FreshCache: Statically and Dynamically Exploiting *Dataless* Ways

Arkaprava Basu Derek R. Hower Mark D. Hill Michael M. Swift

Department of Computer Sciences
University of Wisconsin-Madison
{basu, drh5, markhill, swift}@cs.wisc.edu

Abstract - Last level caches (LLCs) account for a substantial fraction of the area and power budget in many modern processors. Two recent trends — dwindling die yield that falls off sharply with larger chips and increasing static power — make a strong case for a fresh look at LLC design. Inclusive caches are particularly interesting because many commercially successful processors use inclusion to ease coherence at a cost of some data being stale or redundant.

Prior works have demonstrated that LLC designs could be improved through static (at design time) or dynamic (at runtime) use of "dataless ways". The static dataless ways removes the data—but not tags—from some cache ways to save energy and area without complicating inclusive-LLC coherence. A dynamic version (dynamic dataless ways) could dynamically turn off data, but not tags, effectively adapting the classic selective cache ways idea to save energy in LLC but not area. Our data show that (a) all our benchmarks benefit from dataless ways, but (b) the best number of dataless ways varies by workload. Thus, a pure static dataless design leaves energy-saving opportunity on the table, while a pure dynamic dataless design misses area-saving opportunity.

To surpass both pure static and dynamic approaches, we develop the FreshCache LLC design that both statically and dynamically exploits dataless ways, including a predictor to adapt the number of dynamic dataless ways as well as detailed cache management policies. Results show that FreshCache saves more energy than static dataless ways alone (e.g., 72% vs. 9% of LLC) and more area by dynamic dataless ways only (e.g., 8% vs. 0% of LLC).

I. Introduction

The on-chip cache hierarchy plays a crucial role in processor performance, as evidenced by designs that dedicate more than 50% of the die area to last-level caches (LLCs) [20,28,38,39]. Historically, designers found the area and power demands of LLCs acceptable, but two recent trends — increasing static power [2,7,8,18] and diminishing die yields [1,15,22,35] — encourages reconsideration of LLC designs.

Inclusive LLCs [36] present an opportunity for improvement because they replicate the cache blocks contained in upper-level caches (closer to the processor). This design is widely used in commercial CMPs (e.g., Intel's Nehalem, Sandy Bridge, and Ivy Bridge designs) because it simplifies coherence and reduces on-chip traffic [10,36,37]. However, replicating data makes inclusive caches *more area- and energy hungry than they need to be*. The fact that they are used in spite of this waste and viable alternatives — exclusion [14], non-inclusion [23], and tag replication [4] — shows the high value placed on the coherence benefits of inclusion. Thus, an LLC design that reduces area and power overhead *without* sacrificing inclusion is immediately useful.

Design time: To address this waste in inclusive caches, researchers have proposed NCID [41], which uses cache ways built with tag and metadata but *no* data. These ways, which we call *static dataless ways (SDWs)*, can save area and static energy while keeping the coherence benefits of an inclusive cache. However, our analysis (Section III) shows that the opportunity to use dataless ways varies widely across workloads. Since the number of static dataless ways is decided before chip fabrication time, it needs to be conservative to ensure that the worst-case performance degradation across all workloads remains within an acceptable range. Thus, a fixed number of static dataless ways is unable to harness the full potential of dataless ways.

Runtime: This shortcoming can be addressed by creating dataless ways at runtime. The data portion of cache ways can be turned off dynamically to save energy. We call such dataless ways *dynamic dataless ways (DDWs)*. This is inspired by Albonesi's Selective Cache Ways [3], which was among first systems to demonstrate that dynamically resizing caches can save energy. The concept of dynamic resizing of cache is easily extended to LLC and, in fact, a few modern processors allow system software to control the LLC size [12]. Unfortunately, resizing the LLC dynamically gives up the area savings of static dataless ways.

New Hybrid: In this work we present the *FreshCache* LLC design, which seeks to achieve best of both worlds – static dataless ways, provisioned at design time to save area and energy with negligible performance impact, augmented with dynamic dataless ways enabled at run time for further energy savings when opportunity exists. Furthermore, FreshCache minimally changes inclusive cache coherence protocols and provides hardware management of dynamic resizing without software changes or profiling.

At chip design time, FreshCache fixes a given number of cache ways as static dataless ways (e.g., 2 out of 16 ways). Such SDWs save both area and energy of the LLC. The number of SDWs is chosen conservatively to keep the worst-case performance loss acceptable across all workloads.

At run time, FreshCache hardware monitors the workload's performance sensitivity to dataless ways and increases or decreases the number of DDWs *depending upon the opportunity and the constraint*. The number of DDWs at a given time is decided based on a software-provided *maximum performance degradation (MPD)* and the controller's predicted performance loss from different numbers of DDWs. At the runtime, FreshCache's dataless-way-aware LLC controller actively guides

cache blocks with stale data towards dataless ways at runtime (SDWs or DDWs) to minimize performance degradation due to presence of dataless ways. The use of dataless ways instead of turning off whole cache way allows FreshCache to keep benefits of inclusion without reducing the effective capacity of private caches. Importantly, FreshCache achieves this with only minimal changes to the coherence protocol.

In summary, the FreshCache design uses SDWs to save both area and power without possibility of substantially degrading performance of any workloads and uses DDWs at runtime to *opportunistically* save more power if the workload characteristics permit.

