Efficient Synonym Filtering and Scalable Delayed Translation for Hybrid Virtual Caching

Chang Hyun Park, Taekyung Heo, Jaehyuk Huh
School of Computing, KAIST
{*changhyunpark, tkheo*}*@calab.kaist.ac.kr, and jhhuh@kaist.ac.kr*

***Abstract*—Conventional translation look-aside buffers (TLBs) are required to complete address translation with short latencies, as the address translation is on the critical path of all memory accesses even for L1 cache hits. Such strict TLB latency restrictions limit the TLB capacity, as the latency increase with large TLBs may lower the overall performance even with potential TLB miss reductions. Furthermore, TLBs consume a significant amount of energy as they are accessed for every instruction fetch and data access. To avoid the latency restriction and reduce the energy consumption, virtual caching techniques have been proposed to defer translation to after L1 cache misses. However, an efficient solution for the synonym problem has been a critical issue hindering the wide adoption of virtual caching.**

Based on the virtual caching concept, this study proposes a hybrid virtual memory architecture extending virtual caching to the entire cache hierarchy, aiming to improve both performance and energy consumption. The hybrid virtual caching uses virtual addresses augmented with address space identifiers (ASID) in the cache hierarchy for common non-synonym addresses. For such non-synonyms, the address translation occurs only after last-level cache (LLC) misses. For uncommon synonym addresses, the addresses are translated to physical addresses with conventional TLBs before L1 cache accesses. To support such hybrid translation, we propose an efficient synonym detection mechanism based on Bloom filters which can identify synonym candidates with few false positives. For large memory applications, delayed translation alone cannot solve the address translation problem, as fixed-granularity delayed TLBs may not scale with the increasing memory requirements. To mitigate the translation scalability problem, this study proposes a delayed many segment translation designed for the hybrid virtual caching. The experimental results show that our approach effectively lowers accesses to the TLBs, leading to significant power savings. In addition, the approach provides performance improvement with scalable delayed translation with variable length segments.

***Keywords*-address translation, hybrid virtual cache, synonym detection, segmented translation**

I. Introduction

With ever growing system memory capacity and increasing application requirements, traditional memory virtualization has become a performance bottleneck for big memory applications [1], [2]. For such applications, conventional translation lookaside buffers (TLBs) can no longer cover large working sets and generate excessive TLB misses. Furthermore, in the traditional memory hierarchy, the address translation through TLBs must be completed before L1 cache tag matching. Since the address translation is on the critical path of memory operations, it is difficult to increase the TLB capacity arbitrarily to cover more memory pages. In addition, the address translation consumes a significant amount of dynamic power for TLB accesses [3], [4].

An alternative approach to reduce the cost of address translation is virtual caching [3], [5], [6], [7], [8], [9], [10]. Virtual caching postpones the address translation until an L1 cache miss occurs. It eliminates the critical translation overhead between the core and L1 cache, reducing the energy consumption for address translation. However, a critical problem of virtual caching is the synonym problem where different virtual addresses can be mapped to the same physical address, allowing multiple copies of the same data in a cache. Prior studies address the synonym problem with solutions tuned for L1 virtual caches [3], [5], [6], [8], [9], [10]. They invalidate synonyms with a reverse translation to identify existing synonyms [3], [5], [6], or use self-invalidation with coherence protocol supports [7]. A software-oriented approach has also been proposed with software-based cache miss handling [8].

Inspired by the prior virtual caching techniques, this paper proposes a new hybrid virtual caching architecture supporting efficient synonym detection and scalable delayed translation. Unlike the prior virtual caching approaches, the proposed hybrid caching extends the scope of virtual addressing to the entire cache hierarchy for non-synonym addresses. Synonym addresses are detected by a synonym filter, and translated to physical addresses with conventional TLBs. Each physical memory block is addressed only by a single name, either virtual or physical address. As non-synonym private pages account for the majority of memory pages, most of the actual address translations are deferred to after last-level cache (LLC) misses. By extending the scope of virtual caching, the proposed hybrid translation aims not only to reduce the TLB access energy, but also to improve the translation performance, since large on-chip caches often contain cachelines which could have missed the conventional TLBs. Figure 1 describes the overall translation architecture of the proposed hybrid virtual caching. For synonym candidates detected by the synonym filter, TLB accesses occur prior to the L1 cache accesses. For non-synonym pages, delayed translation is applied only after

LLC misses.

To allow such a hybrid translation, a key component is the *synonym filter*. Exploiting the fact that synonym pages are uncommon in real systems, the study proposes a synonym filter design based on Bloom filters with a low latency and low power consumption [11]. The operating system updates the bloom filter values, when it changes the page state to shared (synonym). The synonym filter guarantees the detection of synonym pages, while occasionally causing false-positives. With the synonym filter, any page in the virtual address space can be changed to a synonym page without restriction, unlike the prior approach [3].

Delaying translation can reduce TLB access overheads both in energy and latency significantly. However, for applications with large working sets, the delayed translation may not completely eliminate performance degradation due to a large number of delayed TLB misses. To reduce such translation costs, in addition to the traditional page granularity translation which will eventually reach the coverage limit, this paper proposes a many segment translation mechanism. It extends the prior variable-length segment translation [1], [12]. The prior segment translation proposed for physical caches, maps part of the virtual address space to one or more contiguous physical regions with variable length segments for each process. Unlike the prior segment approaches on the critical core-to-L1 path, providing 10s of concurrent segments [12], the proposed delayed segment translation can support 1000s of concurrent segments efficiently, supporting better OS memory allocation flexibility to mitigate internal and external fragmentation problems.

The proposed architecture differs from the prior approaches in four aspects. First, this study extends virtual caching to the entire cache hierarchy. For non-synonym cachelines, cache blocks are indexed and tagged by their address space identifier (ASID) and virtual address. The coherence mechanism also uses virtual addresses with ASID for non-synonym cachelines. Second, this study proposes an efficient HW and SW-combined synonym filtering technique based on Bloom filters. Unlike the prior pure hardware-oriented hash-based synonym detection [13], in this study, the operating system updates the Bloom filters, not requiring any HW synonym tracking. With the hash-based filtering technique, only the synonym candidates access TLBs before L1 accesses. Third, as the fixed page-granularity delayed translation after LLC misses will reach its coverage limit for large memory applications, this study investigates segment-based delayed translation supporting many concurrent segments. Finally, the proposed scheme mitigates the overhead of HW-based virtual machine supports, since the costly two-step translation is delayed until LLC misses. We propose to extend the synonym filter to detect synonym pages induced by both the guest OS and hypervisor.

The experimental results show that the proposed synonym filter is very effective with negligible false positives. The

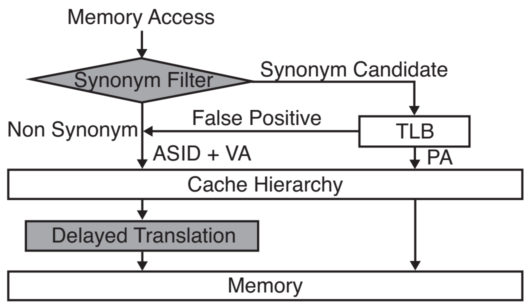

Figure 1. A synonym filter detects synonym candidates and allows non-synonym addresses to bypass translation until LLC miss. The components in gray are newly added by this work.

performance of memory intensive applications is improved by 10.7% compared to the physically addressed baseline system, and the power consumption of the translation components is reduced by 60%. For virtualized systems, the performance gain becomes larger with 31.7%, compared to a system with a state-of-the-art translation cache for two-dimensional address translation.

