Utopia: Fast and Efficient Address Translation via Hybrid Restrictive & Flexible Virtual-to-Physical Address Mappings

Konstantinos Kanellopoulos¹  Rahul Bera¹  Kosta Stojiljkovic¹  Nisa Bostanci¹  Can Firtina¹  Rachata Ausavarungnirun²  Rakesh Kumar³  Nastaran Hajinazar⁴  Mohammad Sadrosadati¹  Nandita Vijaykumar⁵  Onur Mutlu¹

¹ETH Zürich  ²King Mongkut’s University of Technology North Bangkok  ³Norwegian University of Science and Technology  ⁴Intel Labs  ⁵University of Toronto

Abstract
Conventional virtual memory (VM) frameworks enable a virtual address to flexibly map to any physical address. This flexibility necessitates large data structures to store virtual-to-physical mappings, which leads to high address translation latency and large translation-induced interference in the memory hierarchy, especially in data-intensive workloads. On the other hand, restricting the addressing mapping so that a virtual address can only map to a specific set of physical addresses can significantly reduce address translation overheads by making use of compact and efficient translation structures. However, restricting the addressing mapping flexibility across the entire main memory severely limits data sharing across different processes and increases data accesses to the swap space of the storage device even in the presence of free memory.

We propose Utopia, a new hybrid virtual-to-physical address mapping scheme that allows both flexible and restrictive hash-based address mapping schemes to harmoniously co-exist in the system. The key idea of Utopia is to manage physical memory using two types of physical memory segments: restrictive segments and flexible segments. A restrictive segment uses a restrictive, hash-based address mapping scheme that maps virtual addresses to only a specific set of physical addresses and enables faster address translation using compact translation structures. A flexible segment employs the conventional fully-flexible address mapping scheme. By mapping data to a restrictive segment, Utopia enables faster address translation with lower translation-induced interference. At the same time, Utopia retains the ability to use the flexible address mapping to support conventional VM features such as data sharing and avoid storing data in the swap space of the storage device when program data does not fit inside a restrictive segment.

Our evaluation using 11 diverse data-intensive workloads shows that Utopia improves performance by 24% in a single-core system over the baseline conventional four-level radix-tree page table design, whereas the best prior state-of-the-art contiguity-aware translation scheme improves performance by 13%. Utopia provides 95% of the performance benefits of an ideal address translation scheme where every translation request hits in the first-level TLB. All of Utopia’s benefits come at a modest cost of 0.64% area overhead and 0.72% power overhead compared to a modern high-end CPU. The source code of Utopia is freely available at https://github.com/CMU-SAFARI/Utopia.

1 Introduction
Virtual memory (VM) serves as a foundational element in most computing systems, simplifying the programming model by offering an abstraction layer over physical memory [2–24]. In the presence of VM, the operating system (OS) maps each virtual address to its corresponding physical memory address to facilitate application-transparent memory management, process isolation, and memory protection. The virtual-to-physical mapping scheme in conventional VM frameworks allows a virtual address to flexibly map to any physical address. This flexibility enables key VM functionalities, such as (i) data sharing between processes while maintaining process isolation and (ii) avoiding frequent swapping (i.e., avoiding storing data in the swap space of the storage device in the presence of free main memory space). However, a flexible mapping scheme requires mapping metadata for every virtual address and its corresponding physical address, which is stored in the page table (PT). As shown in multiple prior works [25–35], data-intensive workloads do not efficiently use translation-dedicated hardware structures and the processor performs frequent PT accesses, i.e., a process called PT walk (PTW), to resolve address translation requests. Frequent accesses to the PT heavily impact system performance in two ways: they lead to (i) high address translation latency and (ii) interference between program data and the PT data across the memory hierarchy, i.e., CPU caches, interconnect and main memory.

High address translation latency. As data-intensive applications use increasingly larger data sets, the size of the PT grows, which increases the latency of PTWs. For example, modern x86-64 systems use a four-level radix-tree PT that requires up to four serialized memory accesses, to translate a virtual address to its corresponding physical address [36]. For workloads that make scarce use of the main memory capacity, walking the four-level radix table is fast since the PTs are small enough to fit in on-chip caches. However, the large data footprints of emerging data-intensive workloads (e.g., graph analytics [37, 38], recommendation systems [39, 40], generative models [41, 42]) lead to large PTs that do not fit in on-chip caches. For example, given an application with a 2TB dataset, the x86-64 PT’s size can reach up to 4GB, which is much larger than the total caching capacity of a modern high-end CPU [43]. As we demonstrate in §3, even using the state-of-the-art hash-based PT design [44] in a system that supports both 4KB and 2MB pages [45–48], a PTW takes an average of 86 cycles (up to 123) to complete, across 11 data-intensive workloads. High frequency and high latency PTWs lead to high address translation latency and degrade system performance.
Translation-induced interference in the memory hierarchy. During a PTW, the processor issues memory requests to the memory hierarchy in order to fetch the PT. Upon retrieval from main memory, the PT data is stored within the cache hierarchy. As a result, PTWs interfere with the memory hierarchy of a processor in two major ways. First, PTWs consume the scarce on-chip cache hierarchy space (to store translation metadata), which otherwise could be used to cache program data. Second, PTWs increase DRAM row buffer misses due to frequent DRAM accesses for retrieving translation metadata. In §3, we show that (i) data-intensive applications consume up to 38% of the L2 cache capacity only to store PT data, and (ii) memory requests for PT data increase DRAM row buffer misses by 30% compared to an ideal system that uses a perfect translation-lookaside buffer (TLB).

Prior works [15, 49, 50] explore the possibility of restricting the virtual-to-physical mapping (e.g., the physical address can be computed based on a hash function applied to the virtual address) flexibility to reduce the size of the data structures that store translation metadata and reduce the address translation overhead. Restricting the virtual-to-physical mapping drastically reduces the size of the translation data structures, and accordingly lowers the latency of retrieving the virtual-to-physical mapping. For example, as shown in [49], determining the physical location of a virtual page based on a specific set of bits of the virtual address is considerably faster than accessing the x86-64 multi-level PT. However, restricting the address mapping across the entire physical address space in general-purpose systems handicaps core VM functionalities and can cause severe performance overheads. First, two virtual pages from different processes might not be able to map to the same physical page which limits data sharing. Second, the sole use of a restrictive mapping leads to memory underutilization as the system might not be able to freely map virtual pages to the available free physical space. This can cause more pages to be stored inside the swap space of the storage device even in the presence of free physical memory space. Our analysis in §3 shows that restricting the flexibility of the address mapping, as proposed in [49], for the whole main memory of a general-purpose system increases data accesses to the swap space of the storage device by 122%. We conclude that using only flexible or only restrictive hash-based address mapping does not satisfy the requirements of both highly-flexible memory management and high-performance, low-interference address translation.

Our goal is to design a virtual-to-physical address mapping scheme that provides fast and efficient translation via the use of a restrictive hash-based address mapping while still enjoying the benefits of the conventional fully-flexible address mapping. To this end, we propose Utopia, a new hybrid virtual-to-physical address mapping scheme that enables both flexible and restrictive hash-based virtual-to-physical address mapping schemes to harmoniously co-exist in the system. The key idea of Utopia is to manage physical memory using two types of physical memory segments: restrictive segments and the flexible segment. A restrictive segment (called RestSeg) enforces a restrictive, hash-based address mapping scheme, thereby enabling fast and efficient address translation through compact and efficient address translation structures. A flexible segment (called FlexSeg), employs the conventional address mapping scheme and provides full virtual-to-physical address mapping flexibility. By mapping data to a RestSeg, Utopia enables fast address translation with low translation-induced interference in the memory hierarchy whenever flexible address mapping is not necessary (e.g., when optimizing for fast address translation). At the same time, Utopia retains the ability to use flexible address mapping to (i) support conventional virtual memory features such as data sharing and (ii) avoid accesses to the swap space when data does not fit inside a restrictive segment.

Key Mechanism. We study an example implementation of Utopia that uses a set-associative address mapping (similar to how hardware caches work) as the restrictive hash-based address mapping scheme to map data to RestSegs. In contrast to the conventional fully-flexible address mapping, which requires expensive PTWs to resolve a translation request, the set-associative address mapping scheme requires only (i) calculating the set index by applying a hash function over the virtual address, and (ii) performing tag matching using a highly-compact and scalable data structure compared to the conventional PT.

