Reducing Memory Reference Energy with Opportunistic Virtual Caching

Arkaprava Basu Mark D. Hill Michael M. Swift
University of Wisconsin-Madison
{basu, markhill, swift}@cs.wisc.edu

Abstract

Most modern cores perform a highly-associative translation look aside buffer (TLB) lookup on every memory access. These designs often hide the TLB lookup latency by overlapping it with L1 cache access, but this overlap does not hide the power dissipated by TLB lookups. It can even exacerbate the power dissipation by requiring higher associativity L1 cache. With today's concern for power dissipation, designs could instead adopt a virtual L1 cache, wherein TLB access power is dissipated only after L1 cache misses. Unfortunately, virtual caches have compatibility issues, such as supporting writeable synonyms and x86's physical page table walker.

This work proposes an Opportunistic Virtual Cache (OVC) that exposes virtual caching as a dynamic optimization by allowing some memory blocks to be cached with virtual addresses and others with physical addresses. OVC relies on small OS changes to signal which pages can use virtual caching (e.g., no writeable synonyms), but defaults to physical caching for compatibility. We show OVC's promise with analysis that finds virtual cache problems exist, but are dynamically rare. We change 240 lines in Linux 2.6.28 to enable OVC. On experiments with Parsec and commercial workloads, the resulting system saves 94-99% of TLB lookup energy and nearly 23% of L1 cache dynamic lookup energy.

1 Introduction

The current focus on energy efficiency motivates reexamining processor design decisions from the previous performance-first era, including considering some optimizations that challenge compatibility across layers. To this end, this paper uses small virtual memory changes to save substantial translation lookaside buffer (TLB) and L1 cache lookup power.

Almost all commercial processors today cache data and instructions using physical addresses and consult a TLB on every load, store, and instruction fetch. Thus, a TLB access must be performed for each cache access. However, processor designs are increasingly constrained by power, and physically addressed caches lead to energy dissipation inefficiencies. TLB lookup must be fast, rarely miss—often necessitating an energy-hungry highly associative structure. Industrial sources report that 3-13% of core power (including caches) is due to TLB [33], and an early study finds that TLB power can be as high as 15-17% of the chip power [16,18]. Our own analysis shows that a TLB lookup can consume 20-38% of the energy of an L1 cache lookup.

Energy consumption is further exacerbated by efforts to reduce the critical-path latency of a cache access. Most current processors overlap the TLB lookup with indexing the L1 cache and use the TLB output during tag comparison [25,29]. Such a virtually indexed, physically tagged cache requires that the virtual index bits equal the physical index bits, which is only true if the index comes from the page offset. Thus, the L1 cache size divided by its associativity must be less than or equal to the page size. To satisfy this constraint, some L1 cache designs use a larger associativity than needed for good miss rates (e.g., 64KB L1/4KB page size = 16-way), which leads to higher energy consumption (Section 2.1).

Now that energy is a key constraint, it is worth revisiting *virtual caching* [5,38]. While most past virtual cache research focused on its latency benefit [5,13,35], we focus on its potential energy benefits. A virtual L1 cache is accessed with virtual addresses and thus requires address translation only on cache misses. This design makes TLB accesses much less frequent and reduces its energy consumption substantially. Further, it can lower the L1 cache lookup energy by removing the associativity constraint on the L1 cache design described above.

However, virtual caches present several challenges that have hindered their adoption. First, a physical address may map to multiple virtual addresses (called synonyms). An update to one synonym must be reflected in all others, which could be cached in different places. Thus it requires additional hardware or

software support to guarantee correctness. Second, virtual caches store page permissions with each cache block, so that these can be checked on cache hits without a TLB access. When page permissions change, associated cache blocks must be updated or invalidated beyond the normal TLB invalidation. Third, virtual caches require extra mechanisms to disambiguate homonyms (a single virtual address mapped to different physical pages). Fourth, they pose challenges in maintaining coherence, as coherence is traditionally enforced using physical addresses. Finally, virtual caches can be incompatible with commercially important architectures. For example, the x86 page-table walker uses physical addresses to find page-table entries [40], which creates problem for caching entries by virtual address.

We find, though, that many of these problems occur rarely in practice. We analyze the behavior of applications running on real hardware with Linux operating system to understand how synonyms are actually used and to measure the frequency and character of page permission changes. As detailed in Section 3, we find that synonyms *are* present in most processes, but account for only 0-9% of static pages and 0-13% of dynamic references. Furthermore, 95-100% of synonym pages are read-only, for which update inconsistencies are not possible. We also find that page permission changes are relatively rare and most often involve all pages of a process, which allows permission coherence to be maintained through cache flushes at low overhead. Thus, a virtual cache, even without synonym support, could perform well, save energy, and almost always work correctly.

Since correctness must be absolute, we instead propose a best-of-both-worlds approach with opportunistic *virtual caching* (OVC) that exposes virtual caching as a *dynamic optimization* rather than a hardware design point. OVC hardware can cache a block with either a virtual or physical address. Rather than provide complex support for synonyms in hardware [13,35] or enforce limits on which virtual addresses can be synonyms [23], OVC requires that the OS (with optional hints from applications) declare which addresses are not subject to read-write synonyms and can use virtual caching; all others use physical addresses and a normal TLB. This flexibility provides 100% compatibility with existing software by defaulting to physical caching. The OS can then save energy by enabling virtual caching when it is safe (*i.e.*, no read-write synonyms) and efficient (*i.e.*, few permission changes). OVC provides a graceful software adoption strategy, where OVC can initially be disabled, then used only in simple cases (*e.g.*, read-only and private pages) and later extended to more complex uses (*e.g.*, OS page caches).

With simple modifications to Linux (240 lines of code), our evaluation shows that OVC can eliminate 94-99% of TLB lookup energy and saves more than 23% of L1 cache dynamic energy compared to a virtually indexed, physically tagged cache.

This paper makes three contributions. First, we analyze modern workloads on real hardware to understand virtual memory behavior. Second, based on this analysis, we develop policies and mechanisms that use physical caching for backward compatibility, but virtual caching to save energy by avoiding many address translations. Third, we develop necessary low-level mechanisms for realizing OVC.