The rest of this paper is organized as follows. Section II presents virtual caching and its synonym problem with other prior work. Section III presents the proposed hybrid translation and synonym filtering. Section IV describes the segment-based delayed translation. Section V presents how to support system virtualization efficiently. Section VI presents the performance and energy improvements. Section VII concludes the paper.

II. Background & Motivation

A. Virtual Caching

Virtual caching postpones the address translation until a cache miss occurs, allowing caches to store cachelines with virtual address [3], [5], [6], [7], [8], [9], [10], [14]. With such deferred translation, virtual caches can reduce the dynamic power consumption for TLB accesses. A recent study has shown that using virtual L1 caches opportunistically reduces more than 94% of energy consumed by the data TLBs [3]. Although the prior virtual caching studies commonly aim to reduce power consumption for address translation, virtual caching can improve performance too. The cached data do not require address translation and large on-chip caches can potentially have a larger data coverage than the limited TLBs. Such phenomenon was recently measured by Zhang et al. [15], and for several workloads, more than 95% of accesses that cause TLB misses were resident in the caches. For such cases, physically-tagged caches must wait for the TLB miss handling, even if the cachelines are in on-chip caches. Furthermore, by deferring address translation, the capacity of delayed TLBs can be increased as they are no longer restricted by the critical core-to-L1 path latency.

Virtual caching, despite having such desirable properties, has critical problems which have hindered its wide commercial adoption. The main problem is the M:N mapping between the virtual address space and physical address space resulting in two types of problems. The first type of problem, called *homonym*, occurs when two different physical pages map to the same virtual page of different processes. To virtual caches, the virtual page number looks identical, however the underlying physical pages are different. The homonym problem has been fixed by adding the address space identifier (ASID) in each cache tag entry, to identify the cacheline owner for the given virtual page number. The second type of problem occurs when two different virtual addresses map to a single physical address. This problem, called *synonym* or *aliasing*, is the main crippling factor of widespread use of virtual caches. We discuss the synonym problem and prior work in the next section.

B. Prior Work on Synonym Problem

Synonyms can create multiple cachelines with different virtual addresses for the same physical address. The main problem of synonyms is that if a cacheline at one virtual address is updated, the other cachelines pointing to the same physical address need to see the update, otherwise the other cachelines will have stale data. There have been several hardware and/or software approaches to resolve the coherence problem [5], [6], [7], [8]. There are two common components to support coherence for synonyms. First, a *synonym tracking* mechanism finds all cached synonym copies of the same physical block. Second, a *synonym detection* mechanism identifies whether a given address is a synonym address. The synonym tracking can also be used as a detection mechanism. The two mechanisms are used together or in isolation to address the synonym problem.

HW-based synonym tracking: A key component to maintain the coherence of synonyms is to track virtually addressed cachelines that share common physical counterparts. Once the synonyms are identified, they can be invalidated or downgraded by checking all existing copies. To track synonyms, reverse mapping is used either by adding extra address tags in virtual caches [5], or by adding back-pointers in the L2 cache [6]. Goodman proposed to use dual-tag arrays with virtual and physical tags to support coherence through physical addresses for virtual caches [5]. Wang et al. proposed to keep back-pointers in the private physically addressed L2 cache, which track synonyms in the virtual L1 cache [6].

In parallel to our study, Yoon and Sohi proposed to map synonym pages of the same physical address to a leading virtual address. It uses a hash table, much like a Bloom filter used in our work, to identify synonyms and translate to the leading virtual address before accessing the L1 virtual cache [16].

Isolating synonym pages by OS: Instead of detecting synonyms, the OS can isolate synonym pages in certain known virtual addresses. OVC uses virtual address for private data, and physical address for shared data in L1 caches [3]. This approach restricts the OS to use a fixed region of virtual address space for synonym pages. In addition, unlike our approach, OVC limits virtual caching to the L1 cache, as it aims to reduce energy, and it requires reverse translation through extra physical tags in L1s for coherence.

SW-oriented approach: Single address space operating systems use one address space for all processes, and thus the synonym problem does not occur [17]. However, protection across different processes must be supported by an extra protection lookaside buffer. An alternative software-oriented approach is to use virtually addressed caches, but during cache miss handling, the address must be translated to a physical address by a SW-based cache miss handler [8]. Although it simplifies the HW complexity for the synonym problem greatly, a slow SW handler must intervene for cache misses.

Self-invalidation: Kaxiras and Ros proposed to use self-invalidation and downgrade from virtual L1 caches [7]. Using the support from coherence protocols, synonyms in L1 caches are self-invalidated to maintain their coherence. However, a limitation of this approach is that it requires a cache coherence protocol which does not send any coherence request directly to virtual L1 caches. To classify synonym and non-synonym addresses, they use a combined HW/SW approach to mark sharing status in page table entries.

Intermediate address space: Enigma makes use of the intermediate address space, available in the PowerPC architecture, that exists between virtual and physical address spaces[15][1]. Between the core and L1, a virtual address is translated to an intermediate address through a large fixed granularity segment. Memory sharing between processes is achieved by large fixed granularity segments. The cache hierarchy uses the intermediate address space, and the address translation occurs after LLC misses. Synonyms are addressed by the first-level translation with coarse-granularity mapping. The second-level translation after LLC misses uses conventional page-based translation with TLBs. Although Enigma exploits the large on-chip caches to reduce the burden of address translation for performance, its fixed page-based translation between the intermediate and physical address spaces limits translation scalability as shown in our results section.

C. Synonym Pages for Common Workloads

Synonym pages are used infrequently for many common workloads. Table I shows the ratio of r/w shared memory pages out of the total used memory pages, and the ratio of memory accesses to shared regions over all memory accesses. The presented shared memory ratio is an average of the ratios sampled per second. From the entire PARSEC

[1] PowerPC provides three address spaces in the order of logical, virtual, and physical; whilst Enigma renames the virtual address to intermediate address. We name these spaces in the order of virtual, intermediate, and physical for consistency with our work.

Table I
RATIO OF R/W SHARED MEMORY AREA AND ACCESSES TO THE R/W SHARED REGIONS

	Shared area	Shared access
ferret	0.004543%	0.849976%
postgres	66.753033%	15.788043%
SpecJBB	0.328539%	0.023681%
firefox	0.754580%	0.001840%
apache	9.402985%	0.444037%
SPECCPU	0%	0%
Remaining Parsec	0%	0%

suite, only `ferret` uses shared memory regions, but the actual memory area and access frequency are limited. Other than `ferret`, the rest of PARSEC and SPECCPU2006 applications do not exhibit any r/w memory sharing. The proposed scheme treats read only (r/o) shared pages equally to private pages, as r/o shared pages do not cause any coherence problem.

In addition to the SPECCPU and PARSEC applications, we examined four additional applications that have synonym pages. Only `postgres` exhibits a large number of shared memory pages, since it allows multiple processes to share data. However, the other applications have a relatively small amount of memory sharing. Even for `postgres`, the actual memory accesses to the shared region is 16% of the total accesses. In the next section, we will exploit the lack of synonym pages in applications to design an efficient synonym detection mechanism.

A key observation for addressing the synonym problem is that as long as a unique address is consistently used to identify each physical memory block, the coherence problem does not occur. Our hybrid caching uses either virtual or physical address, and guarantees that the same unique address is used for each memory block.