Key Challenges. Integrating Utopia into a conventional system requires addressing three key challenges. First, identifying the pages that are good candidates for storing inside a RestSeg. To address this challenge, we propose a mechanism to predict whether or not a page is costly to translate. Based on this prediction, Utopia determines the potential benefits of allocating the page within a RestSeg (§3.5). Second, efficiently managing the co-existence of RestSegs and FlexSegs in a single physical address space. We address this challenge by extending the operating system to support the creation of RestSegs, allocation of pages in RestSegs and FlexSegs, and migration of pages between the two segments (§5.6). Third, accelerating address translation for pages that reside in a RestSeg. We achieve this by extending the processor with architectural support to efficiently access the data structures of the RestSeg with minimal overhead on top of the existing PTW path (§5.7).

Key Results. We evaluate Utopia with an extended version of the Sniper simulator [51] (which we open-source [52]) using 11 data-intensive applications from five diverse benchmark suites (Graph-BIG [53], GUPS [54], XSBench [55], DLRM [39] and Genomics-Bench [56]). Our evaluation yields five key results that demonstrate Utopia’s effectiveness. First, in single-core (four-core) workloads, Utopia improves performance by 24% (28%) on average over the conventional four-level radix-tree baseline PT design, whereas two prior state-of-the-art translation mechanisms, elastic cuckoo hashing (ECH [44]) and redundant memory mappings (RMM [57]) improve performance by 8% (14%) and 13% (12%), respectively. Second, in single-core workloads, Utopia provides 95% of the performance of an ideal address translation scheme where every translation request hits in the L1 TLB. Third, Utopia reduces address translation latency by 69% over the baseline radix-based page table in single-core workloads, whereas ECH and RMM reduce translation latency by 39% and 15%, respectively. Fourth, Utopia reduces DRAM row buffer misses by 20% compared to the baseline system. Fifth, all of Utopia’s benefits come at a modest cost of 0.74% area overhead and 0.62% power overhead compared to a modern high-end CPU [43]. We make the following major contributions in this paper:

1.i.e., a system where every translation request hits in the L1 TLB.
• We demonstrate that although using a restrictive virtual-to-
physical address mapping reduces the address translation
overhead and memory hierarchy interference, employing a
restrictive address mapping across the entire physical address
space limits key benefits of VM design such as data sharing
between processes and high memory utilization.
• We propose Utopia, a new hybrid virtual-to-physical address
mapping scheme that allows both flexible and restrictive
hash-based virtual-to-physical address mapping schemes to
harmoniously co-exist in the system. This way, Utopia enables
efficient address translation and reduces translation-induced
interference in the memory hierarchy while maintaining the
benefits of a fully-flexible address mapping scheme.
• We devise three key components to integrate Utopia into a
conventional system: (i) a set of techniques to identify costly-
to-translate pages that benefit from a restrictive mapping,
(ii) OS support to manage the co-existence of flexible and
restrictive address mappings in the same physical address
space, and (iii) lightweight architectural support to integrate
Utopia in the processor’s translation pipeline.
• We quantitatively evaluate Utopia in single-core and multi-
core environments and compare it against three state-of-
the-art address translation mechanisms. Our experimental
results show that Utopia significantly reduces the overheads
associated with address translation at a modest low area and
power cost. We open-source Utopia at https://github.com/
CMU-SAFARE/Utopia.

2 Background

2.1 The Virtual Memory Abstraction

Virtual memory is a cornerstone of most modern computing sys-
tems that eases the programming model by providing a convenient
abstraction to the physical memory [2–24]. The operating system
(OS), transparently to application software, maps each virtual mem-
ory address to its corresponding physical memory address. Doing so
provides a number of benefits, including: (i) application-transparent
memory management, (ii) sharing data between applications, (iii)
process isolation and (iv) page-level memory protection. Conven-
tional virtual memory designs allow any virtual page to map to
any free physical page. Such a flexible address mapping enables
two important key features of virtual memory: (i) efficient mem-
ory utilization to avoid frequent swapping, and (ii) sharing pages
between applications. However, such a flexible address mapping
mechanism has a critical downside: it creates the need to store a
large number of virtual-to-physical mappings, since the OS needs
to keep track of the physical location of every virtual page that is
used by each process.

2.2 Page Table (PT)

The PT is a per-process data structure that stores the mappings
between virtual and physical pages. In modern x86-64 processors,
the PT is organized as a four-level radix-tree [36]. Even though the
radix-tree-based PT optimizes for storage efficiency, it requires mul-
tiple pointer-chasing operations to discover the virtual-to-physical
mapping. To search for a virtual-to-physical address mapping, the
system needs to sequentially access each of the four-levels of the
page table. This process is called a page table walk (PTW).

Figure 1 shows the page table walk assuming (i) an x86-64 four-
level radix-tree page table whose base address is stored in the CR3
register, and (ii) 4KB pages. As shown in Fig. 1, a single PTW
requires four sequential memory accesses to discover the
physical page number. The processor uses the first 9-bits of the
48-bit virtual address as offset (Page Map Level4; PML4) to index the
appropriate entry of the page table within the first level of the page
table. The processor then reads the pointer stored in the first
level of the page table to access the second-level of the page table.
It uses the next 9-bit set (Page Directory Page table; PDP) from the
virtual address to locate the appropriate entry within the second
level. This process continues iteratively for each subsequent level
of the multi-level (hierarchical) page table (Page Directory; PD
and Page Table; PT). Eventually, the processor reaches the leaf
level of the page table, where it finds the final entry containing the
physical page number corresponding to the given virtual address.
As shown in multiple previous works [25–35], PTWs incur high
translation latency, which leads to high performance overheads.
ARM processors use a similar approach, with the number of levels
varying across different versions of the ISA [61].

![Four-level radix-tree page table walk in x86-64 ISA.](image)

2.3 Memory Management Unit (MMU)

When a user process generates a memory (i.e., instruction or data)
request, the processor needs to translate the virtual address to its
corresponding physical address. Address translation is a critical op-
eration because it is on the critical path of the memory access flow:
no memory access is possible unless the requested virtual address
is first translated into its corresponding physical address. Given
that frequent page table walks lead to high address translation
overheads, modern cores employ a specialized memory manage-
ment unit (MMU) responsible for accelerating address translation.
Figure 2 shows an example structure of the MMU of a modern
processor [62], consisting of three key components: (i) a two-level
hierarchy of translation lookaside buffers (TLBs), (ii) a hardware
page table walker, (iii) page walk caches (PWCs).

L1 TLBs are highly- or fully-associative translation caches that
directly provide the physical address for recently-accessed virtual
pages at very low latency (i.e., typically within 1 cycle). There are
two separate L1 TLBs, one for instructions (L1 I-TLB) and one for

---

Swap space is a reserved space in the storage device that is used to store pages
that are not currently mapped in the main memory [58–60]. When a page is evicted
from the main memory, the OS stores it in the swap space. When the page is accessed
again, the OS loads it back into the main memory.

CR3 register stores the page table base address in the x86-64 ISA.
3 Motivation

Data-intensive workloads (e.g., graph analytics [37, 38], recommendation systems [39, 40], generative models [41, 42]) use large datasets and exhibit irregular memory access patterns that lead to large and costly-to-access page tables. Multiple prior works [25–35] and large-scale industrial studies [29, 30] demonstrate that a wide range of data-intensive workloads experience high TLB miss rates and high PTW latencies. Figure 3 shows the L2 TLB MPKI of the baseline system, as we increase the L2 TLB size from 1.5K entries up to 64K entries, for 11 memory-intensive workloads. We observe that the baseline 1.5K-entry L2 TLB suffers from high average MPKI, 39 on average and up to 77. Even using a drastically larger 64K-entry L2 TLB, the average MPKI remains high at 24 (and up to 54), resulting in frequent PTWs. Frequent and high-latency PTWs pose two key challenges that significantly impact system performance: (i) high address translation latency and (ii) high address translation-interference in the memory hierarchy.

Figure 3: L2 TLB MPKI across L2 TLBs with different sizes.

High Address Translation Latency. To better understand the performance of address translation in data-intensive workloads, we study the effectiveness of two different systems: (i) a baseline system that uses the conventional four-level radix page table (Radix) and (ii) a system that uses the state-of-the-art elastic cuckoo hash-based page table (ECH) proposed in [44]. Both systems use 4KB and 2MB pages allocated by the Linux Transparent Huge Page (THP) mechanism [45, 46]. Figure 4 shows the average PTW latency (in processor cycles) for Radix and ECH. We observe that Radix spends 137 cycles and ECH 86 cycles, on average, to complete the PTW.