2 Background and Motivation

In this section we first describe the benefits and limitations of existing physically and virtually addressed L1 caches.

2.1 Physically Addressed Caches

A physical L1 cache requires the address translation to finish before a cache lookup can be completed. In one possible design, the address translation completes before L1 cache lookup starts, which places the entire TLB lookup latency in the critical path. However, a more common design is to overlap the TLB lookup with the cache access [25,29]. The processor sends the virtual page number to the TLB for translation while sending the page offset to the cache for indexing into the correct set. Then the output of the TLB is used to find a matching way in the set. Such a design is termed a virtually indexed/physically tagged cache. In both designs, all cache accesses, both instruction and data, require a TLB lookup. When latency (and thus performance) is the single most important design objective, a virtually indexed/physically tagged design is attractive as it hides the TLB lookup latency from the critical path of cache lookups while avoiding the complexities of implementing a virtual cache.

In the remainder of the paper, we focus on the second commonly used design. Henceforth we use the term *physical cache* to refer to a virtually indexed and physically tagged cache.

When power is a first-class design constraint, two aspects of this design lead to higher energy consumption. First, TLB lookups are energy-hungry: they occur frequently and often use a highly associative (or even fully associative) design [3,25].

L1 Data Misses per 1K Cache references			
	4-way	8-way	16-way
Parsec	34.400	33.885	33.894
Commercial	41.634	40.721	39.636
L1 Instr. Misses per 1K Cache references			
	4-way	8-way	16-way
Parsec	1.053	0.991	0.934
Commercial	12.202	12.011	11.938

Table 1. L1 cache (32KB, 64B block) miss ratios with varying associativity.

Limiting associativity can reduce energy consumption, but complicates support for multiple page sizes since indexing bits in a set-associative structure depends upon the page size, which is unknown until the translation completes [34]. TLBs can also cause to thermal hotspots due to high power density [31]. As increasing working sets put further pressure on TLB reach [2], processors may require yet larger TLBs, thus making TLB energy consumption worse. Second, to allow indexing with virtual addresses, the address bits used for cache indexing must be part of the page offset. This requires that cache size ÷ associativity ≤ page size. For example, a 32KB L1 cache requires at least an 8-way set-associative design for 4KB pages.

With the method explained in Section 5.2, we empirically find that such a highly associative L1 cache can lead to energy-inefficient cache designs that provide little hit-rate improvement benefit from their increased associativity. Table 1 shows the number of misses per 1K cache references (MPKR) for Parsec and commercial workloads for a 32 KB L1 cache with varying associativity. For example, when associativity increases from 4 to 8 the MPKR changes very little (e.g., < 1 MPKR). Even ignoring extra latency of higher-associativity lookups, this can at best lead to a 0.2-0.4% speedup when associativity is increased from 4 to 8 and 16 respectively (Table 3). However, Table 2 shows that a cache lookup consumes 30% more energy when the associativity is increased from 4 to 8, and 86% more energy when increased from 4 to 16. While extra misses can burn more energy due to access to lower-level caches, the high L1 hit rates makes this a non-issue. This data shows that designing higher-associativity caches to overlap TLB latency can lead to energy-inefficient L1 caches.

	4-way	8-way	16-way
Read Dynamic Energy	1	1.309	1.858
Write Dynamic Energy	1	1.111	1.296

Table 2. Normalized energy per access to L1 (32KB) with varying associativity (w.r.t. 4-way).

	4-way	8-way	16-way
Parsec	1	0.998	0.998
Commercial	1	0.998	0.996

Table 3. Normalized run time with varying L1 associativity (w.r.t. 4-way).

2.2 Virtually Addressed Caches

A virtual L1 cache is both indexed and tagged by virtual address, and consequently does not require address translation to complete a cache hit. Instead, virtual caches consult a TLB on a miss to pass the physical addresses to the next level in the cache hierarchy. The primary advantage of this design is that TLB lookups are required only on misses. L1 cache hit rates are generally high and thus a virtual L1 cache acts as an effective energy filter on the TLB. Moreover, a virtual L1 cache removes the size and associativity constraints on the L1, which enables more energy-efficient designs.

However, several decades of research on virtual caches have showed that they are hard:

Synonyms: Virtual-memory synonyms arise when multiple, different virtual addresses map to the same physical address. These synonyms can reside in multiple places (sets) in the cache under different virtual addresses. If one synonym of a block is modified, access to other synonyms with different virtual addresses may return stale data.

Homonyms: Homonyms occur when a virtual address refers to multiple physical locations in different address spaces. If not disambiguated, incorrect data may be returned.

Page mapping and protection changes: The page permissions must be stored with each cache block to check permissions on cache hits. However, when permissions change, these bits must be updated. This is harder than with a TLB because many blocks may be cached from a single page, each of which must be updated. In addition, when the OS removes or changes a page mapping, the virtual address for a cache block must change.

Applications	Percentage of application - allocated pages that contains synonyms	Percentage of synonym containing pages that are *read-only*	Percentage of all dynamic user memory accesses to pages with synonyms
canneal	0.06%	100%	0%
fluidanimate	0.28%	100%	0%
facesim	0.00%	100%	0%
streamcluster	0.23%	100%	0.01%
swaptions	5.90%	100%	26%
x264	1.40%	100%	1%
bind	0.01%	100%	0.16%
firefox	9%	95%	13%
memcached	0.01%	100%	0%
specjbb	1%	98%	2%

Table 4. Virtual memory synonym analysis

Cache block eviction: Evicting a block cached with a virtual address requires translating the address to a physical address to perform writeback to physical caches further down the hierarchy.

Maintaining cache coherence: Cache coherence is generally performed with physical addresses. With a virtual cache, the address carried by the coherence messages cannot be directly used to access the cache. Thus a reverse translation (physical-to-virtual) is logically required.