Our evaluation is divided into two parts. First, we present an analysis on why dataless ways can be beneficial. To this end we find that in an inclusive LLC on average 24% of *valid cache blocks can contain stale data* (data that cannot be used), which can be exploited through use of dataless ways. Second, in experiments with PARSEC [5] workloads and three commercial workloads, we show that FreshCache can use SDWs to save 8% of LLC area and with added DDWs up to 72% (average 40%) of LLC and DRAM access energy without significantly affecting performance (1.7% on average, 2.8% in the worst case). We demonstrate that compared to a pure static approach, FreshCache saves more energy for some workloads (e.g., 72% vs. 9% energy savings) without hurting the performance of any workload. Compared to a pure dynamic approach FreshCache could save significant LLC area (e.g., 8% of LLC area savings vs. no area savings).

II. Base system architecture

We describe our design in the context of a base architecture primarily modeled after the Intel Nehalem® architecture [36]. The base architecture, described Table 1, contains three levels of on-chip caches. The L1 and the L2 caches are private to a core, while the last level L3 cache is logically shared among all the cores on the die. The private L2 is exclusive with respect to the L1, and the L3 is inclusive with respect to the private caches. The 1:4 ratio of aggregate L2 to L3 size was chosen to follow Intel Nehalem (Xeon) E5507/5506 core [16] and recent industrial research [17].

We model a "MESI" coherence protocol for on-chip coherence [15]. An on-chip directory located at the L3 is responsible for maintaining coherence. The tags for LLC blocks include state and sharing information required for coherence.

This *in-cache-directory* is similar to many commercially popular x86-64 processors with inclusive LLCs. Table 1 shows that we scaled down the on-chip cache hierarchy size by a factor of two from most commercial architectures. This makes off-chip accesses more frequent that is likely to result in underestimating energy savings and overestimating performance costs for our proposed technique.

III. Stale data in LLCs

This section analyzes how much opportunity there is to utilize dataless ways. FreshCache takes advantage of valid cache blocks with stale data in inclusive LLCs to reduce power. For static dataless ways, a designer must determine the prevalence

Core	8, in-order, 2 Ghz
L1 cache	Private, 16kB 4-way, Split I/D, writeback
L2 cache	Private, 128 kB, 8-way, *exclusive* with L1, writeback
L3 cache	Shared, 4 MB, 16-way, *inclusive* to private caches, writeback
Coherence	MESI Directory protocol, directory co-located with L3 cache blocks
Memory	2 GB , ~ 350 cycle round trip

Table 1. Base system configuration

of stale data across all workloads to avoid major performance impact of using dataless ways. For dynamic dataless ways, the variability in stale data and cache usage must be known. The wide variation in sensitivity to cache size across workloads is well studied and understood [32,40], and hence we focus on understanding the presence of stale data. We analyze the reasons behind the stale data in the LLC and quantify its presence. We then demonstrate variation in amount of stale data across workloads.

First, we describe below an example of how stale blocks can occur, and then present analyses of how often stale blocks can be found. When a private cache requests a cache block with exclusive permission (i.e., a *GETX* request) from the LLC, the LLC controller invalidates the sharers and gives the cache block with exclusive permission to the private cache. Hereafter the data portion of the LLC block serves *no* purpose because the private cache with exclusive permission is free to modify the block without notification. Thus data in the block is *stale*. The LLC forwards subsequent write or read requests from another core to its exclusive owner. The block's data in the LLC cannot be used to satisfy a request, because it may have been modified in the private cache. However, the tag and other meta-data continue to be useful as it help identify the owner of the block.

Frequency of stale blocks. The number of stale blocks is proportional to the overlap between private caches and the LLC; more overlap introduces more stale blocks. To evaluate the magnitude of stale blocks and to find whether they can be exploited, we measure the fraction of valid cache blocks in an LLC holding stale data for varying ratios of aggregate private L2 to shared L3 cache size. For a variety of workloads, we

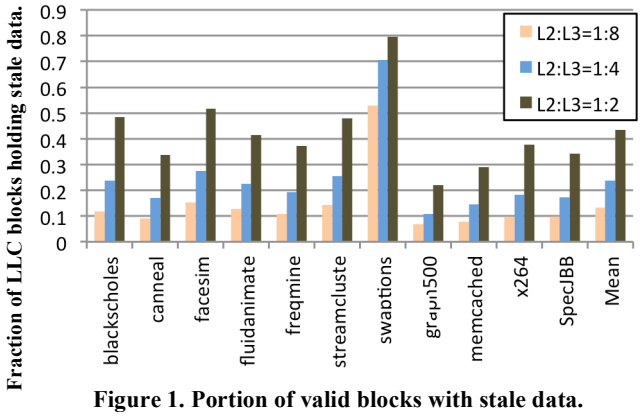

Figure 1. Portion of valid blocks with stale data.

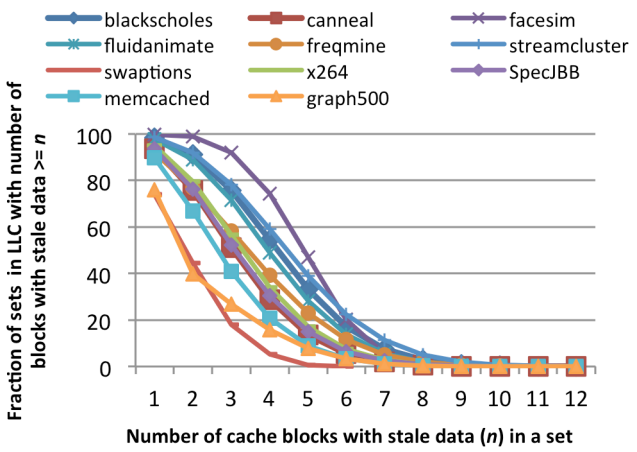

Figure 2. Distribution of stale blocks across sets.

sampled the LLC every 100000 cycles (0.5 micro-sec) and report the average number of stale blocks across the samples. We record the number of stale blocks as a fraction of valid blocks and do not include unused blocks.

