2018 IEEE International Symposium on High Performance Computer Architecture

SIPT: Speculatively Indexed, Physically Tagged Caches

Tianhao Zheng, Haishan Zhu, Mattan Erez
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX
Email: {thzheng, haishanz, mattan.erez}@utexas.edu

***Abstract*—First-level (L1) data cache access latency is critical to performance because it services the vast majority of loads and stores. To keep L1 latency low while ensuring low-complexity and simple-to-verify operation, current processors most-typically utilize a virtually-indexed physically-tagged (VIPT) cache architecture. While VIPT caches decrease latency by proceeding with cache access and address translation concurrently, each cache way is constrained by the size of a virtual page. Thus, larger L1 caches are highly-associative, which degrades their access latency and energy. We propose speculatively-indexed physically-tagged (SIPT) caches to enable simultaneously larger, faster, and more efficient L1 caches. A SIPT cache speculates on the value of a few address bits beyond the page offset concurrently with address translation, maintaining the overall safe and reliable architecture of a VIPT cache while eliminating the VIPT design constraints. SIPT is a purely microarchitectural approach that can be used with any software and for all accesses. We evaluate SIPT with simulations of applications under standard Linux. SIPT improves performance by 8.1% on average and reduces total cache-hierarchy energy by 15.6%.**

I. Introduction

The L1 cache is the most frequently accessed structure in the memory hierarchy and therefore should have low expected access latency. As such, the L1 cache presents challenging tradeoffs between hit rate and access latency. Access latency includes the virtual memory address translation latency (TLB lookup), tag array access and matching, and the data access itself. In order to push latency down, all three components are ideally overlapped. Tag and data accesses are overlapped by accessing all ways simultaneously and delivering only tag-matching data. Overlapping those two accesses with address translation is more challenging because an access can not start before the address is known. The simplest cache design indeed performs translation before L1 access begins. This design is called a *physically-indexed physically-tagged* (PIPT) cache because virtual addresses (VAs) are not used at all in the L1. While simple, the translation overhead is not hidden and access latency is often considered too high.

Current designs reduce latency and enable access and translation overlap in one of two ways. The first relies only on the offset bits of the VA for computing L1 array locations; the offset bits are not translated and can hence be used at the same time translation proceeds. Before data is delivered, the tag is compared to the fully translated physical address (PA). This *virtually-indexed physically-tagged* (VIPT) design is very effective, but constrains L1 design such that the total capacity of each cache way is the system's page size, which is commonly just 4KiB. Thus, high-associativity caches are necessary to attain a high hit rate because associativity determines the cache size. This not only increases the access latency, but also increases cache energy consumption. For example, Intel reports that 12% - 45% of core power is consumed by private caches[1].

The second solution is to translate virtual addresses (VAs) before the L1 is filled, thus accessing the cache purely with VAs. Such *virtually-indexed virtually-tagged* (VIVT) designs eliminate the translation latency for accessing the cache. However, relying purely on VAs for most memory accesses (as most are L1 hits) presents significant complications for cache management and coherence because software maps multiple VAs to the same physical address (*synonyms*) and may also map the same virtual address to multiple physical addresses (*homonyms*). Prior work has developed solutions, but the designs are more complicated than VIPT [2], [3].

Many current processors have adopted the simple and reliable VIPT design including, to the best of our knowledge, all Intel x86 processors, IBM Power processors, and recent implementations from ARM. In this focused paper, we propose and evaluate a new option for cache indexing that relaxes the size vs. associativity tradeoff of VIPT caches while retaining the advantages of VIPT caches. Our *speculatively-indexed physically-tagged* (SIPT) mechanism uses simple predictors to accurately predict the cache index beyond the offset bits and uses PAs for tag matches. Thus, larger lower-associativity caches can be designed with just $1-3$ bits being predicted. If SIPT predicts an incorrect cache index, the physical tag mismatches and the cache is accessed again with the correct bits from the now available PA. We show that misspeculation rates are low and that performance and energy are improved. The predictability of the few index bits may depend on the mapping between virtual and physical memory. However, we show that prediction accuracy is high even when physical memory is stressed and virtual to physical mapping is artificially highly fragmented.

We present three variants of SIPT and evaluate them in the L1 data cache of both out-of-order processors with a three-level hierarchy and in-order processors with a two-

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00020

level cache hierarchy. The first variant simply assumes that an additional $1-3$ address bits will remain the same after translation and always accesses the cache with a speculative index. Because of OS memory allocation schemes, this is often true. However, misspeculation is frequent enough that the extra L1 accesses limit speedup and energy efficiency.

Our second SIPT variant adds a speculation filter to predict whether those additional $1-3$ bits are likely to remain the same after translation. If the bits are predicted to change, speculation is bypassed and the cache is only accessed after translation. Our experiments show that the predictor effectively curbs redundant accesses and improves energy, while rarely bypassing what would be correct speculations. Because some applications exhibit frequent accesses with bits that change during translation, performance with speculation bypassing lags that of an ideal cache.

The third SIPT variant goes further and instead of predicting whether speculation should be bypassed or not, attempts to predict the value of the $1-3$ bits after translation. We design a BTB-like predictor for this purpose, which first predicts whether a change is likely or not and then predicts the bit values for those small number of accesses for which a change is predicted. Our experiments show that this scheme is surprisingly effective across all benchmarks, and improves performance by 8.1% on average in a quad-core out-of-order processor. This aggressive SIPT design comes quite close to an ideal cache which is always accessed with the correct full physical addresses. Moreover, we validate our results with highly-fragmented memory. Unlike many predictors proposed for memory, SIPT does not rely on virtual addresses, thus the prediction can start before the address is generated and does not increase cache accessing latency.

The paper is organized as follows: Section II provides detail about cache indexing options and discusses the most closely related prior work; Section III presents motivating experiments that demonstrate that VIPT caches significantly constrain the design space; Sections IV, V, and VI describe and evaluate the three SIPT variants; Section VII discusses a few sensitivity studies; and Section VIII concludes the paper.

II. Background

We discuss the fundamental tradeoffs between PIPT, VIVT, and VIPT caches and then discuss, and contrast from, prior work that is most relevant to SIPT.

A. PIPT Caches

PIPT caches work purely in the physical address space and require that virtual addresses be translated prior to cache access. PIPT caches are simple to design, validate, and verify. PIPT caches are therefore very common in caches that are further from the core because translation has already (typically) occurred before they are accessed.

B. VIVT Caches

VIVT caches access L1 in the virtual address space and avoid the latency of translation altogether. While this sounds simple and appealing, VIVT caches for fully general-purpose systems are, in fact, complex. The complexity stems from how the operating system (OS) manages physical memory and optimizes its use. It is common for the OS to map multiple VAs to a single PA (synonyms). The OS may also assign a VA to multiple PAs (homonyms), though only when these homonyms are in different address spaces (e.g., processes).

Homonyms are easy to resolve by adding an address-space identifier (ASID) to the tag (at the cost of somewhat more expensive tags) or flushing the cache on a context switch (overhead too high in general CPUs). Synonyms are more challenging because the cache must ensure all cached synonyms are always coherent (or always point to the same physical cache line). Much research has been devoted to the synonym problem. Examples of such prior work include the U-cache [4], Synonym Lookaside Buffer [5], and Dynamic Synonym Remapping [6]. However, synonym handling mechanisms are complex, increasing cost, design effort, and importantly verification effort. Even with synonym resolution at the L1, there are still implications for cache coherence which further complicate the design or constrain it to some specific coherence mechanisms [3], [2].

Even if all these problems are addressed and the design and verification challenges are acceptable, some architecture- and system-specific mechanisms require physically addressed (tagged) caches, else they introduce even more complexity. Examples include the x86 page walker [7] and Linux's Direct Access for files (DAX) [8].

Note that some designs that target scenarios with very high TLB miss rates advocate for VIVT caches because the latency of TLB misses can be very high. However, we are not aware of modern commercial general-purpose CPUs that follow it, though some accelerators, for which the driver tightly controls synonyms, homonyms, and potential coherence pitfalls have suggested the use of VIVT caches [2].

