2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

# SEESAW: Using Superpages to Improve VIPT Caches

Mayank Parasar
*School of ECE*
*Georgia Institute of Technology*
*Atlanta, GA, USA*
*Email: mparasar3@gatech.edu*

Abhishek Bhattacharjee
*Department of Computer Science*
*Rutgers University*
*New Brunswick, NJ, USA*
*Email: abhib@cs.rutgers.edu*

Tushar Krishna
*School of ECE*
*Georgia Institute of Technology*
*Atlanta, GA, USA*
*Email: tushar@ece.gatech.edu*

***Abstract*—Hardware caches balance fast lookup, high hit rates, energy efficiency, and simplicity of implementation. For L1 caches however, achieving this balance is difficult because of constraints imposed by virtual memory. L1 caches are usually virtually-indexed and physically tagged (VIPT), but this means that they must be highly associative to achieve good capacity. Unfortunately, excessive associativity compromises performance by degrading access times without significantly boosting hit rates, and increases access energy.**

**We propose SEESAW to overcome this problem. SEESAW leverages the increasing ubiquity of superpages[1] – since superpages have more page offset bits, they can accommodate VIPT caches with more sets than what is traditionally possible with only base page sizes. SEESAW dynamically reduces the number of ways that are looked up based on the page size, improving performance and energy. SEESAW requires modest hardware and no OS or application changes.**

***Keywords*-Virtual Memory; L1 Caches; Memory systems; Superpages.**

## I. Introduction

L1 caches service the majority of CPU and coherence requests, and are important for overall system performance and energy. Modern L1 caches are designed to balance:

① **Good performance:** L1 caches must achieve high hit rates and low access times. This requires balancing the number of sets and ways in the set. Higher associativity can increase hit rates, but worsen access times.

② **Energy efficiency:** Cache hit, miss, and management (e.g., insertion, coherence, etc.) energy must be minimized. Higher set-associativity may reduce cache misses and subsequent energy-hungry probes of larger L2 caches and LLCs. But higher set-associativity also magnifies L1 lookup energy.

③ **Simple implementation:** L1 cache cycle times are crucial to core timing. For good performance, L1 caches should be simple and fast, with low load-to-use latency.

Unfortunately, the virtual memory subsystem makes it challenging to design L1 caches that simultaneously achieve all these goals. The key problem is that virtual memory necessitates virtual-to-physical address translation. Designers accelerate address translation with per-CPU Translation Lookaside Buffers (TLBs). To accommodate increasingly memory-intensive workloads with large page tables, processor vendors design large TLBs. But this presents a problem for L1 caches, which are physically addressed and require TLB access *prior* to cache lookup. Fig. 1a shows that larger and hence slower TLBs delay L1 cache access.

Architects have historically used virtual-indexing and physical-tagging (VIPT) to overcome this problem. Fig. 1b shows that VIPT L1 caches are searched in parallel rather than in series with the TLB. L1 cache set selection is overlapped with TLB lookup, which must complete by the time L1 cache tag comparisons commence. This approach enables larger TLBs, but it complicates our ability to realize L1 caches that achieve ①-③. The main problem is that VIPT caches require the cache index bits to be entirely subsumed in the page offset field. This restricts the number of sets implementable in the cache. For example, consider x86-64 systems, with 4KB baseline pages. Since the page offset is 12 bits and cache lines are typically 64 bytes, this leaves only 6 bits for the set index. In other words, vanilla VIPT L1 caches for x86-64 systems can integrate at most 64 sets. This means that VIPT L1 caches can be grown only by increasing associativity, rather than by implementing more sets, as Fig. 1c shows. For many real-world workloads, as we show in Section III, increasing associativity provides only diminishing hit rate benefits, but significantly worsens access latency and energy.

Studies have attacked this problem with virtually-indexed, virtually-tagged (VIVT) caches [1, 2], and approaches that implement VIPT but use a subset of the virtual page number bits for cache indexing [3]. Some of these ideas have been implemented in select products (e.g., MIPS R10K) [4]. But such designs can be complex and require dedicated hardware to track down virtual address synonyms[2], particularly on store operations. Since there are cases when OSes use synonyms aggressively to implement operations like memory deduplication, copy-on-write, fork, etc. [5, 6], VIPT L1 caches (which do not suffer from the synonym problem) remain more commonly used in real-world products. Similarly, other approaches reduce L1 cache associativity via opportunistic virtual caching [7], synonym awareness [8–12],

[1]By superpages, we refer to any page sizes supported by the architecture bigger than base page size.

[2]Synonyms are scenarios where multiple virtual addresses map to the same physical address.

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00026

and by pushing part of the index into the physical page number [8]. These are useful but remain complex compared to VIPT.

This paper proposes an alternate approach we call **Se**t-**E**nhanced **S**uperpage **Aw**are (SEESAW) caching. SEESAW is inspired by the following – VIPT L1 caches were originally designed when OSes predominantly allocated (and in some cases only supported) one page size. However, modern systems support and actively use not only these conventional base pages, but also larger superpages (e.g., 2MB and 1GB on x86 systems) [13, 14]. Superpages have historically reduced TLB misses and lowered page table size [5, 15, 16]. But we go beyond and ask, can superpages also be used to realize VIPT caches that better achieve goals ①, ②, and ③?

Conceptually, superpages are an attractive candidate to improve VIPT because they have wider page offsets. For example, x86-64 systems support 2MB and 1GB superpages with 21-bit and 30-bit page offsets. Fig. 1d shows that superpages can therefore accommodate more index bits for VIPT, and consequently permit more cache sets than what is possible assuming only 4KB base pages. SEESAW leverages this observation to dynamically reduce set associativity for accesses to data residing in superpages. In other words, SEESAW supports three types of cache lookups:

① **CPU lookups for data in a superpage:** SEESAW checks fewer L1 cache ways than traditional VIPT caches, reducing hit time and saving energy.

② **CPU lookups for data in a base page:** SEESAW checks the same number of cache ways as traditional VIPT and hence achieves the same performance and energy.

③ **Coherence lookups:** L1 coherence lookups use physical addresses. Consequently, they *do not* need TLB lookup. However, they still needlessly look up many L1 ways since VIPT necessitates high L1 set-associativity. SEESAW solves this problem, allowing *all* coherence lookups, whether they are to addresses in superpages or base pages, to check fewer L1 cache ways. Coherence energy reduces dramatically.

Overall, SEESAW does not replace but instead improves the readily-implementable concept of VIPT. To that end, our contributions are as follows:

- We study cache lookup time and energy as a function of size and associativity with a commercial SRAM memory compiler. We also study average memory access time of real-world cloud and server workloads with associativity. Our results highlight the limitations of traditional VIPT.
- We perform a real-system characterization study on the prevalence of superpages in modern systems. We find that current Linux/x86 systems create ample superpages for SEESAW to be effective.
- We use cycle-accurate software simulation to model SEESAW's performance and energy benefits on an in-order and out-of-order architecture based on Intel's Atom and Sandybridge processors. Against 32KB and 64KB baseline L1 VIPT caches, SEESAW achieves 3-10% better runtime, and 10-20% better memory access energy. We show how these numbers vary with memory fragmentation, which impacts the OS's ability to generate superpages.

Ultimately, SEESAW represents a new way to reap the benefits of superpages. We believe that this is well-aligned with recent trends in OS design, where superpage adoption is becoming ubiquitous and necessary to combat the increasing memory pressure of emerging memory-intensive software [13, 15–17]. SEESAW represents an opportunity to design hardware that piggybacks atop this body of parallel work.

## II. BACKGROUND

SEESAW targets improvements in the interactions between the TLB and L1 cache. We briefly discuss how modern systems implement this layer today.

### A. TLB and L1 Cache Interface

Modern CPUs use TLBs and OS-mantained page tables to translate virtual addresses (VAs) to physical addresses (PAs). TLBs interface with caches in three ways:

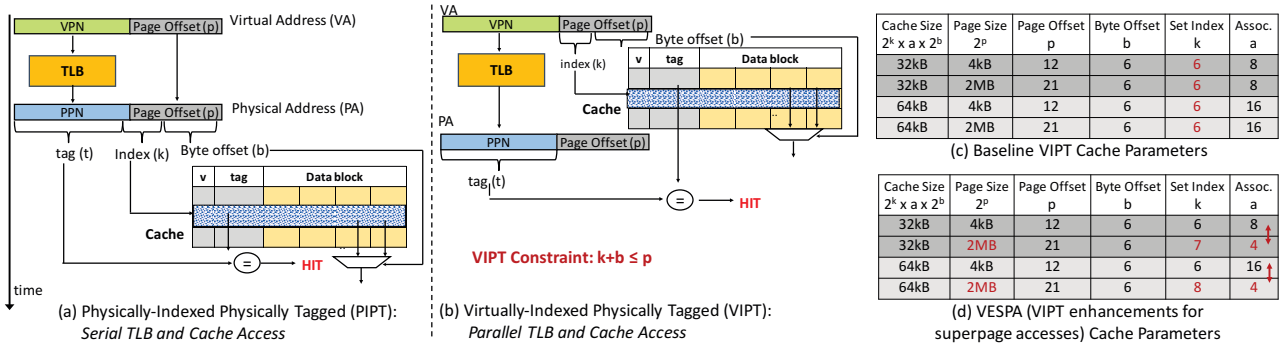
**Physically-indexed, physically-tagged (PIPT):** The L1 cache is looked up after the TLB, making this the slowest approach. PIPT is uncommon in real-world processors.