Figure 4: Average PTW latency in Radix and ECH.

Figure 5 demonstrates the breakdown of the servicing location (DRAM, LLC, L2) of memory requests to access the PT in both Radix and ECH, normalized to Radix. We make two key observations. First, an average of 43% of the PT requests are serviced from DRAM, in Radix. This is the key reason behind the long average PTW latency of Radix (137 cycles). Second, although ECH reduces the fraction of PT requests that hit in the DRAM, it increases the total number of memory requests (to access the PT) by 62% on average compared to Radix. This is because ECH looks up 4 hash tables in parallel and issues multiple memory requests to the memory hierarchy. Yet, only one of the issued requests is actually necessary (i.e., one request will fetch the correct virtual-to-physical address translation). We conclude that although (i) ECH reduces the average PTW latency compared to Radix and (ii) our evaluated system employs both 4KB and 2MB pages, the average PTW latency still remains high.

Figure 5: Breakdown of the servicing location of memory requests issued during PTWs for ECH normalized to Radix.

To better understand the headroom for improving the performance of address translation, we evaluate the performance of an ideal system that employs a perfect L1 TLB (P-TLB). Figure 6 shows the execution time speedup of ECH and P-TLB compared to Radix. We observe that P-TLB outperforms Radix by 30% and ECH by 22%. We conclude that there is room for further improving the performance of address translation.

Translation-induced Interference in Memory Hierarchy. To better understand the impact of PTWs on the system, we evaluate two example measures of memory interference: (i) the fraction of cache blocks that store PT entries across the cache hierarchy, and (ii) the fraction of memory requests that deliver the translation. Figure 5 reports the servicing location of all the memory requests issued by the page table walker. ECH issues more memory requests than Radix as (i) it employs n = 4 hash tables and accesses them in parallel and (ii) the entries of the hash tables are not cached in a specialized component similar to the PWC in Radix. Figure 4 reports the PTW latency only for the requests that deliver the translation. Figure 5 reports the servicing location of all the memory requests issued by the page table walker.

4Section 7 describes in detail our evaluation methodology.
and (ii) how address translation affects DRAM row buffer misses. Figure 7 shows the fraction of cache blocks of two caches (L2, LLC) that store PT data (L1 typically does not store PT entries [66–68]), averaged across 500 epochs of 1M instructions, for Radix and ECH. We observe that both Radix and ECH use significant fraction of cache capacity in the cache hierarchy. For example, Radix and ECH respectively use 33% and 57% of L2’s total capacity for PT entries. The high usage of cache blocks for PT entries reduces the effective capacity of the cache hierarchy, which otherwise could have been used to store the data of (i) the running application and (ii) other applications running on the system if the LLC is shared.

Figure 7: Fraction of cache blocks that contain PT entries.

Figure 8 shows the reduction in DRAM row buffer conflicts provided by ECH and a perfect L1 TLB (P-TLB) compared to Radix. We observe that (i) ECH increases DRAM row buffer conflicts by 50% due to the increase in memory requests sent to DRAM and (ii) P-TLB decreases row buffer conflicts by 30% due to the reduced number of DRAM row activations for translation metadata. We conclude that designing more compact and efficient translation structures (and thus ideally approaching a perfect TLB) can lead to a significant reduction in memory hierarchy interference.

Figure 8: Normalized DRAM row buffer conflicts for ECH and perfect TLB over Radix.

Restrictive Hash-based Mapping Prior works [15, 49, 50] explore the possibility of restricting the virtual-to-physical mapping flexibility (e.g., by computing the physical address using a hash function applied to the virtual address) to reduce the size of the data structures that store translation metadata and thus reduce the address translation overhead. Restricting the virtual-to-physical mapping drastically reduces the size of the translation data structures, and accordingly lowers the latency of retrieving the virtual-to-physical mapping. For example, as shown in [49], determining the physical location of a virtual page based on a specific set of bits of the virtual address is considerably faster than accessing the x86-64 multi-level PT. However, restricting the address mapping across the entire physical address space in general-purpose systems handicaps core VM functionalities and can cause severe performance overheads. First, two virtual pages from different processes might not be able to map to the same physical page, which limits data sharing. Second, the sole use of a restrictive mapping leads to increased swapping activity as the system might not be able to freely map virtual pages to the available free physical space. This can cause more pages to be stored inside the swap space of the storage device even in the presence of free physical memory space.

Figure 9 shows the increase in the number of data accesses to the swap space in a system that uses only the restrictive mapping across the whole memory, similar to [49], compared to the baseline system. We observe that employing a restrictive address mapping in the entire memory space causes a significant increase in swap space accesses, 2.2× on average, since a large number of virtual pages cannot be mapped inside physical memory and need to be stored into and fetched from the swap space. Fetching data from the swap space is orders of magnitude slower than fetching data from DRAM, which leads to significant performance overheads [69].

Figure 9: Accesses to the swap space using a system that employs a restrictive mapping across the entire memory space normalized to the baseline system.

We conclude that neither the sole use of a restrictive address mapping nor the sole use of a flexible address mapping across the entire physical address space is an effective solution for reducing address translation overheads while maintaining the core benefits and functionalities of VM.

4 Utopia: Overview

4.1 Key Idea & Design Overview

In this work, we propose Utopia, a new hybrid virtual-to-physical address mapping scheme that enables both flexible and restrictive hash-based virtual-to-physical address mapping schemes to harmoniously co-exist in the system. The key idea of Utopia is to manage physical memory using two types of physical memory segments: restrictive and flexible. A restrictive segment (called RestSeg) enforces a restrictive, hash-based address mapping scheme, thereby enabling fast and efficient address translation through the use of compact and efficient address translation structures. A flexible segment (called FlexSeg), employs the conventional address mapping scheme and provides full virtual-to-physical address mapping flexibility. Utopia identifies and maps costly-to-translate addresses to RestSegs, thereby enabling fast address translation with low translation-induced interference in the memory hierarchy whenever flexible address mapping is not necessary (e.g., when optimizing for fast address translation). At the same time, Utopia retains the ability to use flexible address mapping to (i) support conventional VM features such as data sharing and (ii) avoid accesses to the swap space when data does not fit inside a restrictive segment.

Figure 10 shows a simplified example of Utopia, in which a FlexSeg and a RestSeg co-exist in the main memory. A virtual
page can be mapped to any of the physical pages (Pages 0-3) in the FlexSeg 1, but address translation incurs high latency due to the costly memory accesses to the conventional page table 2. In contrast, a virtual page can be mapped to only one single physical page (Page 5) inside the RestSeg 3, whose physical page number is calculated using a hash function on the virtual address (e.g., based on the LSBs of the virtual page number). Thus, the RestSeg results in faster address translation but lower flexibility 4, compared to FlexSeg. In Utopia, each virtual page can reside in at most one type of segment (i.e., either in a FlexSeg or in a RestSeg but not both).

To perform address translation in Utopia, after an L1 TLB miss the MMU accesses in parallel (i) the translation structures of each RestSeg, i.e., an operation we call RestSegWalk (RSW), to discover if the data is stored in a RestSeg and (ii) the L2 TLB. If the data is stored neither in a RestSeg nor in the L2 TLB, the system initiates a conventional PTW to discover the physical location of the data in the FlexSeg.

4.2 Design Challenges

To enable the implementation of Utopia in an efficient manner, we need to address three key challenges: (i) how to decide which data should be placed in a RestSeg, (ii) create and maintain RestSegs and FlexSegs, and (iii) how to integrate Utopia in the conventional address translation pipeline.

Challenge 1. To address the first challenge, we design Utopia to place pages that experience high address translation latencies inside a RestSeg. To achieve that, Utopia uses two application-transparent techniques that track costly-to-translate pages and allocate them into a RestSeg: (i) a technique that monitors the PTW cost and PTW frequency of each page and decides if a page is costly-to-translate based on these two metrics, and (ii) a page fault-based technique that directly allocates costly-to-translate pages in a RestSeg after a page fault.

Challenge 2. To address the second challenge, we extend the OS to support the (i) creation and maintenance of RestSegs and FlexSegs, (ii) the allocation of pages inside a RestSeg, and (iii) the migration of data between RestSegs and FlexSegs. In our implementation of Utopia, the OS creates RestSegs during boot time to avoid the overhead of searching for (or creating) contiguous memory regions during runtime.