Figure 1 shows result of this experiment with varying L2:L3 ratios for several PARSEC [5] and three commercial workloads. On average nearly 24% of the cache blocks in the LLC contain stale data with a L2:L3 ratio of 1:4. As expected, the fraction of stale data increases with higher values of L2:L3 ratios because there is more overlap between the LLC and private caches. In a few cases, the fraction of stale blocks is greater than the L2:L3 ratio. This occurs because of the small data footprint of one of the workloads (*swaptions*) does not fill up the entire LLC, so there are few valid blocks and stale blocks make up a large portion of them.

Observation 1: A significant fraction of blocks in the LLC hold stale data at any given time, which adds to power and area costs but without any performance benefit.

Stale block distribution. While the fraction of stale cache blocks is informative, the ability to exploit stale blocks depends upon the distribution of stale blocks across the sets in LLC. Ideally, a processor would configure SDWs for the *minimum* number of stale blocks across all workloads, and DDWs up to the *maximum*.

Figure 2 shows the likelihood that a set will contain at least *n* stale blocks at any time during execution. For example, for *facesim*, on average more than 75% of the sets in the LLC contain four or more stale cache blocks. Across most of the workloads, a majority of the LLC sets contains at least 3 stale cache blocks, indicating a high potential to exploit the stale data phenomenon.

More importantly, we observe that the distribution of number of stale blocks per set of the LLC varies across workloads. For example, *facesim* has at least 4 blocks with stale data in 75% of the sets in the LLC, while for *graph500* only 15% of the sets have 4 or more cache blocks with stale data. Thus, to fully exploit the stale data in LLC the number of dataless ways need to vary dynamically according to the workload characteristics.

Observation 2: The distribution of stale data across cache sets varies across workloads and a design with static dataless ways alone is unlikely to fully exploit stale data in LLC.

IV. FreshCache: Leveraging Stale Data in the LLC

The FreshCache design uses a hybrid of static dataless ways (SDWs) and dynamic dataless ways (DDWs) to design area and energy efficient LLC. SDWs help save both area and energy, while DDWs help save more energy when opportunity exists.

SDWs constitute a fixed number of *contiguous* ways in each set (e.g., two out of sixteen). The data in these ways are omitted from the cache layout. The number of SDWs in a FreshCache design must be chosen conservatively to ensure worst-case performance across all workload remains acceptable.

On the other hand, DDWs are created at run time by turning off power to the data cells of a cache way. DDWs can save power, but not area, and provide dynamic control over the power savings and performance impact. Applications that can tolerate a larger number of dataless ways can use DDWs "for free" without incurring performance penalties, while other applications can maintain high performance with fewer DDWs. For example, Figure 2 shows that for *graph500* only 40% of cache sets have more than one stale cache block. However, a small number of dataless ways limits the savings on programs with more stale data, such as *facesim*. Thus, FreshCache leverages DDWs where the number of DDWs can be controlled automatically by the hardware according to workload characteristics to save more power when the opportunity exists.

FreshCache needs to accomplish two major tasks. First, it needs a dataless-way-aware LLC controller to select which blocks use dataless ways (SDWs or DDWs) and which use conventional (with data) ways. Second, it needs a hardware monitoring mechanism to select the optimal number of DDWs for a given workload at runtime. Next, we describe how FreshCache accomplishes the first task with a modified LLC controller, called the *FreshCache controller* and then delve into details of our online hardware monitoring and management mechanism for DDWs (called the *DDW controller*).

A. FreshCache Controller: Managing Stale Data

Fundamental to a FreshCache design is how to exploit stale data in LLC and manage dataless ways, be it SDW or DDW. There are two primary goals of this design -- 1) keep dataless ways occupied with cache blocks with stale data to hide any potential performance degradation 2) uphold inclusive properties of the LLC without substantially perturbing the coherence protocol.

Dataless ways in the LLC can only store blocks that would otherwise hold stale data, while *conventional ways* hold the blocks with valid data (metadata+data). If a stale cache block cannot be found, then the dataless ways must remain empty, which effectively reduces the cache capacity. FreshCache uses a modified cache controller (the *FreshCache Controller)* that actively guides stale blocks to dataless ways to ensure that they have minimal performance impact.

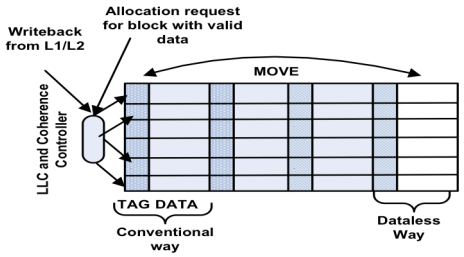

Figure 3. LLC controller with LLC having dataless ways.

When the coherence state of a block changes, the FreshCache controller interprets the new state to infer whether the data in the block is stale. If a valid cache block holds stale data then the controller makes it a candidate for allocating in (or moving to) one of the dataless ways in the set.

The FreshCache controller must consider dataless ways during at least two occasions: first, when a cache block is allocated in the LLC, and second, when a private cache writes back a block to the LLC. In addition, the controller's replacement policy selects a victim from a subset of ways (dataless or conventional) when necessary.

Allocation of a cache block in the LLC: The LLC allocates a cache block with stale data in response to a write (GETX) request from private cache or a read request when a data cache block does not have other requester (sharer).