C. VIPT Caches

VIPT caches are appealing because they combine the strong correctness and simple coherence guarantees of PIPT caches with practically zero-latency translation. All addresses have the full PA available through the tags for correctness and coherence. At the same time, the latency of translation can be fully hidden by accessing the cache arrays in parallel with only the page offset bits that are never modified by address translation. However, the important tradeoff made is the constraints on cache parameters. Specifically, each set is limited in capacity to a single virtual memory page. Therefore, the cache capacity is coupled with its associativity: capacity $=$ #ways $\times$ 4KiB, assuming common 4KiB page granularity. For example, many current processors have 32KiB 8-way set-associative caches [9], [10], [11]. While high associativity

reduces conflicts it also adds latency, which is potentially a suboptimal design point when compared to a larger lower-associativity cache (as we show later, associativity impacts latency more than capacity). Furthermore, L1 cache access energy is also coupled with associativity because all ways are typically accessed in parallel to reduce latency.

D. Related Work

We discuss prior work relating to overcoming VIPT constraints by relying on the OS (page coloring and huge pages), relying on additional hardware translation mechanisms (TLB slice), prediction techniques to reduce the overheads of highly-associative caches (way prediction) and hide access latency (line prediction), hardware de-aliasing mechanisms to eliminate synonyms within a cache, and hybrid VIVT/VIPT caches.

Improving VIPT Caching with Page Coloring. With page coloring, the software memory allocator applies memory placement considerations, grouping and separating pages to manage locality. The popular example usage of coloring is placement to avoid cache set conflicts for pages that are commonly in the same working set. This mechanism is implemented in some operating systems, including FreeBSD [12] and NetBSD [13].

This idea can be applied to constrain virtual to physical mapping such that a VIPT cache can rely on more bits for indexing. For example, all even frames may be mapped to the bottom cache indices while odd frames are mapped to the top indices. However, hardware must rely on software and software must be aware of specific microarchitectural details. This is challenging for correctness and is difficult to maintain. One example of a commercial product that uses this restriction for a VIPT cache is the ARMv6 architecture [14].

In contrast, *SIPT only uses address locality as a hint for improving performance and hardware always guarantees correctness.*

Hugepages. Hugepages [15], [16], [17] are an architectural and OS mechanism for coarse-grained (e.g., 2MiB) pages. They increase TLB reach to reduce TLB misses and improve performance. With a 2MiB page, 21 bits are unchanged by translation, relaxing the constraints on a VIPT L1. However, for backward compatibility and performance reasons, hardware cannot rely on all pages being huge [17], [15], [18].

TLB Slice. The TLB slice [19] is an alternative translation lookaside buffer design used in MIPS R6000 microprocessors. The TLB slice is smaller and faster than a conventional TLB because it only holds a few $(4-8)$ physical page number bits to index an uncommon PIVT (Physicals indexed, virtually tagged) cache. However, the TLB slice simply uses the physical bits of the last access for prediction. To achieve reasonable performance, a primary VIVT cache is required for shielding the TLB slice from most loads and stores.

Cache Way Prediction and Line Prediction. Way prediction [20] has been proposed as a technique to lower the overhead of accessing highly-associative caches. Instead of accessing all ways in parallel (typical for L1), a single way is predicted to hold the cache line being accessed. A correct prediction saves the energy of accessing all ways in parallel. If the prediction is incorrect, extra steps are required to activate all ways, so average latency for an L1 access may increase. A similar idea is to predict which bank of a multi-banked cache contains the accessed data [21].

Like way and bank prediction, SIPT attempts to predict where in the cache an address is stored. However, the motivation, insight, and actual mechanisms are quite different. SIPT can perhaps be thought of as predicting a set and the motivation is relaxing the constraints on VIPT cache design.

In fact, way prediction and SIPT are complementary. Lowering associativity with SIPT can improve way-prediction accuracy. We discuss combining way prediction and SIPT in Section VII and analyze the impact of one on the other.

The Alpha 21264 implemented a line predictor for the I-cache [22]. It learns from branch history and predicts which line contains the next instruction block. The processor fetches and validates the predicted line before the branch/jump target is resolved to hide access latency and jump overhead. However, the absence of similar chaining makes prediction in D-cache difficult. Instead of prefetching, SIPT improves the cache itself (lower latency, larger capacity, or both).

De-aliasing synonyms within cache. Some processors use a design where the cache is indexed with a virtual address (beyond page boundaries), but the tag is still physical. This breaks the set-size constraint but requires additional mechanisms to de-alias synonyms. The AMD Opteron [23] checks all possible locations of aliases on every miss and evicts any aliases before a fill. These accesses can add significant latency, energy, and complexity, which SIPT avoids. The MIPS R10000 similarly uses bits beyond the page size (bits 13:12) to index its L1 cache, and also ensures no synonyms exist before the cache is filled. Instead of checking all possible alias locations, the R10000 adds the two VA bits to the L2 cache tag. The L2 controller uses these to evict aliases when necessary.

SIPT differs significantly from these prior industry solutions. SIPT does not rely on any VIVT addressing, not even limited to a few bits. Instead, SIPT allows synonyms to be cached and ensures correctness by always checking the full tag on a lookup. Performance is improved by effectively speculating when accesses are likely to succeed regardless of possible changes between VA and PA and by successfully predicting such changes.

Hybrid VIVT and VIPT Caches. Opportunistic virtual caching (OVC) [9] dynamically adapts the L1 cache between VIVT and VIPT. Virtual addressing is used to reduce access energy with lower-associativity sets. When used in physical-addressing mode, the cache increases associativity to fulfill VIPT constraints. OVC relies on software to identify memory regions that have no synonyms (e.g., pages private to one

process). The region must be continuously monitored to ensure no synonyms are created and that permissions remain valid. Homonyms still need to be handled by flushing or by extending the tag with an ASID.

SIPT is similar in its attempt to better trade off the benefits of VIPT and VIVT. However, SIPT is a pure microarchitectural technique with no address tracking and no OS interactions. Like the VIVT mode of OVC, SIPT relaxes constraints for VIPT caches, but SIPT does so for all accesses.

III. Motivation

The motivation for SIPT is that VIPT constraints lead to suboptimal L1 cache parameters (latency, capacity, and associativity). We therefore simulate the effects of different cache parameters and analyze their impact on the SPEC CPU applications. We use Intel's Haswell L1 parameters: 32KiB capacity, 8-way set associative, and 4-cycle latency.

We motivate the need for SIPT and demonstrate its effectiveness in the context of data caches, leaving instruction caches for future work. We believe SIPT will at least as well for instruction caches as instruction working sets are typically small compared to data (suggested by the high I-TLB hit rates observed in prior work [24], [25]).

A. Impact of Capacity and Associativity on Latency

Methodology. We explore the capacity and associativity design space with Cacti 6.5 [26]. We simulate L1 caches with the configurations and parameters summarized in Tab. I. We present the latency of the different configurations normalized to the latency of 32KiB 8-way baseline (Fig. 1). For each capacity and associativity, we sweep the number of read ports and the number of banks and show the range and mean of relative latencies for each configuration.

We also use CACTI when we later evaluate the energy of the cache hierarchy. We estimate total cache hierarchy energy by considering both active and passive power (over the duration of a simulation). For the L1 caches, we use high performance transistors and configure the cache for concurrent tag and data array access across all ways. For the L2 and L3 caches, we use low static power transistors and access the tag and data arrays sequentially. We consider the lower level caches as well because modifying the L1 configuration affects lower-level cache accesses. When averaging energy or relative energy metrics, we use the arithmetic mean.

Analysis. While both capacity and associativity affect latency, associativity has the greater impact. This is especially the case when increasing associativity beyond 4 ways. For example, we can reduce the latency of a 32KiB cache to 2 cycles by reducing its associativity to 2 ways, and that of a 64KiB cache to 3 cycles by reducing its associativity to 4 ways. However, these configurations are not possible with VIPT and 4KiB pages because they need more index bits than the 12 offset bits of the page.

Technology	32 nm
Cache line size	64 Bytes
Capacity	16 KiB, 32 KiB, 64 KiB, 128 KiB
Associativity	2-way, 4-way, 8-way, 16-way, 32-way
Access mode	Parallel data and tag access
Ports	1 or 2 for read, 1 for write
Banks	1, 2 or 4 banks

Tab. I: L1 cache configurations.

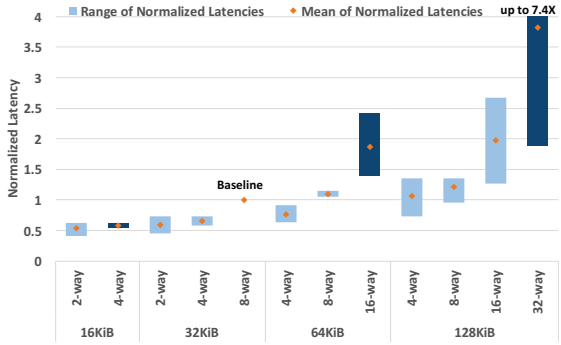

Fig. 1: L1 latency (range and mean) relative to \$32KiB 8-way baseline.

