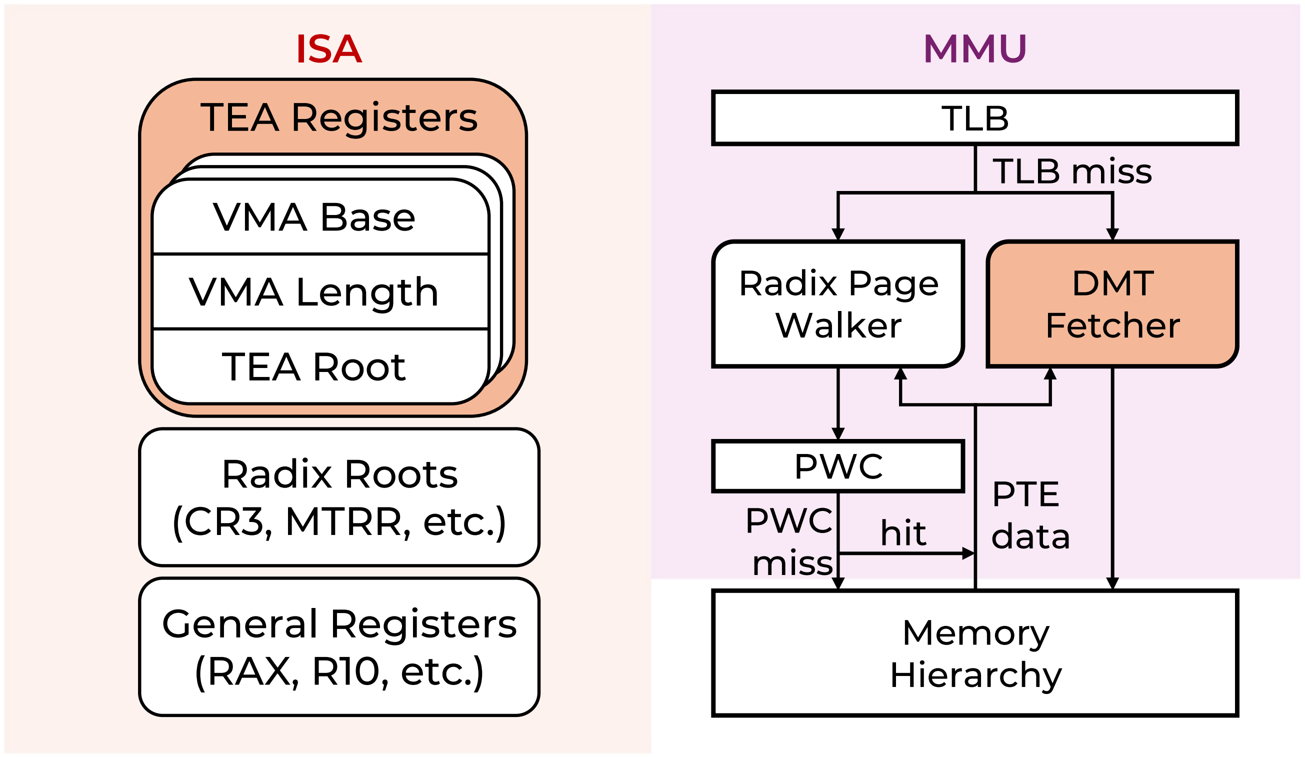


DMT requires minimal hardware support



DMT exposes TEA registers in the Instruction Set Architecture (ISA)

DMT requires minimal hardware support



DMT exposes TEA registers in the Instruction Set Architecture (ISA)
DMT adds a PTE fetcher side-by-side with the existing one in the Microarchitecture

Summary of Evaluation Results

- DMT speeds up page walk by an average of **1.28x** over vanilla x86.
- DMT speeds up application execution by an average of **1.05x**.

Conclusion

- DMT is a SW-HW extension that shortcuts page table walks
  - Native: **4** (x86-64) → **1** (DMT)
- DMT directly fetches the last-level page table entries (PTEs)
  - **Fully compatible** with existing radix page table design (e.g., x86 and ARM)
  - **Flexible and scalable** for OS and HW implementation
- Artifact available at: https://github.com/xlab-uiuc/dmt