Challenge 3. To address the third challenge, we enhance the MMU with lightweight architectural support to integrate Utopia in the address translation pipeline. The MMU is extended in three ways: (i) we incorporate new hardware circuitry to enable access to the translation metadata of RestSegs, (ii) we add two 2KB caches to provide fast access to the recently-used translation metadata of RestSegs and (iii) we parallelize the access to the translation metadata of RestSegs with the L2 TLB access to reduce address translation latency.

5 Utopia: Detailed Design

We describe in detail (i) the key properties of a RestSeg, (ii) how to perform address translation for pages that reside in a RestSeg, (iii) how to resolve address translation in the presence of hybrid address mapping, (iv) how Utopia decides which data should be placed into a RestSeg (Challenge 1), (v) the OS extensions to enable Utopia (Challenge 2), and (vi) the architectural modifications in the MMU to efficiently support Utopia (Challenge 3).

5.1 Segment with Restrictive Address Mapping

We design RestSeg as a physical memory segment that enforces a set-associative address mapping. A virtual page can map only to a specific set of physical pages in the RestSeg, in a similar way that set-associative caches store sets of cache blocks at a particular index value. The set-associative address mapping accelerates address translation for virtual pages mapped in a RestSeg since discovering a virtual-to-physical mapping only requires calculating the set index using a hash operation on the virtual address followed by a tag matching operation. In this section, we discuss the key properties of a RestSeg and the translation structures that enable efficient address translation for pages that reside in a RestSeg.

5.1.1 Key Properties of RestSeg

RestSeg’s key properties enable (i) adaptability to diverse workloads and system configurations, (ii) backward compatibility with existing OS primitives, and (iii) efficient address translation for multiple processes at the same time.

Structural Properties of RestSeg. Figure 11 shows the structure of a RestSeg. RestSeg is a contiguous physical memory segment of associativity $M$ that contains $N$ physical pages 6 of equal size 7 (e.g., 4KB) organized in $N/M$ sets 8 of $M$ ways 9. Each virtual page can map to any of the $M$ ways of the set that it corresponds to. Different RestSegs can be configured at their creation with different size, page size and associativity (as we discuss in §5.6, a RestSeg is created during boot time) to adapt to a diverse set of workloads and system configurations.

Figure 11: Structural properties of a RestSeg.

Multiple RestSegs in the System. In our implementation, Utopia uses two RestSegs, one that stores 4KB and one that stores 2MB pages to retain backward compatibility with existing large page mechanisms [45, 46]. However, it is possible to employ more than two RestSegs in a single system to satisfy the needs of different

\[ i.e., \text{a memory block can map to a specific set of cache blocks.} \]
workloads (e.g., to support three different page sizes [63] or relieve a fully-allocated RestSeg from memory capacity pressure).

Sharing RestSeg Across Processes. A RestSeg can store pages from different processes since the translation structures of a RestSeg are stored per process. Sharing a RestSeg across multiple processes is useful in scenarios where multiple processes benefit from the fast address translation of a RestSeg. At the same time, sharing a RestSeg across multiple processes can lead to efficient memory utilization when the processes have different memory footprints and the RestSeg is not fully utilized by a single process.

5.1.2 Translation Structures of RestSeg

To locate a page in a RestSeg, we introduce two new translation structures: the Tag Array (TAR) and the Set filter (SF). Figure 11 shows the TAR and SF for an example 4-entry 2-way associative RestSeg.

![Diagram of Translation Structures of RestSeg](image)

**Translation Structures of RestSeg**

**Tag Array (TAR)**: TAR stores the tags of every way of every set of the RestSeg. Each tag consists of the virtual page tag and 10 extra bits for various metadata (e.g., access permissions). In the example of Fig. 11, TAR stores the tags and metadata of the ways 0 and 1 of set 0 since only these two ways of the RestSeg are occupied. Each virtual page tag requires 48 - \( \log_2(4KB) \times \log_2(2) \) = 48 - 12 - 1 = 35 bits and the total size of the TAR is \( (4 \times (35 + 10))/8 = 180 \) bits.

**Set Filter (SF)**: SF is used to quickly discover if a set of the RestSeg is empty (i.e., all ways are empty) or not. SF stores an array of sets counters of length \( \log_2(\text{assoc}) + 1 \) that keep track of the cardinality of every set of the RestSeg. Each counter gets incremented/decremented when a new page of a process is added/removed from the corresponding set. In the example of Fig. 11, SF stores two 2-bit counters, one for set 0 and one for set 1. Thus, the total size of the SF is \((16KB/4KB/2)/2 \times (\log_2(2) + 1) = 4 \) bits. The counter of set 0 is equal to 11 since two pages are stored in set 0. The counter of set 1 is equal to 00 since no pages are stored in set 1.

**Scalability of TAR/SF**. Figure 13 shows how the size of TAR/SF scales compared to the radix-based page table, across fully-allocated physical memory segments (i.e., all physical pages are occupied) of increasing sizes. We observe that for the largest allocated memory size (256GB) TAR and SF consume 81% less memory than the radix-based page table. We conclude that Utopia’s new translation structures (TAR and SF) scale efficiently as the size of allocated memory increases.

**Storing TAR/SF**: TAR and SF are stored in kernel memory per process to provide process isolation guarantees (i.e., a process cannot access the TAR/SF of another process). The OS employs a global TAR (stored in kernel memory) to maintain a global view of pages that reside in a RestSeg across all processes. The OS uses the global TAR to discover and allocate free pages in a RestSeg when needed.

5.2 Address Translation for Data in RestSeg

RestSeg uses TAR and SF to discover the physical location of a virtual page inside the physical memory space. We call this process RestSeg Walk (RSW). RSW consists of two operations: (i) tag matching and (ii) set filtering. Figure 14 shows the operations of RSW in a system that employs a 4-entry 2-way associative RestSeg that stores 4KB pages.

![Diagram of RestSeg Walk](image)

**Figure 14: RestSeg Walk: Address translation in RestSeg**

**Tag Matching**: To perform tag matching for a virtual page, a hash function is applied to the virtual page number (VPN) to retrieve the set index and the virtual page tag (virtual page tag is equal to VPNi and set index is equal to 0 in Fig. 14). TAR is looked up to compare the virtual page tag with the tags of all ways of the corresponding set. If the virtual page tag matches with the tag of way i, the virtual page resides in way i (\( \text{tar}[0] = \text{VPN}_i \) and \( i = 0 \)). The physical address is computed directly as \( \text{RestSegBaseRegister} + \text{set index} \times \text{associativity} + i \text{ where the RestSeg Base Register points to the beginning of the RestSeg in the physical address space (VPN is stored in physical page 0 in physical address 0x0FFF)). If the virtual page tag does not match with any of the tags (stored in any of the ways) in the set, the virtual page does not reside in the RestSeg.

**Set Filtering**: For every tag matching operation, all the tags of the set need to be looked up even if they are invalid (i.e., the set is empty).

---

8The general formula to compute the size of the SF is: \( \frac{\text{RestSegSize}}{\text{PageSize} \times \text{Associativity}} \times (\log_2(\text{PageSize} \times \text{Associativity}) + 1) \).

9The general formula to compute the size of the TAR is the following: \( \left( \frac{\text{RestSegSize}}{\text{PageSize}} \right) \times \left( 48 - \log_2(\text{PageSize}) - \log_2 \left( \frac{\text{RestSegSize}}{\text{PageSize} \times \text{Associativity}} \right) + 10 \right) \).
To avoid looking up the TAR when sets are empty, RSW looks up the SF to quickly discover if a virtual page does not reside in the RestSeg. The SF is indexed using the set index to retrieve the counter of the corresponding set \( set_{index} = 0 \) and \( set_{filter} = 01 \). If the counter is 0, tag matching is skipped since all ways of the set are invalid (i.e., the virtual page does not reside in the RestSeg). If the counter is > 0, the system proceeds to tag matching to identify whether the virtual page is kept in any way of the set (the set contains VMs). SF comes with two key benefits: (i) SF is smaller than TAR and hence enjoys better temporal locality than TAR (ii) SF enables skipping tag matching for sets that are empty, thereby avoiding expensive lookups to TAR.

RSW provides two key benefits over the conventional four-level PTW. First, RSW generates only two parallel memory accesses: one for SF and one for TAR. In contrast, a conventional PTW always requires four sequential memory accesses. Second, as we show in §8, TAR and SF take better advantage of hardware caching as they are smaller and exhibit high spatial/temporal locality compared to the PT.