Here, the FreshCache controller first looks for a free dataless way, and if that is not available it tries a conventional way before invoking the replacement policy to make a free block. Conversely, when allocating a cache block with valid data, the controller first seeks a free conventional way and then looks for a conventional way with stale data that can be moved to a free dataless way. If there are no free ways, it invokes the replacement policy. In this way the controller minimizes evictions by keeping dataless ways occupied with stale cache blocks.

Writeback to a cache block in LLC: The LLC can receive a writeback from a private cache in three cases: (1) when a block held with exclusive permission is victimized from the private cache, (2) when the exclusive permission is relinquished by a private cache in response to a read request by another core, and (3) when the LLC back-invalidates a block in a private cache to ensure inclusion. In the third case, the LLC does not store the written-back data and thus no new mechanism is needed. However, in the first two cases if the block in the LLC resides in a dataless way then the writeback cannot proceed since there is no space for the data. In this case, the controller moves the block to a conventional way and replaces an occupied conventional way if needed. A writeback to a block in a conventional way proceeds normally.

Figure 3 depicts a LLC with dataless ways and the FreshCache controller that uses intra-set block movement to keep the dataless ways occupied with blocks with stale data.

LLC Replacement policy: Unlike conventional caches, the FreshCache may need to pick a victim from one of two classes of cache ways. During allocation of a cache block with valid data or when handling writeback to a block in a dataless way, it may be necessary to choose a victim only from conventional ways. To simplify the design, the locations of the dataless ways in each set are kept contiguous. Thus, existing victim selection mechanisms can be trivially extended to choose a victim from just the conventional ways. For example, a binary tree-based pseudo-LRU [9] replacement mechanism, commonly used in highly associative LLCs, can select a victim from within conventional ways by constraining the tree traversal to a sub-tree structure of the conventional ways in a given set.

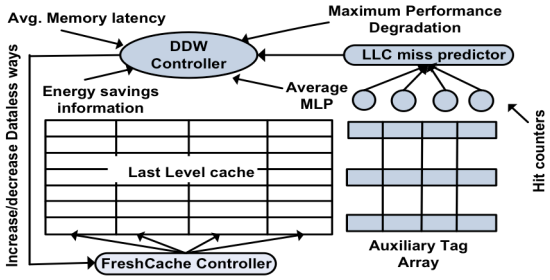

Figure 4. Hardware Control for Dynamic Dataless Ways (DDW). Additions are shaded.

B. Managing Dynamic Dataless Ways

For workloads with few stale blocks in the LLC and high LLC usage, DDWs should be kept low (or even zero) to avoid significant performance degradation, while they should be used more for workloads with many stale blocks to save energy. DDWs effectively reduce LLC capacity when it is not needed, which provides additional power savings similar to dynamic cache-sizing techniques [3]. However, turning off only the data (but not metadata) in the LLC leaves data in the private caches. In contrast, if entire ways (metadata+data) are disabled then inclusion requires eviction of the corresponding data from private caches.

In the following we describe the implementation details pertaining to creation of DDWs and hardware monitoring mechanisms to decide the number of dataless ways.

1) Creating Dynamic Dataless Ways

Dynamically enabling dataless ways requires mechanisms to designate and disable the data portion of selects ways. First, data ways in the LLC must be modified to support turning them on/off. Second, the FreshCache controller should be able to designate a set of contiguous data ways in the LLC to turn off. The FreshCache always keeps all dataless ways contiguous as it enables simpler design of cache block replacement mechanism as mentioned Section IV.A. Finally, the controller flushes out any dirty data from those ways to the memory. The flush operation is carried out in the background without blocking other requests.

2) DDW Controller: Provisioning DDWs

At a high level, the DDW controller monitors current cache performance against an user-specified policy goals, and adjusts the number of DDWs up or down to achieve that goal.

Depending upon execution environment and the purpose, the relative importance of performance and energy savings can vary. Thus, FreshCache design enables a user to provide the relative importance of energy savings and performance by

specifying a *maximum performance degradation* (MPD) value. The FreshCache aims to minimize cache energy as long as the percentage performance degradation relative to the baseline design with conventional LLC remains within this user-provided MPD value. A high value of MPD indicates the user's willingness to save more energy at cost of potentially larger performance degradation, while a low value of MPD indicates greater importance for performance. The DDW controller will find the number of DDWs that saves the most energy as long as estimated performance degradation stays within this limit. In our implementation MPD is expressed as integer percent performance degradation over the baseline with a conventional LLC. The software provides the desired MPD value to the hardware by writing to a designated register.

As depicted in Figure 4, the DDW controller is built from three components: (1) a *miss-rate estimator* to predict cache behavior with different numbers of dataless ways, (2) configured *miss latency* and *energy savings* values, and (3) a *memory-level parallelism estimator* to calculate the performance cost of misses. With these components, the controller predicts the performance loss and energy savings from different numbers of dataless ways and selects the greatest savings with performance above the MPD threshold.

We use a slightly modified version of Qureshi et al.'s cache utility monitoring mechanism [32] to estimate the number of off-chip misses with a given number of dataless ways. As depicted in Figure 4, the monitor adds an auxiliary tag array of the same set-associativity as the LLC but containing only one of every 32 sets using set-sampling [31]. This structure simulates hits and misses for each way in the set in the recency order. Counters keep track of the hit count for each way. We modify Qureshi's proposal by incrementing the hit counter for a way only when there is a cache hit that a dataless way could not have served (*e.g.*, read miss for shared data from a private cache but not for exclusive data in another private cache) instead of on all hits. The hit counter values provide an estimation of the number of misses in an LLC when a given number of ways are rendered dataless. The estimated miss numbers for each possible number of dataless ways are then fed to DDW controller.