Backward compatibility: Virtual caches can break compatibility with existing processor architectures and operating systems. For example, OSes on x86, such as Linux, update a page table entry (PTE) using cacheable virtual addresses. However, the x86's hardware page-table walker uses only physical addresses to find PTEs in caches or memory [40]. With a virtual cache, it is unclear how to make x86's page table walker work both correctly (a virtual L1 cache entry is like a synonym) and efficiently (if caching of PTEs is disabled). Moreover, virtual caches often break compatibility by requiring explicit OS actions (*e.g.*, cache flushes on permission changes) to maintain correctness.

These challenges hinder the adoption of virtual L1 caches despite their potential energy savings. In this work, we seek an ideal situation that provides most of the benefits of virtual caches by using it as *a dynamic optimization* while avoiding their complexities to an extent possible and maintaining compatibility.

3 Analysis: Virtual Cache Opportunity?

We set out to determine how often the expensive or complex virtual cache events actually happen in the real world by studying several modern workloads running on real x86 hardware under Linux. First, we measure the occurrences of virtual-memory synonyms to determine how often and where they occur in practice. As noted in Section 2.2, synonyms pose a correctness problem for virtual caches. Second, we measure the frequency of page protection/mapping changes, as these events can be more expensive with virtual caches.

We measured applications drawn from Parsec benchmark suite [30], as well as some important commercial applications (workloads explained in Section 5.2) listed in Table 4 running on Linux. We identified synonym pages by analyzing the kernel's page tables, and measured dynamic references to synonyms using PIN [22].

3.1 Synonym Usage

Table 4 presents a characterization of synonyms for our workloads. A page with a synonym is a virtual page whose corresponding physical page is mapped by at least one other user-space virtual address. We make three observations from this data. First, all but one application had synonym pages, but very few pages (0.06-9%) had synonyms. Second, the dynamic access rate of synonym pages was low (0-26%), indicating that virtual caching could be effective for most references. Finally, most synonym pages are mapped read-only, and therefore cannot introduce inconsistencies. This occurs because these pages were often from immutable shared library code (95-100% of the synonym pages).

We also found that the OS kernel sometimes uses synonyms in the kernel virtual address space to access user memory. For example, to process a direct I/O request that bypasses the operating system's page cache (used by databases), the kernel copies user data using a kernel address-space synonym for the user-space page. Kernel space synonyms are also used during a copy-on-write page fault to copy content of the old page to the newly allocated page. These kernel-space synonyms are temporary but can introduce inconsistency through read-write synonyms.

Finding 1: *While synonyms are present in most applications, conflicting use of them is rare. This suggests that virtual caches can be used safely for most, but not all, memory references.*

	Mean time between TLB invalidation in ms (avg. TLB invalidations per sec per core)	Fraction of TLB invalidations to a single page
canneal	132.62 (7.5)	0%
facesim	75.64 (13.2)	0%
fluidanimate	52.63 (19.2)	0%
streamcluster	55.53 (18)	0%
swaptions	51.81 (19.3)	0%
x264	111.11 (9.4)	0%
bind	7.571 (132.1)	0.00%
firefox	4.761 (210.3)	0.10%
memcached	2.325 (430.1)	0%
specjbb	39.011 (25.6)	2.50%

Table 5. Frequency of TLB invalidations

3.2 Page Mapping and Protection Changes

The operating system maintains coherence between the page-table permissions and the TLB by invalidating entries on mapping changes or protection downgrades, or by flushing the entire TLB. Table 5 presents the average inter-arrival time of TLB invalidations for our workloads (and its reciprocal – the TLB invalidation request per sec). The inter-arrival time of TLB invalidations varies widely across the workloads, but we make two broad observations. First, even the smallest inter-arrival time between invalidations (2.325ms for memcached) is an order of magnitudes longer than the typical time to flush and refill a L1 cache (~ 5µs). Hence, flushing the cache is unlikely to have much performance impact. Second, we observe that almost all TLB invalidations (97.5-100%) flush the entire TLB rather than a single entry. Most TLB invalidations occur on context switches that invalidate an entire address space, and only a few are for page protection/permission changes. Consequently, complex support to invalidate cache entries from a single page may not be needed.

Finding 2: *TLB invalidations that occur due to page mapping or protection changes are infrequent and are thus unlikely to create much overhead.*

4 Opportunistic Virtual Caching

The empirical analyses in the previous section suggest that while virtual cache challenges are real, they occur rarely in practice. All the applications studied provide *ample dynamic opportunities* for safe (*i.e.*, no read-write synonyms) and efficient (*i.e.*, no page permission/protection changes) use of virtual caches. Unfortunately, correctness and backward compatibility must be absolute and *not* "almost always".

To benefit from virtual caches and while sidestepping their dynamically rare issues, we propose *opportunistic virtual caching* (OVC). OVC hardware can cache a block with *either* virtual or physical address (Section 4.1). Virtual caching saves energy (no TLB lookup on L1 hits and reduced L1 associativity). Physical caching provides compatibility for read-write synonyms and caching page-table entries (and other structures) accessed by the processor with physical addresses.

To reap the benefits of OVC, the operating system must enable virtual caching for memory regions that are amenable to the use of virtual caching (Section 4.2). Importantly, we find that the OS kernel (Linux in this study) already possesses most of the information needed to determine which memory regions are suitable for virtual caching and which are not. While OVC defaults to physical-only caching to enable deployment of unmodified OSes and applications, changes to support virtual caching affected only 240 lines of code in the Linux kernel (version 2.6.28-4).

4.1 OVC Hardware

OVC requires that hardware provide the following services to realize the benefits of caching with virtual addresses – (1) determining when to use virtual caching and when physical caching, (2) reducing power when possible by bypassing the TLB and reducing cache associativity (3) handling-virtual memory homonyms and page permission/protection changes, and (4) handling coherence requests for virtually cached blocks.