5.3 Segment with Flexible Address Mapping

The structure of FlexSegs is similar to that of the conventional flexible segments used in modern VM designs (which we discussed in detail in §2). FlexSeg uses a fully-flexible virtual-to-physical address mapping: a virtual page can map to any physical page. We call the process of looking up the PT to perform address translation FlexSeg Walk (FSW) (same as PTW in modern VM).

For each process in the system, the OS maintains a single PT that stores the virtual-to-physical mappings of all the pages of the process, regardless of the number of FlexSegs that the process uses.

5.4 Address Translation Flow in Utopia

Figure 15 shows a high level description of how address translation is performed in a system that employs Utopia with two RestSegs, one that stores 4KB pages and one that stores 2MB pages, and one FlexSeg. On an L1 TLB miss, the system in parallel (i) performs two RestSeg walks, one for each RestSeg (i.e., one for the RestSeg that stores 4KB pages 1 and one for the RestSeg that stores 2MB pages 2) and (ii) probes the L2 TLB 3. If the physical address is found during either RSWs 4 or in L2 TLB 5, the translation request is resolved without performing an FSW. If not, an FSW is performed 6 to discover the physical address 7.

![Figure 15: Address translation flow in Utopia.](image)

5.5 Data Placement in a RestSeg

We design Utopia to place pages that experience high address translation latencies inside a RestSeg. To achieve that, Utopia uses two application-transparent techniques to allocate pages into a RestSeg:

(i) a Page-Fault-based technique that directly allocates pages inside a RestSeg and (ii) a PTW-Tracking-based technique that migrates pages to a RestSeg based on the PTW cost and frequency of each page.

5.6 Operating System Support for Utopia

Creation of RestSeg. The OS creates the RestSegs during boot time. Doing so avoids the runtime overheads of compacting memory during runtime in order to create contiguous physical memory segments.

Data Allocation in a RestSeg. The OS directly allocates a page in a RestSeg when servicing a page-fault (PF) for that page (as we describe in §5.5). During the page fault, a hardware interrupt hands the control to the OS. The OS computes the set index of the page in the RestSeg by applying a hash function to the virtual page number (VPN). Using the set index, the OS accesses the global TAR to search for a free way in the set. If the set has a free way, the OS places the page in the set and updates the TAR and SF of the process.

Eviction of a Page from RestSeg to FlexSeg. During a page allocation in a RestSeg, if the corresponding set has no free ways, the OS handles the conflict and evicts a page from the set. The OS employs a replacement policy (we use SRRIP [70] in our evaluation) to decide which page to evict from the set. The OS triggers a page migration to move the evicted page from the RestSeg to a FlexSeg. When the migration is complete, the OS updates the translation structures of the RestSeg and the global TAR.

Migration of a Page from FlexSeg to RestSeg. When the PTW-Tracking migration policy discovers a costly-to-translate page (as we describe in §5.5), the MMU sends an asynchronous interrupt to the OS, so that the OS migrates the page into the RestSeg without stopping program execution. If there is free space in the corresponding set of the RestSeg, a single migration is performed, from the FlexSeg to the RestSeg. If there is no free space in the corresponding set of the RestSeg, the OS performs (i) the migration of the

---

11We use the term FSW to clearly distinguish the process of address translation in a FlexSeg from the process of address translation in a RestSeg.
12The system does not use a separate PT per FlexSeg since the virtual-to-physical mappings of all FlexSegs can be stored in the same PT.
costly-to-translate page from the FlexSeg to the RestSeg and (ii) the migration of the evicted page from the RestSeg to the FlexSeg.

**Performing a Page Migration.** The OS takes four steps to migrate a page to/from a RestSeg to/from a FlexSeg. Figure 16 shows the migration process of a page from a RestSeg to a FlexSeg and vice versa. First, the OS performs a TLB shootdown (and a TAR/SF cache shootdown, as described in §5.7) 1 to maintain coherence and locks the corresponding entries of the translation tables 2 to make sure the program cannot access the translation tables while the migration is happening. Second, the OS flushes all the dirty cache lines of the migrated page from the cache hierarchy 3 to ensure that the data in the main memory is not stale. Third, the OS copies the page to the destination memory region using the Direct Memory Access (DMA) engine to avoid stalling the CPU during the migration process 4. 13 Fourth, the OS updates all the translation tables, PT, TAR and SF and “unlocks” them so that the application can fetch the corresponding cache lines from the cache hierarchy 3. While the migration happens, the cache lines of the migrated page cannot be accessed by any running application.

![Figure 16: Steps followed during a page migration.](image)

5.7 Architectural Support in MMU for Utopia
The MMU is extended with a hardware-based RestSeg walker to accelerate address translation for data in RestSegs. RestSeg walker consists of two components: (i) a new hardware FSM that can access the TAR and SF of the RestSegs and (ii) two (2x2KB) SRAM caches, the TAR and SF cache, that store recently-accessed TAR and SF entries. The address translation flow of the new MMU is shown in Figure 17. We describe how the MMU performs address translation in every possible scenario: (i) the address mapping is cached in the TLB hierarchy (TLB hit), (ii) data is stored in a RestSeg and the physical address is determined by the RestSeg walker (RestSeg Walk) and (iii) data is stored in a FlexSeg, the address mapping is not cached in the TLB hierarchy and the physical address is determined by the FlexSeg walker (FlexSeg walk).

![Figure 17: MMU extensions and operations to support Utopia.](image)

6 System Integration
6.1 Context Switches in Utopia
The MMU accesses the translation structures of a RestSeg (TAR and SF) of a process using specialized registers that store the base address of each data structure (one for TAR and one for SF), similarly to how the CR3 register works in the x86-64 ISA. After a context switch, the TAR and SF registers are reloaded by the OS to point to the translation structures of the new process. The contents of the TAR and SF caches do not need to be flushed during a context switch since both caches operate using physical addresses.

In conventional systems, whenever a virtual-to-physical mapping gets modified (e.g., due to a page migration or a page deallocation), all the affected TLB entries of all the running processes are invalidated to maintain TLBs coherent. When Utopia triggers a page migration to/from a RestSeg from/to a FlexSeg, the OS gets invoked and sends an INVLPG instruction [76] to the MMU. The corresponding TAR/SF cache entries are invalidated to maintain TAR/SF coherence, in the same way as TLB entries are invalidated in conventional systems. To invalidate the TAR/SF entries that correspond to the modified virtual-to-physical mapping, MMU computes the indices to the TAR/SF data structures by applying a hash function on the virtual address of the modified page. The TAR/SF caches are probed using the calculated indices and the corresponding entries get invalidated.

6.2 Area & Power Overhead
Utopia extends the MMU with two 2KB SRAM structures, the TAR cache and SF cache (§5.7). We measure the area and power overhead of extending the MMU to support Utopia using the 45-nm library of
We evaluate Utopia using the Sniper Simulator [51]. This simulator is freely available at https://github.com/CMU-SAFARI/Utopia. We extend Sniper to accurately model: (i) TLBs that support multiple page sizes, (ii) the conventional radix page table walk, (iii) page walk caches, (iv) memory management (e.g., allocation using the buddy allocator [78]), (v) page migration latency, and (vi) the functionality and timing of all the evaluated systems. Table 1 shows the simulation configuration of (i) the baseline system and (ii) all evaluated systems.

### Evaluation Methodology

We evaluate Utopia using the seven workwalk caches, (ii) memory management (e.g., allocation using the buddy allocator [78]), (iii) page migration latency, and (iv) the functionality and timing of all the evaluated systems. Table 1 shows the configuration of (i) the baseline system and (ii) all evaluated systems.