III. EFFICIENT SYNONYM FILTERING

This section describes the proposed hybrid address translation architecture which allows delaying address translation for the majority of memory accesses.

A. Hybrid Address Translation

Cache Tag Extension: The hybrid virtual caching uses address space identifier (ASID) concatenated to virtual address (VA) for non-synonym pages, and physical address for synonym pages. The cache tags are illustrated in Figure 2. The cache tag entry contains a *synonym* bit which distinguishes a synonym (or physical address) from non-synonym (or ASID+VA). ASID bits are added to prevent the homonym problem of non-synonym cachelines. The ASID is configured to 16 bits which allow 65,536 address spaces, and such a large number of address spaces will be required for large systems and virtualized systems. For synonym cachelines, physical addressing is used, ignoring ASID. Note

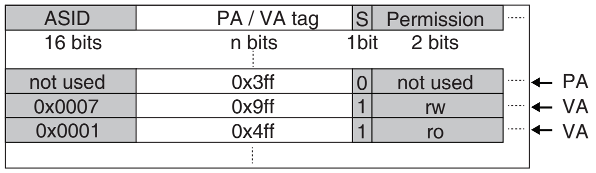

Figure 2. Cache tag extension for ASID and status/permission bits

that the PA/VA tag portion is shared both by synonym and non-synonym cachelines, and the PA/VA tag width is determined by the maximum bits required for either virtual or physical address space. For example, AMD systems use a large 52 bit physical address space, which allows the virtual tag to fit in the physical tag. Permission bits are added for non-synonym cachelines to check access permission. The additional tag bits induce negligible overheads. The results from CACTI [18] estimates 1.7-3.7% static and 0.16-0.76% dynamic power increase for all three levels of the cache.

Address Translation: The first step of the address translation is to access the synonym filter. The synonym filter checks whether the address is a synonym candidate. An important property of the synonym filter is that it must be able to detect all synonym addresses correctly, although it can falsely report a non-synonym address as a synonym page (*false-positive*).

If an address is determined to be a non-synonym address, the ASID and virtual address are concatenated to access the L1 cache (`ASID+VA`). `ASID+VA` is used throughout the entire cache hierarchy for non-synonym addresses. Even the coherence protocol uses the `ASID+VA` address. When the external memory needs to be accessed, the `ASID+VA` is translated to the actual physical address through *delayed translation*. The delayed translation can use conventional TLBs with fixed page sizes, or variable-length segments as we will discuss in Section IV. Since the majority of memory accesses are to non-synonym addresses, the accesses to the synonym filter and the L1 cache with `ASID+VA` can be overlapped, hiding the synonym filter latency.

If an address is predicted to be a synonym address by the filter, a normal TLB access occurs. For a TLB hit, the address is translated to a physical address, if it is truly a synonym. If the detection was false positive, the TLB entry for a non-synonym page will report the false positive status[2]. For the false-positive case, the address is not translated to a physical address, and the L1 cache block accessed with `ASID+VA` is used. If a TLB miss occurs, the HW page walker will translate the address and fill the TLB entry. For a false-positive caused by the synonym filter, the non-synonym TLB entry is added to the TLBs to quickly correct potential false-positives for future accesses to the address.

The proposed hybrid design does not require any reverse

[2] The page table entries need to add a single sharing bit for page mappings to mark a page sharing or non-sharing. Such information is easily accessible by the kernel and reserved bits are available for use.

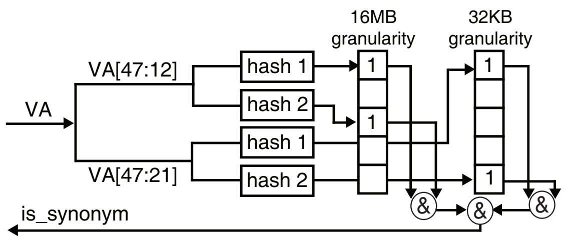

Figure 3. A synonym filter is composed of two Bloom filters each with different granularities. Each filter uses two hash functions. The synonym filter only returns true, thus a synonym candidate when all four bits are set to one.

mapping to track synonyms. It is guaranteed that a single address (either `ASID+VA` or `PA`) is used for a physical cacheline in the entire cache hierarchy. With such a single address for each memory block, cachelines are kept coherent by the HW coherence mechanism either with `ASID+VA` or `PA`, eliminating the synonym problem entirely. With the hybrid address design, as long as the synonym filter can identify all synonym pages, its correctness is guaranteed. The pages used for direct memory access (DMA) by I/O devices are also marked as synonym pages, and they are cached in physical address.

Page Deallocation and Remap: Virtual page mapping or status changes may require selective flushing of cachelines addressed in `ASID+VA`. If the synonym status changes from non-synonym (private) to synonym (shared), the cachelines in the affected page must be flushed from the cache hierarchy, although such changes will be rare. The deallocation of a physical page from a virtual page or remapping of a virtual page also requires the invalidation or flushing of cachelines in the page.

On a modification of a virtual mapping, current systems issue a TLB shootdown which is broadcasted via inter-processor interrupts to all cores. In the hybrid virtual caching, depending on the previous state of the mapping (synonym or non-synonym), the shootdown can be directed to the per core TLB structure (synonym), or the per core TLBs and shared delayed TLB (non-synonym).

Permission Support: For permission enforcement, each cacheline holds the permission bits which are set through the delayed translation for non-synonym pages. For synonym pages, the TLB translation will check the permission status before cache accesses. For non-synonym pages, permission violation such as writes to a read-only (`r/o`) page will raise an exception. When the permission of a non-synonym page changes, the permission bits in cached copies must be updated along with the flush of the delayed translation TLB entry for the page.

B. Hash-based Sharing Detection

The proposed synonym detection uses Bloom filters [11] to detect synonym pages efficiently with a short access latency and low power consumption. Each address space has a set of two Bloom filters maintained by the operating system. The Bloom filters are stored in the OS memory region, and for each context switch, the hardware registers for the starting addresses of the Bloom filters must be set by the OS, along with the conventional page table pointer. Setting the filter registers will invoke the core to read the two Bloom filters from the memory and store them in the on-chip filter storage of the core. Woo et al. proposed using Bloom Filters to filter out synonyms to reduce extra cache lookups to find synonyms, with a pure hardware-oriented mechanism [13]. This study uses a HW and SW-combined approach.

The proposed scheme uses two filters for each address space to reduce false positives as shown in Figure 3. The first coarse-grained filter checks the synonym region at 16MB granularity, and the second fine-grained filter checks at 32KB granularity. The synonym filter reports a synonym candidate only when both filters are set to true for the corresponding entry for an address. To further reduce false-positive, we used two hash functions for each Bloom filter as shown in the figure.

The Bloom filters are cleared during the creation of an address space (process). When the operating system changes the status of a virtual page to a shared one (synonym), it must add the page to the coarse and fine-grained Bloom filters. Updating the Bloom filter pair will require synchronizing all cores running the same ASID. Such a status change is rare, and it uses the same mechanism as TLB shootdowns. Note that conventional systems also require TLB shootdowns to synchronize the TLB entries on page table updates. Changing the status of a page from the synonym state to the non-synonym state does not clear the Bloom filters, since multiple pages may share the same bit. However, such changes are rare, and if such changes exceed a certain threshold and generate too many false positives, the OS can reconstruct the filter set for the process.

