Data Placement Across the Cache Hierarchy: Minimizing Data Movement with Reuse-Aware Placement

Andreas Sembrant, Erik Hagersten, and David Black-Schaffer
Uppsala University, Dept. of Information Technology, Sweden
{andreas.sembrant, erik.hagersten, david.black-schaffer}@it.uu.se

***Abstract*—Modern processors employ multiple levels of caching to address bandwidth, latency and performance requirements. The behavior of these hierarchies is determined by their approach to data placement and data eviction. Recent research has developed many intelligent data eviction policies, but cache hierarchies remain primarily either exclusive or inclusive with regards to data placement. This means that today's cache hierarchies typically install accessed data into all cache levels at one point or another, regardless of whether the data is reused in each level. Such data movement wastes energy by installing data into cache levels where the data is not reused.**

This paper presents Reuse Aware Placement (RAP), an efficient data placement mechanism to determine where to place data in the cache hierarchy based on whether the data will be reused at each level. RAP dynamically identifies data sets and measures their reuse at each level in the hierarchy. This enables RAP to determine where to move data upon installation or eviction to maximize reuse. To accomplish this, each cache line is associated with a data set and consults that data set's policy upon eviction or installation. The RAP data placement mechanism is orthogonal to the replacement policy, and can be combined with any number of proposed eviction mechanisms.

By itself, the RAP data placement mechanism reduces traffic in the cache hierarchy by 21 to 64%, depending on the level, without hurting performance. As a result of this traffic reduction, RAP reduces dynamic cache energy by 28% and total cache energy by 17%.

1. Introduction

An effective cache hierarchy is essential for performance and efficiency in modern processors. Two key aspects of cache hierarchy design are *how to place data* across the levels in the hierarchy and selecting *which data to evict* when new data is installed. While there has been a significant amount of research into intelligent eviction mechanisms [1], [2], [3] bypassing [4], [5], and NUCA placement [6], [7], today's memory hierarchies blindly move data through many different cache levels, spending energy to read and write data each time it is moved. This data movement consumes a large portion of the cache hierarchy energy, which can be as high as 50% of the total processor energy [8], [9]. In this work we demonstrate how a *reuse-aware placement mechanism* can move data upon eviction or installation to cache levels where the data is likely to see reuse, thereby dramatically reducing traffic and saving energy, without hurting performance. This data placement mechanism is orthogonal to the mechanism used to select data for eviction, can be combined with virtually any replacement policy, and is applied globally across the whole cache hierarchy.

The data *placement* policies in today's cache hierarchies are primarily *exclusive* or *inclusive*. Exclusive hierarchies install data into the level closest to the CPU on an L1 cache miss and move the data out through each level on eviction. This approach keeps only one copy of the data in the hierarchy, but requires moving data through all levels of the hierarchy on eviction, regardless of whether there is any benefit to doing so. Inclusive hierarchies initially install data into all levels. While this initial data replication eliminates the need for data movement on the eviction of clean lines (since the data is known to be present in the next level, so-called *silent evictions*), it reduces the effective cache capacity by replicating data and can lead to undesirable priority inversion across the hierarchy.

The inefficiencies due to inclusive and exclusive data placement when used with LRU eviction have become more severe as cache hierarchies have become deeper. This is particularly apparent in the shared LLC of multiprocessors, where inappropriate evictions from the shared cache can cause remote evictions from private caches due to inclusivity. Many proposals have sought to address these issues by changing the local *eviction mechanism* at the shared cache level [2], [10], while keeping the data placement policy the same, or bypassing *individual levels* of the hierarchy [4], [5].

This work addresses the inefficiencies in *data placement* across the full hierarchy. The *Reuse Aware Placement* (RAP) cache design reduces memory hierarchy traffic and energy by only placing data in levels where it is likely to be reused. RAP avoids the excessive data movement of exclusive designs without incurring the capacity reduction and priority inversion of inclusive ones. To accomplish this, the RAP mechanism tracks the behavior of an application's data sets and learns where their data is reused throughout the hierarchy. This information drives a policy that places data in the appropriate level on installation and eviction, thereby dramatically reducing data traffic by avoiding reads and writes to levels where the data will see no reuse. By placing data intelligently, RAP eliminates unnecessary data traffic and significantly reduces energy in the memory hierarchy.

This paper makes the following contributions to data placement within cache hierarchies:

- A mechanism for learning and applying a reuse-aware data placement policy by observing cache line reuse on eviction.
- A global data set tracking mechanism that monitors data set behavior across the whole cache hierarchy and is agnostic to the local eviction/replacement mechanism.
- A dramatic (21 to 64%) reduction in cache traffic compared to standard exclusive data placement, resulting in a 28% reduction in dynamic energy, with only a 0.8% performance decrease and 2.7% increase in on-chip storage.