| Table 1: Simulation Configuration and Simulated Systems |
|--------------------------------------------------------|
| **Baseline System**                                    |
| Core                                                   |
| L1 TLB: 128-entry, 8-way associ, 1-cycle latency       |
| MMU                                                    |
| L1 TLB: 4KB (K): 64-entry, 4-way associ, 1-cycle latency|
| L1 TLB: 2MB (MB): 32-entry, 4-way associ, 1-cycle latency|
| L2 TLB: 1536-entry, 12-way associ, 12-cycle latency     |
| L1 Cache                                               |
| L1 Cache: 32 KB, 8-way associ, 4-cycle access latency   |
| L1 D-Cache: 32 KB, 8-way associ, 4-cycle access latency  |
| LRU replacement policy; IP-stride prefetcher [79]       |
| L2 Cache                                               |
| 2 MB, 16-way associ, 16-cycle latency                   |
| L3 Cache                                               |
| 2 MB/core, 16-way associ, 35-cycle latency              |
| Main Memory                                            |
| 32 GB, DDR4-3200-fgJ-gp=12.5ns, fgP=2.5ns               |
| Migration Latency: 2 DRAM Full row reads/writes         |
| Transparent Huge Pages (THP) [45, 46]                   |
| Debian 9.14.2 10-node cluster                          |
| Memory per node: 256GB-1TB                             |
| Evaluated Systems                                      |
| POM-TLB [81]                                           |
| 64KB entry L3 software-managed TLB, 16-way associ       |
| RMM [57]                                               |
| 32-entry Range Lookaside Buffer, Eager paging allocator |
| Elastic Cuckoo Hash Table (ECH) [44]                   |
| 8192-entries/way, 4-way, Scaling: 10, Occupancy: 0.6,   |
| Hash function: CTTY [82] 2-cycle lat.                  |
| Transparent Cuckoo Walk Caches, 2-cycle latency         |
| Utopia                                                 |
| TAR Cache: 2KB, 2-cycle latency, SF Cache: 2KB, 2-cycle latency |
| Hash function: modulo hash                             |
| Perfect TLB (P-TLB)                                    |
| Translation requests always hit in the L1 TLB           |

**Workloads.** Table 2 shows all the benchmarks we use to evaluate Utopia and the systems we compare Utopia to. We select applications with high L2 TLB MPKI (> 5), which are also used in previous works [44, 66, 83, 84]. We evaluate our design using seven workloads from the GraphBig [53] suite, XSBench [55], the Random access workload from the GUPS suite [54], Sparse Length Sum from DLRM [39] and kmer-count from GenomicsBench [56]. We extract the page size information for each workload from a real system that uses Transparent Huge Pages [45, 46] with both 4KB and 2MB pages. The fraction of 2MB pages is shown in Table 2. We create five mixes of 2, 4, and 8 benchmarks to evaluate multi-programmed workloads. Each benchmark is executed for 500M instructions.

**Evaluated Systems.** Table 1 shows the configuration of the simulated systems. We evaluate five different systems: (i) **Radix:** baseline x86-64 system that uses (1) the conventional fully-flexible address mapping and (2) a conventional four-level radix-page table. (ii) **POM-TLB:** a system that employs a large 64K-entry software L3 TLB [81] to increase TLB reach and reduce the number of PTWs. (iii) **ECH:** a system that uses (1) the conventional fully-flexible address mapping and (2) the state-of-the-art hash-based page table, Elastic Cuckoo Hash Table [44] (ECH). ECH employs n different hash tables and issues n memory requests in parallel to each one of the hash tables to increase parallelism and reduce PTW latency. ECH makes use of additional Cuckoo Walk Caches to avoid looking up all hash tables. We implement an optimistic version of ECH that (1) does not require allocating large contiguous physical memory blocks to store the page table and (2) performs migrations between page tables without any performance penalty. Thus, we provide and upper bound estimate of ECH’s performance as described in [85]. (iv) **RMM:** a system that uses multiple dynamically-allocated contiguous physical regions, called ranges [57], to provide efficient address translation for a small number of large memory objects used by the application. RMM uses a hardware Range Lookaside Buffer (RLB) to cache the mappings of the ranges and allocates ranges using a custom memory allocator [57]. (v) **Utopia:** a system that employs Utopia. Utopia uses both the page-fault-based allocation policy and the PTW-Tracking-based migration policy (§5.5). Utopia employs (i) two 512MB RestSegs, one for storing 4KB pages and one for storing 2MB pages and (ii) the rest of memory is organized as a FlexSeg. (vi) **Perfect TLB:** a system where every address translation requests hits in a perfect L1 TLB (P-TLB). P-TLB provides an upper bound estimate of the performance gains possible by accelerating address translation.

We provide additional 2MB pages (1GB in total) to Radix, ECH, and POM-TLB, to match the size of the contiguous allocated RestSegs and conduct a fair comparison against Utopia. For RMM, we provide an additional 1GB contiguous physical memory block. In all evaluated systems, the L2 TLB access is performed in parallel with the PTW to conduct a fair comparison against Utopia.

### Evaluation Results

**8.1 Single-Core Results**

Figure 18 shows the execution time speedup of POM-TLB, ECH, RMM, Utopia and P-TLB compared to Radix, in the single-core
configuration, across 11 workloads. We make two key observations. First, Utopia on average outperforms Radix, POM-TLB, ECH, and RMM by 24%, 21%, 16%, and 11%, respectively. Second, Utopia achieves 95% of the performance of the Perfect-TLB. To better understand the performance speedup achieved by Utopia, we next examine the impact of Utopia on (i) address translation latency and (ii) translation-induced interference in main memory.

Translation Latency. Figure 19 shows the reduction in address translation latency provided by POM-TLB, ECH, RMM and Utopia over Radix. We observe that Utopia significantly reduces address translation latency by 63%, 47%, 29% and 14% compared to Radix, POM-TLB, ECH, and RMM, respectively. This is because (i) RSWs are on average 7.6× faster than PTWs and (ii) Utopia reduces the number of PTWs by 78% over Radix, on average across all workloads.

To better understand the sources of Utopia’s address translation efficiency, Figure 20 shows the breakdown of the servicing location (L2, LLC or DRAM) of memory requests issued by the MMU to access the translation structures of POM-TLB, ECH, RMM and Utopia, normalized to Radix. We make three key observations. First, Utopia issues on average 88% fewer memory requests to the memory hierarchy compared to Radix. Second, Utopia reduces the number of memory requests sent to DRAM, by 78%, 82%, 86% and 40% compared to Radix, POM-TLB, ECH and RMM, respectively. Third, 51% of memory requests issued by Utopia (during a FSW or RSW) hit the L2 cache. We conclude that Utopia significantly reduces the number of memory requests issued to the memory hierarchy to access translation structures, which leads to a significant reduction in address translation latency.

Memory Interference. Figure 21 shows the reduction of DRAM row buffer conflicts (RBC) for POM-TLB, ECH, RMM, Utopia and P-TLB normalized to Radix. We make two key observations. First, due to the reduced number of DRAM row activations to fetch translation metadata (Fig. 20), Utopia reduces the total number of DRAM RBCs (i.e., considering row activations to access both application data and translation structures) by 20%, 15%, 70% and 4% over Radix, POM-TLB, ECH and RMM, respectively. Second, Utopia causes only 9% more DRAM RBCs compared to Perfect-TLB. We conclude that Utopia significantly reduces translation-induced interference in the main memory.

8.2 Multi-Programmed Results

We evaluate Utopia in 2-, 4- and 8-core systems using multi-programmed workloads to demonstrate its effectiveness in systems where multiple applications compete for memory space in the same RestSeg. Figure 22 shows the average (of 5 mixes) performance speedup of ECH, POM-TLB, RMM, Utopia and P-TLB compared to Radix. First, we observe that Utopia outperforms Radix, POM-TLB, ECH and RMM across all multi-core systems. In the 8-core system, Utopia outperforms the second-best performing mechanism (RMM) by 5% and achieves 91% of the performance of P-TLB. We conclude that Utopia provides high performance benefits, even when multiple applications compete for memory space in the same RestSeg.

8.3 Analysis of Utopia

8.3.1 Effectiveness of TAR and SF Caches. To better understand why RSWs are more efficient than PTWs, we examine the effectiveness of the TAR/SF caches. Figure 23 demonstrates the hit rate of the TAR cache and the SF cache. We observe that 81% of TAR requests hit in the TAR cache while 98% of SF requests hit in the SF cache. This is because TAR/SF entries experience high reuse (both data structures consume 545KB in total for a 512MB RestSeg). TAR cache has a lower hit rate compared to SF cache because TAR is 31x larger than SF (i.e., 528KB vs 17KB) and the 2KB TAR cache cannot cover as many TAR entries as the SF cache does. We conclude that TAR/SF caches are highly effective and lead to low-latency RSWs.
8.3.2 Effect of Utopia’s Page Migrations. To better understand how page migration affects memory requests (§5.6), we plot (in Figure 24) the fraction of memory requests that get stalled by page migrations across RestSegs with different sizes. We observe that less than 0.001% of the memory requests are affected due to migration, even for the smallest RestSeg (i.e., 1MB). This is due to two reasons. First, the number of page migrations is low (i.e., 0.8 migrations per kilo instructions on average). Second, 82% of the page migrations occur due to evictions from a RestSeg to a FlexSeg. As we show in Table 1, Utopia chooses pages with low reuse for eviction (and eventually migration) candidates (i.e., SRRIP replacement policy [70]). Thus, the probability of accessing a migrated page is low, especially during the migration process. We conclude that Utopia’s page migrations minimally interfere with regular memory requests.