In this paper, we use a 1K-bit Bloom filter for both 32KB and 16MB filters. We chose the 32KB granularity, since shared pages are commonly allocated in 8 consecutive 4KB pages. The hash functions partition the address bits (trimmed by 15 bits or 24 bits according to the Bloom filter granularity) into two parts. One function partitions them by 1:1 ratio and the other by 1:2 ratio. For each hash function, 5 bits hash result is generated from a partition by exclusive-oring the bits in the partition. The two 5-bits from the partitions are concatenated into 10 bits, which is the index bit for the filter.

C. Performance of Synonym Filter

Methodology: In this section, we use a trace-based model to evaluate a long execution of applications with a large amount of synonym pages. We used the Pin tool [19] and implemented our synonym filter, TLBs, and delayed

Table II
False positive rates, TLB access and miss reduction

	false positive rates	TLB access reduction	total TLB miss reduction
ferret	0.000756%	99.1%	20.4%
postgres	0.427410%	83.7%	-6.1%
SpecJBB	0.000019%	99.9%	42.6%
firefox	0.521944%	99.4%	63.2%
apache	0.000000%	99.5%	69.7%

translation TLBs (delayed TLBs). We also modeled the cache hierarchy to feed only cache misses into the delayed translation. In the model, false positives are also inserted into the TLBs. We experimented on a system with Linux kernel 3.12 and glibc 2.19.

In addition to the Pin tool, we identified workloads with read-write sharing by extracting and analyzing system call traces. We investigated the five applications with synonyms shown in Table I. The baseline conventional TLBs have a 64-entry L1 TLB backed by a 1024-entry 8-way L2 TLB. Compared to the baseline, the synonym TLB is a 64-entry 4-way associative single level TLB. Our delayed TLB is a 1024-entry 8-way TLB for the results in this section, to have the same overall TLB area as the conventional system. The cache is an 8MB shared cache, and the simulations of the workloads are multi-programmed or multithreaded (depending on each workload).

Results: Table II presents the effectiveness of the proposed synonym filter with two 1K-entry Bloom filters. The second column shows false-positive access rates, due to the hash conflicts in the Bloom filters. Among all accesses, such false-positive accesses are very small, less than 0.5%. The third column shows the reduction of TLB accesses by bypassing TLBs for non-synonym pages. Except for `postgres` which has a significant amount of shared pages, all the other applications can reduce TLB accesses by 99%, reducing the power consumption significantly. Even for `postgres` with 66% of shared memory, the TLB access reduction is significant with an 84% decrease.

The fourth column shows the TLB miss reduction achieved by the synonym TLB and delayed TLB, compared to the baseline system with two levels of TLBs. Even though the total size of TLBs is equal in the proposed system and the baseline, the proposed system can significantly reduce the TLB misses by up-to 70% (`apache`). This is due to the large underlying last-level cache which filters out unnecessary translation requests for resident cachelines. The reason for the miss increase in `postgres` is due to the false positives and smaller 64-entry TLB for the synonym candidates compared to the conventional TLBs.

In this section, we investigated only the applications with shared pages, and even for such adverse applications to the proposed scheme, the results show that the synonym filtering and delayed translation work effectively. More diverse application results will be presented in Section 6.

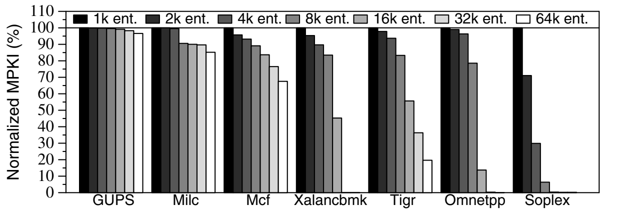

Figure 4. Normalized TLB miss rates (MPKI) for different TLB sizes

D. Efficient Support for Read-only Sharing

As shown in the previous section, synonym pages are not common. One possible source of synonyms not evaluated in the previous section is content-based memory sharing. If a large number of memory pages become synonym pages due to content-based memory sharing, the synonym filters will become less effective, resulting in a significant increase of physically-addressed pages in caches.

However, exploiting the read-only property of such content-based memory sharing, we propose a mechanism to eliminate the need for physical addresses for the read-only (r/o) shared pages. Even if r/o synonyms exist in caches, r/o permission prevents coherence problems of r/o synonyms. As discussed in the previous section, cache tags are extended with permission bits. When the hypervisor or OS sets a page to be a content-shared page, it changes the status to read-only. The r/o permission bit is carried in all copies of the cacheline, and if a write operation occurs, a permission fault is raised. Upon receiving a permission fault, the hypervisor or OS assigns a new physical memory page, copies the content of the r/o shared page to the newly assigned page, and changes the permission of the new page to r/w, making it a private r/w page.

When the status of a page changes from non-synonym to r/o shared, all the cachelines of the page must be either invalidated or their cacheline status must be changed to read-only. However, such status changes do not trigger updates of synonym filters, and do not degrade the filtering capability.

IV. Scalable Delayed Translation

This section presents the limitation of page-based delayed TLBs, and proposes many segment translation architecture to support many variable length segments.

A. Variable Length Segment Translation

1) Delayed TLB translation: For non-synonym pages, address translation occurs on LLC misses. Delayed address translation can use conventional page-based translation with a TLB lookup on each LLC miss. With large on-chip caches, cache-resident data will not cause TLB misses, even if their entries are not in the TLBs. However, such fixed granularity page-based translation, *delayed TLBs*, will eventually reach its limit as the memory working sets of applications increase. Figure 4 shows the limitation of fixed granularity pages for delayed translation. TLB requests are filtered by a 2MB

LLC. Only LLC misses access the delayed TLB ranging from 1K to 32K entries. For `GUPS`, `mcf`, and `milc`, the increase in TLB size does not reduce the number of misses effectively, as their page working sets are much larger than the delayed TLB capacity. Even with a large 32K-entry TLB, 32 times larger than the current 1K-entry L2 TLBs, there are significant TLB misses per 1K instructions.

As the memory requirements for big memory applications increase, the delayed translation must also scale with the requirements. On-chip caches can filter out some translation requests with the hybrid virtual caching, but eventually, the translation efficiency must be improved to support big memory applications. Traditional fixed page-based translation cannot provide such scalability of address translation.

2) Prior Segment Translation Approaches: This section discusses the prior segment translation approaches for conventional physically-addressed caches. To mitigate the limitation of the current TLB-based translation, direct segment was proposed as an alternative to page-based translation [1]. In the direct segment, each process has a set of segment-supporting registers: `base`, `limit` and `offset`. Using the registers, a segment maps a variable length virtual memory partition to a contiguous physical memory region. With such a low complexity, a single segment support can be easily added to the existing page-based systems to allow static memory allocation of large contiguous memory. If the virtual address lies outside the segment region, traditional paging is used.

Redundant memory mapping (RMM) extends on the limitation of a single direct segment, supporting multiple concurrent segments for each process [12]. The operating system can allocate multiple contiguous memory regions with variable lengths for each process, allowing flexible memory management. Traditional paging is used redundantly to RMM. Since the address translation is still on the critical core-to-L1 path, RMM limits the number of segments to 32 operating at the latency of seven cycles, equivalent to the L2 TLB latency.

B. Many Segments for Delayed Translation

Delayed translation can improve segment-based translation by supporting 1000s of segments efficiently. Such many segment translation can provide the operating system with the improved flexibility and efficiency of memory management.