**Virtually-indexed, virtually-tagged (VIVT):** L1 cache accesses can proceed without a prior TLB lookup. While VIVT caches have been implemented in some real-world products, they remain sufficiently complex (particularly in synonym management) to preclude ubiquitous adoption.

**Virtually-indexed, physically-tagged (VIPT):** This remains the "typical" way to implement L1 caches but is constrained in its ability to grow using larger numbers of sets. VIPT cache capacities can usually be increased by adding more ways per set. This is reflective on modern chips like Intel's Skylake [18] which uses 8-way set-associativity for its 32KB L1 since the base page size is 4KB.

### B. System Support for Superpages

For years, Linux, FreeBSD, and Windows have supported 2MB and 1GB superpages (in addition to 4KB base pages) on x86-64 systems. Similarly, ARM systems support 1MB and 16MB superpages. While the adoption of superpages from research studies [16, 19] to production systems has taken decades, the advent of big-data workloads has meant that superpages are widely used today [4, 13–15, 20–22]. In particular, support for transparent 2MB superpages is now sufficiently mature that it is enabled by default on production OSes like Linux [13]. Naturally, there may still be situations



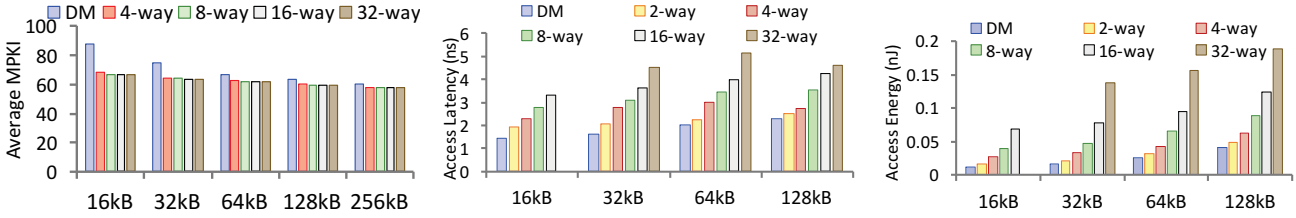

| Cache Size $2^k$ x a x $2^b$ | Page Size $2^p$ | Page Offset p | Byte Offset b | Set Index k | Assoc. a |
|---|---|---|---|---|---|
| 32kB | 4kB | 12 | 6 | 6 | 8 |
| 32kB | 2MB | 21 | 6 | 6 | 8 |
| 64kB | 4kB | 12 | 6 | 6 | 16 |
| 64kB | 2MB | 21 | 6 | 6 | 16 |

(c) Baseline VIPT Cache Parameters

| Cache Size $2^k$ x a x $2^b$ | Page Size $2^p$ | Page Offset p | Byte Offset b | Set Index k | Assoc. a |
|---|---|---|---|---|---|
| 32kB | 4kB | 12 | 6 | 6 | 8 |
| 32kB | 2MB | 21 | 6 | 7 | 4 |
| 64kB | 4kB | 12 | 6 | 6 | 16 |
| 64kB | 2MB | 21 | 6 | 8 | 4 |

(d) VESPA (VIPT enhancements for superpage accesses) Cache Parameters

Figure 1: Overview of SEESAW compared to traditional VIPT caches. SEESAW dynamically changes its associativity for superpages.




(a) Avg. Miss-per-kilo-instructions (MPKI) (b) Cache Access Latency (c) Cache access energy

Figure 2: Effect of MPKI, cache latency, and energy as a function of associativity for different cache sizes.

when base pages are preferred. This happens, for example, when finer-grained protection is necessary [5, 23]. Multiple page sizes are even being integrated on mobile OSes like Android [24].

Hardware vendors have responded to the ability of modern OSes to more aggressively deploy superpages by designing TLB hardware with greater capacity to cache superpage translations. For example, in some architectures (particularly ARM and Sparc processors), fully-associative L1 TLBs maintain translations for multiple page sizes concurrently. In other architectures (typical of Intel processors and some AMD processors), L1 TLBs are implemented as set-associative structures, with separate L1 TLBs for different page sizes. For example, Intel Sandybridge, Haswell, and Skylake chips maintain separate L1 TLBs for 4KB, 2MB, and 1GB pages [22, 25]. The L1 TLBs are backed by larger L2 TLBs. As OS superpage support has improved, L1 TLBs for 2MB/1GB pages have doubled from Sandybridge to Skylake, and L2 TLBs are now large 1536-entry structures that support not just 4KB pages, but also 2MB superpages. In other words, processor vendors are already tailoring their TLB architecture to use superpages more aggressively. We believe that it is time to architect L1 caches to follow suit.

## III. MOTIVATION

### A. Impact of Associativity on MPKI

One might expect L1 cache miss rates to reduce appreciably for high levels of associativity. But Fig. 2a shows different trends. We collect memory traces from a collection of workloads from *Spec*, *Parsec*, *Cloudsuite* (i.e., *tunkrank*), and *Biobench* (i.e., *mummer* and *tigr*) as well as other server workloads like *graph500*, the *Nutch* Hadoop workload, the *Olio* social-event web service, the *Redis* key value store, and *MongoDB*. Fig. 2a plots MPKI averaged across the applications as a function of cache associativity, for 16-128KB caches. Increasing associativity beyond 4 does not significantly reduce miss rates. This is because L1 caches are small and service requests only from one hardware context (or two-four if we are using SMT). Low associativity is enough to reduce conflict misses, after which the L1 is limited by capacity misses [7]. In this regard, L1 caches fundamentally differ from LLCs, which are order of magnitude larger and typically require 8-16 ways to mitigate set conflicts for requests from multiple cores. Nevertheless, VIPT constraints dictate that L1 caches can only be grown by increasing associativity despite modest hit rate benefits. Perniciously, access time and energy go up, as we discuss next.

### B. Impact of Associativity on Latency/Energy

We worked with SRAM designers to model state-of-the-art L1 caches. We used an SRAM compiler from for TSMC 28nm [26] to create data and tag arrays, and RTL to handle indexing and tag comparisons across different configurations. We synthesized multiple L1 configurations, varying size and set-associativity using Synopsys Design Compiler. Since L1 caches are tightly coupled to the CPU pipeline, we went for designs that optimize latency (rather than energy) by (i) modeling parallel data and tag lookups to optimize access

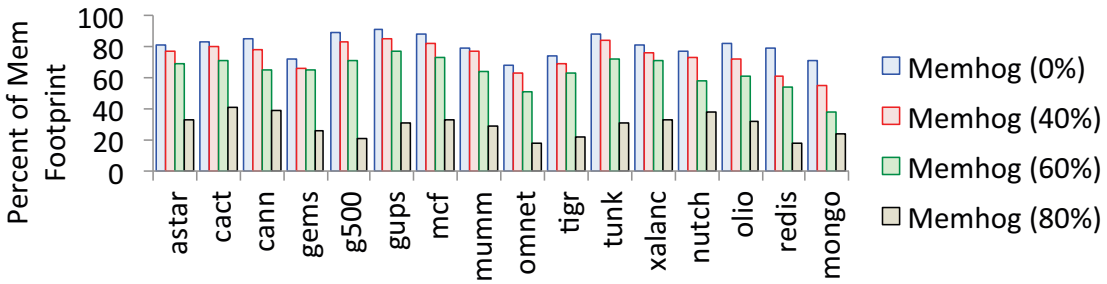



Figure 3: Fraction of total memory footprint allocated with 2MB superpages on a real 32-core Intel Sandybridge system with 32GB of memory. We show how superpage allocation varies as memory is fragmented with the memhog workload.

latency, and (ii) constraining the synthesis tool to aggressively optimize to meet timing, steadily increasing the clock period if it fails.