Determining when to use virtual caching: The hardware defines a one-bit register named ***ovc_enable*** that an operating system can set to enable OVC (default is unset). When OVC is enabled, we take advantage of large virtual address space of modern 64-bit OSes to logically partition the address space into two non-overlapping address ranges (partition $\boldsymbol{P_{physical}}$ and $\boldsymbol{P_{virtual}}$). The highest order bit of the virtual address range (e.g., VA_{47}, the 48th bit in Linux for x86-64) determines the partition in which a virtual address of a cache lookup belongs to (in $\boldsymbol{P_{physical}}$ if VA_{47} is unset and $\boldsymbol{P_{virtual}}$ otherwise). Only cache lookups with virtual address in the partition $\boldsymbol{P_{virtual}}$ can use the virtual address to cache data. Thus, there is no added lookup cost to determine how an address is cached.

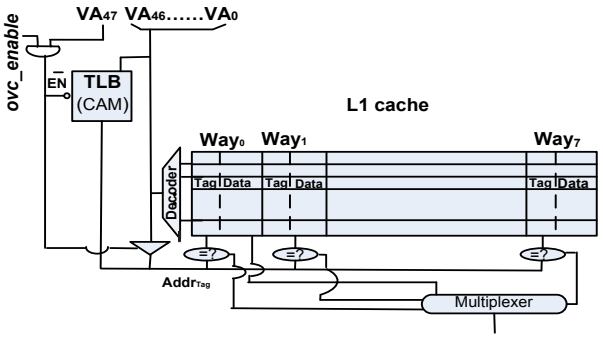

Figure 1. OVC L1 cache and TLB organization allow opportunistically bypassing TLB lookups.

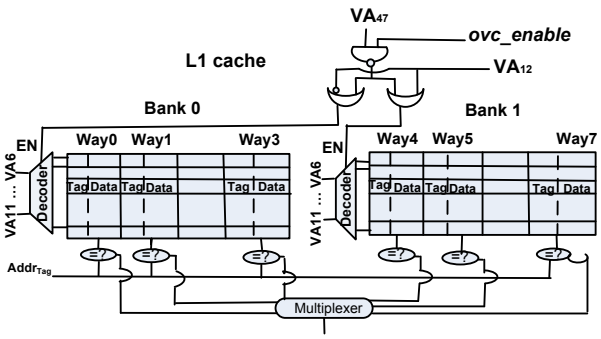

Figure 2. Opportunistic smaller associative L1 cache lookup using banked organization.

Opportunistically reducing lookup energy: When data can be cached using virtual address we take advantage of it in two ways. First, we avoid TLB lookups on L1 cache hits. Second, we allow lower associativity L1 cache lookups. As shown in Figure 1, when cache lookup address falls in partition $\boldsymbol{P_{virtual}}$ (i.e., ***ovc_enable*** and VA_{47} are set), the TLB lookup is disabled and part of the virtual address is used for cache tag match. Otherwise, conventional physical cache lookup is performed where the TLB is performed in parallel with indexing into the L1 cache. On a miss to an address in $\boldsymbol{P_{virtual}}$ a TLB lookup is required before sending the request to the next cache.

Second, OVC dynamically lowers the associativity of L1 cache lookups. We note that the cache associativity constraint of a physical cache, described in Section 2.1, need not hold true for virtually cached blocks. Figure 2 shows an example of how a banked L1 cache organization can be leveraged to allow lower-associativity cache lookup for a 32KB, 8-way set associative cache. The 8-way set-associative cache is organized in two banks each holding 4-ways of each set. For virtual addresses (*i.e.*, ***ovc_enable*** and VA_{47} are set), the processor only accesses one of the two banks (*i.e.*, 4 ways) based on the value of a single virtual-address bit from the tag (VA_{12} in the example). For other accesses using physical addressing, the processor performs a full 8-way lookup as in a conventional cache.

Handling homonyms and page permission changes: OVC implementation uses conventional address-space identifiers (ASIDs) to distinguish between different mappings of the same virtual address and avoids cache flushes on context switches. Both the ASID and the tag need to match for a cache hit to occur. OVC uses an all-zero ASID for blocks cached under the physical address (which results in an ASID match for any physical cache access). To handle the kernel address space, which is shared by all processes, we copy the *global* bit of the x86 PTE (which is set for globally shared kernel memory) to each cache block. Privileged mode access for blocks with this bit set do not need an ASID match. ASID overflow can be handled by modifying Linux's existing ASID management code to trigger a cache flush before reusing an ASID.

Page permissions (*e.g.*, read, write, execute, privileged) augment the coherence state permissions for each cache block and are checked along with coherence permissions. A page permission miss-match (*e.g.*, write request for a block with read permission) triggers a cache miss, which results in access to the TLB. It is then handled appropriately as in conventional physical cache for page permission miss-matches. Page mapping or permission downgrades trigger a cache flush.

Cache block eviction: Eviction of a dirty L1 block invokes a write-back to a physical L2 cache. OVC—like most virtual caches—logically augments each virtually-tagged block with a physical tag to avoid deadlock issues with doing an address translation at eviction. This physical tag adds a small state (e.g., 28 bits on 544 bits state, tag, and data) and can either be stored (a) in the L1 cache or (b) an auxiliary structure (not shown) that mirrors L1 dimensions, but is accessed only on less frequent dirty evictions.

Coherence: L2 caches and beyond typically process coherence with physical addresses. To access virtually-tagged L1 blocks, incoming (initiated by other cache controllers) back-invalidations and forwarded requests may require reverse address translation (physical to virtual). Reverse translation can be avoided by serially searching physical tags (added for cache block eviction) for all sets that might hold a block. Since OVC already provides the processor with an associative lookup on physical addresses, it

Figure 3. OVC overheads per L1 cache block. Additions are shaded.

associatively handles incoming coherence lookups with the same mechanism. For example, an incoming coherence request to the cache depicted in Figure 2, would simply access the physical tags in both banks (8-way total). Further, this action may be handled with an auxiliary structure (option (b) for handling eviction) and our empirical results find this occurs less than once per 1K L1 cache accesses due to high L1 hit rates and low read-write sharing. Note that, coherence messages received due to local cache misses (*e.g.*, data reply, acks) use miss-status handling register entries to find the corresponding location in the cache and hence do not require reverse translation lookup.