978-1-5090-5142-7/16/$31.00

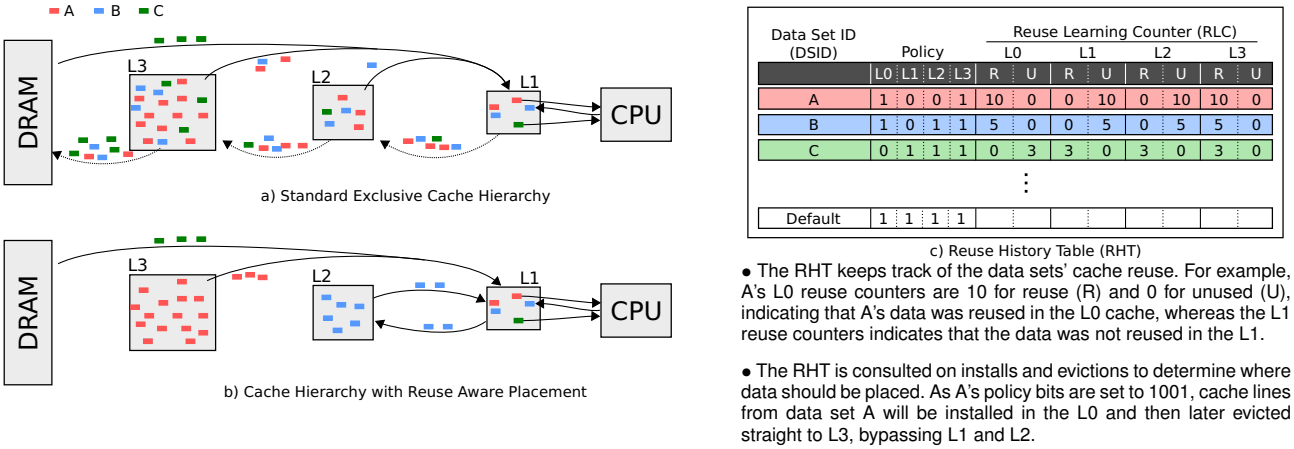

Figure 1. Reuse Aware Data Placement

2. Reuse Aware Placement

The goal of Reuse Aware Placement (RAP) is to save energy by reducing unnecessary data movement between cache levels. To do so, RAP learns where it expects a data set to see reuse, and only places it in those cache levels, bypassing those where it does not expect reuse. It is important to note that RAP is a data *placement* mechanism, and does not address the issue of which data should be evicted, leaving that instead to the chosen eviction/replacement policy.

To understand the impact of placing the data where it will be reused, consider the application shown in Figure 1a. Here we see an application with three different working sets executing on a processor with a traditional three-level cache hierarchy. The red and blue data sets fit into the L3 and L2 caches, respectively, while the green data set is streaming and does not fit entirely in any level of the cache hierarchy. In a standard hierarchy with exclusive data placement and LRU eviction, as shown in Figure 1a, the data from all three working sets is moved to the L1 upon request, and then moved from the L1 to the L2, then from the L2 to the L3, and finally back to DRAM on eviction. This wastes energy by installing data in levels where it will not be reused (e.g., the red data in the L2 and the green data in the L2 and L3) and the streaming data is likely to evict the other two data sets, thereby hurting performance and increasing energy consumption further. More advanced eviction policies [1], [2], [3] can detect that a data set is streaming and reduce cache pollution by preferentially evicting it from the L3 cache, but they still pay the energy cost of installing and evicting at that level as they affect the eviction mechanism, and not the data placement directly.

Figure 1b shows how Resource Aware Placement places data only in the levels where it will be reused. In this case, RAP learns that the green data is not reused anywhere in the hierarchy and moves it directly to DRAM upon eviction, thereby avoiding the overheads of polluting (performance) and accessing (energy) the L2 and L3. The red data set is only reused in the L3, and as a result it is moved directly from the L1 to the L3 when the L1's replacement policy decides to evict it. By learning how data is reused, RAP can apply appropriate data placement to minimize data traffic and cache pollution. In more complex hierarchies, such as those with an L0 cache for better energy efficiency, RAP can also identify whether to install the data in the L0 or L1 (see Section 4.1).

2.1. RAP Fundamentals

To place data in a reuse-aware manner, we need to be able to: 1) track applications' data sets, 2) detect reuse of the data sets at each level in the hierarchy, and 3) use this information upon eviction and installation to intelligently place the data.

1. Tracking Data: The Reuse History Table. The central element of RAP is the Reuse History Table. The Reuse History Table (RHT) is responsible for tracking the behavior of data sets and is consulted when data needs to be moved (installed or evicted) to determine the correct level to which the data should be moved. Conceptually, the RHT stores the degree to which each data set is reused in each level in the hierarchy. This information allows us to implement a policy wherein we only move data to a given level if we have seen sufficient reuse of that data set at that level before.

2. Detecting Data Set Reuse. To determine data set reuse, RAP tracks whether a cache line was reused in its current level when it is evicted. This allows the RHT to keep track of the reused vs. unused counts of evicted cache lines at each level in the hierarchy. This tracking is not performance critical and can be done with low overhead as it takes place at eviction. The RHT combines data from multiple cache lines associated with a common data set to determine the likelihood of reuse at each level in the hierarchy. The ratio of reused-to-unused cache lines for each data set at each level allows the RHT to determine whether it would be beneficial to move it to each level.

To learn where data is reused, and adapt to changes in application and data behavior, RAP continually allows a subset of the data to flow through all levels of the cache hierarchy independent of the RAP policy. These *learning cache lines* allow the RHT to measure the reuse potential at each level and over time.

3. Intelligent Data Movement. When a cache line is requested by the CPU or evicted by a cache's local replacement policy, the RHT is consulted based on the cache line's data set to determine to which level the line should be moved. In an eviction, the cache line is moved to the next level closer to the DRAM that shows a significant degree of reuse in the RHT. If the data is requested by the CPU, similar counts for the L1 (and/or L0) are used to determine where to install it. For this approach to work, the RHT needs to be able to learn reuse behavior on data evictions, but associate it with data in such a way that the placement policy can be applied when other data from that data set is moved or installed.