Configurations that are not feasible with VIPT are shaded light blue while those that are feasible are shaded dark blue. Unfortunately, the most desirable configurations are infeasible. While CACTI is a rough model, we expect generally the same trends (though different values) for real designs.

B. Impact of L1 Configuration on Performance

Methodology. Our analysis of L1 configurations shows that there are significant tradeoffs in capacity, associativity, and latency. The question we answer now is what impact these tradeoffs have on application performance. We simulate application performance with the cycle-based Macsim simulator [27] coupled with DRAMSim2 [28]. Macsim's trace generator is modified to capture virtual address, physical address, and page flags for every memory access. We use Linux's `pagemap` and `kpageflags` [29] interfaces to collect physical page number and page-flag information (I.e., whether allocated as a Hugepage).

We run entire the SPEC CPU 2006 [30] suite with reference inputs, except for xalancbmk, namd, and dealII because of simulation issues (though we do run xalancbmk from SPEC 2017). We also include all SPEC CPU INT 2017 [31] applications, except for gcc, omnetpp, and specrand_is, In addition to the large-memory footprint applications of SPEC INT 2017 (>8GiB for many), we also evaluate large-memory big-data applications: graph500 (graph processing) [32] and DBx1000 with the ycsb workload (database) [33], each configured to use more than 4GiB of memory. For each application we collect 500 million instructions at a SimPoint [34]. While different applications exhibit different characteristics, both big-data applications and the large-footprint SPEC 2017 applications are not outliers w.r.t. SPEC CPU 2016 applications. Because of space constraints, we show individual

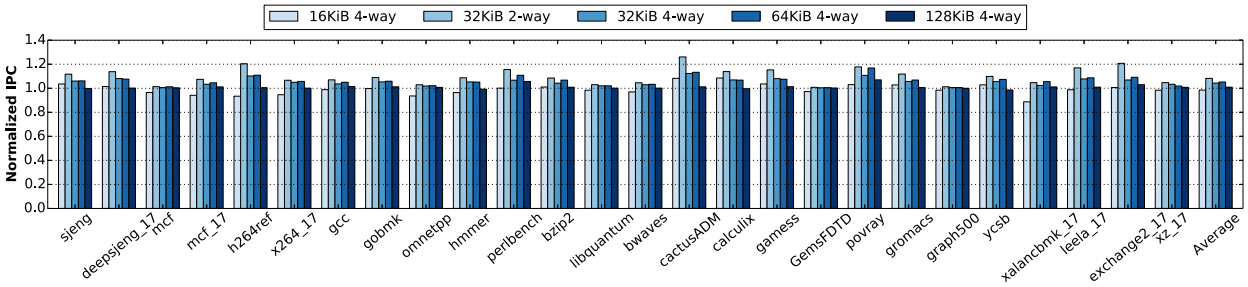

Fig. 2: IPC with various L1 cache configurations for an OOO core, normalized to the baseline L1.

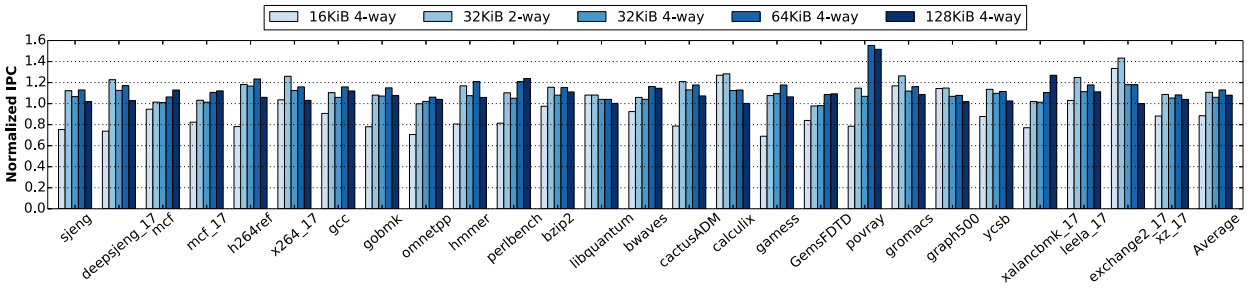

Fig. 3: IPC with various L1 cache configurations for an in-order core, normalized to the baseline L1.

results from a representative subset of applications, however, we include all results in any average number.

We simulate both an OOO core with a 3-level cache hierarchy and an in-order core with a 2-level cache hierarchy with the configuration parameters summarized in Tab. II. Based on the results shown in Fig. 1, we select four desirable configurations (Tab. II). Note that these configurations are not feasible due to the VIPT cache-indexing constrains. However, we model them as ideal caches, assuming the index bits are always correct. In addition, we add a 16KiB cache that trade capacity for lower associativity (4-way) and latency (2-cycle). We report IPC normalized to the baseline L1 and use harmonic means for speedup averages.

	In-Order processor with 2-level cache hierarchy		**Out-of-Order processor with 3-level cache hierarchy**	
Core	2-wide, in-order 3.0 Ghz		6-wide issue, OOO 192-ROB, 3.0 GHz	
TLB	L1: 64-entry, 4KiB pages; 32-entry, 2 MiB pages, 2-cycle L2: 1024-entry unified, 7-cycle			
L1	Configuration	Latency	Energy per access	Static power
	32KiB 8-way VIPT	4-cycle	0.38 nJ	46 mW
	32KiB 2-way SIPT	2-cycle	0.1 nJ	24 mW
	32KiB 4-way SIPT	3-cycle	0.185 nJ	30 mW
	64KiB 4-way SIPT	3-cycle	0.27 nJ	51 mW
	128KiB 4-way SIPT	4-cycle	0.29 nJ	69 mW
	Slow access in SIPT starts right after TLB access			
L2	None		256 KiB, 8-way, 12-cycle, private, 0.13 nJ per access, 102 mW static power	
LLC	1 MiB, 16-way, 20-cycle, shared, 0.29 nJ per access, 532 mW static power		2 MiB 16-way, 25-cycle, shared, 0.35 nJ per access, 578 mW static power	
DRAM	8-bank, 4-channel, DDR3, 16 GiB total			
Note	LLC size increase proportional to core count for multi-core evaluation.			

Tab. II: Simulated system configurations

Analysis. Fig. 2 shows the sensitivity of performance to L1 cache configuration on an OOO core with a 3-level cache hierarchy. Latency has a significant impact and the configuration with shortest latency (32KiB 2-way with 2-cycle latency) performs best, except for xalancbmk_17, which exhibits minor benefits with a 64Kib 4-way L1. Overall, the low-latency configuration can improve performance by 8.2% on average. A 16KiB, 4-way latency cache, which also has a 2-cycle latency and meets VIPT constraints, outperform the baseline in some application (e.g., cactusADM) but overall performs worse than baseline (1.5% slower on average).

However, with an in-order core (Fig. 3), the configuration with balanced latency and capacity (64KiB 4-way with 3-cycle latency) achieves the best performance improvement, 13% on average. In some applications such as calculix and exchange2_17, latency still shows noticeable impact, but most applications benefit more from larger capacity. This could be due to the absence of L2 caches and because an in-order pipeline is less capable of hiding cache misses. As expected, the performance degradation with a 16KiB 4-way cache is much larger (11.3% worse than baseline on average). While the performance improvements are significant, these cache configurations are not feasible because of the VIPT indexing constrains.

IV. Speculatively Indexed Physically Tagged Caches

The simplest variant of SIPT always speculatively accesses the cache assuming that all necessary index bits will remain the same after translation, including those beyond the page granularity. SIPT, Like VIPT, performs address translation in parallel to accessing the cache arrays (Step 1 in Fig. 4).

For performance, all ways are accessed together so that overall latency can be reduced. After address translation, all cache tags read in Step 1 are compared with the physical address to select the correct cache line. At the same time, SIPT compares those index bits that were speculated with their values after address translation (Step 2 in Fig. 4). If all speculated bits indeed are the same the "fast" access completes (Step 3). If any of the speculated bits do not match, the cache request must be repeated with the correct index bits from the PA (Step 4), slowing down the access. Fast accesses are as fast as a VIPT cache, or faster if the relaxed design constraints enable a lower-latency configuration. A slow access, on the other hand, only issues after address translation like a PIPT cache. In addition every slow access wastes energy and contends for the L1 cache port. Note that there are no coherence implications because only the L1 cache is accessed speculatively and no action (other than another access) is taken on a misprediction (tag mismatch).