Space and Power Cost: As depicted in Figure 3, OVC's space overhead in the L1 cache stem primarily from the addition of an ASID (16 bits) and physical tag (28 bits) per cache block. The primary tag must be extended (8 bits) to accommodate larger virtual address tag. We also add page permission/privileged bits (3 bits) and a global bit. This totals approximately 10% space overhead for the L1 assuming 64-byte cache blocks. Given that L1 caches comprise a small fraction of the total space (and thus transistor count) for the cache hierarchy, which is dominated by larger L2 and L3 caches, the overall static power budget (which is grows roughly in proportion to transistor count) of the on-chip caches barely changes: ~ 1% overhead for the cache hierarchy in Table 6. Furthermore, the extra physical tag is accessed only for uncommon events: back invalidations, forwarded coherence messages and dirty evictions. L1 cache lookups and L1 cache hits do not accesses this physical tag. As a result, it leads ~ 1% energy overhead on L1 cache lookups, because most of the energy is spent on data access, which has not changed. We will show that this overhead is outweighed by the benefits of OVC. We also note that cycle time is not affected as data lookup latency overshadows the tag lookup latency.

4.2 OVC Software

The operating system for OVC hardware has three additional responsibilities: (1) predicting when virtual caching of an address is desirable (safe and efficient); (2) informing the hardware of which memory can use virtual caching; and (3) ensuring continued safety as memory usage changes. We extend the Linux virtual-address allocator to address the first two and make minimal changes to the page-fault handler and scheduler for the third.

Deciding when to use virtual caches: The OS decides whether virtual caching may be used at the granularity of memory regions. These are an internal OS abstraction for contiguous virtual-address ranges with shared properties, such as for program code, the stack, the heap, or a memory-mapped file. When allocating virtual addresses for a memory region the OS virtual address range allocator predicts whether the region could have read-write synonyms (unsafe) or frequent permission/mapping changes (inefficient), and if so, uses addresses that allows physical caching and otherwise uses virtual caching.

While predicting future memory usage may seem difficult, we observe that the OS *already possesses* much of the information needed. The kernel virtual-address allocator defines flags specifying how the memory region will be used, which guides its assignment of page permissions for the region. For example, in Linux, the VM_PRIVATE flag indicates pages private to a single process, VM_SHARED indicates a region may be shared with other processes, and VM_WRITE/VM_MAYWRITE indicates that a region is writable. From these flags, the kernel can easily determine that read-write synonyms occur only if the VM_SHARED and VM_WRITE/ VM_MAYWRITE flags are set, which causes the kernel to use physical caching. For all other memory regions kernel predicts to use virtual caching without possibility of read-write synonyms. This enables a straightforward identification of which memory regions can use virtual caching.

Unfortunately, these flags do not provide hints about efficiency: some regions, such as transiently mapped files, may observe frequent page-mapping or protection changes (*e.g.*, through *mprotect()* and *mremap()*) that can be expensive with virtual caches. We thus add an additional flag, MAP_DYNAMIC, to the virtual-address allocator to indicate that the mapping or page permissions are likely to change. Applications can use this flag while allocating memory to indicate frequent protection/mapping changes or the need for physical caching for other semantic or performance reasons.

Communicating access type to hardware: The kernel uses the prediction techniques described above to select either virtually or physically cached addresses for a region. If MAP_DYNAMIC is specified, physical caching is used irrespective of the prediction. We minimally extend the OS virtual address

range allocator to allocate addresses from two non-overlapping address pools, partitions $P_{physical}$ and $P_{virtual}$ (described in Section 4.1), depending on whether physical or virtual caching is to be used.

Ensuring correctness: While the kernel only uses virtual caching when it predicts that conflicting synonyms will not arise, they may still be possible in some rare cases. First, the kernel itself may use temporary kernel address space synonyms to access some user memory (Section 3.1). Second, the kernel allows a program to later change how a memory region can be used (e.g., through Linux's *mprotect()* system call). We provide a fallback mechanism to ensure correctness in these cases by detecting when the change occurs, and then flushing the cache between conflicting uses of memory.

We insert two checks into the Linux kernel for conflicting synonyms. Within the page fault handler, we add code to check whether a virtually cached page is being mapped with write permissions at another address in another process. Similarly, we put a check in the kernel routine that creates temporary kernel mappings to user memory to detect conflicting synonyms. If the above checks detect possibility of a conflicting synonym in the page-fault handler, the OS marks the process with write access to a synonym as *tainted*, meaning that when it runs, it may modify synonym pages. We modify the OS scheduler to flush the L1 cache before and after the tainted process runs. If hyper-threading is enabled, scheduler needs to prohibit tainted process from sharing the same core (and thus L1 cache) with another process. This ensures that address synonyms between the kernel and user-mode code similarly: the kernel flushes caches before and after using kernel-space synonyms.

For frequent and performance-sensitive synonym uses, such as direct I/O, a program can prevent these flushes by mapping I/O buffers using the MAP_DYNAMIC flag, which will use physical caching. However, even if a user fails to do so, the above mechanism ensures correctness anyways. We also note that it is possible to have read-write synonyms within single process's address space (*e.g.*, if same file is simultaneously memory mapped by a single process at different places in writable mode). If such cases ever occur (we have encountered none), we propose to turn off OVC capability (unset ***ovc_enable***) for the offending process.

CPU	4-core, in-order, x86
L1 TLB	Private, Split Data and Instruction L1 TLB, 64 entries, Fully associative
L2 TLB	Private, 512 entries, 4-way set associative
L1 cache	Private, Data and Instruction L1 Cache, 32 KB, 8-way set associative
L2 cache	Private, 256KB, 8-way set associative
L3 cache	Shared, 8MB, 16-way set-associative, MESI Directory cache coherence

Table 6. Baseline system parameters.

5 Evaluation

5.1 Baseline architecture

We modeled a 4-core system with an in-order x86 CPU detailed in Table 6. The simulated system has two levels of TLB and three levels of caches. Each core sports separate L1 data and instruction TLB and a unified L2 TLB. The cache hierarchy has a split L1 instruction and data cache private to each core. Each core also has a private L2 cache that is kept exclusive to the L1 cache. The L3 cache is logically shared among all the cores, while physically distributed in multiple banks across the die.