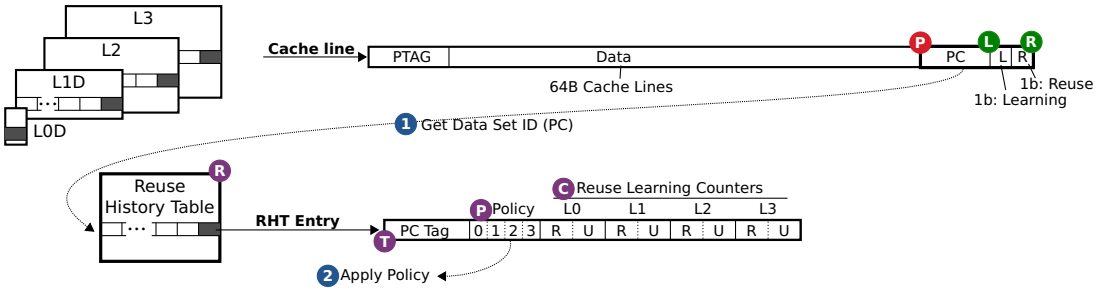

Figure 2. RAP Design Overview. Each cache line is extended with a learning bit (L), a reuse bit (R), and the cache miss PC. The Reuse History Table (RHT) keeps track of where data is reused in the cache hierarchy.

Summary. The Reuse Aware Placement (RAP) design presented here reduces data movement by only *placing data* in the cache level where it is likely to experience reuse. This is done by tracking the ratios of reused-to-unused evicted cache lines for each data set at each level. This information is stored in the Reuse History Table (RHT), which is consulted upon installations and evictions to find the right destination for the data. For this approach to work, the cache requires a mechanism to identify the data set for a cache line upon its eviction such that it can find the correct RHT entry to update and consult for moving the data. By placing data where it is likely to see reuse, RAP significantly reduces data traffic and energy.

3. Design and Implementation

The RAP memory hierarchy is shown in Figure 2, with changes to the baseline indicated by darker lines. The hierarchy has four levels: L0D, L1D, unified L2, and a shared L3, with similar instruction caching. The CPU can directly access data in the L0 and L1. The underlying design is non-inclusive/non-exclusive, such that data is by default (when the reuse is not known) installed in all levels upon demand, but does not have to be present in any particular combination of levels at any time.

3.1. Reuse History Table

The Reuse History Table (R in Figure 2) is an associative structure that stores reuse information for each data set. In this design we use the *PC of the access that caused a DRAM fetch* (i.e., last level cache miss PC) to define a data set T. This definition works well as it categorizes data by the instructions that first use it, which results in data with similar usage behavior being grouped together. Other definitions could be used as well, including ones that incorporate address information and/or page offsets [11], [12], [3].

The RHT stores information about the reuse of a data set at each level in the hierarchy. This data is stored in a set of Reuse Learning Counters C, which track the number evicted cache lines from the data set that were reused (or not) before eviction. If the number of cache lines that were reused before eviction is significantly higher than the number that were not reused, then this is an indication that the data set would benefit from being stored in that level. In addition to the Reuse Learning Counters, the RHT stores Policy Bits P for each cache level. The Policy Bits are the current policy that is applied on an eviction or install, and are periodically updated to reflect what the Reuse Learning Counters have learned. A set Policy Bit indicates that reuse is expected in that level, and that data should be moved there on an installation or eviction. (See Figure 1c.) Providing separate Policy Bits and Reuse Learning Counters avoids the need to constantly compare the counter values to identify the policy and allows us to continue updating the counters until we reach a threshold without changing the active policy.

3.2. Detecting Data Set Reuse

RAP detects data set reuse by looking at whether a cache line has been reused in its current level when it is evicted, and updating the Reuse Learning Counter (reused and not-reused counts) for the cache line's data set and level based on this information. To do so, each cache line is extended with a reuse bit R that is set on read hits to that cache line. This operation can be readily integrated into the replacement policy update. However, to update the RHT on eviction, we need to know the data set for the cache line, which is defined by the PC of the access that caused DRAM fetch in this implementation. To support this we extend each cache line with PC information P, which allows us to update the correct Reuse Learning Counter in the RHT upon eviction.

To learn data set reuse behavior we need to move data through all levels of the cache hierarchy and record where it is reused. This is accomplished by choosing a subset of cache lines as *learning cache lines* and not applying reuse aware placement to them. The learning cache lines will then follow the standard inclusive data placement path through the hierarchy (install in each of L0, L1, L2, L3, and then writeback to L1, L2, L3, DRAM on eviction) and the information about whether the data is reused at each level will give a relatively unbiased view of where the data sets will experience reuse. Learning cache lines are marked by a learning bit L for each cache line. If the learning bit is set, then upon eviction the RHT Reuse Learning Counters are updated, but the Policy Bits are ignored. To choose learning cache lines, a fraction of DRAM fetches are selected to be learning cache lines. This use of learning lines allows RAP to adapt to changes in application and data set behavior.

The use of learning cache lines that traverse the whole hierarchy avoids the difficulties of local sampling at each cache level. For example, if the L2 is being heavily bypassed, then we would expect many fewer evictions from L2 and sampling those evictions that do occur might provide a very biased result. By tracking a learning cache lines across the whole hierarchy, we obtain a view of how that data moves without any policy intervention for the current execution context. Under RAP, this global view of data reuse goes beyond what a local eviction mechanism would see, as RAP will avoid sending data to a cache level at all if it will not experience reuse there.

Frequency	2.5 GHz
Fetch / Issue / ROB / IQ / LSQ / Regs.	3 / 8 / 32 / 32 / 16 / 128
L0 Instruction / Data Caches	4kB, 64B, 1-way, DM, 3 c
L1 Instruction / Data Caches	32kB, 64B, 8-way, LRU, (+1) 4 c
L2 Unified Cache	256kB, 64B, 16-way, LRU, (+8) 12c
L3 Shared Cache	8MB, 64B, 16-way, LRU, (+16) 28c
DRAM	4 GB, 30 ns, 12.8 GB/s
RHT *(RAP only)*	256, 4-way, LRU

Table 1. PROCESSOR CONFIGURATIONS.