Potential benefits of many segments: Table III presents the segment counts for each application, including the segment counts produced in RMM [12]. Our analysis uses a different memory allocator library, `glibc` instead of customized `TCmalloc` used by RMM, which is the main cause of the difference in segment counts. In the table, some applications use few segments while some other applications use a very large number of segments. `Memcached` for example,

Table III
MAXIMUM NUMBER OF SEGMENTS IN USE BY EACH APPLICATION, RMM MPKI, AND MEMORY UTILIZATION

Bench.	RMM [12]	Reproduced	MPKI	Usage(%)
astar	33	16	0	96.5
mcf	28	4	0	72.5
omnetpp	27	106	0.02	99
cactus.	70	56	0	83.2
GemsFD.	61	143	0	100
xalancbmk	N/A	244	0.13	96.5
canneal	46	22	0	84.7
stream.	32	16	0	24.7
mummer	61	3	0	100
tigr	167	90	22.93	99.5
memcached	86	839	0.08	100
NPB:CG	95	5	0	100
gups	62	7	0	99.9

requests for more memory on demand (in 64MB requests) instead of provisioning large chunks of memory.

Using a limited 32 segments may not be able to provide efficient translation when the segments are thrashing in RMM. Workloads such as `tigr, xalancmbmk` and `memcached` caused considerable MPKIs (segment misses per 1K instructions) for 32 segments in our experiments. If a segment miss occurs, either a SW or HW segment walker must fill the segment in RMM.

Another inherent issue of segment-based translation is the low utilization of eagerly allocated memory regions. Instead of the widely used demand paging, segment translation uses *eager allocation*, which allocates contiguous memory segments immediately on application request. Eager allocation increases the contiguity of allocated memory to reduce the number of resulting segments, but may cause internal fragmentation. The final column of Table III shows the utilization of segmented memory regions. Although many workloads use most of the allocated regions, four applications do not utilize 17-75% of their allocated memory. Reservation-based allocation can be used to handle such cases by reserving a large contiguous segment, but internally dividing the segment into smaller segments [20]. Only on actual accesses, the smaller sub-segments are actually allocated to the process. Adjacent sub-segments can be merged as they are promoted from reserved to allocated. However, reservation-based allocation requires more segments to support the reservation functionality.

C. Many Segment Architecture

Figure 5 presents the overall translation flow of the proposed many segment architecture after an LLC miss occurs. The translation mechanism consists of *segment table*, *index cache*, and *segment cache*. Figure 6 shows the internal organization of segment table and index cache.

Segment Table: The OS maintains a system-wide *in-memory segment table* that holds all the segments allocated by the OS. Each segment entry has the starting `ASID+VA` address(`base`), `limit`, and `offset`. The table is indexed

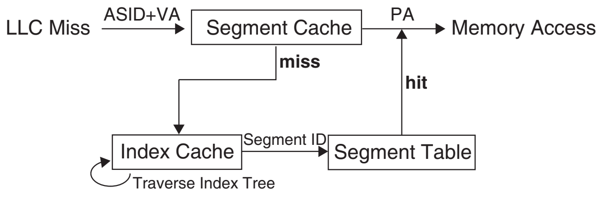

Figure 5. Scalable delayed translation: The overall translation flow from ASID+VA to PA

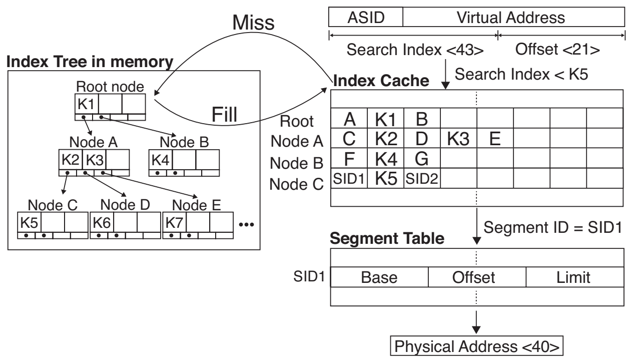

Figure 6. Organization of segment table and index cache

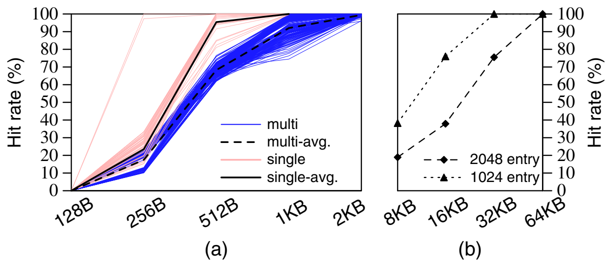

Figure 7. Index cache size sensitivity study. (a) shows index cache hit rate for actual workloads. (b) shows synthetic worst case benchmark.

by the *segment-ID* and holds 2K segment entries, as shown in Figure 6. A hardware structure, *segment table*, mirrors the in-memory segment table. If an incoming `ASID+VA` address is not covered by the segments in the segment table, an interrupt occurs and the operating system fills the table entry. However, segment misses occur only for cold misses, as the size of HW table is equal to the in-memory segment table size to simplify implementation.

Index Cache: One of the key design issues is to find the corresponding segment for an incoming `ASID+VA` address from the segment table efficiently. Unlike TLBs with fixed mapping granularity, segments have varying sizes which complicate the lookup mechanism. A naive way of searching for a segment is to serially search all entries, which look ups all table entries in the worst case. Since the translation latency is important, we propose a hardware-based efficient search mechanism backed by the operating system.

The operating system maintains a B-tree indexed and sorted by the `ASID+VA` for all segments in the segment table, called an *index tree*. Each B-tree node has a key to compare the incoming address, and pointers to the nodes of the next level (the value). The resulting value, which may exist on the leaf or on intermediate nodes, is an index to the segment table, or the *segment-ID*. Thus, a traversal through the index tree yields the segment-ID of the segment that the incoming address belongs to.

For every LLC miss, the index tree must be accessed to find the segment-ID which points to the corresponding segment in the segment table. Since accessing the in-memory index tree for every LLC miss is not feasible with long latencies, a hardware cache for the index tree, *index cache*, stores recently accessed entries from the index tree. For each LLC miss, in the worst case, the index cache must be accessed by the number of the tree depth to reach the leaf node containing the requested segment-ID.

The index cache is a regular cache of 64 byte blocks addressed by physical address. The index tree nodes are cache block aligned. Each node contains six keys (starting address of a segment) and seven values (pointer to the next level node, or the segment-ID). 1024 and 2048 segment entries can fit in an index tree with depth of four, when spanned by a factor of seven (seven values). A hardware walker will bring in a cache block from the memory and will compare the incoming address to all six keys in parallel. The leftmost pointer which satisfies the comparison (address>key) will be used for the next node lookup.

Segment Cache (SC): To hide the latency of accessing the index cache and segment table, the delayed translation can be done simultaneously with LLC accesses. Although the parallel accesses to the delayed translation and LLCs can improve the performance, such parallel accesses can increase the energy consumption for delayed translation unnecessarily. To reduce the energy overhead, an alternative way is to access delayed translation serially only after LLC misses. However, to further reduce the latency overhead of serial delayed translation, this architecture adds a small 128-entry segment cache. The segment cache is a simple TLB-like component with 2MB granularity. For an SC miss, the translation results from the segment table will be used to fill the fixed granularity SC entry for next accesses.

Translation Flow: As shown in Figure 5, the incoming address from an LLC miss checks the 2MB granularity segment cache (SC) first. If it hits SC, the translation is completed. For a miss, the translation traverses the index tree by looking up tree nodes. The segment-ID resulting from the index tree traversal is used to index the HW segment table to find the segment information. Using the segment entry, the address is checked against the base and limit values of the segment. Finally, the offset value is used to translate the incoming `ASID+VA` to PA.