5.2 Methodology and Workloads

We used x86 full system simulation with gem5 [4] to simulate a 4-core CMP with the configuration listed in Table 6. We modified the Linux 2.6.28-4 kernel to implement the operating system changes required for leveraging OVC. We used CACTI 6.5 [28] with the 32nm process for computing energy numbers. For TLBs, L1 caches, and L2 caches, we used high performance transistors ("itrs-hp"), while low static power transistors ("itrs-lstp") were used for L3. L1 and L2 caches lookup both tag and data array in parallel for providing faster accesses. However, L3 caches lookup the tag array and data array in sequence.

We use several of RMS workloads (*canneal, facesim, fluidanimate, streamcluster, swaptions, x264*) from

	L1 Data TLB	L1 Instr. TLB
canneal	72.253	99.986
facesim	96.787	99.999
fluidanimate	99.363	99.999
streamcluster	95.083	99.994
swaptions	99.028	99.989
x264	95.287	99.304
specjbb	91.887	99.192
memcached	94.580	98.605
bind	97.090	98.310
Mean	93.484	99.486

Table 7. Percentage of TLB lookup energy saved by OVC

Parsec [30]. We also use a set of commercial workloads: *SpecJBB* 2005 [41], a server benchmark that models Java middle-tier business-logic processing; *memcahed* [26], an open source in-memory object store used by many popular web services including Facebook and Wikipedia; and *bind*, the BIND9 Domain Name Service (DNS) lookup service [9]. We also analyzed the open-source web browser Firefox [27] synonym usages and TLB invalidation characterization. However, as an interactive workload, it does not run on our simulator.

5.3 Results

To evaluate OVC, we seek to answer three questions: (1) How much TLB lookup energy is saved? (2) How much of L1 cache lookup energy is saved? (3) What is the performance impact of the OVC?

In our evaluation we focus on dynamic (lookup) energy as TLBs and L1 caches are frequently accessed, but relatively small, making OVC's static-energy impact insignificant.

TLB Energy savings: Table 7 shows the percentage of L1 data and instruction TLB dynamic energy saved by the OVC. We observe that more than 94% of the L1 data TLB energy and more than 99% of L1 Instruction TLB lookup energy is saved by OVC. To analyze this result, we first note that the cache accesses that use virtual addresses and hit in the L1 cache avoid burning energy for TLB lookups. Table 8 shows the percentage of data and instruction accesses that can complete without needing address translation, while the L1 cache hit rates for accesses using virtual addresses are listed in Table 9. We observe that on average 97% of data accesses and 100% of

	Data V-addr access perct.	Instr V-addr access perct.
canneal	80.791	100
facesim	99.843	100
fluidanimate	99.925	100
streamcluster	98.575	100
swaptions	99.990	100
x264	99.933	100
specjbb	96.650	100
memcached	99.291	100
bind	98.97	100
Mean	97.116	100

Table 8. Percentage of access that use Virtual address

instruction accesses complete without needing address translation, while a very high fraction these accesses (0.96 and 0.99 respectively) hit in the cache, saving TLB lookup energy.

L1 cache energy savings: OVC saves L1 cache lookup energy by accessing only a subset of the ways in a set when using virtual addresses (Section 2). Table 10 presents percentage savings in dynamic energy by OVC from opportunistic use of partial lookups (4-ways out of 8-ways) in the L1 cache. The second column shows that on average more than 22% of the dynamic energy spent on L1 data cache lookups is saved, while the third column shows similar savings for an instruction cache. The rightmost column provides a more holistic view of the energy savings in the chip by showing how much of dynamic energy of TLBs and all the three levels of on-chip caches taken

	L1 Data $	L1 Instr. $
canneal	0.894	0.999
facesim	0.969	0.999
fluidanimate	0.994	0.999
streamcluster	0.964	0.999
swaptions	0.990	0.999
x264	0.953	0.993
specjbb	0.950	0.991
memcached	0.952	0.986
bind	0.980	0.983
Mean	0.961	0.994

Table 9. L1 cache hit rates for virtual accesses

	L1 Data $ dynamic energy savings	L1 Instr. $ dynamic energy savings	TLBs + $ hierarchy dynamic energy savings
canneal	17.381	22.800	9.989
facesim	22.252	22.800	18.575
fluidanimate	22.727	22.801	30.672
streamcluster	21.805	22.802	16.709
swaptions	22.797	22.807	32.542
x264	27.737	23.230	25.446
specjbb	23.229	22.771	17.547
memcached	23.352	23.155	16.765
bind	22.812	22.784	28.283
Mean	22.624	22.883	19.546

Table 10. Percentage of Dynamic energy savings in the caches

together is saved. On average, more than 19% of the dynamic energy spent on the on-chip cache hierarchy and the TLBs is eliminated by the OVC. The savings can be as high as 32% (*swaptions*) for applications with small working sets that rarely access L2 or L3 caches. In total, OVC saves a considerable portion of on-chip memory subsystem dynamic energy through lower associative L1 cache lookups and TLB lookup savings as these two frequent lookups account for most of the dynamic energy in the on-chip memory.