Figure 24: Fraction of memory requests that get stalled due to migrations, across RestSegs with different sizes.

8.3.3 Utopia’s Effectiveness in Discovering Costly-to-Translate Pages. To understand the effectiveness of Utopia in discovering costly-to-translate pages, Figure 25 plots the fraction of pages that experience different PTW latencies for four workloads in the baseline system Radix: (i) less than 300 cycles, (ii) 300-500 cycles, (iii) 500-1500, (iv) more than 1500, and (v) more than 3000. We observe that more than 50% of pages experience latencies between 500 and 3000, while more than 25% of pages experience latencies larger than 3000 cycles. The PTW-Tracking-based migration policy (§5.5) identifies and migrates to RestSeg costly-to-translate pages (>500-cycle PTW latency) with 82.9% accuracy. We conclude that the evaluated workloads have a significant number of costly-to-translate pages which Utopia can effectively identify and migrate to a RestSeg and thus reduce the address translation latency.

Figure 25: Distribution of PTW latencies across pages in Radix.

8.3.4 Effectiveness of RestSeg’s Replacement Policy. Figure 26 shows the reuse-level distribution of (4KB) pages while they reside in the RestSeg. This reuse level corresponds to how many times RSWs resolve address translation before evicting the page from the RestSeg. We make two key observations. First, nearly 0% of the pages in the RestSeg are evicted without being reused, which indicates that Utopia saves at least one PTW for almost every page that is allocated or migrated to a RestSeg. Second, we observe that more than 50% of pages experience reuse larger than 5 and 27% of the pages experience reuse larger than 20 before getting evicted from the RestSeg. This relatively high reuse is due to two reasons. First, the PTW-Tracking-based migration policy (§5.5) migrates pages that experience highly-frequent PTWs to a RestSeg, which converts the slow PTWs to fast RSWs. Second, the SRRIP replacement policy [70] (employed by the RestSeg) effectively estimates the re-reference interval (analogous to reuse) of pages and evicts pages with low reuse. We conclude that Utopia using (i) the PTW-Tracking-based migration policy and (ii) the SRRIP replacement policy in the RestSeg, retains pages with high reuse in the RestSeg which leads to (i) fast address translation for pages that otherwise experience frequent PTWs in the baseline system and (ii) efficient utilization of the RestSeg.

Figure 26: Reuse-level distribution of (4KB) pages that reside in the RestSeg.

8.3.5 Sensitivity to RestSeg Size. Fig. 27 shows the execution time speedup provided by Utopia over Radix across different RestSeg sizes. We make three key observations. First, larger RestSegs lead to higher performance benefits, up to 27% for the 2GB RestSeg. Second, the performance of the 512MB RestSeg is within 1.3% of the performance achieved by the 2GB RestSeg. Third, the 1MB RestSeg achieves the same performance as the baseline configuration, since it does not provide enough space to store costly-to-translate pages. We choose the 512MB RestSeg in our evaluation setup since it delivers similar performance gains to the 2GB RestSeg with 4x lower memory consumption.

Figure 27: Speedup provided by Utopia over Radix, across different RestSeg sizes.

8.3.6 Effect of Performing RSW in Parallel to L2 TLB Access. To understand the benefits of performing the RSW in parallel with the L2 TLB access, Figure 28 shows the execution time speedup of: 1) Radix that performs the PTW in parallel with the L2 TLB access (Radix-Parallel) and 2) Utopia that performs the RSW in serial/parallel with the L2 TLB access (Utopia-Serial/Utopia-Parallel), all normalized to the performance of Radix that performs the PTW in serial with the L2 TLB access (Radix-Serial). We make two key observations. First, Utopia-Serial outperforms Radix-Serial by 21% on average across all workloads. Second, Utopia-Parallel outperforms Utopia-Serial by 3% on average. We conclude that Utopia (i) outperforms Radix in both serial and parallel L2 TLB/RSW configurations and (ii) parallelizing the RSW with the L2 TLB access provides a noticeable performance benefit compared to serializing the RSW with the L2 TLB access.

8.3.7 Effect of Utopia on Non-Translation-bound Workloads. To understand the performance impact of Utopia on workloads that do not experience high performance overheads due to address translation, we evaluate Utopia and Radix using eight workloads...
from the SPEC CPU2017 benchmark suite [86] that exhibit low (i.e., smaller than 2) L2 TLB MPKI. Figure 29 shows the performance loss of Utopia compared to Radix. We observe that Utopia causes less than 0.05% average performance loss. This very low overhead is due to two factors: (i) non-translation-bound workloads experience high L1/L2 TLB hit rates (more than 95% on average across all workloads) and Utopia does not affect the L1/L2 TLB hit rate or latency and (ii) Utopia does not trigger many migrations from a FlexSeg to a RestSeg (i.e., most of the pages are correctly estimated to be not costly-to-translate) and thus avoids evicting high-locality data from the cache hierarchy. We conclude that Utopia has negligible performance impact to applications that do not experience high translation overheads.

Figure 29: Performance overhead of Utopia compared to Radix in non-translation-intensive workloads.

8.3.8 Sensitivity to RestSeg Address Mapping Function. To understand the impact of the address mapping function that determines the location of a page inside a RestSeg, Figure 30 plots the performance of Utopia using four different hash functions: (i) modulo hashing (MOD), (ii) prime displacement hashing [87], (iii) XOR-based hashing [88] and (iv) Mersenne modulo hashing [89]. We observe that the modulo hash function performs similarly to more sophisticated hash functions while requiring minimal hardware support. As such, the modulo hash function provides the best performance/complexity trade-off and we use it in our evaluation setup.

Figure 30: Speedup provided by Utopia with different RestSeg address mapping functions.

8.3.9 Sensitivity to Context Switches. To evaluate the impact of context switches on Utopia’s performance, we evaluate Utopia and Radix using different context switch quanta (CSQ) while multiple workloads execute in a single core in a round-robin manner. Each workload is executed for CSQ time units before the OS performs a context switch to the next workload.

Figure 31 shows the performance of Utopia and Radix using 5 different CSQs, ranging from 20ms to 100ms, across 5 mixes of 5 different workloads. We make two observations. First, Utopia provides on average 24.9% higher performance compared to the baseline system for the smallest CSQ (i.e., 20 ms), across all 5 workload mixes. Second, we observe that increasing the CSQ does not affect the performance of Utopia. This is because Utopia does not affect context switch overhead and Utopia’s operations are not affected by the CSQ.

Figure 31: Speedup provided by Utopia over Radix while executing multiple applications in a single core for different context switch quanta values.

9 Related Work

To our knowledge, Utopia is the first hybrid virtual-to-physical address mapping technique that enables both flexible and restrictive hash-based virtual-to-physical address mapping schemes to harmoniously co-exist in the system. We already comprehensively compared Utopia to systems that employ (i) large software-managed TLBs [81], (ii) state-of-the-art page table designs [44] and (iii) contiguity-aware translation mechanisms [57] in §8.1. In this section, we qualitatively compare Utopia to other related prior works that propose solutions to reduce address translation overheads.

Hash-based Address Mapping. Multiple works [15, 49, 50] leverage the concept of hash-based address mapping to accelerate address translation. Picorel et al. [49] and Gosakan et al. [50] propose employing a restrictive hash-based virtual-to-physical address mapping across the entire main memory to reduce the overheads of address translation. Although such techniques drastically reduce address translation overheads, they generally handicap core VM features such as (i) sharing pages and (ii) the flexibility of allocating pages in free memory space to avoid swapping. For example, as we show in §3, the mapping scheme proposed by Picorel et al. [49] leads to a large increase (2.2× on average) in accesses to the swap space over the baseline system. In contrast, Utopia accelerates address translation while supporting all the key features enabled by the conventional virtual memory framework.