The Policy Bits for each level are decided based on that cache level's Reuse Learning Counters. While this decision uses the information specific to its particular cache level, it takes into account the global behavior of the cache, as the Reuse Learning Counters are updated by learning cache lines that traverse the whole hierarchy. This allows them to obtain a decent global picture of where they experience reuse throughout the hierarchy with very little overhead.

3.3. Intelligent Data Movement

When moving a cache line (either towards the processor on an install or away on an writeback/eviction) we use its PC ❶ to consult the Policy Bits in the RHT to determine which cache levels have shown reuse and ❷ use that information to determine where to move the data. The policy bits are set by comparing the reused and not-reused counts in the Reuse Learning Counters to a given threshold whenever its counter overflows. This naturally updates the policy as the program executes, thereby adapting to changes in data and application behavior. When data is moved, a set policy bit indicates that reuse is expected at a given level and that the data may be moved to that level, otherwise the level is bypassed. By addressing the issue of data *placement* we avoid needing to move data into a cache level if we expect no reuse, which saves energy and capacity over approaches that rely on the *eviction* mechanism to intelligently move the data out once it has been installed.

4. Results

To evaluate the effectiveness of Reuse Aware Placement we compare it with standard exclusive and non-exclusive/non-inclusive data placement policies. We investigate data traffic at each cache level, the amount of data installed and reused in each cache level, and the resulting impact on energy and performance.

4.1. Methodology

We use the gem5 x86 full-system simulator [13] to evaluate performance. Table 1 lists the processor and memory system configurations. We focus on energy-efficient processors and configured the gem5 simulator to resemble a Qualcomm-based ARM system running at 2.5 GHz with L0, L1 and L2 caches, and extend it to include a larger L3 to model a deeper cache hierarchy for more advanced mobile systems. The caches have local LRU eviction policies to isolate the effects of the reuse-aware placement. The DRAM stride prefetcher is enabled for the final evaluation (Section 4.8) but not for the initial studies to avoid confusing policy behavior and prefetching behavior.

We use the CACTI 6.5 [14], [15] cache simulator (22 nm process, one read/write port) to compute access times and energy. The L0 and L1 are on the critical path and use high-performance transistors (itrs-hp) and optimize for access time. The L2, L3, L2, and RHT, etc., are not performance critical, and use low static power transistors (itrs-lstp) and optimize for area and read energy. All data structures holding metadata and their accesses are modeled and included in performance and power measurements.

We run 570 simulation points from the SPECcpu-2006 [16] benchmarks with reference inputs, SPECjbb-2005 [17] and TPC-C [18] (MySQL/InnoDB). For each benchmark we use 10 uniformly distributed simulation points, with 250 M instructions of cache warming per simulation point followed by 100 M simulated instructions.

We compare RAP with two standard data placement designs: exclusive (Base-Ex) and non-exclusive/non-inclusive (Base-In). All configuration use phased L1, L2, and L3 caches, where the L0 and L1 tags are accessed at the same time. For more discussion on the impact of phased vs. parallel caches, the effect of adding the L0, and alternate eviction policies please see the discussions in Section 4.7.

4.2. Hardware Configuration and Costs

From the design space exploration, we chose an RAP configuration with a 256-entry, 4-way RHT (see Section 4.1). The RHT is 1.8 kB of SRAM with a 12-bit PC (6b tag + 1 valid bit), four policy bits, and eight 6-bit reuse counters.

The cache lines are extended with 14 bits of metadata (1b reuse, 1b learning and 12b PC). The reuse bits can be readily integrated into the replacement policy logic, and the learning bit and PC are accessed only on evictions and cache misses. The cache metadata adds 112B , 896B, 7 kB, and 224 kB of extra SRAM to the L0, L1, L2, and L3, respectively.

In all, RAP requires about 2.8% more SRAM than a standard cache hierarchy, in exchange for a dramatic reduction in data movement and energy, and no performance loss.

4.3. Cache Traffic

Data traffic between all four cache levels is shown in Figure 3, normalized to the Base-Exclusive data placement policy. The figure has a row for each cache level and shows the distribution of accesses to that level across hits, inserts, and writebacks, as a percentage of the total accesses. The bars show, from left to right, Base-Ex, Base-In, and RAP data placement policies. The labels above the bars indicate the absolute number of accesses to each cache level per thousand instructions for Base-Ex.

Figure 3 shows that RAP is able to successfully identify data sets and intelligently place the data in the correct cache level across a wide range of applications. Across all benchmarks, RAP reduces the cache traffic compared to an exclusive cache placement policy by 21% (L0 ❶), 51% (L1 ❷), 64% (L2 ❸), 28% (L3 ❹)[1].

The significant reductions in L2 and L3 traffic are due to RAP's support for silent evictions and cache level bypassing. The benefit of silent evictions can be seen at the L2 level, where Base-In, which supports silent evictions, also shows a significant traffic reduction (31% lower for Base-In vs. 64% lower for RAP). Note that the

1. Note that RAP's memory hierarchy traffic reduction may be particularly valuable for designs where the Network on Chip (NoC) bandwidth is a bottleneck.