Performance impact: We quantify the performance implications of OVC in Table 11 which show the number of misses per 1K cache reference (MPKR) for the baseline and the OVC L1 data and instruction caches. For the L1 data cache, the change in the number of misses is within a negligible 0.7 misses per 1K cache reference, while changes for instruction caches are even smaller. Two of the workloads (*specjbb, memcached*) experience larger L1-D cache miss rate decrease with OVC (~2 misses per 1K reference, which translates to a minuscule hit-rate difference), while the L1 I-cache miss rate increases for one workload (*bind*). We note that cache hit/miss patterns are slightly perturbed due to use of a single bit from the virtual page number in selection of the bank where an access should go when virtual address is used under OVC. More importantly from Table 11 (right-most column), we observe that OVC hardly changes run time compared to the baseline system (within 0.017%). The unchanged run time, coupled with OVC's small static power overhead to the whole on-chip cache hierarchy (Section 4.1) indicates that OVC leaves the static power consumption of the on-

	Baseline L1D MPKR	OVC L1D MPKR	Baseline L1I MPKR	OVC L1I MPKR	Norm. runtime
canneal	105.62	105.68	0.120	0.133	0.9994
facesim	30.476	30.613	0.084	0.093	0.9999
fluidanim.	5.735	5.622	0.003	0.006	0.9999
streamclus.	35.436	35.421	0.037	0.037	1.00001
swaptions	9.668	9.716	0.106	0.106	1.0004
x264	47.329	46.492	6.53	6.977	1.00099
specjbb	51.704	49.289	7.683	8.008	0.99330
memcached	49.699	47.349	14.235	13.947	0.99632
bind	20.527	19.630	13.981	16.893	1.00701
Mean	39.576	38.879	4.745	5.133	1.00017

Table 11. Miss ratio and runtime comparison between Baseline and OVC

chip memory subsystem largely unchanged while saving substantial dynamic energy. Furthermore, for these workloads, the operating system never needed to use the taint bit (Section 4.2) as they do not use direct I/O or make system calls to change page protection. Moreover, there were no cache flushes due to memory-mapping changes.

6 Related Work

There has been decades of research on implementing virtual caches, which are summarized by Cekleov and Dubois for both uniprocessor [5] and multiprocessor systems [6]. Here we discuss a few of the most related work on virtual caches. We also discuss relevant work on reducing TLB power.

Goodman proposed an all-hardware solution for handling synonyms by introducing dual-tag store for finding reverse translations on possible synonyms [13], while a similar technique uses back-pointers in L2 physical caches for finding synonyms in L1 virtual caches [35]. Kim et al. proposed the U-cache in which a small physically-indexed cache was added to hold reverse translations for pages with possible synonyms [19]. A few other works advocate for sidestepping the problem of synonyms by constraining sharing (and thus synonyms) through shared segments only [10,38] or through constrained virtual to physical memory mapping (page coloring) to ensure synonyms always fall in the same cache set [23]. Qiu et al. [32] proposed a small synonym lookaside buffer in place of a TLB to handle synonyms in a virtual cache hierarchy. On the other side of the spectrum,

single address space operating systems like Opal [8,12] and Singularity [20] propose a new OS design philosophy that does away with private per-process address spaces altogether (and thus no possibility of synonyms). Although many of the above techniques for virtual caching are used in OVC; we expose virtual caching as an optimization rather than a design point to leverage benefits of virtual caching when suitable and defaulting to physical cache when needed for correctness, performance or compatibility.

Several past hardware proposals addressed the problem of TLB power consumption through TLB CAM reorganization [16], by adding hardware filter or buffering for TLB access [7,14] or by using banked TLB organization [7,24]. Kadayif et al. [17] proposed adding hardware translation registers to hold frequently accessed address translations under compiler directions. Ekman et al. [11] evaluated possible TLB energy savings by using virtual L1 caches as a pure hardware technique while also proposing a page grain structure to reduce the coherence snoop energy in the cache. Wood et al. [38] advocated doing away with TLBs by using virtual caches and using in-cache address translation. Jacob et al. [15] proposed handling address translation with software exceptions on cache miss to also get rid of the TLB. OVC, on the other hand, is a software-hardware co-design technique that aims to maintain full backward compatibility with existing software while opportunistically allow both TLB and L1 cache lookup energy reduction.

Woo et al. [37] proposed using bloom filter to hold synonym addresses to save L1 cache lookup energy by allowing lower associativity. Ashok et al. [1] proposed compiler directed static *speculative* address translation and cache access support to save energy. Different from their work, we do not burden the hardware with the onus of ensuring correctness for static miss-speculation; neither do we require recompilation of application to take advantage of OVC. Zhou et al. [39] proposed heterogeneously tagged (both virtual and physical tag) to allow cache access without TLB access for memory regions explicitly annotated by the application. Unlike their work, application modification is not necessary for OVC. Furthermore, OVC saves L1 cache lookup energy through reduced associativity. Lee et al. [21] proposed to exploit the distinct characteristics of accesses to different memory regions of a process (e.g., stack, heap etc.) to statically partition the TLB and cache resources to save energy. OVC does not require static partitioning of hardware resources and instead opportunistically use virtual caching to allow substantial energy benefits.

Some older embedded processors from ARM also allowed virtual caches, but had to flush caches on context switch/page permission changes [36].

7 Conclusion

Our empirical analysis shows that virtual cache challenges are real, but occur rarely in practice. To benefit from virtual caches and yet sidestep their rare correctness issues, we proposed the opportunistic virtual cache (OVC) that can cache a block with either a virtual (saving power) or physical address (ensuring compatibility). We show that small OS changes enable effective OVC virtual caching, while OVC facilitates its own adoption by operating correctly with no software changes.

8 Acknowledgement

We thank Derek R. Hower and Haris Volos for their thoughtful comments and help during the project. We thank Wisconsin Computer Architecture Affiliates for their feedback on early version of the work. We thank Megan Falater and Elizabeth Hagermoser for proof-reading our drafts.

This work is supported in part by the National Science Foundation (CNS-0720565, CNS-0916725, and CNS-1117280, CNS-0834473), Sandia/DOE (#MSN 123960/DOE890426), and University of Wisconsin (Kellett award to Hill). The views expressed herein are not necessarily those of the NSF, Sandia or DOE. Hill has a significant financial interest in AMD and Swift has a significant financial interest in Microsoft.

9 References

1. Ashok, R., Chheda, S., and Moritz, C.A. Cool-Mem: combining statically speculative memory accessing with selective address translation for energy efficiency. *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.
2. Barr, T.W., Cox, A.L., and Rixner, S. SpecTLB: a mechanism for speculative address translation. *Proc. of the 38th Annual Intnl. Symp. on Computer Architecture*, 2011.
3. Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. Accelerating two-dimensional page walks for virtualized systems. *Proc. of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.
4. Binkert, N., Beckmann, B., Black, G., et al. The gem5 simulator. *Computer Architecture News*, 2011.