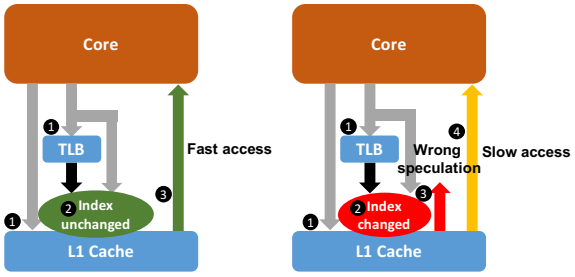

Fig. 4: SIPT cache access when speculated index is unchanged (left) and changed (right).

A. *Speculation Accuracy*

Fig. 5 shows the percentage of memory accesses that are speculated correctly, depending on how many speculative index bits are required. Each component of a stacked bar in the figure represents fast accesses for the number of bits required and all other accesses are slow. The most strict scenario, *hugepage* includes only those accesses from huge pages (for which 21 address bits are guaranteed not to change).[1] While some applications (e.g., libquantum and GemsFDTD) have most accesses targeting transparently mapped huge pages, many others have the vast majority of accesses to normal 4KiB pages. Those applications with a low correct speculation rate are likely to suffer performance degradation with SIPT compared to the VIPT baseline.

Reasonable L1 configurations, however, do not use 2MiB ways (2^{21}) and only require $1-3$ index bits beyond the page granularity. In these scenarios, the correct prediction rate is much higher overall. If only a single speculative index bit is

[1]We run Linux 3.12 with transparent hugepage management turned on and collect our traces on a system that is regularly used and which had an uptime of weeks. We later discuss a system which is artificially extremely fragmented.

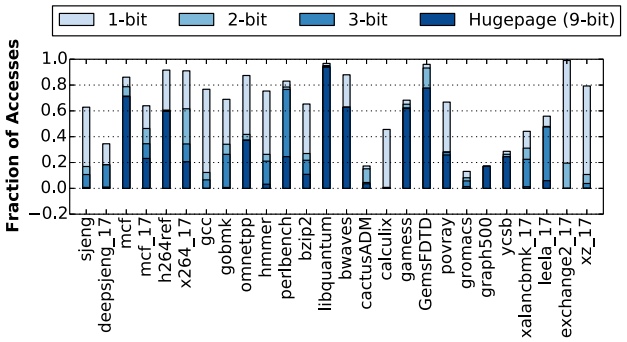

Fig. 5: Fraction of correct speculations vs. the number of index bits that are speculated as unchanged.

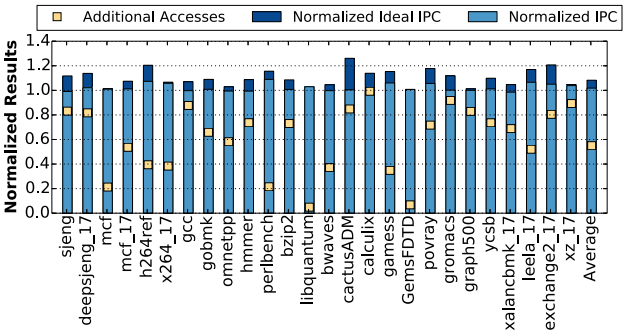

Fig. 6: IPC and additional L1 accesses with a naive SIPT 32KiB/2-way/2-cycle cache for an OOO core normalized to the baseline L1.

needed (e.g., for a 32KiB 4-way L1), all but seven applications (deepsjeng_17, CactusADM, calculix, graph500, ycsb, xalancbmk_17, and gromacs), have majority fast accesses.

B. *Naive SIPT Performance*

To evaluate the performance of this naive version of SIPT, which always speculates, we evaluate different SIPT configurations with an OOO core, and compare to the baseline L1. For brevity we only show the results for the 32KiB 2-way SIPT configuration (with 2 extra index bits), which performs the best as an ideal cache in an OOO processor (Section III). Fig. 6 summarizes the results and shows the IPC normalized to the baseline, and compares with the ideal cache. The figure also shows relative extra accesses due to misspeculation $\left(\frac{\text{accesses}_{SIPT}}{\text{accesses}_{baseline}} - 1\right)$.

SIPT caches with lower associativity and shorter latency achieve IPC improvement in many applications; e.g., h264ref, and perlbench, exhibit 7.3% and 8.9% IPC speedup. However, because speculative bits are used, naive SIPT suffers a high misspeculations rate. For example, in some applications, e.g., calculix and gromacs, less than 5% speculations succeed with 2 extra index bits. When misspeculations happen, SIPT generates slow accesses.

Similarly in Fig. 7, we show the relative energy $\left(\frac{E_{\text{SIPT}}}{E_{\text{baseline}}}\right)$ of the whole cache hierarchy and also compare with the

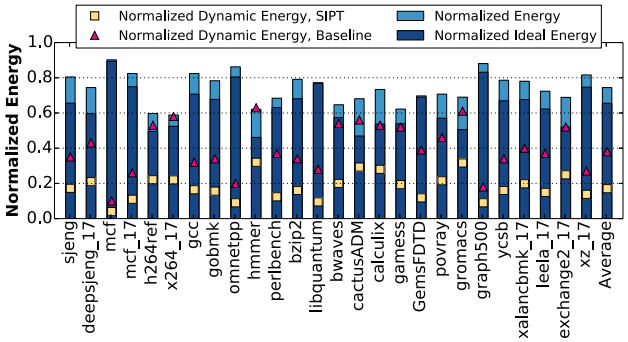

Fig. 7: Cache hierarchy energy of naive SIPT 32KiB/2-way/2-cycle for an OOO core normalized to baseline.

ideal cache. Some applications such as libquantum and GemsFDTD exhibit energy savings close to ideal. However, many other applications exhibit significant gaps between naive SIPT and ideal. On average, naive SIPT reduces total cache energy to 74.4%, which is 8.5% worse than ideal. We also show the relative dynamic energy $\left(\frac{E_{\text{dynamic}}}{E_{\text{baseline\_total}}}\right)$ for both SIPT and baseline. SIPT reduces dynamic energy significantly.

Overall, due to a high misspeculation rate, the naive SIPT is far from ideal. We now aim to both reduce extra L1 accesses and increase the number of fast accesses.

V. Misspeculation Prediction

While the naive SIPT design is simple, speedup and energy savings are hampered by misspeculation overheads. In this section we evaluate a light-weight predictor that determines whether a fast access is likely to succeed and speculation should proceed, or whether cache access should wait until after translation. Our goal is to make a speculate/no-speculate binary decision early in the pipeline to hide its latency. We present the evaluation of a small PC-based Perceptron predictor [35]. We experimented with simpler counter-based predictors, but their accuracy is inferior.

We base our Perceptron predictor design (Fig. 8) directly on the smallest global-history configuration proposed by Jimenez and Lin [35]. We add a global history register $x_1x_2...x_h$ that tracks the last h speculation outcomes as ones and zeros (fast access success or extra cache access failure). The predictor itself has 64 entries each being a perceptron of $h+1$ weights $w_0w_1...w_h$. We use the memory operation program counter (PC) to index the 64-entry predictor table. Because we only use the PC, the prediction can be overlapped with other pipeline stages.

Perceptron calculates a prediction (y) by performing a dot product of the history and the weights of a specific entry in the table plus a learned bias: $y = w_0 + [x_1x_2...x_h] \cdot [w_1...w_h]$. If y is positive, we predict the index will not change and will continue with a fast access using the speculative index. If y is negative we bypass speculation and wait for the physical address before accessing the cache. Other than the smaller number of entries, all details, training algorithm, and other parameters precisely follow those of Jimenez and Lin [35], and we do not describe them in this paper.

We estimate the overhead of this perceptron predictor at just 624B of storage and a small amount of logic (6b weights, 13 weights per perceptron, 64 perceptrons). We model perceptrons as RAM with Cacti [36] (Tab. I). The dynamic energy for reading a perceptron is only 0.34% of a baseline L1 cache access. The static power is only 0.0007% of the baseline L1 cache. Song et al. [37] suggest a 32-bit integer addition consumes $\frac{1}{10}$ the energy of reading 32 bits from a register file. Since $x_1x_2...x_h$ are ones and zeros, $y = w_0 + [x_1x_2...x_h] \cdot [w_1...w_h]$ is essentially adding $h+1$ (13 in our implementation) 6-bit integers and therefore estimated to consume less energy than reading the perceptron. Training consumes similar energy. This predictor introduces no extra latency and only negligible area and energy overheads.