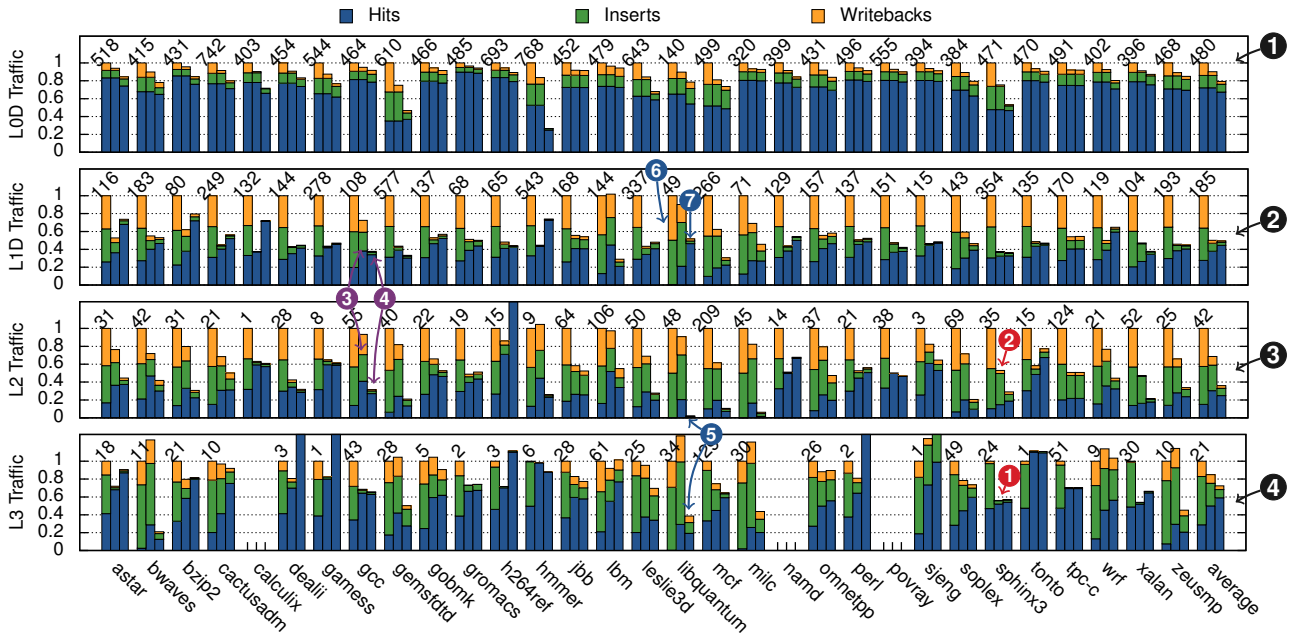

Figure 3. Data movement normalized to Base-Exclusive (left bar) for the configurations in Table 1. For each cache, the colors indicate the hits, inserts, and writebacks. From left to right: Base-Ex, Base-In, and RAP. Labels indicate number of cache accesses per thousand instructions for Base-Ex, with values less than 1 not shown.

reductions at the L2 and L3 levels are largely independent of the presence of an L0 (see Section 4.7).

At the L0 and L1 levels, the traffic reductions from RAP are smaller and vary significantly across benchmarks. The overall improvements are smaller at the L0 level due to CPU accesses that cannot be filtered. However, RAP is successful in reducing L0 traffic by only moving data to the L0 if it expects it to be reused several times (21% reduction) and by bypassing the L1 (51% reduction).

Example: Silent Evictions. Base-In and RAP reduce writeback traffic by keeping several copies of the data in the cache hierarchy, thereby enabling silent evictions for clean data. This can be seen in sphinx3 where L3 traffic is reduced by 40% (1) by virtually eliminating the write backs from the L2 and instead sending the data directly to DRAM. However, while Base-In placement supports silent evictions, it also installs copies of data in all levels of the hierarchy. The effect of this inclusive installation policy is seen in the large number of inserts (2), but few hits in the L0 and L2. RAP detects that this data is not reused in the L2 and bypasses it, resulting in significantly lower traffic to the L2, and increased traffic directly from the L3 to the L1 while not installing in the L0.

Example: Reuse Aware Placement. RAP only installs data where it expects it to be reused. For gcc, the baseline exclusive placement spends most of the cache accesses moving data through the hierarchy as seen by the large insert and writeback bars in the L3, L2, and L1. This results in a small number of hits as data is flushed out. Inclusive placement reduces data movement through silent evictions (smaller writeback bars for L0, L1, and L2) but still has a significant amount of data insertions due to its inclusive insertion policy (3). With RAP the data is placed where it will see reuse, which eliminates the need to constantly move it up and down the hierarchy. This data placement virtually eliminates the inserts and writebacks from the L1, L2, and L3 (4), and increases the number of hits over exclusive placement. The number of hits is higher with inclusive placement, but many of these are likely writebacks that hit due to the inclusive installation.

Example: Streaming Data. RAP detects that streaming data is not reused in the L2 or L3 and therefore installs it directly in the L0 or L1 (bypassing L2 and L3 on install) and evicts it straight to memory (bypassing L2 and L3 on eviction). This effect can be seen in libquantum, where data is fetched to L0, then worked on in L0 or L1, and finally evicted directly to memory bypassing both L2 and L3 (5). The baseline exclusive placement uses the L1 purely for data movement between the L0 and the L2, as seen by the lack of any hits and the large number of writebacks and inserts (6). The baseline inclusive placement does somewhat better, both in reducing the total accesses to the L1 and managing to achieve a few hits, but these are most likely due to writebacks from having inclusively installed the data in the L1 on the way to the L0. RAP, however, virtually eliminates the data movement through the L1 as shown by the very small number of inserts and writebacks. Further, the large number of hits in the L1 (7), without corresponding inserts and writebacks, indicates that this data is reasonably stable, and is therefore appropriately placed in the L1 for reuse.

4.4. Cache Utilization

To look at how effectively RAP installs cache lines that will be reused in a given level, we examine the number of cache lines installed and the percentage of them that are reused before eviction in Figure 4. The data is normalized to a Base-Ex placement (no bypassing) and shows the proportion of reused data in dark bars and the number of installed cache lines per thousand instructions for Base-Ex above the bars.