5. Cekleov, M. and Dubois, M. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro 17*, 5 (1997).
6. Cekleov, M. and Dubois, M. Virtual-Address Caches, Part 2: Multiprocessor Issues. *IEEE Micro 17*, 6 (1997).
7. Chang, Y.-J. and Lan, M.-F. Two new techniques integrated for energy-efficient TLB design. *IEEE Trans. Very Large Scale Integr. System 15*, 1 (2007).
8. Chase, J.S., Levy, H.M., Lazowska, E.D., and Baker-Harvey, M. Lightweight shared objects in a 64-bit operating system. *Object-oriented programming systems, languages, and applications*, 1992.
9. Consortium, I.S. *Berkeley Internet Name Domain (BIND)*. http://www.isc.org/software/bind .
10. Diefendorff, K., Oehler, R., and Hochsprung, and R. Evolution of the PowerPC Architecture. *IEEE Micro 14*, 2 (1994).
11. Ekman, M., Dahlgren, F., and Stenstrom, P. TLB and Snoop Energy-Reduction using Virtual Caches in Low-Power Chip-Multiprocessors. *In Proceedings of International Symposium on Low Power Electronics and Design*, 2002, 243–246.
12. Eric J. Koldinger, J.S.C. and Eggers, S.J. Architecture support for single address space operating systems. In Proc. of the *5th international conference on Architectural support for programming languages and operating systems*, 1992.
13. Goodman, J.R. Coherency for multiprocessor virtual address caches. *Proc. of the 2nd international conference on Architectural support for programming languages and operating systems*, 1987.
14. J. H. Lee, C.W. and Kim, S.D. Selective block buffering TLB system for embedded processors. *IEE Proc. Comput. Dig. Techniques 152*, 4 (2002).
15. Jacob, B. and Mudge, T. Uniprocessor Virtual Memory without TLBs. *IEEE Trans. on Computer 50*, 5 (2001).
16. Juan, T., Lang, T., and Navarro, J.J. Reducing TLB power requirements. *Proc. of the international symposium on Low power electronics and design*, 1997.
17. Kadayif, I., Nath, P., Kandemir, M., and Sivasubramaniam, A. Reducing Data TLB Power via Compiler-Directed Address Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 26*, 2 (2007).
18. Kadayif, I., Sivasubramaniam, A., Kandemir, M., Kandiraju, G., and Chen, G. Generating physical addresses directly for saving instruction TLB energy. *Proc. of the 35th annual ACM/IEEE international symposium on Microarchitecture*, 2002.
19. Kim, J., Min, S.L., Jeon, S., Ahn, B., Jeong, D.-K., and Kim, C.S. U-cache: a cost-effective solution to synonym problem. *1st IEEE symposium on High-Performance Computer Architecture*, (HPCA) 1995.
20. Larus, G.H.J., Abadi, M., Aiken, M., et al. *An Overview of the Singularity Project*. Microsoft Research, 2005.
21. Lee, H.-H.S. and Ballapuram, C.S. Energy efficient D-TLB and data cache using semantic-aware multilateral partitioning. *Proc. of the international symposium on Low power electronics and design*, 2003.
22. Luk, C.-K., Cohn, R., Muth, R., et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Proc. of the SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005.
23. Lynch, W.L. *The Interaction of Virtual Memory and Cache Memory*. Stanford University, 1993.
24. Manne, S., Klauser, A., Grunwald, D., and Somenzi, F. *"Low power TLB design for high performance microprocessors*." University of Colorado, Boulder, 1997.
25. McNairy, C. and Soltis, D. Itanium 2 Processor Microarchitecture. *IEEE Micro 23*, 2 (2003), 44–55.
26. memcached - a distributed memory object caching system. www.memcached.org.
27. Mozilla, M. *Firefox,*. http://www.mozilla.org/ .
28. Muralimanohar, N., Balasubramonian, R., and Jouppi, N.P. *CACTI 6.0*. Hewlett Packard Labs, 2009.
29. Patterson, D.A. and Hennessy, J.L. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2005.
30. *Princeton Application Repository for Shared-Memory Computers.* http://parsec.cs.princeton.edu/.
31. Puttaswamy, K. and Loh, G.H. Thermal analysis of a 3D die-stacked high-performance microprocessor. *16th ACM Great Lakes symposium on VLSI*, 2006.
32. Qiu, X. and Dubois, M. The Synonym Lookaside Buffer: A Solution to the Synonym Problem in Virtual Caches. *IEEE Trans. on Computers 57*, 12 (2008).
33. Sodani, A. *Race to Exascale: Opportunities and Challenges*. MICRO 2011 Keynote talk.
34. Talluri, M., Kong, S., Hill, M.D., and Patterson, D.A. Tradeoffs in Supporting Two Page Sizes. *Proc. of the 19th Annual International Symposium on Computer Architecture*, 1992.
35. Wang, W.H., Baer, J.-L., and Levy, and H.M. Organization and performance of a two-level virtual-real cache hierarchy. *Proc. of the 16th annual international symposium on Computer architecture*, 1989.
36. Wiggins, A. and Heiser, G. Fast Address-Space Switching on the StrongARM SA-1100 Processor. *Proc. of the 5th Australasian Computer Architecture Conference*, (1999).
37. Woo, D.H., Ghosh, M., Özer, E., Biles, S., and Lee, H.-H.S. 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.
38. Wood, D.A., Eggers, S.J., Gibson, G., Hill, M.D., and Pendleton, J.M. An in-cache address translation mechanism. *ISCA'86 : 13th annual international symposium on Computer architecture*, (1986).
39. Zhou, X. and Petrov, P. Heterogeneously tagged caches for low-power embedded systems with virtual memory support. *ACM Transactions on Design Automation of Electronic Systems (TODAES) 13*, 2 (2008).
40. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Part1, Chapter 2*. 2009.
41. *SpecJBB 2005*.http://www.spec.org/jbb2005/.