Fig. 2b plots access latency as we vary 16-128KB caches from direct-mapped to 32-way. Consistent with current systems like Intel's Sandybridge/Skylake, for 32KB 8-way set-associative caches we see 3-4 cycle access latencies. In general, for each cache size, we observe access latency increasing 10-25% at each step with associativity across all cache sizes. Naturally, some of these caches configurations are simply infeasible because of high access latencies – i.e., 64KB/128KB caches with 32-way associativity. It may in fact be more judicious at that point to modify the caches from VIPT to PIPT, with lower associativity, but also lower access latecy, for the best performance. Our quantitative analysis of SEESAW takes these other design points into account.

Fig. 2c plots the total (dynamic and leakage) energy for varying cache sizes and associativity. We found the SRAM data array to be the dominant contributor. The energy graph is correlated with the latency graph with a steady increase with associativity. But the percentage increase can be 40-50% on average at each step. We observed that this is due to the synthesis tool aggressively trying to meet timing, which becomes harder as associativity increases. To validate this, we tried synthesizing all caches at 200MHz (not a viable design point for high-performance caches) and found the latency values to be much higher (given the available slack), and energy increase with associativity flatter.

These trends match Figures 2.3 and 2.4 in Hennessy and Patterson's 5th edition [27] and we also validated them using Cacti 6.5 [28]. We also scaled the cache latency numbers to 22nm and 14nm, using publicly available [29] L1-DCache access latency numbers from Intel's SandyBridge (32nm), IvyBridge (22nm), and Skylake (14nm). We found that absolute cache access times have gone down by 3% and 17% respectively, over the 2 generations. However, the relative trend between associativities remains the same. We use the scaled numbers at 22nm in our evaluations.

This study reveals that increasing associativity beyond a certain point hurts cache access latency and energy without commensurately improving hit rates. The exact associativity at which the latency crosses the target clock period or energy becomes prohibitively high will depend on the technology node. However, L1 caches today cannot be designed for a low associativity since that is dictated by VIPT which constrains the cache's ability to support arbitrary numbers of sets (as Fig. 1 showed). This is the problem SEESAW solves.

### C. Superpages in Modern Systems

To quantify the prevalence of superpages, we profile a 32-core Sandybridge system with 32GB RAM, running Ubuntu Linux with the 4.14 kernel series. We focus on 2MB superpages as modern OSes typically have better support for 2MB pages than 1GB superpages (whose transparent support is an area of active study) [5, 15]. The system had been heavily loaded for over a year with user-level applications and system activity (e.g., networking stacks, etc.). To further load the system, we ran a memory-intensive workload called *memhog* in the background. *Memhog* is a microbenchmark for fragmenting memory that performs random memory allocations, and has been used in many prior works on virtual memory [5, 21, 22, 30, 31]. For example, *memhog (50%)* represents a scenario where *memhog* fragments as much as half of system memory. We enabled Linux's transparent superpage support [13], which attempts to allocate as much anonymous heap memory with 2MB pages as possible. The more fragmented the system, the harder it is for the OS to allocate superpages.

Fig. 3 plots the fraction of the workload's memory footprint allocated to 2MB superpages. With low fragmentation (i.e., *memhog (0-20%)*), as much as 65%+ of the memory footprint is covered by 2MB superpages for every single workload. In many cases, this number is 80%+. Even with non-trivial memory fragmentation (i.e., *memhog (40-60%)*), superpages continue to remain ample. This is not surprising, since Linux – and indeed, other OSes like FreeBSD and Windows – use sophisticated memory defragmentation algorithms to enable superpages even in the presence of non-trivial resource contention from co-running applications. It is only when contention increases dramatically (*memhog (80-90%)*) that OSes struggle to allocate superpages. Nevertheless, even in the extreme cases, some superpages are allocated.

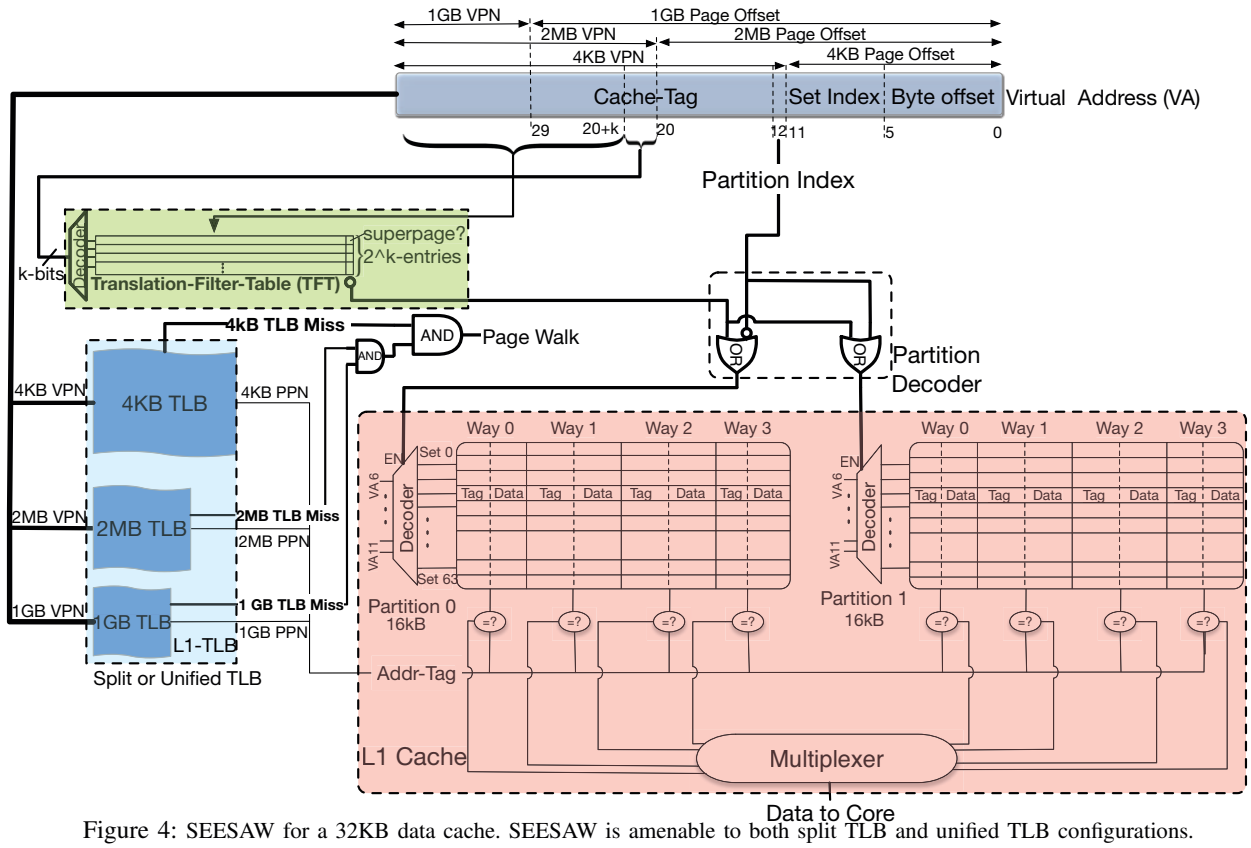



Figure 4: SEESAW for a 32KB data cache. SEESAW is amenable to both split TLB and unified TLB configurations.

We also studied FreeBSD and Windows and we found that 20-60% of the memory footprint of our workloads, running on long-running server systems which have seen lots of memory activity, are covered by superpages. On average, the number is roughly 48%, matching similar observations about superpage prevalence in recent work [20–22].

## IV. SEESAW Microarchitecture

We showcase SEESAW with an example 32KB, 8-way L1 cache operating at 1.33 GHz for x86 with 4KB base pages, and 2MB superpages. We focus on 2MB superpages because transparent OS support for 1GB pages is still in its infancy. However, this approach generalizes readily to 1GB superpages too. Moreover, we report SEESAW's benefits on other cache organizations in Section VI.

### A. Hardware Augmentations

*1) L1 cache microarchitecture:* Fig. 4 shows SEESAW's microarchitecture, which borrows from the well-known idea of way-partitioning. In conventional way-partitioning, a cache line is initially mapped to a set, but it is then also mapped to a subset of the ways in the set. The specific ways that the line can be mapped to depends on the particular hash function that one might use. We propose a variant of this approach to implement SEESAW. In our approach, each set is way-partitioned. The number of ways in each partition is chosen for its desirable latency and energy characteristics. For our example 32KB cache, we use data from Section III-B to target 4 ways per partition per set. We then use the bits immediately more significant than the *set_index* as the *partition_index*. In our example, bit 12 of the virtual address therefore serves as the *partition_index*. Therefore, SEESAW first uses the conventional VIPT approach to select the desired set with the *set_index*. After that, VIPT approaches are different for accesses to lines residing in superpages versus base-pages.