D. Index Cache Efficiency

We conducted three sensitivity studies on the index cache size. The first study investigates single-threaded applications, and the second one for four threads of applications modeling a multi-programmed quad-core system. To stress the index cache, we artificially broke a segment into 10 segments, adding the effect of external fragmentation. Without the added external fragmentation effect, a smaller size of index cache will be good enough for many real applications. The last study investigates the worst case scenario, where all accesses are randomly issued. The results from the three studies are shown in Figure 7. The index cache is an 8-way associative cache ranging in size as labeled in the x-axis.

The single and multi-core application simulations were conducted using the Pin-based simulation of workloads described in Section VI-A. The multi-core evaluation was conducted by interleaving four Pin traces (quad core system). Ten applications causing most misses were chosen, and a total of 210 quad-core mixes were generated and executed. The lighter lines show the miss curves of single thread applications, and the darker lines show multi-threaded applications. The multi-threaded workloads cause more conflict misses compared to single applications, thus the curves show slightly more misses for the same index cache size. The access patterns of real applications exhibit locality, thus the index cache does not suffer misses even with a modestly sized index cache of 8KB.

For the worst case, we distributed the physical address space of 40 bits to 1024 or 2048 segment entries equally. We inserted all entries into the index tree, and simulated one million random accesses to the entire physical address space. This workload shows no spatial locality, and is thus the absolute worst case. Even in this worst case, 32KB index cache can almost eliminate index cache misses for 1024 segment entries, and provide 75.5% hit rates for 2048 segment entries.

Using CACTI 6.5 [18], we estimated the access latency of the index cache. For a 3.4GHz machine, both 32KB and 64KB 8-way index cache have the latency of 3 cycles. Also the access latency of the segment table of 2048 entries is seven cycles [3]. Thus we can estimate that four accesses or less to the index cache (four level index-tree) followed by an access to the segment table is about 19 cycles. Based on the analysis, we assume that the delayed many-segment translation takes 20 cycle.

To support 2048 segments, the segment table size is about 48KB (base, offset, length), and the index cache size is 32KB. Compared to the LLC size, the extra 80KB structure does not add a significant area overhead. Furthermore, a multi-core processor needs to have only one index cache and segment table shared by multiple cores.

[3]The segment table is configured to use low standby power configuration. All others use the high performance configuration

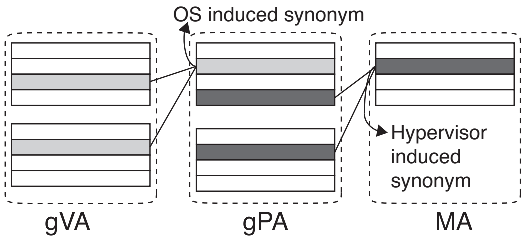

Figure 8. Difference of synonyms incurred by the OS and hypervisor

V. Virtualization Support

The overall translation architecture for virtualized systems is similar to non-virtualized systems. For non-synonym pages, the ASID must include the virtual machine identifier (VMID) in addition to the process ID within a VM. With the ASID extension, a VM cannot access virtually-addressed cachelines of another VM, since their ASIDs do not match.

A. Extending Synonym Detection

In virtualized systems, a guest virtual address (gVA) is translated to a guest physical address (gPA), and the guest physical address is translated to a machine address (MA). The guest operating system maintains the mapping between the guest virtual and guest physical address spaces, and the hypervisor maintains the mapping between the guest physical and machine address spaces. In non-virtualized systems, the synonym filters are maintained by the operating system. However, for virtualized systems, both the OS and hypervisor can cause synonyms for memory pages. In addition to OS-induced synonyms, the hypervisor can make two different pages in the same VM or different VMs share a physical page. Figure 8 illustrates how two types of synonyms differ.

To address the additional hypervisor-induced synonyms, the synonym filters are updated both by the OS and hypervisor. Similarly to the two dimensional page walks supported by recent x86 architectures, the OS and hypervisor maintain different filters, the `guest` and `host` filters. For a guest process context switch, the guest OS switches the guest filter as in a native system. For a VM context switch, the hypervisor switches the host filter. When looking up the synonym filter, both filters are looked up. If either one of the filters reports a filter hit, the accessed page is identified as a synonym candidate. The subsequent flow for synonym and non-synonym pages are identical to native systems.

One of the challenges for the synonym detection is that the filters must be set in gVA for both the guest and host Bloom filters, since both filters are looked up using the guest virtual address. When a guest virtual page becomes a synonym due to a hypervisor-induced sharing, the hypervisor is responsible for setting the host filters by the guest virtual address.

When the hypervisor changes a guest physical page to a synonym page (shared page), it traces the corresponding guest virtual addresses in the virtual machine, and updates the host filter content. To facilitate the process, the hypervisor may maintain an inverse mapping from gPA to gVA for each virtual machine.

On a special case where the guest OS changes the mapping of a guest virtual page from a private guest physical frame to a hypervisor-induced shared physical frame, the guest is not aware of the guest physical frame being shared by the hypervisor. Thus, it is the responsibility of the hypervisor to mark the newly mapped gVA as a synonym. Our solution is simple for the case. For such remapping of a guest virtual page, the guest OS must flush the corresponding cachelines anyways, regardless of the hypervisor-induced synonym problem. Access to the newly mapped page causes an LLC miss, causing delayed translation. The address will be identified as a synonym address during the two dimensional walk, and an exception will be raised for the hypervisor to handle the newly identified synonym.

B. Segment-based 2D Address Translation

The current full virtualization of the x86 architecture uses two separate page tables in the guest OS and hypervisor to map the virtual memory to the actual system memory [21]. A hardware two-dimensional page walker walks both guest and hypervisor page tables and fills the TLB with the translation from gVA to MA. One of the benefits of the hybrid virtual caching for virtualized systems is that it hides the cost of the two-dimensional translation. By removing the translation from the critical core-to-L1 path, much of the delayed translation can be filtered by LLC hits. For the proposed hybrid virtual caching, a similar 2D page walker will be used for the page-based delayed translation, which will walk the guest and host page tables for translation cache misses.

To support full virtualization for many segment translation, segment translation must be supported for guest and host segments, which are governed by the guest OS and hypervisor respectively. The guest OS can update only the guest segments. Full virtualization incurs two overheads if it is done before L1 cache accesses as in RMM [12]. First, segment-based translation needs to be done twice for guest and host segments. The two steps are required because the hypervisor cannot guarantee to allocate equally sized physical segments for every guest OS segment allocation requests, and may serve a guest OS segment allocation with multiple host segments. Second, the per-core segments limited in number are further divided into guest and host segments, reducing the number of segments available for each user process.

The proposed many segment translation does not suffer from the limited number of segments for guest and host segments. The additional latency incurred by the two dimensional translation can be overlapped with LLC accesses to mitigate additional latency at the expense of minor extra energy consumption. However, in this study, to reduce energy consumption, we employ serial accesses to the LLC and segment translation. To further reduce the additional latency, a 128-entry segment cache (SC) is used to store direct translation from gVA to MA for 2MB regions, skipping the gPA.