The controller computes the estimated performance degradation for each number of DDWs by multiplying the estimated number of misses with the expected LLC miss latency (provided) and dividing this total miss latency by the estimated memory level parallelism. The parallelism is calculated as the fraction of misses across different cores.

Finally, the controller computes the energy savings using configured values for the static energy saved by turning off data ways and estimated energy cost of each off-chip accesses from a miss. While worse performance also increases energy due to running longer, the current implementation of the DDW controller does not incorporate this cost. From the predicted energy savings and predicted performance degradation, the controller then chooses the number of DDWs with performance cost less than the MPD and with most energy savings.

This analysis is carried out periodically every 50M cycles, at which point the controller signals the FreshCache controller to increase or decrease the number of DDWs as depicted in Figure 4. The additional hardware structures needed for predicting cache misses adds 12KB of state overhead for a LLC with 4MB data capacity (< 0.3%).

C. Putting all together

In summary, FreshCache uses static dataless ways (SDWs) to save area and energy and uses dynamic dataless ways (DDWs) to opportunistically save more energy as and when workload characteristics permit. At runtime, the FreshCache controller actively guides cache blocks with stale data towards dataless ways (SDWs and DDWs) to hide potential performance loss. The number of SDWs is fixed conservatively at design time to ensure acceptable worst case performance across range of workloads while allowing reasonable area and energy savings. At runtime, the DDW controller monitors the workload characteristics and chooses the number of DDWs against a user-specified upper limit on performance degradation to enable the highest energy savings possible.

V. Evaluation

We evaluate the FreshCache design to quantify its benefits:

- How much energy and area can be saved by FreshCache?
- How big are the benefits of FreshCache's hybrid approach in reducing LLC area and power?

Further, we measure the performance overhead due to FreshCache.

A. Simulation Methodology

We use the gem5 full system simulator [6] to model an x86-64 machine running Linux (kernel version 2.6.28.4).

We simulated a multi-core chip with 8 cores and three levels of caches. The parameters for simulation are shown in Table 1. The L2:L3 ratio is 1:4. The absolute sizes of the simulated caches are scaled down by at least a factor of two compared to real processors for better simulation speed. However, shrinking caches likely to make our gains conservative as the performance cost of dataless ways for larger caches likely to be lower than for our experiments.

We extended CACTI 6.5 [27] to model the power and the area of our proposed LLC designs with dataless ways. We plugged its estimates into the full-system simulation to obtain power consumption. For the LLC, we used low-power transistors with a 32 nm process. We estimate that an LLC with the configuration in Table 1 draws 0.8 watt of static power while each off-chip access costs 16 nJ of energy.

B. Workloads

We use a mix of programs from Parsec [30] and three commercial-like multithreaded workloads to evaluate FreshCache. For all the Parsec workloads we use the native (largest) input set. We simulated *SpecJBB 2005* [42], which models the middle-tier business logic of a three-tier web service; *memcached* [26], a memory cache frequently used by web services; and *graph500* [25], a graph traversal algorithm useful in HPC environments.

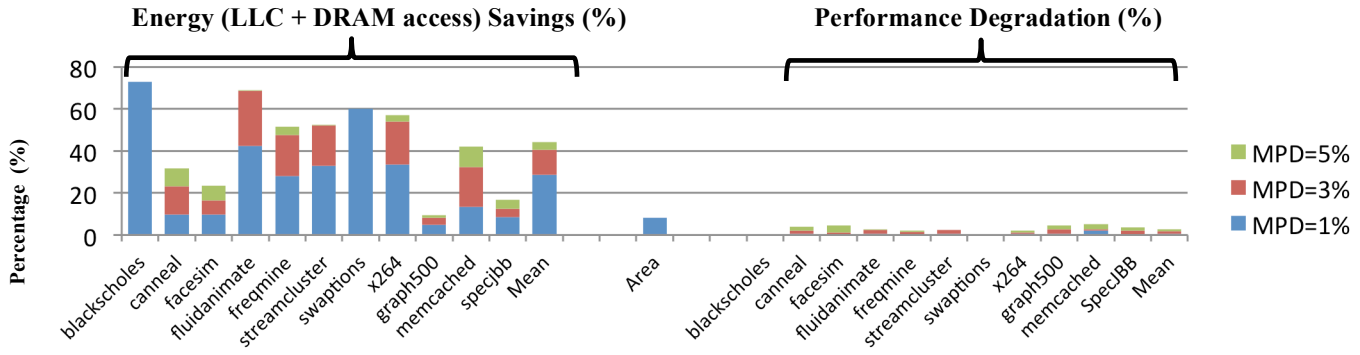

Figure 5. Energy/area savings and performance degradation with FreshCache.

C. *FreshCache Savings*

In this section we present the results of our evaluation of a FreshCache design that uses two SDWs, and up to 14 DDWs selected dynamically at runtime by the DDW controller. We use 2 SDWs because our experiments showed that it incurs negligible performance cost (0.08% average, 0.46% worst case), while larger values had more than 1% worst case cost. Thus, using 2 SDWs has low risk of negatively impacting performance while still providing useful area and power savings.