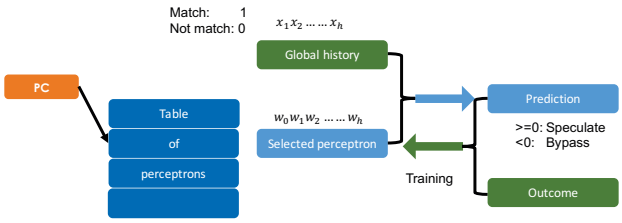

Fig. 8: Perceptron-based predictor.

We evaluate the predictor by considering 4 possible outcomes. If the speculated bits are unchanged by translation and the predictor decides to speculate, we call it *correct speculation*. If the speculated bits are changed by translation and the predictor decides to bypass, we call it *correct bypass*. If the speculated bits are unchanged by translation but the predictor chooses bypass, an opportunity for fast access was squandered, we call this *opportunity loss*. Finally, if the speculated bits are changed by translation and the predictor chooses to speculate, an *extra access* is generated.

This simple perceptron predictor achieves more than 90% accuracy in all applications; in fact most applications have far fewer than 5% extra L1 accesses and negligible opportunity loss (Fig. 9). We also evaluated the sensitivity of the predictor parameters such as increasing the number of perceptrons and increasing the history length. Our experiments did not show strong sensitivity to these parameters, most likely because the prediction rate is already high. We do not warm up the predictor and the results include all mispredictions. We also evaluated various counter-based predictors, but their average accuracy is only $\sim$ 85% and not consistent across applications. We omit the results for brevity and because the perceptron already has low overhead.

While the predictor practically eliminates extra accesses due to SIPT and thus saves significant energy, it fails to reduce the extra latency from slow accesses and hence cannot improve performance.

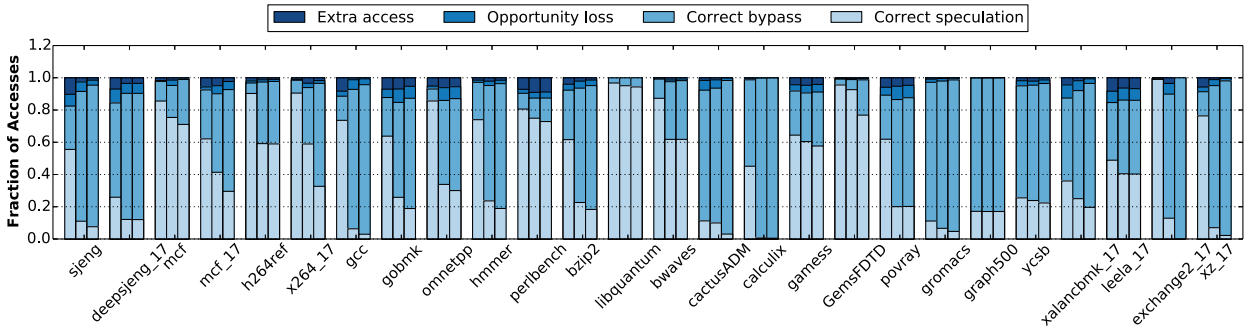

Fig. 9: Breakdown into four possible prediction outcomes; each group of 3 bars represents 1, 2, and 3 speculated index bits (left to right, respectively).

VI. Partial Index Prediction

Intuitively, if the speculation bypass predictor is so accurate at predicting whether speculative indexing can proceed, perhaps it can be extended to predict the actual post-translation index value. Consider an SIPT design that requires a single speculative index bit. In this case, if the bypass predictor predicts to bypass, it is in effect indicating that the speculative bit is most likely not remaining the same. Therefore, flipping the bit in these cases will lead to the correct post-translation index. Because the prediction accuracy is so high, few extra accesses are added by this technique.

When there are multiple speculative tag bits, we need to predict their exact values, which is generally hard because with 3 speculative bits, it is likely that they may take any of 8 possible values. This requires a complex predictor or resulting many misspeculations. However, in the context of SIPT, predicting values is doable because of spatial locality in memory address mapping. Prior work [38], [39] establishes that memory addresses are usually mapped in coarse-grained blocks even without considering huge pages. And Pham et al. [40], [41] suggests the spatial contiguity between the virtual page numbers and physical page numbers.

Linux manages free pages using the buddy algorithm. Free pages are grouped into $1, 2, 4, \ldots 1024$ contiguous page frames and page groups of each size are then stored in linked lists. This scheme keeps the overhead of tracking free pages low.

Consider the scenario in which a user program allocates a large number of pages. This is common behavior when programs set up data structures during their initialization. The number of pages at fine-grained groups is unlikely to satisfy such requests. As a result, the buddy algorithm has to break large groups to satisfy bursts of memory allocation requests, which can lead to a significant amount of contiguous physical pages being mapped to a contiguous virtual address space. Other allocators, such as slab and eager paging [38], maintain contiguity explicitly. OS features such as Hugepages and page coloring also increase the occurrence of contiguously mapped memory blocks; page coloring tries to maintain the same low-order address bits between the VA and PA to maximize usage of the LLC.

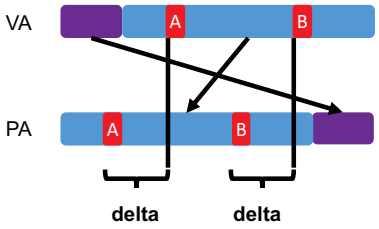

Fig. 10: Deltas between virtual and physical addresses are constant within a single large block.

The fact that large contiguous blocks exist aids with predicting speculative index bits. For all addresses in one contiguous range (A and B for example in Fig. 10), the delta between a virtual address and its corresponding physical address is the same. Software may even optimize for SIPT, though prediction rates are already high.

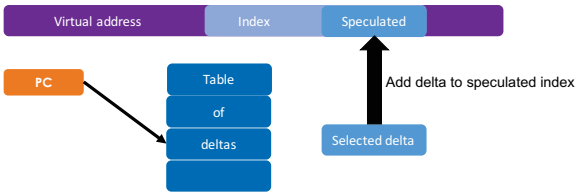

Fig. 11: The index delta buffer predicts the delta between the speculative virtual address bits and corresponding physical address bits.

The prediction benefits from, but does not solely rely on, a coarse-grained memory mapping. Even when memory is highly fragmented and lacks multi-page contiguity, or when applications make only small allocations, deltas within each page are fixed. Thus only the first access to a page will mispredict; there are typically many L1 accesses per page.

Instead of predicting a value, we predict the VA to PA delta, which is the same for the entire range. Specifically for SIPT, only the delta of the speculative partial tag bits is required. We propose the *index delta buffer* (IDB) to predict these narrow delta values. Similar to a branch target buffer (BTB), the IDB is a PC-indexed table with each entry storing a (speculative)

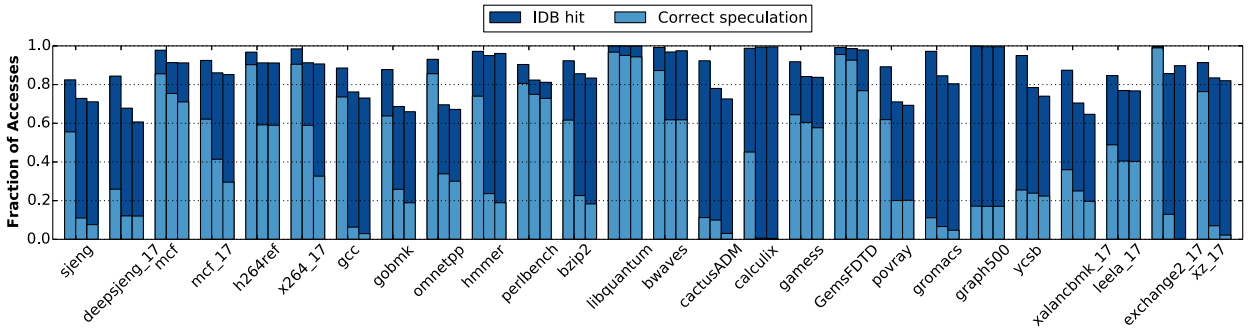

Fig. 12: Prediction accuracy of the combined predictor when attempting to predict 1, 2, and 3 speculative index bits (the left, middle, and right bar within each group, respectively).

index delta.