Overall RAP reduces the number cache lines installed across the hierarchy by 26% (L0), 36% (L1), 47% (L2) and 62% (L3). RAP is also successful at only installing data that is reused before

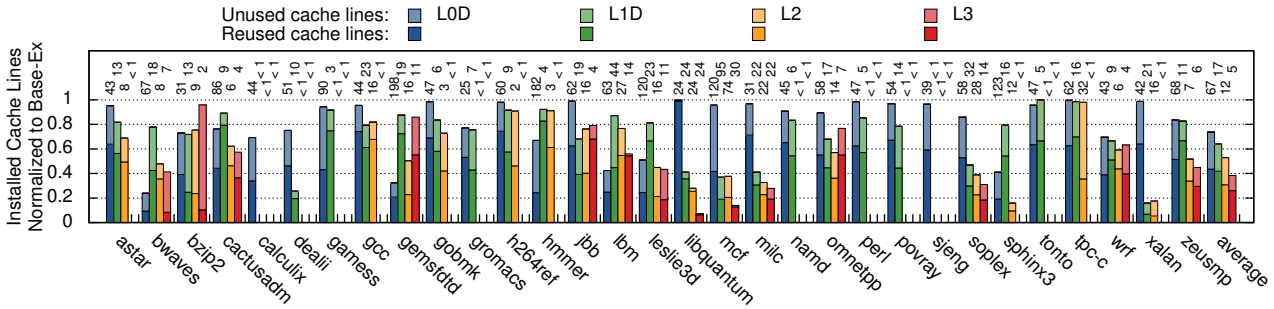

Figure 4. Number of installed cache lines for RAP normalized to Base-Ex (no bypassing). Left-to-right: L0, L1, L2, L3. The darker colors indicate the proportion of cache lines that are reused before eviction. The numbers give the absolute number of installed cache lines per thousand instructions for Base-In, with values less than 1 not shown.

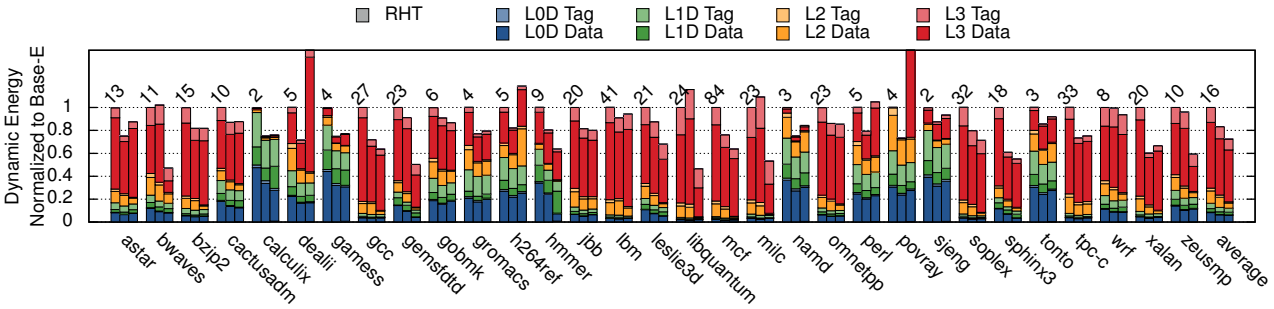

Figure 5. Dynamic energy normalized to Base-Ex. From left to right: Base-Ex, Base-In, and RAP. Labels show cache pJ per instruction for Base-Ex. (Note that dealli and povray see an increase in L3 hits, while h264 sees an increase in L2 hits under RAP. See Figure 3)

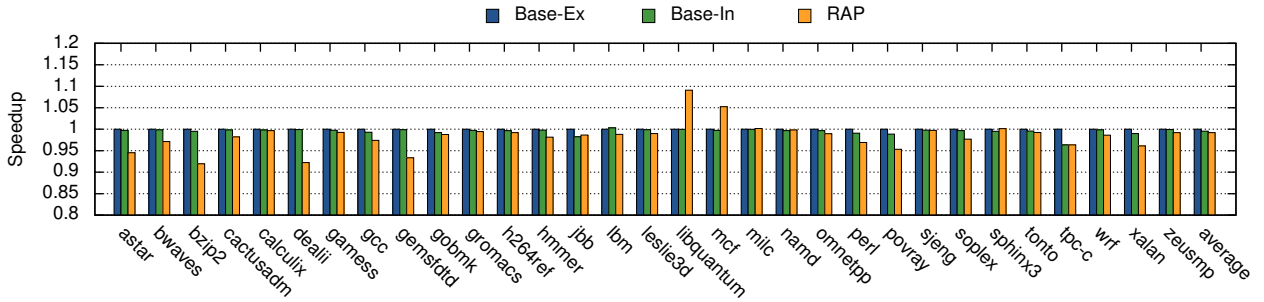

Figure 6. Execution time normalized to Base-Ex. From left to right: Base-Ex, Base-In, and RAP.

eviction, with 59% (L0), 66% (L1), 58% (L2) and 68% (L3) of data seeing reuse, compared to 46% (L0), 54% (L1), 40% (L2) and 44% (L3) for the inclusive cache. While RAP cannot reach 100% reuse due to the learning cache lines (which do not follow the RAP policy) and varying behavior within each data set, this data shows that RAP is very successful in only placing data where it will see reuse.