**Superpages:** SEESAW exploits the fact that superpages have wider page offset bits. Since the *partition_index* bits reside in the page offset (i.e., bit 12 falls within the 21-bit page offset for 2MB pages), they can be used to directly index into the desired partition inside the set. The main benefit of this is that only the ways in the partition, rather than the full set, needs to be looked up, saving latency and energy.

**Base-pages:** Base-pages do not have sufficiently wide page offsets to guarantee that the *partition_index* from the virtual page number remain unchanged in the physical address space. Consequently, all ways in all the partitions in the set must be searched, just like traditional VIPT. Note that waiting for address translation to complete first in order to identify

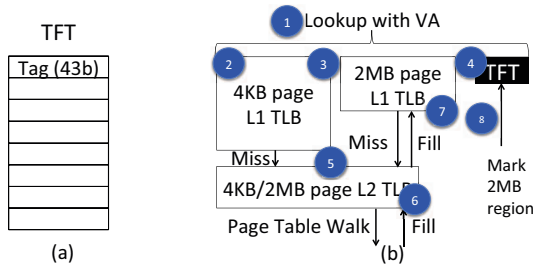



Figure 5: (a) The TFT consists of a list of 2MB virtual address regions that backed by 2MB superpages; and (b) the TFT is looked up in parallel with the L1 TLBs and is updated (or marked) whenever an entry is filled into the 2MB L1 TLB.

*partition_index* defaults to the slower serial PIPT case. We show how SEESAW avoids this subsequently.

We present the policy for indexing into one of the partitions (for superpages) versus reading all ways (for base pages) next. Table III presents the access latencies in the cache for baseline and superpages across various cache sizes and clock frequencies, to point to the robustness of this idea.

*2) TLB-L1 cache interface:* For our 32KB cache example, SEESAW must determine whether bit 12 in the virtual address is unchanged in the physical address to know whether it can be treated as a *partition_index*. This rules out the possibility of waiting until TLB lookup completes to identify whether the access is to a superpage. Instead, SEESAW predicts whether the access is to a superpage in parallel with TLB lookup.

Recent studies investigate hardware page size predictors [4, 25] to predict whether accesses are to superpages or base pages. Most of these approaches have been used to optimize the TLB hierarchy in the past, but could readily be co-opted for SEESAW too. However, we use an even simpler variant that consumes a fraction of the hardware from prior approaches. Figure 5 shows our page size predictor, dubbed the TRANSLATION FILTER TABLE (TFT).

**TFT Structure.** The TFT records a list of 2MB virtual address regions backed by 2MB superpages in physical memory. For 64-bit systems, this means that each entry essentially stores a 43-bit "tag" or identifier for a 2MB region of the virtual address space. If a tag exists in the TFT, this means that this 2MB region is backed by a superpage. Figure 5(a) shows that we use a direct-mapped TFT (although set-associative implementations are possible) as we find that this performs sufficiently well. Furthermore, a small per-core TFT with 16 entries, totaling 86B of memory per core, accurately tracks the vast majority of 2MB superpage accesses. For comparison, this is roughly the size of an 8-entry L1 TLB.

**TFT Lookup.** Figure 5(b) shows how the TFT is looked up in the TLB hierarchy. We show a system with split L1 TLBs for different page sizes but the approach generalizes to any TLB hierarchy (including those with fully-associative L1 TLBs unified for all page sizes). In ①, the CPU pipeline makes a memory reference. The appropriate bits from the virtual address are used to look up the L1 TLBs in ②-③. In parallel, the TFT is looked up in ④ by hashing bits 64-21 of the virtual address (which identify the unique 2MB region of the virtual address space). Many hash functions are possible, but we find that a simple function that performs *VA(64:21) MOD (# of TFT entries)* provides good performance. If the tag matches in the TFT, this confirms that this address is backed by a superpage, and this information is sent to the SEESAW for a fast lookup we explain later.

**TFT Fill.** TFT fills can occur in multiple ways. Consider a scenario where all L1 TLBs miss, prompting lookups of the L2 TLB in ⑤. If the L2 TLBs misses, we perform a page table walk, at which point we know the page size of the desired translation. Suppose that this translation corresponds to a 2MB superpage – as the translation is filled into the L2 and L1 TLBs in ⑥-⑦, we also add this 2MB region into the TFT ⑧. We also permit TFT updates whenever the 2MB page L1 TLB is filled. This includes cases with L2 TLB hits. Since the TFT is direct-mapped, fills kick out the current entry without needing any replacement policy.

SEESAW leverages the TFT as follows. All memory references look up the TLB hierarchy and L1 cache in parallel. However, unlike conventional VIPT, SEESAW performs the L1 cache lookup *speculating a superpage access*; i.e., SEESAW assumes at this stage that the *partition_bit* in the virtual address remains unchanged in the physical address and can thus be used to select the desired partition. In parallel with the lookup of the ways in the partition, the TFT is looked up. Since the TFT is small, it quickly (in about a quarter of the cycle time at 1.33 GHz) establishes whether this virtual address maps to a superpage region. If the TFT suffers a miss, we do not know whether the lookup address is to a superpage or not. Therefore, conservatively, the L1 cache logic begins a lookup of the remaining partitions in the set. Accesses to lines on superpages known by the TFT thus finish faster, while those for base pages takes the same time as traditional VIPT. Table I lists the cache lookup timeline on a case-by-case basis for a 32kB L1 at 1.33GHz on an x86 machine with 4KB base pages and 2MB/1GB superpages. A base-page takes 2-cycle access similar to conventional VIPT, while superpage access takes only 1-cycle. The behavior for other configurations (Table III) and page sizes can accordingly be derived. Note a TFT never sees hits for non-superpage accesses – i.e., the TFT always misses for 4KB page accesses.

*3) Relationship with Way-Partitioning:* SEESAW is readily-implementable because it uses a variant of way-partitioning, which is already available on commercial chips. Traditional way-partitioning is used to achieve resource fairness when caches are accessible by multiple workloads. A hashing function is applied to the memory address, and the output determines the target ways in a set that the line can possibly map to. This approach reduces inter-workload cache

Table I: Anatomy of a lookup using SEESAW.

| Page Size | TFT | Cache | Cycle 1 | Cycle 2 | Savings over Baseline |
|---|---|---|---|---|---|
| 2MB | Hit | Hit | Partition lookup using partition_index (bit 12 of the VA). Tag matches. This is the same case as a traditional VIPT for a 4-way cache. | Not Required. | Latency + Energy |
| 2MB | Hit | Miss | Partition lookup using partition_index (bit 12 of the VA). Tag mismatch triggers cache miss. | Not Required. | Energy |
| 2MB | Miss | * | Partition lookup using partition_index (bit 12 of the VA). TFT miss signal triggers a read of the remaining 4-ways of the adjacent partition (i.e., assume a 4kB page). | Other partition is read. L1 TLB misses trigger Level-2 TLB (if present) lookup which may trigger a page table walk. | None |
| 4KB | Miss | * | Appropriate partition is looked up using the partition_index (bit 12 of the VA). The TFT miss signal triggers a read of the remaining 4-ways of the adjacent partition. | Other partition is read. L1 TLB misses trigger Level-2 TLB (if present) lookup which may trigger a page table walk. | None |

interference and enables QoS guarantees [32]. The downside is an effective associativity reduction.

SEESAW realizes a form of selective way-partitioning; i.e., the superpages are way-partitioned. For our 32KB SEESAW cache example, we use bit 12 as the *partition_bit*. This means that successive 4KB regions in a superpage are strided across the two partitions in each set. However, base pages remain mapped to either of the two partitions.

SEESAW is orthogonal to and can be built on top of traditional way-partitioning. Note that traditional way-partitioning is applied to LLCs, targeting scenarios where a single cache services the requests of multiple workloads (e.g., applications, VMs, foreground versus background tasks, etc.). SEESAW targets per-core VIPT L1 caches instead.

*4) Implementation Overheads:* Our 32KB SEESAW cache example dynamically changes associativity from 8- to 4-way. The additional hardware needed for this is similar to traditional way partitioning. Specifically, we need a partition decoder (two 2:1 OR gates in Fig. 4). We also need an extra 2:1 mux at the end of the partitions choose between them. Within each partition, SEESAW needs a 4:1 mux instead of the 8:1 mux used by baseline VIPT for the entire set. The actual SRAM arrays for data and tags remain unchanged. Moreover, the TFT requires only 86 bytes per core. We implemented a SEESAW cache in RTL using the Synopsys SRAM compiler for TSMC 28nm [26]. Access time increases by less than 1%, leaving cache cycle time unaffected at 1.33GHz. Lookup energy for a 4-way access in SEESAW increases by just 0.41%, which is still 39.43% lower than that for 8-way access in the baseline.

### B. Design Optimizations

*1) Cache Line Insertion Policy:* In our 32KB SEESAW cache example, there are two potential line insertion policies.