In addition to updating the prediction history, we also update the expected deltas, which remain stable as long as the same regions are accessed.[2] To compute the speculative index, we add the delta to the virtual address and truncate if it overflows. Fig. 11 shows the design of IDB. For simplicity, we keep the same number of IDB entries as the perceptron-based bypass predictor. The storage overhead of IDB is very small because each entry is the same size as the (already small) number of speculative bits. The IDB is also accessed during fetch or decode and is off the critical path. The predicted 3-bit delta is added to the VA after address generation. The latency of the 3-bit add, which does not propagate the carry, should not increase cycle latency.

A. Combined Speculation Bypass and Index-Bit Value Prediction

Our overall index-bit predictor proceeds in two stages. First, the perceptron predictor is queried. If perceptron predicts to speculate, the speculative index is used immediately. If perceptron predicts to bypass speculation, the IDB is queried and its predicted index bit values are used to access the cache with a speculative index. Like naive SIPT, this combined predictor always accesses the L1 before translation. As long as IDB predicts correctly, slow accesses are converted to fast ones. However, because we more aggressively access the L1, more extra accesses are likely. When there is only one speculative index bit, we do not use the IDB and follow the intuitive reversed prediction technique explained earlier.

Fig. 12 presents the accuracy and effectiveness of the combined speculation bypass and IDB predictor. We consider three possible outcomes. The first is are correctly-speculated fast access by the bypass predictor (in which case the IDB is not accessed). The second is the fraction of accesses that were predicted to bypass speculation and for which the IDB predicted the correct speculative index bit values (*IDB hits*); these would be slow accesses without the IDB that are fast accesses with it. All remaining accesses are slow and generate extra L1 accesses. Note that we also label as IDB hits those fast accesses that use the reversed bypass prediction. As with the perceptron bypass predictor, IDB consumes little area and power. We estimate the total overhead of the combined predictor at $< 2\%$ of L1 cache area and energy.

When only a single bit value needs to be predicted, over 90% (and usually close to 100%) of all accesses are fast accesses. This is in contrast to the speculation bypass predictor alone, which leaves up to 80% of accesses waiting for address translation to complete; all seven applications with low speculation rate, now have majority fast accesses (e.g., CactusADM and gromacs both go from under 20% fast accesses to more than 95% fast accesses). With 2 and 3 speculative index bits, the combined predictor successfully convert many slow accesses into fast ones. For example, gcc, calculix and xz_17 exhibited nearly no fast accesses with the bypass predictor because of poor locality between virtual and physical address. With the IDB, however, more than 70% of accesses are fast.

For brevity we discuss only the performance of the OOO configurations in detail. In-order performance is presented as a sensitivity study in Section VII. Fig. 13 shows that SIPT with IDB approaches the performance of the ideal cache. The IDB enables many more fast accesses and the slow L1 accesses do not significantly hamper performance. Overall, the 32KiB 2-way SIPT cache (with 2 speculative index bits) achieves an average (harmonic mean) of 5.9% IPC speedup, only 2.3% away from ideal. In some applications (e.g., h264ref, cactusADM, calculix, leela_17, exchange2_17, and gromacs), SIPT shows more than 10% performance improvement and never underperforms baseline.

Fig. 14 tells a similar story for energy that the SIPT with combined predictor approaches the efficiency of an ideal cache. The energy numbers are a bit further from ideal (2.4%) than speedup because of the extra L1 accesses generated by the aggressive index-bit value speculation.

Again, even we only show the result for the SIPT configuration that performs the best in Section III, the trend that

[2]Deltas may also change when memory is remapped explicitly with `munmap`, by copy-on-write, or on major page faults. We find that such events are very infrequent and omit the analysis for brevity. Our evaluation includes configurations that are far-worse for IDB.

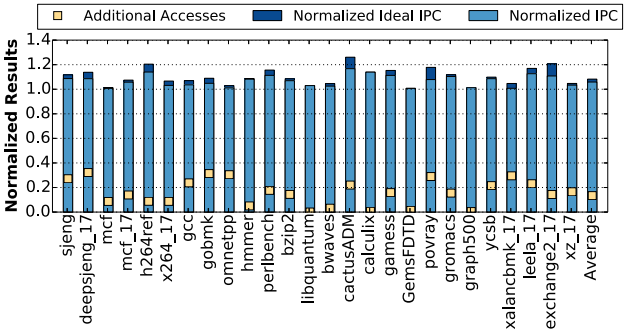

Fig. 13: IPC and additional L1 accesses with a 32KiB/2-way/2-cycle SIPT cache with IDB for an OOO core normalized to the baseline L1.

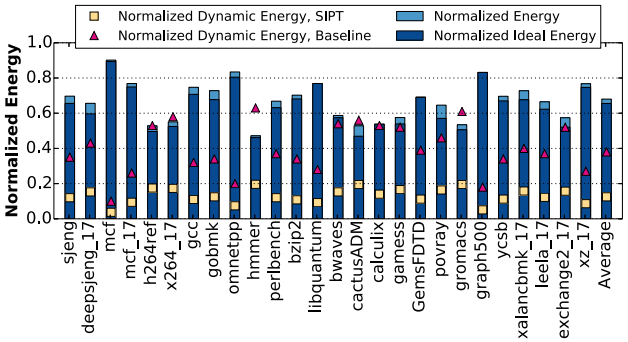

Fig. 14: Cache hierarchy energy with a 32KiB/2-way/2-cycle SIPT+IDB for an OOO core normalized baseline.

SIPT with combined speculation bypass and IDB prediction yield speedup and energy saving very close to the ideal cache configurations exist in all SIPT configurations.

B. Multicore Evaluation

We focus on single-core evaluation because the L1 is so tightly integrated with the core. We also evaluate SIPT in a multicore system. We simulate a quad-core processor and also quadruple the capacity of the last-level cache. We construct 11 multi-programmed workloads by mixing applications used in the single-core evaluation; every application is used at least once and the workloads are listed in Tab. III. We recycle traces until the last core completes its initial trace to maintain a consistent level of resource contention. We report sum-of-IPC speedup, which is a simple metric that captures overall throughput improvement (speedup is relative to the multicore with the baseline cache, not a single-threaded baseline).

Fig. 15 examine SIPT for an OOO quad core. The most obvious difference between the two sets of results is that using application mixes decreases the variability in SIPT impact between different workloads. This is expected. Also as expected, using a multicore does not change any of the conclusions about the benefits of SIPT for private L1 caches.

The 32KiB 2-way cache performs the best of all configurations. Overall, the average IPC improvement is 8.1%, slightly better than we observed with a single core. The reason for the increased speedup is larger pressure on the LLC and main memory, such that L1 cache performance has a greater role. In fact, individual application speedup on each core is nearly-identical to the single-core experiments. We expect this because there is no sharing and no contention in this multiprogrammed environment.

Mix0	h264ref, hmmer, perlbench, povray
Mix1	mcf, gcc, bwaves, cactusADM
Mix2	gobmk, calculix, GemsFDTD, gromacs
Mix3	astar, libquantum, lbm, zeusmp
Mix4	mcf, perlbench, leslie3d, milc
Mix5	h264ref, cactusADM, calculix, tonto
Mix6	gcc, libquantum, gamess, povray
Mix7	sjeng, omnetpp, bzip2, soplex
Mix8	graph500, ycsb, mcf, povray
Mix9	mcf_17, xalancbmk_17, x264_17, deepsjeng_17
Mix10	leela_17, exchange2_17, xz_17, xalancbmk_17

Tab. III: Multi-programmed workloads.

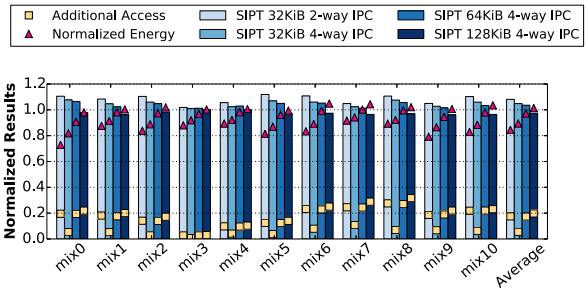

Fig. 15: IPC, extra L1 accesses, and cache hierarchy energy of SIPT with IDB for an OOO quad core; IPC and energy normalized to the baseline L1 cache.

SIPT is also able to reduce the total cache energy, but to a smaller degree than with a single core. We attribute this to the longer overall run times resulting from interference at other levels of the memory hierarchy; the static energy component is relatively larger in the multicore so the impact of SIPT is lessened.