4.5. Energy

Figure 5 compares the impact of RAP's reduction in data traffic on dynamic energy in the cache hierarchy with the Base-In and Base-Ex placement policies. It is clear from this data that while the hierarchy does a very effective job of filtering out accesses to the L3, the L3's size makes each access sufficiently expensive that it is still responsible for the majority of the dynamic energy consumption. This emphasizes the importance of placing data correctly in hierarchy: eviction mechanisms must pay the energy cost of installing and evicting data at each level, even if they can evict it intelligently to avoid the performance penalty due to pollution, while a placement mechanism can avoid that cost in the first place by not moving the data. We found that installing data that is unlikely to be reused in the LRU position, instead of bypassing it, increased energy by 11% with RAP.

The results across all benchmarks show that silent evictions of clean data reduce dynamic energy by 17% (Base-In vs. Base-Ex) while RAP is able to reduce dynamic energy by 28% due to more intelligent placement and bypassing, with 2.6%, 1.8%, 7.7%, 15.6% of the total energy savings coming from the L0, L1, L2, and L3, respectively. This indicates that just over half of the energy savings comes from intelligently managing what data we place in the L3 cache, which is not unexpected given the large size of the L3 for our design point. The corresponding relative energy reductions for each level are 30%, 24%, 58%, 22%, for the L0 to L3.

The energy overhead of RAP is small, with the RHT consuming less than 1% of the total energy due to its small size and limited reading and writing (only writing to the Reuse Learning Counters on a learning cache line eviction and primarily reading only the policy bits). However, as 60% of Base-Ex's total cache energy is dynamic, RAP reduces the total cache energy by 17%. Note that

RAP also improves the effective cache capacity by only installing data that is likely to be reused. This reduction in cache utilization opens up interesting possibilities for reducing cache size either dynamically or at design time without hurting performance.

4.6. Performance

Figure 6 shows the execution time normalized to Base-Ex. Both Base-In and RAP have slightly worse execution time compared to Base-Ex for SpecJBB and TPC-C. This is due to the benchmarks' increased instruction pressure and the reduced effective cache capacity of the non-exclusive L0 and L1. gemsfdtd shows a slight slowdown due to overly-aggressive bypassing, which results in decreased hit ratios in the L1, L2, and L3 caches. RAP shows a slight speedup for libquantum, mcf, and milc due to significant bypassing of L2 and L3 for datasets that do not fit in the caches.

On average, RAP achieves its goal of not hurting performance (0.8% slowdown) compared to Base-Ex data placement, while demonstrating significant decreases in data movement and energy.

4.7. Discussion: L0/L1, Phased and Parallel

To investigate the impact of the L0 cache we ran experiments without it and found that the L0 cache reduced the total dynamic cache energy by 13% for Base-In, but increased the energy by 2.3% for Base-Ex, since it has more L0 to L1 traffic. This demonstrates that an L0 cache can be very effective at reducing cache energy if it can be managed correctly. The reduction in data movement at the L2 and L3 levels are largely independent of an L0 as the similar filtering would be done in an RAP design with only an L1. In addition we investigated parallel tag and data lookup in the L1 for lower latency and found that it increased energy in Base-In and Base-Ex by 40% and 27%, respectively. As a result of these experiments we chose a baseline with an L0 and a phased L1.

4.8. Discussion: Prefetching

The preceding evaluations were done without a prefetcher to avoid confusing installation policy and prefetcher behavior. To analyze the impact of a prefetcher, the performance and energy experiments we ran the experiments with a strided DRAM-to-L1 prefetcher. Across all designs the only significant change is a roughly 10% performance increase and a small (<6%) increase in unused data. The increase in unused data could be avoided by integrating the prefetcher with RAP by having it return the miss PC for the prefetched cache lines, and thereby enabling RAP to learn how to best place the prefetched data.

Summary. To evaluate RAP's success in intelligently placing data we compared three different data placement policies for traffic, cache usage, energy and performance. As expected, we found that standard inclusive placement (Base-In) reduce traffic compared to exclusive (Base-Ex) by supporting silent eviction of clean data, but sacrifice cache capacity. RAP shows the benefits of inclusive placement, but moves data more optimally by bypassing levels that show little reuse, thereby dramatically reducing traffic (28% lower for the L3) and energy (28% lower dynamic, 17% lower total), without hurting performance.

5. Related Work

Cache Bypassing. Cache bypassing has been studied extensively. Most recent work has focused on last-level cache insertion and bypassing [19], [20], [21], [22], [23], [10], [24], [4]. For example, Albericio et al. [4] only install data in the last level cache when the data is reused, whereas Zahran et al. [5] do the opposite and install in the last level cache but bypasses the first level cache. Such methods focus on a single cache level whereas RAP addresses the whole cache hierarchy. Many of these techniques orthogonal to RAP and can be used in combination with RAP to improve the replacement policy within a cache level.

Cache Replacement. There has been a significant amount of work done on intelligent cache eviction/replacement policies. Johnson et al. [25] develop a reuse-aware replacement policy to reduce conflict misses in low-associative cache hierarchies, instead of reducing energy by minimizing data movement. On a cache miss, they look up the hit count for the missing block and the hit count for the to-be-replaced block, and keep the one with the higher hit count in the cache. RAP tracks data based on PC and only sets a reuse bit on a cache hit, whereas [25] use macro-blocks (a group of cache lines) and increment hit counters on every memory access.

For shared caches, there are many policies that decide what data to evict from the cache based on local reuse information, but do not decide where to place the data globally as RAP does. The most similar of these approaches is Wu et al.'s [3] SHiP, a PC-based technique to improve last level cache replacement performance. SHiP predicts whether a cache lines will hit or not at the local cache, and inserts the cache line in the LRU position for standard LRU caches (distant prediction for SDRIP caches) if it predicts that the cache line will not be reused. RAP also uses the PC to predict cache line behavior, but uses it to reduce energy across all cache levels by not placing it in the levels it will not be reused. It would be possible to extend the SHiP work with cache bypassing (although the cache hierarchy used in their work required inclusive placement) and by using it at each level in the hierarchy. We investigated the energy cost of installing data that was unlikely to be reused in the LRU position (instead of bypassing it all together as RAP does), and found that RAP with LRU install instead of bypass improved performance by 1% at a 11% cost in energy.