① 4way-8way insertion policy: On a cache miss to a line in the superpage (see Table I), the victim line is chosen from the partition that the line maps to. However, if there is a miss to a line in a base page, the replacement victim is chosen across either partition using LRU. SEESAW behaves like a 4-way associative cache for superpages and a 8-way associative cache for base pages from an insertion policy perspective.

② 4way insertion policy: The victim is chosen using LRU from the particular partition that the line maps to, regardless of whether the line is in a superpage or a basepage. The 4way policy uses a local replacement policy within the 4 ways of the concerned partition, instead of a global replacement within 8 ways of the original set, irrespective of page size.

We use 4way insertion in SEESAW for four reasons. First, we use it for correctness, to handle scenarios where a page is mapped both as a base page and a superpage. 4way-8way can lead to the same line getting installed twice in the cache. A uniform insertion policy for both base and superpages avoids this problem. Second, we use 4way insertion for energy reasons. LRU is simpler and saves energy on each cache-line installation due to tracking and lookup of fewer ways. Third, 4way insertion achieves good performance. As an academic exercise, we ran all our experiments with both policies, and noticed only a 1% difference drop in hit rate with the 4way policy, in line with the earlier observations in Fig. 2a. Finally, 4way insertion is useful for coherence lookups by reducing lookup time and energy, as we discuss later in Section IV-C.

*2) Relationship with Way-Prediction:* SEESAW "filters" out lookup of ways in a partition where we know that a superpage cannot reside. This is symbiotic with past work on way-prediction [33, 34]. Way-prediction predicts which way in a cache set is likely to be accessed in the future. When prediction is correct, access energy latency are reduced, as the cache behaves like it is direct-mapped. Past work proposes using schemes with MRU, the PC, or XOR functions to achieve accurate prediction [34]. However, predictor accuracy can vary, with good results when locality is good and poorer results for emerging workloads with poor access locality (e.g., like graph processing).

SEESAW presents an effective additional design point to way-prediction. When accurate, way-prediction can help reduce L1 cache access energy. However, it may not always reduce access latency, since the L1 cache still needs to wait for the TLB access to finish before doing a tag comparison. Combining SEESAW with way-prediction can allow SEESAW to present the right partition to the way-predictor, which can then predict a way within the partition. This can reduce both access latency and energy for superpage accesses. Moreover, SEESAW can also help reduce the way-predictor's misprediction penalty for superpage accesses, as the remaining ways in only that partition need to be looked up, not the entire set. We evaluate the benefits of combining way-prediction with SEESAW in subsequent sections.

*3) Instruction Scheduling Issues:* SEESAW is a general technique that can be used on in-order and out-of-order architectures. On out-of-order architectures, it is important to consider interactions between the instruction scheduling logic and SEESAW. In particular, modern out-of-order architectures speculatively issue program loads/stores assuming that they are cache hits. Instructions dependent on these memory references are scheduled for execution assuming the that, for example, the load will complete cache access in some number of cycles. On a cache miss, the load/store and its dependent instructions are squashed and rescheduled. The challenge with a variable-hit-latency cache like SEESAW is that instruction squashing and reissuing takes some number of cycles. If the difference in cycles between the "fast" and "slow" cache hit times is not sufficient, the primary source of performance overhead can be from instruction squashing. If these overheads become high, they can cripple the benefits of variable-hit-latency caches. This challenge is not unique to SEESAW, other variable-hit-latency optimizations such as way-prediction face it as well in modern CPUs.

To address this, the instruction scheduler in SEESAW begins by assuming the "fast" hit time, since SEESAW inherently speculates that an access is to a superpage. If we discover that the access is actually to a base-page, we squash dependent instructions and replay them, this time assuming the "slow" hit time. We also apply an optimization to this approach to handle situations when superpages are scarce, which otherwise suffer from frequent instruction squashing and rescheduling. Specifically, we add one counter for the superpage TLB which tracks the number of its valid entries. If this value is too low, we know that superpages are scarce. In these cases, the scheduler begins by assuming a "slow" hit time. By sweeping our workloads, we have found that setting the threshold of the counter to a quarter of the number of superpage TLB entries achieves good performance. We implement this policy in our instruction scheduler for out-of-order cores based on Intel's Sandybridge (see Section VI).

It is also possible to always conservatively assume a "slow" hit time in the scheduler, like the baseline. In such a design, a faster hit due to SEESAW may not translate to overall runtime reduction, but will still provide the same energy benefits.

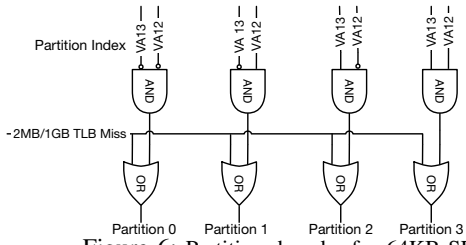


Figure 6: Partition decoder for 64KB SEESAW.

*4) Scalability:* When/if processor vendors scale cache size, they will have to balance factors including (but not restricted to) – desired L1/L2 TLB sizes, maximum permissible L1/L2 TLB latency/energy, maximum permissible L1 cache latency/energy, and so on. Depending on their optimization goals and target workloads, designers may opt for VIVT, PIPT, or VIPT caches. What SEESAW provides is the flexibility of considering an additional design point that achieves the performance and flexibility of VIPT, but does so in a more scalable manner. To that end, SEESAW's operation for larger L1 caches mirrors the description of 32KB L1 caches. The difference is that the number of partitions increases. This changes partition decoder circuitry. Fig. 6 shows a possible circuit of a bank decoder for a 64KB SEESAW cache. Decoders for 128KB caches can be similarly built. The number of ways in each partition is a design choice depending upon the cache's latency-energy profile as a function of associativity. In this work, we assume 4-way (16KB) partition size.

### C. System-Level Issues

*1) Cache coherence:* All coherence lookups (invalidations/probes) use physical addresses. With traditional VIPT, they pay high-associative lookup costs. With SEESAW using the 4-way insertion policy, the physical address can directly identify the correct cache set and partition. Thus SEESAW enables all coherence messages to pay the energy and lookup costs for a 4-way lookup (instead of 8-way). These coherence benefits apply to not just superpages, but also base pages.

*2) Page table modifications:* OSes can modify the page table, which can splinter superpages into base-pages (or vice-versa). SEESAW must handle these changes correctly. Suppose that a superpage is broken into base pages. Lines that belonged to the superpage must remain accessible. SEESAW naturally achieves this as accesses to base-pages automatically look up the partition that the superpage originally mapped to. Additionally, however, the old 2MB superpage may have an entry in the TFT, which must be invalidated. This can be easily accommodated however; when a 2MB page is splintered, the OS code executes a software instruction to invalidate the corresponding TLB entries (e.g., *invlpg* in x86-64, or *tlbi* in ARM) , which are now stale. These instructions take as an argument the virtual page number. We bootstrap off this existing instruction and modify the microarchitecture

to invalidate the TFT entry tagged with this virtual page number as well, in parallel with the TLBs. Therefore, there are no correctness issues when 2MB pages are splintered to 4KB pages.

Alternately, several base pages may be promoted to create a superpage. Since SEESAW probes fewer ways, it is possible a line from one of the prior base pages may be cached in a partition that is no longer probed, which is a problem if that line is dirty. We use a simple solution for this. Normally, when the OS promotes base pages to a superpage, it has to invalidate all the base page translation entries in the page table. For correctness, OSes then execute TLB invalidation instructions. These instructions take 150-200 cycles to execute (we determine these latencies using microbenchmarks, and these numbers are consistent with measurements form Linux kernel developers). We extend this instruction so that it triggers a sweep of the L1 cache, evicting all lines mapping to each invalidated base page. We have found 150-200 cycles ample to perform a full cache sweep. We model such activities in our evaluation infrastructure and find that page table modifications events only minimally affect performance.

*3) TFT Process IDs:* Modern TLBs are ASID-tagged, meaning that they need not be flushed on context switches [22, 31]. Consequently, we studied the need to add ASIDs to the TFT. We ultimately found that the area overheads were high (i.e., almost doubling total area). We also found that the performance overheads of not supporting ASIDs and having to flush the TFT on context switches were less than 1% of total performance. Therefore, we opted for a TFT without ASID tags.