In Figure 5 we show the energy, area, and performance impact of FreshCache with varying MPD values (1, 3, and 5%). The first cluster of bars in Figure 5 shows the energy saved in the LLC and DRAM access normalized to a baseline system with *no* dataless ways. The top of each stack in the stacked bars shows the percentage energy savings for the corresponding MPD values (indicated by the legend) for the given workload. For example, on average, 28% of energy is saved with MPD=1% and above 44% with MPD=5%. We observe that across all workloads substantial energy is saved by FreshCache; however, savings varies widely across workloads. For example, with MPD=3%, FreshCache can save nearly 69% of the LLC and the DRAM energy for *fluidanimate*, but only 8% of energy savings for *graph500*.

We also observe that across almost all workloads, energy benefits begin to diminish as the MPD increases. Higher performance losses result in longer run times, which results in static energy use for a longer time, and more off-chip accesses, which use more dynamic energy. Above a threshold, energy saving from the DDWs is unable to offset the increase due to longer runtimes and off-chip misses.

The singleton bar in the middle shows the percentage of the area of a conventional LLC eliminated by FreshCache. The area savings are due to SDWs in the FreshCache and do not change with workload or MPD values. As mentioned earlier in the section we evaluated FreshCache with 2 SDWs. This saves 8.2% of LLC area, which is substantial given that LLCs often account for more than 50% of the chip area.

The third cluster of stacked bars in the Figure 5 shows the percentage performance loss for each value of MPD relative to the baseline with a conventional LLC. For example we observe that for MPD=3%, on average performance dropped 1.7%. Importantly, we observe that across all workloads the DDW controller is able to keep the performance degradation within the limit stipulated by the MPD. We also observe that the actual performance loss was often much below the specified MPD value.

This occurs for many reasons. First, above a certain threshold, the static-energy savings from DDWs are unable to offset the energy consumption increase from more off-chip misses and a longer run time. Thus, even if a user accepts more performance degradation, it would not save more energy. Second, the DDW controller never lets performance *for a single period* (50M cycles here) drop below the threshold. This is a stricter condition than the average MPD for a full run of the program.

Putting all three clusters together, we see that FreshCache can save significant energy and non-negligible area at the cost of small or negligible performance loss, well within the user specified limits to performance degradation. With MPD=3%, FreshCache reduces energy on average by 41% and area by 8.2% for a mere 1.7% actual reduction in performance.

Is the hybrid approch of FreshCache necessary?

FreshCache proposes a hybrid of a static chip design time and a dynamic runtime technique to utilize the dataless ways to enable area and energy savings in LLC. Here, we compare FreshCache against a pure static (like NCID [41]) and a pure dynamic approaches (like Selective Cache Ways [3]) to understand whether the hybrid aproach is justified or not.

Figure 6 depicts the tradeoffs of purely static design, purely dynamic and FreshCache in terms of energy savings, area saving and performance impact. For static designs we evaluated two configurations: a conservative configuration with 2 SDWs (*Static-2*) and an agressive configuration with 8 SDWs (*Static-8*). These two designs do not use DDWs. To undestand the potential of purely dynamic approach we profiled each application to select the best number of DDWs for each application for MPD=3% (*DynamicOffline-3*). Finally, *FreshCache-3* is FreshCache design with MPD=3%. Similar to Figure 5, the first set of bars show energy savings over the conventional LLC. We observe that *Static-2* yields the least energy savings across all the configurations studied (8.2%), while, as expected, *Static-8* provided better energy savings (36%). However, this is still well below *FreshCache-3*, which provides 40.7% energy savings. We note that FreshCache lies between the optimal offline settings (*DynamicOffline-3*, with 44% saving) and the aggressive static design but does so without requiring software profiling. The second

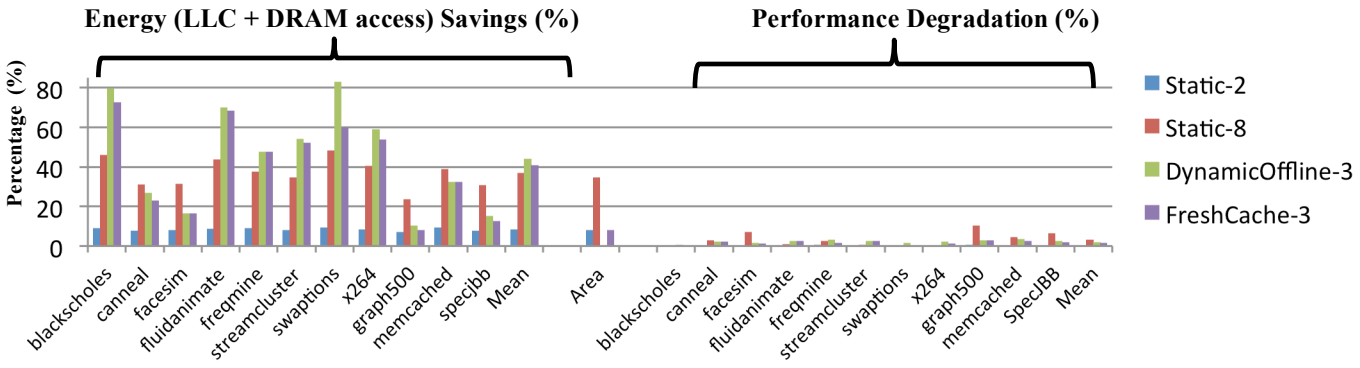

Figure 6. Energy savings and performance degradation for dynamic dataless ways under hardware control.

set of bars in Figure 6 depicts the percentage of LLC area savings. As expected, the greatest area savings (35%) comes from the aggressive static design (*Static-8*). The FreshCache and the conservative static design (*Static-2*) both provide non-negligible (8.2%) LLC area savings. A pure dynamic approach saves ***no*** area.