Table IV
SIMULATED BASELINE SYSTEM CONFIGURATIONS

Parameter	Value
Processor	Out-of-order x86 ISA, 3.4GHz 128-entry ROB, 80-entry LSQ 5-issue width, 4-commit width 36-issue queue, 6-ALU, 6FPU
Branch Predictor	4K entry BTB, 1K entry RAS Two-level branch predictor
L1 I/D Cache L2 Cache L3 Cache	2/4-cycles, 32KB, 4-way, 64B block 6-cycles, 256KB, 8-way, 64B block 27-cycles, 2MB, 16-way, 64B block
TLB L1 TLB L2	1-cycles 64 entry, 4-way, non-blocking 7-cycles 1024 entry, 8-way, non-blocking
Memory	4GB DDR3-1600, 800MHz, 1 memory controller

VI. EXPERIMENTAL RESULTS

A. Methodology

To validate the performance of the proposed system, we used MARSSx86 [22] + DRAMSim2 [23] which is a cycle accurate full system simulator running a linux image, with an accurate DRAM simulator. Table IV shows the simulated system configuration. For the baseline system, we modeled the TLBs after the Haswell architecture of Intel [24]. The evaluated experiments run SPECCPU2006, Graph500 (sized 22), NPB benchmarks (C sizes, and B size for `NPB_DC`), and `tigr` of the BioBenchmark suite [25]. Additionally we conducted experiments of GUPS (with size 30), a random access benchmark. One billion instructions were simulated for evaluation. However, for `tigr` we had to reduce the number of instructions executed to 500 million, due to the elongated simulation time resulting from a very low IPC.

B. Performance

Native Systems: In Figure 9, the first experimental results show the performance improvement of delayed translation in non-virtualized systems. The results were normalized to the baseline system. The evaluated configurations are the baseline, fixed granularity delayed TLB translations varying the TLB size from 1K to 32K entries (henceforth labeled as delayed TLB), the delayed many-segment translation (without and with 128-entry segment cache), and finally the ideal TLB performance. The ideal TLB depicts the potential performance of a system without TLB misses. The graph is divided into three parts by the dotted vertical line.

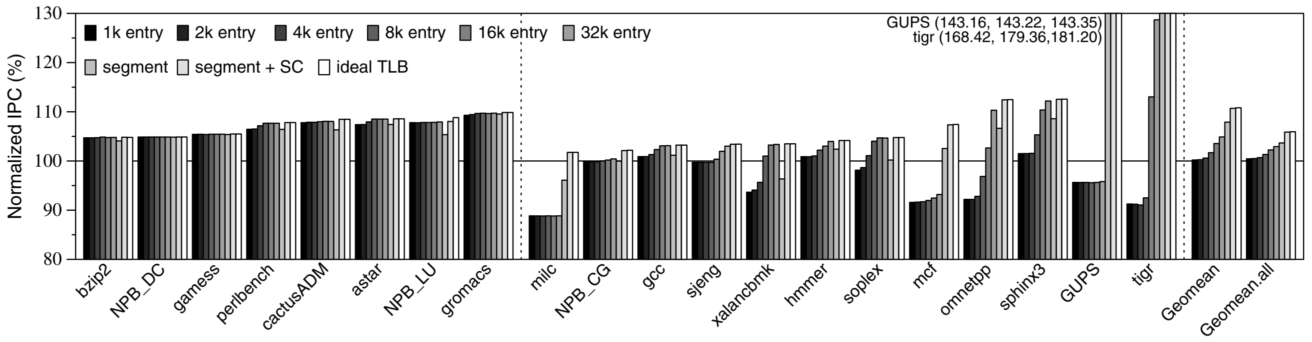

Figure 9. Normalized performance of our proposed system to the baseline system. Workloads on the left are workloads which benefit from delayed translation. The results in the center show workloads in which fixed granularity delayed translation causes overhead but improves as more delayed TLB entries are provided.

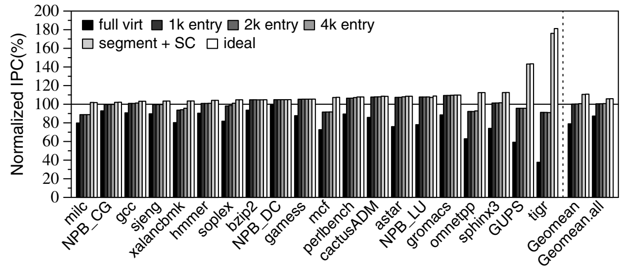

Figure 10. Performance of full-virtualized baseline and delayed translation normalized to the baseline native system

For brevity, only the workloads sensitive to the different configurations are plotted. The remaining workloads are aggregated in the `Geomean.all` plot.

The leftmost part of the figure shows the workloads which benefit directly from delayed translation of virtual caching. These workloads efficiently utilize the cache hierarchy, and delayed translation effectively removes unnecessary translation requests which occurred in the baseline system. The performance of these workloads is higher than the baseline, and consistent for different sizes of delayed TLB, as the 1K TLB can absorb most of the delayed translation requests.

In the applications of the center part of the figure, a naive TLB-based delayed translation causes performance reduction. These applications exhibit significant LLC misses, and delayed translation adds extra latencies for each miss. Most of the applications in the center part suffer from a certain performance reduction with the delayed TLBs, compared to the baseline. The reason for performance degradation is that the delayed TLB miss handling can take longer latencies than the conventional TLB miss handling in our conservative model of delayed TLBs. In conventional TLBs, a TLB miss handler (page table walker) can access L1, L2, and L3 caches to fetch the required page table entries during page walks. Among the accesses, a significant number of accesses are L1 or L2 cache hits. However, in the delayed TLBs, we conservatively assume that the page table walker can access only the LLC (L3), since the TLB miss handler is located along with the LLC miss handler. Therefore, even if the TLB miss rates are equal in the baseline and delayed 1KB TLBs, the performance degradation can occur with the delayed TLBs. A simple remedy for this problem is to use a translation caching scheme which caches non-leaf tree entries of multi-level page table for delayed translation. For the performance comparison in this section favoring the baseline, we did not evaluate the translation caching support for the delayed TLBs.

For a subset of workloads such as `sjeng`, `xalancbmk`, `hmmer`, `soplex`, `omnetpp`, and `sphinx`, the performance is improved for larger delayed translation TLBs. However, `milc`, `mcf`, and `GUPS` still suffer from performance drops, as the larger TLBs cannot cover the working sets. The results are consistent to the Pin-based study presented in Figure 4. For the workloads with high performance drops even with large TLBs, our proposed many segment translation shows its potential. Many segment translation with a small segment cache (SC) can almost eliminate the cost of delayed translation. Across all the workloads evaluated, the performance with many segment translation with SC matches the performance of ideal translation. For `GUPS` and `tigr`, the performance improvements are 43% and 79% compared to the baseline. On average, the many segment translation scheme improves the performance of selected workloads by 7.9% and many-segment with SC shows an average of 10.7% performance gain. Although not shown in the figure, parallel accesses to the many segment translation and LLCs can provide near ideal performance for all the applications, 0.1% less than the ideal runs. For the selected workloads, parallel accesses achieve an average of 10.8% performance gain compared to the baseline.