## V. METHODOLOGY

**Simulator.** We use a Simics-driven cycle-accurate software simulator to model SEESAW for both out-of-order and in-order cores (modeled similar to Intel Sandybridge and Atom respectively) [35]. Table II presents our target system. For each case, we evaluate three sizes of L1 data caches (32KB, 64KB, and 128KB) and three operating frequencies (1.33GHz, 2.80GHz, and 4GHz). Table III shows access latency of each configuration. Although not shown, we find a 16-entry TFT, totaling 86 bytes of memory per core, to work well for all these designs. This structure can be accessed in a single cycle at all frequencies. Furthermore, the numbers are scaled to 22nm from our 28nm SRAM results presented earlier in Section III-B using standard scaling factors [36] to be consistent with 22nm models for other components in our simulator. Note that our design assumes split TLBs, as are used on Sandybridge and Atom processors, but SEESAW works with unified TLBs too. We assess performance improvements by considering improvements in IPC, and also extract energy improvements of the entire memory hierarchy (the L1 cache, as well other caches and memory). We apply SEESAW on the data cache, although it is also possible to apply it to the instruction cache. This may be valuable with the advent of cloud workloads [37, 38] that use considerably larger instruction-side footprints.

Table II: System Parameters.

| | |
|---|---|
| **CPU Models** | |
| **Out-of-Order** | ∼Intel Sandybridge: 168-entry ROB, 54-entry Instruction Scheduler, 16 byte I-fetches per cycle |
| **In-order** | ∼Intel Atom: Dual-Issue, 16-stage pipeline |
| **Memory System** | |
| **L1 Cache** | Private Split L1I (32kB) + L1D (Table III) |
| **TLB (Atom)** | L1 (64-entry for 4kB, 32-entry for 2MB), 512-entry L2 |
| **TLB (Sbridge)** | Split L1 (128-entry for 4kB, 16-entry for 2MB) |
| **LLC** | Unified, 24MB |
| **DRAM** | 4GB, 51ns round-trip access latency |
| **System Parameters** | |
| **Technology** | 22nm |
| **Frequency** | 1.33 GHz, 2.80 GHz, 4.0 GHz |
| **Cores** | 32, 64, 128 |
| **Coherence** | MOESI directory |

As Table III shows, the larger caches support unacceptably high access latencies (e.g., 128KB, 32-way caches have 42-cycle access latencies). Naturally, one might consider alternate designs at this point, such as PIPT caches with lower associativity. Our quantitative evaluation compares SEESAW against such alternatives.

**Workloads.** We collect memory traces from a real Sandybridge system running Ubuntu v4.4. To ensure that our system has memory fragmentation, and hence superpage allocation frequency of realistic servers with long uptimes, we collect traces on a system that has been running for several months. Our workloads are a mix of Spec [39], Parsec [40], Cloudsuite [37], and other important cloud workloads like *Olio*, *Nutch*, *Redis*, *MongoDB*, and *graph500*. We use a modified version of Pin [41] to collect traces. Each trace contains 10-billion instructions from the target application, in addition to instructions of other applications running in parallel, such as memhog (Sec. III-C, and system-level activity. This lets us study the performance of SEESAW in the presence of real OS activity affecting superpage allocation (such as fragmentation and defragmentation) rather than in in isolation (which might undersell or oversell the scheme).

We profile the percentage of the memory references that are to lines in superpages and discover that this number always ranges from 53-95% in our workloads. Generally, workloads like *Nutch*, *Olio*, *Redis*, *MongoDB*, *graph500*, and *tunkrank*, which are representative of modern cloud workloads, see 70-95% of their references going to superpages.

## VI. EVALUATION

### A. Performance

Fig. 7-Fig. 9 plot percent runtime improvements in the out-of-order and in-order architectures we model. To plot all these data points, we performed three experiments. First,

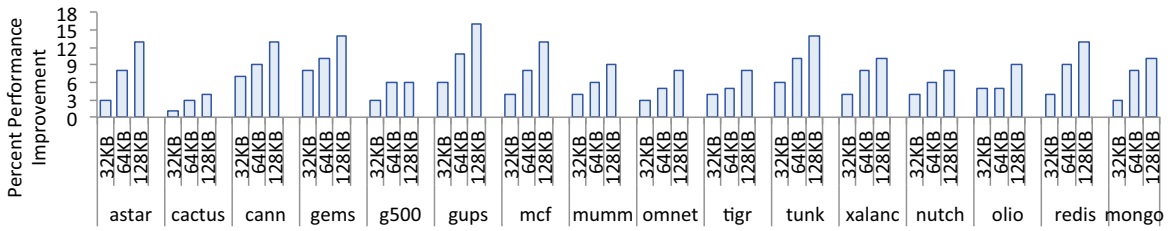



Figure 7: Percent improvement in runtime using SEESAW on an OoO processor versus baseline VIPT. (Freq = 1.33GHz, L1 cache: 32KB to 128KB.

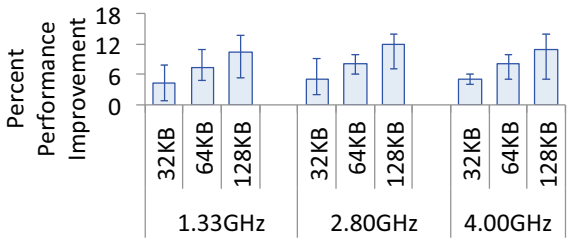



Figure 8: Percent improvement in runtime using SEESAW on an out-of-order processor versus baseline VIPT. We show the average, minimum, and maximum improvements across all workloads for all cache sizes, across 1.33GHz, 2.80GHz, and 4.00GHz operating frequency.

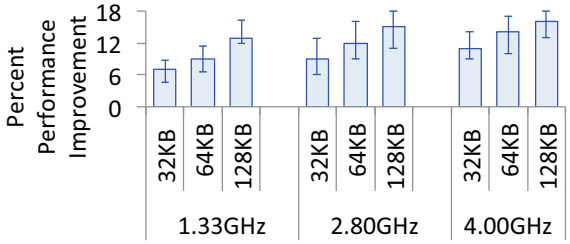



Figure 9: Runtime improvement using SEESAW on an in-order processor versus baseline VIPT. We show the average, minimum, and maximum improvements across all workloads for all cache sizes, across 1.33GHz, 2.80GHz, and 4.00GHz operating frequency. Benefits are higher on an in-order versus out-of-order processor.

we quantified performance on the baselines presented in V. Second, we measured the area overhead of SEESAW (i.e., TFTs, additional cache circuitry, etc.) and added this area to the L1 cache in the baseline (i.e., it does not perform support SEESAW). Third we quantified the performance of SEESAW. The objective of the second experiment was to make sure that the area spent on SEESAW was worth it; i.e., we would not have been better off simply making the baseline L1 cache bigger. We found that the second scenario improved performance over the baseline by less than 0.01% in all cases. We therefore ignore these results for the remainder of this evaluation, focusing on SEESAW instead.

Fig. 7 shows performance improvements on an out-of-order processor as we vary cache size, for a fixed 1.33GHz clock. Every single one of our workloads benefits from SEESAW, despite the fact that the variable-hit-latencies can create some amount of instruction squashing and rescheduling. Generally,

Table III: L1 Cache Configurations.

| | | | Access Latency (cycles) | | |
|---|---|---|---|---|---|
| **Cache Size (kB)** | **VIPT Assoc-iativity** | **Fre-quency (GHz)** | **TFT** | **L1 base-page** | **L1 super-page** |
| 32 | 8 | 1.33 | 1 | 2 | 1 |
| 32 | 8 | 2.80 | 1 | 4 | 2 |
| 32 | 8 | 4.00 | 1 | 5 | 3 |
| 64 | 16 | 1.33 | 1 | 5 | 1 |
| 64 | 16 | 2.80 | 1 | 9 | 2 |
| 64 | 16 | 4.00 | 1 | 13 | 3 |
| 128 | 32 | 1.33 | 1 | 14 | 2 |
| 128 | 32 | 2.80 | 1 | 30 | 3 |
| 128 | 32 | 4.00 | 1 | 42 | 4 |

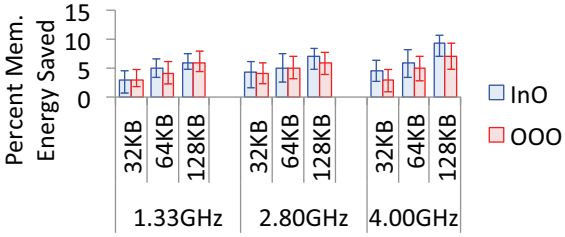



Figure 10: Percent improvements in the energy spent on the entire memory hierarchy using SEESAW compared to a baseline VIPT. We separate in-order (InO) and out-of-order (OOO) results.

the larger the cache, the more the performance improvement since baseline VIPT becomes even more highly-associative and slow in these cases. Average performance improvements range from 5-11% for 32-128KB caches. Furthermore, these benefits are particularly notable for emerging cloud workloads like *redis*, *olio*, *tunkrank*, and *mongoDB*. Furthermore, Fig. 8 shows that these performance benefits can increase with higher clock frequency as the number of cycles taken for the baseline VIPT grows in these cases.