Since RAP is agnostic to the underlying replacement policy, these local eviction policies could be combined with RAP's global data placement in many interesting combinations.

Cache Traffic. Kumar et al. [26] dynamically resize the cache line width, which enables them reduce traffic and use the cache capacity more efficiently by only storing the parts of a cache line that are actually used. Such an approach could be combined with RAP to move words instead of whole cache lines. Traffic between different cores and shared data have also been studied extensively, where the most recent work include [27], [28]. RAP focus on private data movement, but can could readily be extended/combined to optimize for shared data as well.

Inclusive vs. Exclusive Hierarchies. Exclusive cache hierarchies provide more cache capacity since data is not replicated, but they see increased traffic as a result. Sim et al. [29] propose FLEXclusion, a cache design that is able to switch between exclusive and inclusive mode depending on workload. A problem with inclusive cache hierarchies are forced evictions due to the inclusion requirement and priority inversion. To reduce forced evictions, Jaleel et al. [30] propose TLA, a set of cache management

techniques that use temporal locality in the cores' private caches in order to find a victim that is not present in private caches. RAP is a non-inclusive/non-exclusive hierarchy that allows data to reside wherever reuse is expected, thereby enabling both the cache capacity of exclusive caches and the traffic reductions of inclusive caches. However, TLA could be used together with RAP to improve the L3's replacement policy.

NUCA Caches. NUCA caches have been developed to deal with growing wire delays in large last level caches [7], [31], [32], [6]. To optimize for the different access latencies, several policies have been suggested to migrate data closer to where it is needed. These techniques focus on one large cache, whereas RAP crosses different cache level boundaries and optimizes placement across the levels.

6. Conclusions

This paper presented Reuse Aware Placement (RAP), a new mechanism for reducing data traffic and energy in the memory hierarchy through more intelligent data *placement*. RAP learns where data sets are reused and uses this information to only move data to cache levels where it will see reuse.

We compared RAP's data movement, cache utilization, energy, and performance against traditional inclusive and exclusive data placement designs. RAP demonstrated dramatic reductions in data movement across a wide range of benchmarks (51%, 64%, and 28% for the L1, L2, and L3 caches, respectively) resulting in a significant dynamic (28%) and total (17%) energy savings, with less than 1% performance degradation. The cost of RAP is a 2.7% increase in on-chip storage over a traditional hierarchy. As RAP is agnostic to the underlying local cache replacement policy, it opens up a new design space for integrating data placement and eviction policies across the memory hierarchy.

Acknowledgments. This work was financed by the Swedish Foundation for Strategic research under grant FFL12-0051 and the UPMARC research center. Simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX).

References

[1] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2007.

[2] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2010.

[3] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "SHiP: Signature-based Hit Predictor for High Performance Caching," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2011.

[4] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería, "The Reuse Cache: Downsizing the Shared Last-level Cache," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2013.

[5] M. Zahran, K. Albayraktaroglu, and M. Franklin, "Non-Inclusion Property in multi-level Caches Revisited," *International Journal of Computers and Their Applications Special Issue on Techniques and Architectures for High Performance and Energy Efficient Computing Systems*, vol. 14, no. 2, jun 2007.

[6] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2009.

[7] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. Internationl Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[8] A. Sodani, "Race to Exascale: Opportunities and Challenges," in *MICRO 2011 Keynote*, 2011.

[9] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *Proc. International Solid-State Circuits Conference (ISSCC)*, 2014.

[10] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, "Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches," in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[11] B. Calder, D. Grunwald, and J. Emer, "Predictive Sequential Associative Cache," in *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 1996.

[12] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos, "Accurate and Complexity-Effective Spatial Pattern Prediction," in *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2004.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Comput. Archit. News*, 2011.

[14] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," Hewlett Packard Labs, Tech. Rep., 2009.

[15] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2009.

[16] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, 2006.

[17] SPECjbb2005, *http://www.spec.org/jbb2005/*.

[18] Transaction Processing Performance Council, *http://www.tpc.org/*.

[19] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and Insertion Algorithms for Exclusive Last-level Caches," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2011.

[20] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng, "Optimal Bypass Monitor for High Performance Last-level Caches," in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[21] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The Evicted-address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[22] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving Cache Management Policies Using Dynamic Reuse Distances," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2012.

[23] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2012.

[24] S. Gupta, H. Gao, and H. Zhou, "Adaptive Cache Bypassing for Inclusive Last Level Caches," in *Proc. International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.

[25] T. L. Johnson and W.-m. W. Hwu, "Run-time Adaptive Cache Hierarchy Management via Reference Analysis," in *Proc. International Symposium on Computer Architecture (ISCA)*, 1997.

[26] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2012.

[27] G. Kurian, O. Khan, and S. Devadas, "The Locality-aware Adaptive Cache Coherence Protocol," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2013.

[28] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas, "Protozoa: Adaptive Granularity Cache Coherence," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2013.

[29] J. Sim, J. Lee, M. K. Qureshi, and H. Kim, "FLEXclusion: Balancing Cache Capacity and On-chip Bandwidth via Flexible Exclusion," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2012.

[30] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer, "Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2010.

[31] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2003.

[32] B. M. Beckmann and D. A. Wood, "Managing Wire Delay in Large Chip-Multiprocessor Caches," in *Proc. International Symposium on Microarchitecture (MICRO)*, 2004.