The final set of bars shows the performance loss of using these approaches compared to a conventional LLC design. Except for the aggressive static design (*Static-8*), all other designs limit the worst case performance degradation to 3% across all workloads, and often much less. However, the aggressive static design can lead to more than 10% performance degradation (*graph500*), which may be unacceptable. Further, 3 of the workloads (*facesim, memcached, specJBB*) suffer at least 6.5% performance degradation. In contrast, FreshCache limits performance loss for all workloads, since it exploits dataless ways to save more energy *only* when the opportunity exists.

Summary: If a conservative static design is used then energy savings are moderate and opportunity to save more is lost for many workloads. If an aggressive static design is used then it leads to large performance degradations for some workloads. If a pure dynamic approach is used, then we get the energy savings and high performance, but lose out on the area savings. Thus, only the hybrid approach put forth by FreshCache enables both chip area savings and significant energy savings.

VI. Related Work

Researchers have previously proposed cache designs that decouple tags and data in a last level cache [4,41]. In particular, NCID [41] make use of dataless ways to bring the snoop filtering benefits of inclusive LLC designs to exclusive/non-inclusive caches. On the other hand, FreshCache maintains an inclusive coherence protocol with only a slight change, and the remaining changes are localized to the cache controller without affecting the protocol state machine. More importantly, NCID seeks to reduce invalidations to private caches and to support QoS in the LLC, while the FreshCache provides both power and area savings. Finally, we show how to dynamically vary the number of dataless ways to take advantage of workload characteristics, while NCID is a purely static design.

FreshCache bears similarity to Albonesi's *Selective Cache Ways* [3], with which software can turn off a desired numbers of ways in L1 cache. However, our FreshCache design targets the LLC and exploits the availability of stale cache blocks to minimize any increase in off-chip accesses. More importantly, unlike Selective Cache Ways, FreshCache can save substantial on-chip area as well. Several other proposals also looked into selectively turning off cache ways at runtime to save energy [13,42,44]. However, none of these techniques save area.

Several researchers have suggested predicting and exploiting *dead* blocks in caches [19,21,24,34]. A cache block is dead from the time it is last referenced until it is evicted from the cache. Our notion of a valid cache block with stale data is different from a dead block, as a valid cache block with stale data in the LLC may not be dead; it could possibly be accessed again after a private cache gives up its exclusive rights. Unlike these works, which require predicting when a cache block becomes dead, it is easy to know when a cache block contains stale data by interpreting its coherence state. While these works focus on enhancing the performance of the cache, we focus instead on designing an area- and power-efficient LLC.

Qureshi et al.'s *V-way* cache [33] proposed a decoupled, pointer-linked tag and data store for set-associative caches where number of tags is a multiple of the number of data ways in order to reduce the number of conflict misses in the cache. Chishti et al.'s *CMP-NuRAPID* [11] also uses decoupled, pointer-linked tag and data store to allow for controlled replication and capacity management in a NUCA cache to get the best of both shared and private organization of large caches. Similar to FreshCache, these works have more tag than data, but for different purposes than our objective of area and power efficiency.

Jaleel et al. proposed a novel cache replacement policy to bridge the performance gap between inclusive and exclusive caches [17]. They observe that the performance difference between inclusive and non-inclusive design stems from the bad replacement decisions made at an inclusive LLC that back invalidates "hot" blocks from the private caches. They address this by proposing an LLC replacement policy that is aware of the temporal locality in private caches. Their policy can also be applied in FreshCache to improve performance.

FreshCache also bears similarities to victim caches and exclusive/non-inclusive caches, which like FreshCache, may not keep a copy of a data present in the private cache. However, as mentioned earlier, FreshCache keeps the simplicity of inclu-

sive coherence protocol with no or negligible changes. Whereas a LLC designed as victim cache or exclusive cache requires very different cache and coherence controller.

Researchers have proposed circuit techniques like Gated-Vdd [29] to selectively turn off cache blocks by adding extra gated-transistors to SRAM cells. Flauntner et al. proposed Drowsy caches [13], where multiple supply voltages are used to enable SRAM cells to go into a low power mode where they keep the data but cannot be accessed immediately. FreshCache in contrast, proposes a reorganization of the cache architecture that enables considerable area savings via static dataless ways.

VII. Conclusion

FreshCache statically and dynamically reduces power through dataless ways. FreshCache also makes the LLC more area efficient. The design comes from the observation that in inclusive LLCs a significant fraction of valid blocks contain stale data. Rather than give up the coherence benefits of inclusion, we instead take advantage of stale data. At design time, FreshCache uses static dataless ways to save area and power, while at runtime uses dynamic dataless ways to further reduce substantial amount of power when opportunity exists.

VIII. Acknowledgements

This work is supported in part by the National Science Foundation (CNS-0720565, CNS-0916725, and CNS-1117280, CNS-0834473), Google, and the University of Wisconsin (Kellett award and Named professorship to Hill). The views expressed herein are not necessarily those of any sponsor. Hill has a significant financial interest in AMD, and Swift has a significant financial interest in Microsoft.