Fig. 9 shows that these performance benefits are even higher on in-order cores. This makes sense, since L1 cache access latency cannot be overlapped with useful work via out-of-order techniques. Hence, SEESAW achieves 3-5% higher performance on in-order cores versus out-of-order cores.

### B. Energy

Fig. 10 quantifies SEESAW's energy benefits. We plot average, minimum, and maximum percent improvements while varying cache sizes from 32KB to 128KB, and frequencies from 1.33GHz to 4.00GHz. Note that we focus on the energy expended on the entire memory hierarchy (rather than just the L1 cache), since changes to L1 cache

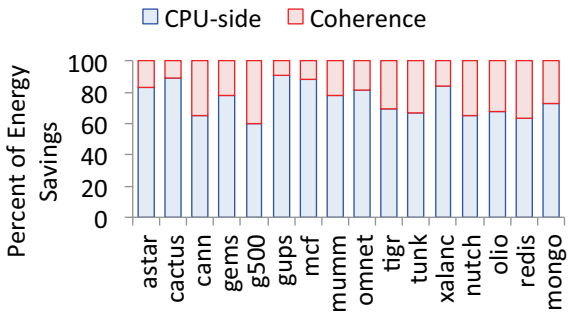



Figure 11: Percent of the energy savings that can be attributed to savings in CPU-side lookups versus coherence lookups. Results are collected on an out-of-order system with 64KB L1 caches at 1.33GHz.

hit rates can affect access rates and energy of the bigger caches and memory. We separate results for in-order and out-of-order processors.

Fig. 10 SEESAW always improves the energy spent on the memory hierarchy. Generally, in-order processors save slightly more energy, but out-of-order processors do well too. Energy savings come from a variety of sources. Workloads whose memory footprints fit comfortably in the L1 cache (e.g., the Spec applications) benefit mostly from savings in dynamic access energy from SEESAW. Other workloads with larger working sets, like *Redis*, *MongoDB*, and *Olio* benefit from both dynamic access energy and decreased leakage energy because the application runs faster. Energy savings can be attributed to both savings in CPU-side accesses as well as coherence lookups. Fig. 11 showcases the fraction of energy savings that are attributed to savings from CPU-side lookups versus coherence lookups. We show per-workload results for 64KB caches at 1.33GHz for the out-of-order system but these trends remain the same for other organizations.

Fig. 11 shows that *all* applications, whether they are single- or multi-threaded, benefit from more energy-efficient CPU-side lookups but also coherence lookups. For example, over 10% of the energy savings of applications like *astar* and *mcf* are from coherence lookups. This is because our simulation framework also captures cache coherence from not just the application, but also system-level activity (e.g., the OS, the network stack, etc.) that can exercise the coherence protocol. Naturally, coherence savings become even more important for multi-threaded applications. Roughly a third of the energy benefits of workloads like *canneal* and *tunkrank* come from energy-efficient coherence lookups. Note, furthermore, that these results are collected using a directory-based cache coherence protocol, where the coherence directory eliminates many spurious L1 cache coherence lookups. We have also experimented with snoopy coherence protocols and find that in many cases, energy improvements can increase by an additional 2-5% in multi-threaded applications.

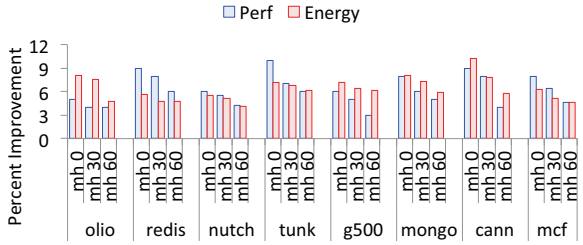



Figure 12: We vary memory fragmentation using memhog, which uses 0%, 30%, and 60% of memory. We show percent performance and energy savings on the memory hierarchy by using SEESAW.

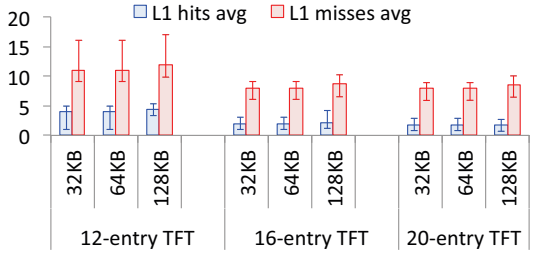



Figure 13: Percentage of superpage accesses that are missed by the TFT, for varying TFT size (12- to 20-entry) and cache size (32KB, 64KB, 128KB). Results are separated for TFT misses for accesses that hit and miss in the data cache. Average, min, and max results shown.

### C. Effect of Memory Fragmentation

We have also studied SEESAW's benefits as load is stressed further, making it even harder for OSes to allocate superpages. While the results presented thus far are already on a system with significant load and uptime, Fig. 12 shows what happens when load is exacerbated. These experiments focus on 64KB L1 caches at 1.33GHz. While we focus on five important cloud workloads, we see the same trends for our other workloads too. For each workload, we vary fragmentation levels by running, like prior work [5, 22, 30, 31], a memory-intensive microbenchmark (i.e., *memhog*), giving it 0%, 30%, and 60% of the total memory capacity. The more memory given to *memhog*, the more challenging it is for Linux to allocate superpages. For example, *Olio* and *Redis* see over 80%, 72-76%, and 48-52% of their memory footprint covered by superpages when giving *memhog* 0%, 30%, and 60% of the total memory area. Despite this, however, SEESAW continues to provide energy benefits even as the number of superpages decreases. As expected, the raw performance and energy benefits decrease but still remain in the 4-6% range in the presence of heavy fragmentation (i.e., *memhog* of 60%). In other words, OS support for superpages has improved sufficiently that even under high memory load, there are ample superpages allocated for SEESAW to be useful.

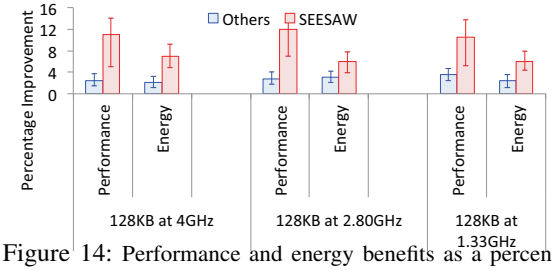



Figure 14: Performance and energy benefits as a percentage of the baseline configuration for SEESAW versus other approaches (e.g., PIPT with varying associativity, varying TLB sizes, etc.)

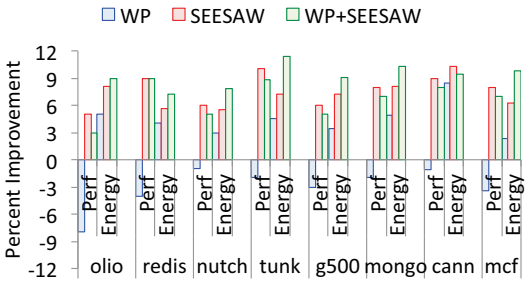



Figure 15: Performance and energy improvements, versus a baseline 64KB VIPT cache at 1.33GHz. We show way-predictor data (WP), SEESAW and a combined approach (WP+SEESAW).

### D. TFT Analysis

TFT hit rates are critical to the success of SEESAW. The key metric to measure is the fraction of cache accesses to superpage regions that the TFT does not identify as superpage accesses. These are the only situations where SEESAW looks up more partitions (and ways) in the data cache than it needs to. Figure 13 quantifies this metric by showing the percentage of total superpage accesses missed by the TFT. We vary TFT size from 12- to 20-entry, and show data cache sizes changing from 32KB-128KB. Furthermore, we separate the TFT misses into situations where the memory reference is ultimately discovered in the data cache (the blue bars or L1 hits) and those that miss (the red bars or L1 misses).

As shown, a TFT size of 16-entry drives miss rates to under 10% even in the worst case. Beyond this, a 20-entry TFT does not yield much better prediction rates (and although not shown, this slightly higher prediction rate yields less than 0.1% additional performance). This is why we settle on a 16-entry TFT. Furthermore, Figure 13 shows that by far, the bulk of the TFT misses occur to superpage accesses that actually *miss* in the L1 cache. This means that even though we pay the penalty of the additional partition access in SEESAW, this additional latency pales in comparison to the subsequent L2 cache lookup (and beyond). Consequently, most of these misses do not adversely impact performance.