Efficient TLBs and Page Walk Caches (PWCs). Many prior works focus on reducing address translation overheads through efficient TLB and PWC designs [16, 27, 90–107]. Such techniques include: (i) prefetching TLB and page table entries [99–104], (ii) TLB-specific replacement policies [90, 108], (iii) employing software-managed TLBs [14, 81, 104, 109–114], (iv) sharing TLBs across cores [106, 107, 115], (v) employing efficient PWCs [27, 105, 116], and (vi) PT-aware cache management [66–68] (e.g., specialized cache replacement policies for PTEs [67]). Although such techniques may offer notable performance improvements, their effectiveness reduces as the page table size increases. This is because they rely on (i) the existing scarce TLB resources, which are unable to accommodate the very large number of virtual-to-physical mappings...
required by data-intensive applications or (ii) new hardware/software translation structures that pose a significant trade-off between performance and area/power/energy efficiency. In contrast, Utopia fundamentally reduces the size of the address translation structures to enable efficient address translation. However, we believe that Utopia can be used in combination with techniques that improve the efficiency of the TLB hierarchy and PWCs, to further reduce address translation overheads. For example, caching TLB entries in the L2 cache [117], prefetching TLB and PT entries [99–104], and employing efficient PWCs [27, 105, 116] can significantly reduce the L2 TLB miss latency (when the needed data does not reside in a RestSeg) and further accelerate address translation in Utopia.

Alternative Page Table Designs. Various prior works propose (i) hash-based and (ii) flattened PT designs to reduce PTW latency [33, 44, 49, 50, 66, 83, 85, 118–121]. For example, Skarlatos et al. [44] and Stojkovic et al. [85] propose replacing the radix-tree-based page table with a Cuckoo hash table [122] to parallelize accesses to the PT and reduce PTW latency. Park et al. [66] propose a flat PT design in combination with a PT-aware replacement policy to reduce PTW latency. In §8, we show that Utopia significantly outperforms a standalone version of ECH (i.e., in a system that employs the fully-flexible virtual-to-physical address mapping). We believe that Utopia can be used in combination with alternative PT designs to further reduce the latency of FSWs (flexible segment walks) and further accelerate address translation.

Employing Large Pages. Many works propose hardware and software mechanisms for efficient support for pages of varying sizes [63, 64, 123–136]. For example, Panwar et al.[64, 127] propose new OS techniques to improve the efficiency of mechanisms that enable application-transparent allocation of large pages. As we discuss in §5.1, Utopia is backward compatible with large page mechanisms and the OS can create multiple RestSegs to accommodate pages of different sizes.

Contiguity-Aware Address Translation. Many prior works enable and exploit virtual-to-physical address contiguity to perform low-latency address translation [25, 29, 57, 137–141]. For example, in [25], the authors propose pre-allocating arbitrarily-large contiguous physical regions (10-100’s of GBs) to drastically increase the translation reach for specific data structures of the application. Karakostas et al. [57] propose the use of multiple dynamically-allocated contiguous physical regions, called ranges, to provide efficient address translation for a small number of large memory objects used by the application. Although these works can significantly increase translation reach and reduce address translation overheads, their effectiveness heavily depends on the availability of free contiguous memory blocks. Utopia allocates RestSegs during system boot time to avoid the need to find or create free contiguous memory blocks during runtime. In §8, we show that, given the same amount of available contiguity in the system, Utopia outperforms (by 11%) a system that employs multiple segments with contiguous virtual-to-physical address mappings [57]. We believe that Utopia can be naturally extended to incorporate segments with contiguous virtual-to-physical address mappings to further reduce address translation overheads.

Address Translation in Virtualized Environments. Various works propose techniques to reduce address translation overheads in virtualized environments [128, 142–146, 146–149]. For example, Ghandi et al. [142] propose a hybrid address translation design for virtualized environments that combines shadow paging and nested paging. Utopia can be employed to support and accelerate address translation in virtualized environments by placing costly-to-translate guest-virtual as well as host-virtual pages in RestSegs. We leave the evaluation of Utopia in virtualized environments as future work.

Virtual Caching & Intermediate Address Spaces. Another class of works focuses on delaying address translation by using techniques such as virtual caching [16, 150–153, 157–159] and intermediate address spaces [1, 84, 154, 155]. Virtually-indexed caches reduce address translation overheads by performing address translation only after a memory request misses in the LLC [16, 152, 153, 156]. Gupta et al. [84] propose mapping large virtual memory regions to an intermediate address space to enable fast virtual-to-intermediate address translation and delay intermediate-to-physical address translation until an LLC miss (for the corresponding data access). Hajinazar et al. [155] propose the use of virtual blocks to enable fast virtual-to-intermediate address translation and extend the memory controller with a programmable core that handles (i) data allocation in physical memory and (ii) intermediate-to-physical address translation, based on properties of each virtual block. Utopia is orthogonal to these techniques and can be used to accelerate address translation of costly-to-translate pages that reside either in the intermediate address space or in the virtual address space, regardless of when the address translation takes place (e.g., after an LLC miss or before the L1 cache access).

10 Conclusion

We propose Utopia, the first hybrid address mapping technique that allows both flexible and restrictive address mapping schemes to harmoniously co-exist in a system with virtual memory. By restricting the virtual-to-physical address mapping, Utopia alleviates the need to store a large number of virtual-to-physical mappings and enables faster address translation via compact translation structures. At the same time, Utopia retains the ability to use the flexible address mapping to support conventional virtual memory features. Our extensive evaluations using data-intensive workloads show that Utopia leads to fast and efficient address translation, improving application performance in both single-core and multi-core systems. We believe that Utopia is also applicable to virtualized environments, address translation in GPUs and specialized accelerators, and address translation for I/O data. To enable further research in these and other directions, we open source Utopia at https://github.com/CMU-SAFARI/Utopia.

Acknowledgments

We thank Jisung Park and Nika Mansouri-Ghiasi for their valuable feedback on this work. We thank Konstantinos Sigouras for his valuable help with the development of the simulation infrastructure. We thank the anonymous reviewers of ISCA 2022, MICRO 2022, ISCA 2023, and MICRO 2023 for their feedback. We thank the SAFARI Research Group members for providing a stimulating intellectual environment. We acknowledge the generous gifts from our industrial partners: Google, Huawei, Intel, Microsoft, and VMware. This work is supported in part by the Semiconductor Research Corporation and the ETH Future Computing Laboratory.
Mohammad Agbarya, Idan Yaniv, Jayneel Gandhi, and Dan Tsafrir. Predicting Execution Times with Partial Simulations in Virtual Memory Research: Why and How. In MICRO 2020.

Narayanan Ganapathy and Curt Schimmel. General Purpose Operating System Support for Multiple Page Sizes. In ATC 1998.

Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In ISCA 2019.

Chloe Alverti, Stratos Psmoulakis, Vassilios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In ISCA 2020.

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.

Dongwei Chen, Dong Tong, Chunj Yang, Jiangfang Yi, and Xu Cheng. FlexPointer: Fast Address Translation Based On Range TLB and Tagged Pointers. In TACO 2023.

Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In MICRO 2012.

Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In ISCA 2016.

Binh Pham, Jan Vesely, Gabriel H Loh, and Abhishek Bhattacharjee. Using TLB Speculation to Overcome Page Splintering in Virtual Machines. In Technical Report Rutgers Univ. 2015.

Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In MICRO 2014.

Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In ASPLOS 2008.

Zi Yan, Jan Vesely, Guilherme Cox, and Abhishek Bhattacharjee. Hardware Translation Coherence for Virtualized Systems. In ISCA 2017.

Dimitrios Skarlatos, Umut Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Joes Torrellas. Babelfish: Fusing Address Translations for Containers. In ISCA 2020.

Artemiy Margartov, Dmitrii Ustingov, Amna Shahab, and Boris Grot. PTE: Hierarchical Physical Memory Reservation for Faster Page Walks in Public Clouds. In ASPLOS 2021.

Arish Parwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K Gopinath, and Jayneel Gandhi. Fast Local Page-tables for Virtualized Numa Servers with vmiptos. In ASPLOS 2021.

Stefanos Kaxiras and Alberto Ros. A New Perspective for Efficient Virtual-Cache Coherence. In ISCA 2013.

Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. SEESAW: Using Superpages to Improve VIRT Cache. In ISCA 2018.

Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In ISCA 2012.

Michel Cekleov and Michel Dubois. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. In IEEE Micro 1997.

Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In ICS 2010.

Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavou, Vivek Seshadri, and Onur Mutlu. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework. In ISCA 2020.

Michel Cekleov and Michel Dubois. Virtual-Address Caches Part 2: Multiprocessor Issues. In IEEE Micro 1997.

James R. Goodman. Coherency for Multiprocessor Virtual Address Caches. In ASPLOS 1987.

W. H. Wang, J.-L. Baer, and H. M. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In ISCA 1989.

Bob Wheeler and Brian N. Bershad. Consistency Management for Virtually Indexed Caches. In ASPLOS 1992.