VII. Further Discussion

A. Way Prediction

Way prediction [20] saves access energy compared to set associative caches where all cachelines in the same set are fetched in parallel. By predicting the data location within the cache set, only the predicted cache line is fetched. If the prediction is correct, no additional access latency is introduced. When the prediction is wrong, a second access is required to search the remaining cachelines in the set.

We evaluate the simple way prediction mechanism described in [20] that the MRU way in a set is always predicted. A small amount of metadata (3 bit per set for an 8-way set associative cache) is accessed before cache is accessed. Although, fancy predictors may increase the accuracy of way prediction, we find that the accuracy of this simple predictor is already high and robust across applications. Unlike SIPT, way prediction requires the virtual address, thus cannot be

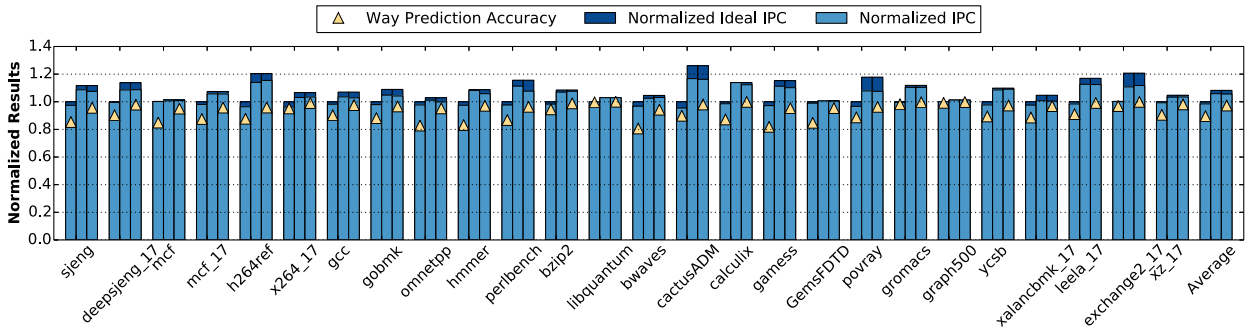

Fig. 16: IPC normalized to the baseline L1 and way prediction accuracy. Each group from left to right: baseline L1 with way prediction, 32KiB/2-way/2-cycle SIPT with IDB, and 32KiB/2-way/2-cycle SIPT with both IDB and way prediction.

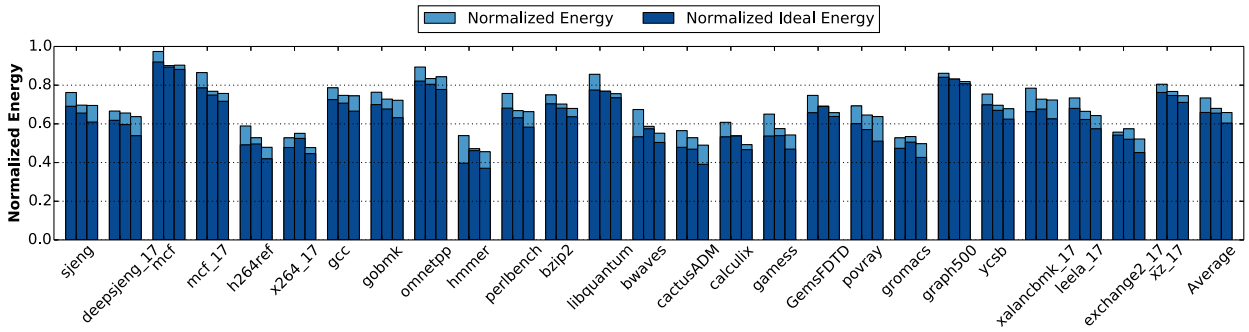

Fig. 17: Cache hierarchy energy normalized to the baseline L1. Each group from left to right: baseline L1 with way prediction, 32KiB/2-way/2-cycle SIPT with IDB, and 32KiB/2-way/2-cycle SIPT with both IDB way prediction.

fully overlapped with early pipeline stages. Employing more complex metadata may introducing extra latency to cache accesses. We stay with this simple prediction mechanism and optimistically modeled no extra latency for accessing way prediction metadata.

As mentioned in Section II, way prediction and SIPT are complementary. We apply way prediction to both the baseline and our 32KiB/2-way/2-cycle SIPT cache with combined speculation bypass and IDB prediction. The results are shown in Fig. 16. And Fig. 17 shows the normalized cache energy. In addition to being ideally indexed, ideal caches also assume way prediction always accesses the correct way. We model the energy of way prediction by reducing the relative dynamic energy according proportionally to the associativity (e.g., one-eighths of dynamic energy consumed in the baseline 8-way cache on a way prediction hit).

When applied to the baseline cache, way prediction achieves 89% accuracy and reduces cache energy by 24% on average. However, the remaining misses still reduce performance by 2% overall. When applied on top of SIPT, because of reduced associativity (from 8 in baseline to 2 in SIPT), the way prediction accuracy increases to 97.3% on average, and there is only a 0.3% performance drop compared to SIPT alone. At the same time, it saves 2.2% additional cache energy (compare to SIPT alone), superior than ideal way prediction. By reducing associativity to 2, SIPT alone already saves significant cache energy, and there is only limited space for further reduction. The saving from applying way prediction on top of SIPT is very stable among all applications because of the robust and high accuracy.

B. *Predictability of Partial Index Bits*

The efficiency of SIPT relies on the accuracy of partial index-bit prediction. We evaluate SIPT with a regularly used machine that has an uptime of weeks. However, this is not necessarily the worst condition an application faces. We now discuss a few sensitivity studies with more severe operating conditions, including running applications with artificially highly fragmented physical memory and with Linux's transparent huge page mechanism turned off (thus only fine-grained 4KiB pages). We also include the impact of these parameters on the performance of the in-order machine first presented in Section III.

Fragmented Memory. On a long-running system with a large number of co-running applications, the physical memory may be fragmented. When running applications on fragmented physical memory, the lack of physical memory contiguity may decrease the predictability of partial index bits. We use a tool from Kwon et al. [42] to fragment memory and quantify fragmentation using the *unusable free space index* [43], a value between 0 (unfragmented) and 1 (highly fragmented). Importantly, this index does not represent lack of memory, but rather an inability to satisfy large contiguous requests. It can be calculated with $F_u(j) = \frac{TotalFree - \sum_{i=j}^{i=n} 2^i k_i}{TotalFree}$

where $TotalFree$ is the size of the free space, 2^n is the largest allocation that can be satisfied, j is the order of the desired allocation size (2MiB for THP), and k_i is the number of free page blocks of size 2^i. We maintain an unusable free space index > 0.95; an extreme level of fragmentation at nearly all times and not representative of typical operation. Again, we never run out of physical memory in any experiment.

To evaluate SIPT under highly fragmented physical memory, we repeat the same simulations we conducted before with traces collected under this extreme condition.

4KiB pages. Most Linux distributions enable transparent huge pages (THP) by default. We are aware that under certain circumstances, THP hurts performance [44] and should be disabled but believe this is not the general case. However, an interesting question is what impact disabling THP has on SIPT prediction accuracy and performance. Similarly, we repeat the same simulations with THP disabled, which forces all pages to 4KiB.

Removing $>$ 4KiB contiguity. A more challenging (but unrealistic) condition is to force zero contiguity beyond 4KiB pages such that all 4KiB pages are mapped with different deltas. This essentially eliminates all benefits from contiguous memory mapping, thus IDB only works for references within the exact same page. We force this in simulation by tracking the page number of the last access for each IDB entry. We then apply delta prediction only if the same page is accessed and choose a random delta if a different page is accessed; this mimics zero contiguity without modifying the OS. Note that this scenario presents locality and randomness that exceed that of any reasonable system and dynamic VA to PA remappings.

Fig. 18 shows the average IPC, cache energy normalized to the baseline, and the prediction accuracy (the percentage of *correct speculation* and *IDB hit*) for all four SIPT configurations on both OOO and in-order cores. As expected, running with fragmented physical memory or disabling THP does degrade the behavior of SIPT. However, the degradation is not significant. For instance, with a 32KiB 2-way SIPT cache (2-bit speculation) on OOO core, the prediction accuracy drops from 86.7% to 84% when running with fragmented physical memory, 83.1% when THP is disabled, and 73% when no $>$ 4KiB contiguity. The IPC improvement also drops from 5.9% to 5.3%, 4.8% and 3.8%, and cache energy increases from 67.8% to 68.3%, 70% and 71.2%. A similar trend can be observed in all SIPT configurations on both OOO and in-order cores. As mentioned in Section VI, our prediction mechanisms benefits from contiguous memory mapping but does not solely rely on it. Using extremely fragmented memory or disabling THP has limited impact.