### E. Scaling L1 Caches

When we presented the configurations in Table III, we found that as we scale L1 caches, baseline VIPT approaches had unacceptably high access latencies; e.g., Baseline cache access latencies for 128KB caches at 1.33GHz, 2.80GHz, and 4.00GHz were 14, 30, and 42 cycles respectively. While SEESAW does improve the performance of these approaches substantially, one might imagine other ways to do so too; e.g., changing the cache from VIPT to PIPT and reducing any combination of associativity or TLB size to save access latency. We have swept through a range of such design parameters (i.e., a range of PIPT associativities and TLB sizes) and compare our results to SEESAW in Figure 14. We separate performance and energy benefits of the two approaches (showing average and min/max with the error bars). SEESAW consistently outperforms these approaches because it strikes a balance – it can accommodate higher cache associativity (eking out slightly better cache hit rates) while achieving similar cache access latency as these other approaches. Moreover it does so without shrinking TLB sizes (which reduces TLB hit rates), which other approaches frequently need to do. Consequently, SEESAW achieves both better performance and energy savings than these other approaches.

### F. Impact of Way Prediction

We also quantify SEESAW's relationship with way-prediction strategies. We have studied 32KB-128KB caches on an out-of-order processor at our three target frequencies. Because trends are similar, we focus on 64KB caches at 1.33GHz for this discussion. Fig. 15 compares the percent performance and energy improvements that an MRU-based way predictor based on prior work [33] achieves, compared to SEESAW. We also consider the case where the way-predictor and SEESAW are combined (WP+SEESAW). All results are normalized to a baseline VIPT L1 cache. Furthermore, because of a lack of space, we show results for five of our cloud workloads, although the trends hold across the other workloads too.

Fig. 15 shows that, as expected, the way predictor alone degrades performance. This is expected since way-prediction trades access latency for better performance. When prediction accuracy is good (e.g., for nutch which has prediction accuracy over 85%), performance degradation is marginal. But when MRU prediction suffers because workloads use pointer-chasing memory access patterns with poorer access locality (e.g., *graph500* and *olio*), way prediction can increase runtime significantly. In contrast, SEESAW never degrades performance. At worst, it maintains baseline performance in the absence of superpages. Far more commonly (as Fig. 15 shows), performance improves.

Second, way-prediction can improve energy. However, SEESAW typically saves even more energy since superpages are ample. The best energy savings are achieved when it is combined with way-prediction. We have found this trend to be consistent across all the workloads. The key is that

SEESAW is able to counteract latency overheads when the way-predictor mispredicts. It also achieves energy savings in these cases, when the access is to a superpage. We intend studying advanced schemes that dynamically choose when to combine SEESAW and way-prediction, in future work.

## VII. Related Work

Perhaps closest in spirit to our work is recent work on speculatively-indexed physically-tagged (SIPT) caches [42]. Like our study, this work shows that VIPT constraints force designers to grow L1 caches by increasing associativity, which can hurt access time and energy. Unlike our work, the authors of the SIPT paper approach this problem by speculating/predicting which bits in memory addresses remain in the same in virtual and physical addresses. We view this approach as related – however, the reliance on speculation and rollback mechanisms presents a separate set of design challenges than our work.

VIVT caches [1, 2, 8, 11, 43] are an alternative to VIPT, obviating the need for TLB lookup before L1 cache access. While VIVT caches are attractive because they decouple the TLB and L1 cache, they are hard to implement of problems with synonyms, multi-processing, context switches, and interactions with cache coherence protocols which operate on physical addresses. Recent work does present effective solutions to the problems of synonyms [1, 8] and cache coherence [43], but requires non-trivial modifications to L1 cache and datapath design. *Opportunistic virtual caching* [7] proposes an L1 cache design that caches some lines with virtual addresses, and others (belonging to synonym virtual pages) as physical addresses.

## VIII. Conclusion

L1 caches are critical for system performance as they service every cacheable memory access from the CPU and coherence lookups from the memory hierarchy. L1 caches desire fast lookups, low access energy, high hit rates, and simplicity of implementation. In this work, we identify the opportunity presented by superpages, to optimize current VIPT L1 caches. Our design, SEESAW, provides performance improvements, and energy reduction for all L1 lookups, whether they are initiated by the CPU or coherence activity.

## IX. Acknowledgements

We thank the National Science Foundation, which partially supported this work through grants 1253700 and 1337147. We thank Jan Vesely, Zi Yan, Guilherme Cox, and Gabriel Loh for their feedback on this work.

## References

[1] C. Park *et al.*, "Efficient synonym filtering and scalable delayed translation for hybrid virtual caching," in *ISCA*, June, 2016.

[2] B. Jacob, "Segmented addressing solves the virtual cache synonym problem," in *Technical Report UMD-SCA-1997-01 December, 1997*, 1997.

[3] P. Mishra *et al.*, "A study of out-of-order completion for the mips r10k superscalar processor," in *Technical Report, UC Irvine, 01-06*, 2001.

[4] B. Pham *et al.*, "Tlb shootdown mitigation for low-power many-core servers with l1 virtual caches," in *Computer Architecture Letters (CAL)*, 2017.

[5] ——, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *MICRO*, 2015.

[6] V. Seshadri *et al.*, "Page overlays: An enhanced virtual memory framework to enable fine-grained memory management," in *ISCA*, 2016.

[7] A. Basu *et al.*, "Reducing memory reference energy with opportunistic virtual caching," in *ISCA*, 2012.

[8] H. Yoon and G. S. Sohi, "Revisiting virtual l1 caches: A practical design using dynamic synonym remapping," in *HPCA*, 2016.

[9] W. Wang *et al.*, "Organization and performance of a two-level virtual-real cache hierarchy," in *ISCA*, June, 1989.

[10] D. Wood *et al.*, "An in-cache address translation mechanism," in *ISCA*, 1986.

[11] A. Hsia *et al.*, "Energy-efficient synonym data detection and consistency for virtual cache," in *Microprocessors and microsystems*, 2016.

[12] X. Qiu and M. Dubois, "The synonym lookaside buffer: A solution to the synonym problem in virtual caches," in *IEEE Transactions on Computers*, 2008.

[13] A. Arcangeli, "Transparent hugepage support," in *High Performance Computing track San Francisco, CA*, 2011.

[14] J. Navarro *et al.*, "Practical, transparent operating system support for superpages," in *OSDI*, 2002.

[15] Y. Kwon *et al.*, "Coordinated and efficient huge page management with ingens," in *ISOSDI*, 2016.

[16] J. Navarro *et al.*, "Practical, transparent operating system support for superpages," in *Operating Systems Review (ACM)*, 2002.

[17] A. Basu *et al.*, "Efficient virtual memory for big memory servers," in *ISCA*, 2013.

[18] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, June, 2016.

[19] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," *ACM SIGPLAN Notices*, 1994.

[20] F. Gaud *et al.*, "Large pages may be harmful on numa systems," in *USENIX ATC*, 2014.

[21] A. Bhattacharjee, "Translation-triggered prefetching," in *ASPLOS*, 2017.

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

[23] E. Witchel *et al.*, "Mondrian memory protection," in *ASPLOS*, 2002.
[24] X. Dong *et al.*, "Shared address translation revisited," in *Proceedings of EuroSys*, 2016.
[25] M. Papadopoulou *et al.*, "Prediction-based superpage-friendly tlb designs," in *HPCA*, 2015.
[26] Synopsys, "DesignWare IP Embedded Memory for TSMC 28-nm," https://www.synopsys.com/dw/doc.php/ds/es/DW-28-nm-DS.pdf.
[27] J. Hennessy and D. Patterson, "Computer architecture: A quantitative approach," in *Morgan Kauffman*, 2012.
[28] N. Muralimanohar *et al.*, "Cacti 6.0: A tool to model large caches," in *HP Laboratories*, 2009.
[29] Intel, "Intel sandybridge, ivybridge, haswell, skylake," http://www.7-cpu.com/cpu/SandyBridge.html.
[30] B. Pham *et al.*, "Colt: Coalesced large-reach tlbs," in *MICRO*, 2012.
[31] ——, "Increasing tlb reach by exploiting clustering in page translations," in *HPCA*, 2014.
[32] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient cache partitioning," in *ISCA*, 2011.
[33] M. Powell *et al.*, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *ISCA*, 2001.
[34] F. Sleiman *et al.*, "Embedded way prediction for last-level caches," in *ICCD*, 2012.
[35] "Wind river simics," https://www.windriver.com/products/simics/.
[36] D. Chinnery and K. Keutzer, *Closing the gap between ASIC & custom: tools and techniques for high-performance ASIC design*. Springer Science & Business Media, 2002.
[37] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *ASPLOS*, 2012.
[38] L. Wang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *HPCA*, 2014.
[39] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, 2006.
[40] C. Bienia *et al.*, "The parsec benchmark suite: characterization and architectural implications," in *PACT*, 2008.
[41] C. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *CPLDI*, 2005.
[42] Z. Tianhao *et al.*, "Sipt: Speculatively indexed, physically tagged caches," in *HPCA*, 2018.
[43] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *ISCA*, 2013.