References

1. Agarwal, A., Paul, B.C., Mahmoodi, H., Data, A., and Roy, K. A process-tolerant cache architecture for improved yield in nanoscale technologies. *IEEE Trans. Very Large Scale Integrated. Systems. 13*, 1 (2005).
2. Ahmed, K. *Transitor Wars*. http://spectrum.ieee.org/semiconductors/devices/transistor-wars.
3. Albonesi, D.H. Selective cache ways: on-demand cache resource allocation. *In MICRO 1999*.
4. Barroso, L.A., Gharachorloo, K., McNamara, R., et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. *In ISCA* 2000.
5. Bienia, C., Kumar, S., Singh, J.P., and Li, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *In PACT* 2008.
6. Binkert, N., Beckmann, B., Black, G., et al. The gem5 simulator. *Computer Architecture News (CAN)*, 2011.
7. Borkar, S. Design Challenges of Technology Scaling. *IEEE Micro 19*, 4 (1999).
8. Butts, J.A. and Sohi, G.S. A static power model for architects. *In MICRO* 2000.
9. Chen, T., Liu, P., and Stelzer, K.C. Implementation of a pseudo-LRU algorithm in a partitioned cache. 2006.
10. Chen, X., Yang, Y., Gopalakrishnan, G., and Chou, C.-T. Reducing Verification Complexity of a Multicore Coherence Protocol Using Assume/Guarantee. *Proceedings of the Formal Methods in Computer Aided Design*, 2006.
11. Chishti, Z., Powell, M.D., and Vijaykumar, T.N. Optimizing Replication, Communication, and Capacity Allocation in CMPs. *Proceedings of the ISCA* 2005.
12. *Intel Atom Processor Z5xx Series Datasheet 2010, Section 2.2*. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/atom-z540-z530-z520-z510-z500-45-nm-technology-datasheet.pdf.
13. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., and Mudge, T. Drowsy caches: simple techniques for reducing leakage power. *In ISCA* 2002.
14. Gaur, J., Chaudhuri, M., and Subramoney, S. Bypass and insertion algorithms for exclusive last-level caches. *In ISCA* 2011.
15. Hennessy, J.L. and Patterson, D.A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th ed, 2007.
16. Intel Corp. Intel Xeon E5507 Specification. www.x86-guide.com/en/cpu/Intel-Xeon-E5507-cpu-no4089.html.
17. Jaleel, A., Borch, E., Bhandaru, M., Jr, S.C.S., and Emer, J. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. *In MICRO* 2010.
18. Kanter, D. *Nvidia's Kal-El Goes Asymmetric*. http://www.realworldtech.com/page.cfm?ArticleID=RWT100511155012&p=1 .
19. Khan, S.M., Daniel A. Jimnez, D.B., and Falsafi, B. Using dead blocks as a virtual victim cache. In *PACT 2010*
20. Kurd, N.A., Bhamidipati, S., Mozak, C., et al. Westmere: A family of 32nm IA processors. *Proceedings of IEEE International Conference on Solid-State Circuits (ISSC)*, (2010).
21. Lai, A.-C., Fide, C., and Falsafi, B. Dead-Block Prediction & Dead-Block Correlating Prefetchers. *In ISCA* 2001.
22. Lee, H., Cho, S., and Childers, B.R. Exploring the interplay of yield, area, and performance in processor caches. *Proceedings of ICCD* 2007.
23. Lepak, K.M. and Isaac, R.D. *Mostly Exclusive Shared Cache Management Policies*. U.S. Patent 7,640,399, 2009.
24. Liu, H., Ferdman, M., Jaehyuk Huh, and Burger, D. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. *In MICRO* 2008.
25. *graph500 --The Graph500 List*. http://www.graph500.org/.
26. memcached - a distributed memory object caching system. www.memcached.org.
27. Muralimanohar, N., Balasubramonian, R., and Jouppi, N.P. *CACTI 6.0*. Hewlett Packard Labs, 2009.
28. Naffziger, S., Stackhouse, B., Grutkowski, T., et al. The implementation of a 2-core, multi-threaded itanium family processor. *IEEE Journal of Solid-State Circuits 41*, 1 (2006).
29. Powell, M., Yang, S.-H., Falsafi, B., Roy, K., and Vijaykumar, T.N. Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories. *In ISLPED* 2000.
30. Princeton, P. Princeton Application Repository for Shared-Memory Computers (PARSEC). http://parsec.cs.princeton.edu/ .
31. Qureshi, M.K., Lynch, D.N., Mutlu, O., and Patt, Y.N. A Case for MLP-Aware Cache Replacement. *In ISCA* 2006.
32. Qureshi, M.K. and Patt, Y.N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. *In MICRO 2006*.
33. Qureshi, M.K., Thompson, D., and Patt, Y.N. The V-Way Cache: Demand Based Associativity via Global Replacement. *Proceedings of ISCA* 2005, 544–555.
34. Samira Manabi Khan, Y.T. and Jimenez, D.A. Sampling Dead Block Prediction for Last-Level Caches. *Proceedings of MICRO*, 2010.
35. SEMATECH, S. Critical Reliability Challenges for The International Technology Roadmap for Semiconductors (ITRS). 2003.
36. Singhal, R. Inside Intel Next Generation Nehalem Microarchitecture. 2008. http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf
37. Wang, J.-L.B.W.-H. On the inclusion properties for multi-level cache hierarchies. *In ISCA* 1988.
38. Wendel, D., Kalla, R., Cargoni, R., et al. The implementation of POWER7TM: A highly parallel and scalable multi-core high-end server processor. *In ISSC* 2010.
39. Wilkerson, C., Alameldeen, A.R., Chishti, Z., Wu, W., Somasekhar, D., and Lu, S.-L. Reducing cache power with low-cost, multi-bit error-correcting codes. *In ISCA* 2010.
40. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., and Gupta, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proceedings of ISCA 1988,* 24–37.
41. Zhao, L., Iyer, R., Makineni, S., Newell, D., and Cheng, L. NCID: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. *In Computing Frontiers* 2010.
42. *SpecJBB 2005*. http://www.spec.org/jbb2005/