C. Implications for Instruction Schedulers

Modern processors commonly support speculative scheduling to enable back-to-back execution of dependent instructions [45], [46]. The speculation relies on deterministic

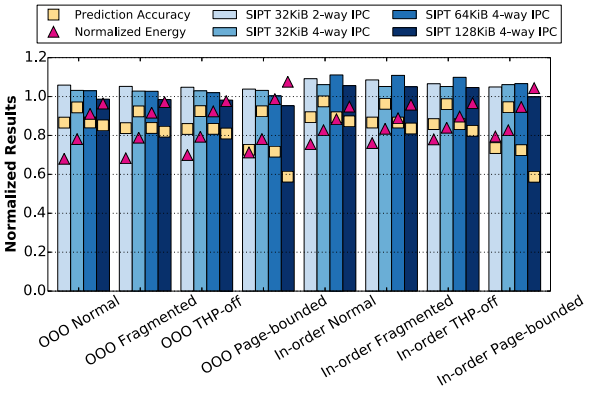

Fig. 18: IPC, cache hierarchy energy, and prediction accuracy on OOO and in-order core with various operating conditions; IPC and energy normalized to the baseline L1.

instruction latencies with support for rare instruction replay when latency varies for certain instructions. Load instruction latency is variable because of cache misses and way prediction and SIPT introduce another source of variability. As suggested in [46], an ideal selective replay mechanism (replay only necessary instructions) is not feasible for wide issue OOO processors and replay mechanisms require trading off design complexity (HW resources) and accuracy (performance impact). SIPT is very accurate and can use existing speculative-scheduling approaches to recover from its rare mispredictions, which are a fraction of cache misses that are already addressed by the scheduler.

Impact on the scheduler can be further reduced because SIPT has a built-in confidence estimator. Similarly to prior designs, expensive selective-replay resources can be reserved for more challenging loads [45], [47], [46]. Loads that SIPT predicts their speculated bits to be unchanged, an alternative simpler replay mechanism with larger penalty can be used instead of selective replay. In many applications (e.g. libquantum, zeusmp) nearly all loads do not require selective replay.

VIII. Conclusion

We show that popular VIPT L1 cache indexing constrains the cache design space, leading to a suboptimal L1 data cache configuration. We argue that it is undesirable to adopt VIVT caches, with their added hardware, design, and verification complexity or expose hardware tradeoffs to software. Instead, we develop a novel pure-microarchitecture mechanism that relaxes design constraints for a physically-addressed L1.

Speculatively Indexed Physically Tagged cache indexing allows arbitrary capacity and associativity. When the number of index bits exceeds virtual page granularity, SIPT predicts values for those speculative index bits. We discuss three ways in which those index bits can be predicted and conclude that a simple predictor, inspired by the perceptron and BTB branch direction and target predictors, achieves very high prediction accuracy at very low cost; less than 2% of L1 area and energy and with a short latency that is hidden by the pipeline.

We demonstrate, through extensive experimentation, that SIPT improves performance and saves energy. On average, SIPT improves performance (IPC speedup) by 13.4% and 8.1% on in-order and out-of-order quad-core processors, respectively. SIPT reduces the total cache hierarchy energy by 11.9% and 15.6% for those same designs. More importantly, the design space is unconstrained, which may enable different tradeoffs in other scenarios, such as more aggressive and dynamic partitioning and power-management techniques.

REFERENCES

[1] A. Sodani and C. Processor, "Race to exascale: Opportunities and challenges," in *Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture*, 2011.

[2] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 535–546, ACM, 2013.

[3] J. R. Goodman, "Coherency for multiprocessor virtual address caches," in *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, (Los Alamitos, CA, USA), pp. 72–81, IEEE Computer Society Press, 1987.

[4] J. Kim, S. L. Min, S. Jeon, B. Ahn, D.-K. Jeong, and C. S. Kim, "U-cache: a cost-effective solution to synonym problem," in *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pp. 243–252, 1995.

[5] X. Qiu and M. Dubois, "The synonym lookaside buffer: A solution to the synonym problem in virtual caches," *IEEE Transactions on Computers*, vol. 57, pp. 1585–1599, Dec 2008.

[6] H. Yoon and G. S. Sohi, "Revisiting virtual l1 caches: A practical design using dynamic synonym remapping," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 212–224, March 2016.

[7] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Part1, Chapter 2." http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html.

[8] Linux Kernel Documentation, "Direct Access for files." www.kernel.org/doc/Documentation/filesystems/dax.txt.

[9] A. Basu, M. D. Hill, and M. M. Swift, "Reducing memory reference energy with opportunistic virtual caching," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 297–308, IEEE Computer Society, 2012.

[10] Bruce Jacob, "The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It," 2009.

[11] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface," 1994.

[12] freebsd.org, "Page Coloring." https://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/page-coloring-optimizations.html.

[13] netbsd.org, "Improving NetBSD/mips." http://www.netbsd.org/~matt/bsdcan2012.pdf.

[14] Jacob Bramley, "Page Colouring on ARMv6 (and a bit on ARMv7)." https://community.arm.com/groups/processors/blog/2012/05/14/page-colouring-on-armv6-and-a-bit-on-armv7.

[15] M. J. Bligh and D. Hansen, "Linux memory management on larger machines," in *Proc. Linux Symposium*, 2003.

[16] Linux Kernel Documentation, "Huge TLB Page Support in Linux." https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[17] A. Arcangeli, "Transparent hugepage support," in *KVM Forum*, 2010.

[18] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 1–12, ACM, 2015.

[19] G. Taylor, P. Davies, and M. Farmwald, "The tlb slice-a low-cost high-speed address translation mechanism," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, (New York, NY, USA), pp. 355–363, ACM, 1990.

[20] K. Inoue, T. Ishihara, and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," in *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, ISLPED '99, (New York, NY, USA), pp. 273–275, ACM, 1999.

[21] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *Proceedings of the 26th International Symposium on Computer Architecture (ISCA-26)*, May 1999.

[22] R. E. Kessler, "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, pp. 24–36, Mar. 1999.

[23] H. de Vries, "Understanding the detailed architecture of amds 64 bit core," *Chip Architecture*, 2003.

[24] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pp. 29–40, Sept 2009.

[25] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift, "Performance analysis of the memory management unit under scale-out workloads," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–12, Oct 2014.

[26] Muralimanohar, Naveen and Balasubramonian, Rajeev and Jouppi, Norman P, "CACTI 6.0: A Tool to Model Large Caches." www.hpl.hp.com/techreports/2009/HPL-2009-85.html.

[27] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide,"

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

[29] Linux Kernel Documentation, "Pagemap, from the userspace perspective." https://www.kernel.org/doc/Documentation/vm/pagemap.txt.

[30] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[31] spec.org, "SPEC CPU 2017." https://www.spec.org/cpu2017.

[32] Graph 500, "Graph 500." http://www.graph500.org/.

[33] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proc. VLDB Endow.*, vol. 8, pp. 209–220, Nov. 2014.

[34] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, (New York, NY, USA), pp. 45–57, ACM, 2002.

[35] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, (Washington, DC, USA), pp. 197–, IEEE Computer Society, 2001.

[36] HP Labs, "CACTI Pure RAM Interface." http://quid.hpl.hp.com:9081/cacti/sram.y.

[37] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, (Cambridge, MA, USA), pp. 1135–1143, MIT Press, 2015.

[38] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 66–78, ACM, 2015.

[39] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 237–248, ACM, 2013.

[40] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 258–269, IEEE Computer Society, 2012.

[41] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 558–567, Feb 2014.

[42] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (Berkeley, CA, USA), pp. 705–721, USENIX Association, 2016.

[43] M. Gorman and A. Whitcroft, "The what, the why and the where to of anti-fragmentation," in *Ottawa Linux Symposium*, vol. 1, pp. 369–384, Citeseer, 2006.

[44] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on numa systems," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 231–242, USENIX Association, 2014.

[45] A. Perais and A. Seznec, "Eole: Paving the way for an effective implementation of value prediction," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 481–492, IEEE Press, 2014.

[46] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," in *Software, IEE Proceedings-*, pp. 198–209, Feb 2004.

[47] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 428–439, Feb 2014.