Delayed TLB exploits the LLC to filter away unnecessary TLB accesses, improving TLB caching efficiency. TLB miss reduction from the baseline 1K-entry TLB to the delayed TLB of 1K-entry is 99.3% on average. In other words, 99.3% of L2 TLB misses on conventional systems can be filtered away if the translation is delayed to after an LLC miss. `GUPS`, `tigr`, and `mcf` have lower TLB reductions of

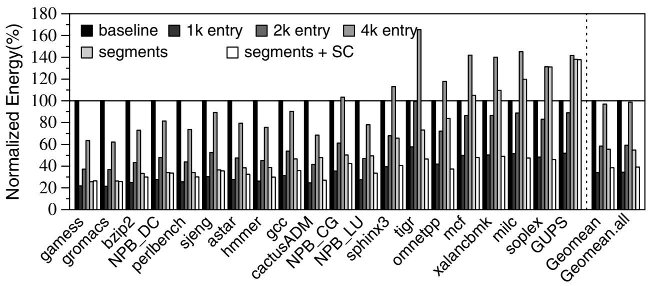

Figure 11. Normalized power consumption of delayed translation over baseline system (power consumption by translation components)

45.5%, 61%, and 76.4%, respectively. The workloads above show random access patterns that are not effectively cached in the LLC, resulting in lower TLB reductions.

Full Virtualization: The second experimentation is to evaluate the performance of fully virtualized systems. We modeled an x86-like fully virtualized system, denoted as `full virt`, and delayed TLB translation which does two dimensional page walks on a miss with TLB sizes ranging from 1K to 4K entries. The 2D many segment translation is denoted as `segment + SC`. The ideal TLB performance is also shown. The results are normalized to the native non-virtualized system performance.

Full virtualization incurs page walk overheads as up-to 24 memory accesses are required, instead of four in native systems. To shorten the page walk latency, commercial architectures support translation caches to cache non-leaf translation entries of multi-level page tables [26], [21]. For the experiment in this work, we have added a page walk cache (PWC) to speed up the page table walks for the `full virt` system. However, with the conservative assumption on our own technique, PWC is not used for delayed TLB translation.

Figure 10 shows the performance result of virtualized systems. The fully virtualized system shows significant performance drops due to the overhead of two-dimensional page walks. Delayed TLB improves performance, and our many segment translation (with 128-entry segment cache) shows the best performance, close to the ideal system. As with the native results, the delayed TLB shows a slight performance improvement with more TLB entries for some workloads, while other workloads such as `milc`, `GUPS` and `mcf` do not benefit from larger TLBs. Many segment translation overcomes such limitations. On average, many segment translation performs 18.5% better than the current fully virtualized architecture for all workloads, and 31.6% better for the selected workloads.

C. Energy Consumption

Figure 11 presents the dynamic and static power consumption of translation components of proposed mechanisms normalized to the baseline TLB system. The power models are generated from CACTI 6.5 [18]. Delayed translation with TLBs and many segments can achieve reduction in overall power consumption, because the power consumed for accessing the synonym filters[4] is lower than that for accessing L1 TLBs, and the number of accesses to the delayed TLB is smaller than that to conventional TLBs, because most of accesses are filtered by the LLC.

Delayed many segment translation generally has lower power consumption compared to the delayed TLB configurations due to the higher static power consumption of larger delayed TLBs. The delayed TLB structures that are larger than 2K entries consume more static power compared to the index cache and segment table, resulting in higher overall power consumption over the delayed many segment translation. However, for the workloads with high LLC misses, delayed many segment without SC consumes more power than even the baseline system.

Using a segment cache reduces the power consumption significantly, as it reduces accesses to the index cache and segment table, and also achieves modest performance gains as observed in Figure 9. The segment cache miss rate is 0.05%, effectively buffering multiple lookups to the index cache and segment table. One exception is `GUPS`, which incurs a high segment cache miss rate (of 96.5%), as the workload has inherently random access patterns. Other noticeably high miss rates were observed in `tigr`, `NPB_LU`, and `NPB_CG` with miss rates of 31.3%, 12.4%, and 10.8%, respectively.

Summary: For many common workloads with low LLC miss rates, hybrid virtual caching can reduce the translation power consumption significantly, as it can eliminate most of the conventional TLB accesses. It can improve the performance too, if the LLC can contain cachelines which could have missed the conventional TLBs. However, for the applications with high LLC misses, the many segment translation can increase the power consumption due to increased accesses to the index cache and segment table. However, although translation power consumption increases for such cases, the performance of the applications can potentially improve significantly with a much more effective translation mechanism than the conventional system, if the traditional TLBs cannot provide a sufficient translation coverage for the applications. The aforementioned two scenarios show the main advantage of the proposed techniques. The hybrid virtual caching can bring either power, performance, or both benefits by virtual caching and delayed translation depending on application TLB and LLC miss behaviors.

VII. Conclusion

This study proposed a new memory virtualization architecture with hybrid virtual caching. The key component

[4] Due to restrictions of the CACTI system, the modeled synonym filter reads in byte granularity instead of bits granularity which causes over estimation of dynamic power per filter access. We expect optimized hardware implementation to be able to conserve more power per access by accessing the synonym filter by bits.

enabling the efficient virtual caching is the synonym filter which can identify synonym pages efficiently with few false positives. With the synonym filter, a physical cacheline can exist in caches only with a single name either by its physical address (synonym case) or by its `ASID+VA` (non-synonym case). By extending hybrid virtual caching to the entire cache hierarchy, the proposed mechanism reduces both energy consumption and translation latency. However, as the memory requirements increase, even delayed translation with page-based TLBs cannot scale effectively. This paper proposed a many segment translation architecture for better translation scalability and more flexible memory allocation by the OS.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2013R1A2A2A01015514)

References

[1] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, Jun 2013.

[2] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[3] A. Basu, M. D. Hill, and M. M. Swift, "Reducing memory reference energy with opportunistic virtual caching," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[4] A. Sodani, "Race to exascale: Challenges and opportunities," MICRO 2011 Keynote.

[5] J. R. Goodman, "Coherency for multiprocessor virtual address caches," in *Proceedings of the second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS)*, 1987.

[6] W. H. Wang, J. L. Baer, and H. M. Levy, "Organization and performance of a two-level virtual-real cache hierarchy," in *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA)*, 1989.

[7] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[8] B. Jacob and T. Mudge, "Software-managed address translation," in *Proceedings of the IEEE 3rd Symposium on High-Performance Computer Architecture (HPCA)*, 1997.

[9] B. Jacob and T. Mudge, "Uniprocessor Virtual Memory Without TLBs," *IEEE Trans. Comput.*, vol. 50, no. 5, pp. 482–499, May 2001.

[10] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Publishers, 2008.

[11] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[12] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Unsal, "Redundant Memory Mappings for Fast Access to Large Memories," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, Jun 2013.

[13] D. H. Woo, M. Ghosh, E. Ozer, S. Biles, and H.-H. S. Lee, "Reducing energy of virtual cache synonym lookup using bloom filters," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2006.

[14] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton, "An in-cache address translation mechanism," in *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA)*, 1986.

[15] L. Zhang, E. Speight, R. Rajamony, and J. Lin, "Enigma: Architectural and operating system support for reducing the impact of address translation," in *Proceedings of the 24th ACM International Conference on Supercomputing (ICS)*, 2010.

[16] H. Yoon and G. S. Sohi, "Revisiting virtual L1 caches: A practical design using dynamic synonym remapping," in *Proceedings of the IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[17] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architecture support for single address space operating systems," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[18] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," Tech. Rep., 2009.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[20] M. Talluri and M. D. Hill, "Surpassing the TLB Performance of Superpages with Less Operating System Support," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[21] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[22] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A Full System Simulator for x86 CPUs," in *Proceedings of the 48th Design Automation Conference (DAC)*, 2011.

[23] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[24] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel, Jul. 2013.

[25] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "Biobench: A benchmark suite of bioinformatics applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2005.

[26] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel, Jun. 2013.