Probabilistic Cache Filtering

Yoav Etsion  Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem, Israel

Abstract

Distinguishing transient blocks from frequently used blocks enables servicing references to transient blocks from a small highly-associative auxiliary cache structure. By inserting only frequently used blocks into the main cache structure, we can reduce the number of collision misses, thus achieving higher performance and allowing the use of direct mapped caches which offer lower power consumption and lower access latencies. In this paper we introduce a simple probabilistic filtering mechanism to identify and select the frequently used blocks. We show that a 16K direct-mapped L1 cache, augmented with a fully-associative 4K filter, achieves on average 11% more instructions per cycle than a regular 16K, 4-way set-associative cache, and even 4% higher IPC that a 32K, 4-way cache, while consuming about 60–70% less dynamic power than either of them.

1 Introduction

The increasing gap between processor and memory speeds witnessed in recent years has exacerbated the CPU’s dependency on the memory system performance — and especially that of L1 caches with which the CPU interfaces directly. One result of this ongoing trend is the increase in the capacity of L1 and L2 caches, in an effort to bridge the memory-processor gap and improve overall system performance. This improvement, however, comes at a price of increased power consumed by the caches — estimated at more than 10% of the overall power consumed by the CPU [13], and expected to rise.

Processor power consumption has become an increasing concern due to two conflicting trends: On one hand, increased clock frequency, CPU complexity, and the number of transistors per chip naturally leads to increased overall power consumption [23, 32]. On the other hand, the growing popularity of mobile, battery powered computing is driving a quest for power efficiency. As a result designers are often confronted with a power-performance tradeoff.

In this paper, we claim that exploiting well-known workload characteristics may help alleviate this tradeoff. Specifically, memory usage is known to be highly skewed, with most references directed at a relatively small subset of the address space. By identifying these references and servicing them using power-efficient, direct-mapped L1 caches, we can potentially increase CPU performance while at the same time reducing the power consumption.

Direct-mapped caches are faster and consume less energy than set-associative caches typically used in L1 caches [10]. However, they are more susceptible to collisions misses than set-associative caches, thus suffering higher miss-rates and achieving lower performance. This deficiency led to abandoning direct-mapped L1 caches in favor of set-associative ones in practically all but embedded processors.

The main contribution of this paper is based on analyzing the memory reference workload and showing it can be characterized using a statistical phenomenon called mass-count disparity [5, 7]. Based on this
observation, we design a simple probabilistic L1 cache filter that uses a simple coin toss to preferentially
insert frequently used blocks into the cache proper, thus reducing the number of collision misses in the
cache; the rest of the references are serviced from the filter itself, which is a small fully-associative auxiliary
structure. We show that this mechanism can harness the speed and low power traits of direct-mapped caches
to reduce the overall L1 power consumption, while still improving overall performance. While using an
auxiliary structure to filter memory references has been explored in the past [14, 16, 17, 18, 20, 21, 24,
25, 30], as far as we know we are the first to harness a simple statistical phenomenon to filter both L1
reference streams efficiently enough to use a direct-mapped structure for L1 caches, thus both reducing
power consumption and improving performance.

The remainder of this paper is organized as follows. The next section describes the mass-count disparity
phenomenon and its application to characterizing L1 reference streams. After reviewing our methodology
in Sect. 3 we present the general design of a dual-structured cache in Sect. 4, and our specific realization
based on probabilistic sampling in Sect. 5. We then describe the effect of probabilistic sampling on L1 cache
behavior (Sect. 6), and the overall impact on power and performance (Sect. 7). Finally, we review the body
of work related to this study (Sect. 8), and conclude in Sect. 9.

2 The Skewed Distributions of Memory Access Patterns

Locality of reference is one of the best-known phenomena in computer workloads [6]. In a nutshell, this
refers to the fact that when a certain address is referenced, there is a high probability that the same address
or a nearby one will be referenced again soon. But this is actually the result of two distinct properties:
that nearby references are correlated, and that some addresses are much more popular than others [15].
Importantly, references to blocks that are seldom accessed can still be grouped together; we call such blocks
transient.

A good way to visualize skewed popularity is by using mass-count disparity plots [7]. These plots
superimpose two distributions. The first, which is called the count distribution, is a distribution on addresses,
and specifies how many times each address is referenced. Thus $F_c(x)$ will represent the probability that an
address is referenced $x$ times or less. The second, called the mass distribution, is a distribution on references;
it specifies the popularity of the address to which the reference pertains. Thus $F_m(x)$ will represent the
probability that a reference is directed at an address that is referenced $x$ times or less.

A problem with the above definition is that it considers all the references to each address, throughout
the duration of the run. But the relative popularity of different addresses may change in different phases
of the computation, so the instantaneous popularity may be more important for caching studies. A possible
solution is to use a certain window size, and only consider references made within this window. This in turn
suffers from a dependence on the window size. Our solution is therefore not to count all the references to
each address, but to count only the number of references made during a single cache residency. Thus, if a
certain block is referenced 100 times when it is brought into the cache for the first time, is then evicted, and
finally is referenced again for 200 times when brought into the cache for the second time, we will consider
this as two instances with popularities 100 and 200, rather than as a single block with popularity 300. The
distributions of reference counts shown for the SPEC2000 benchmarks were calculated in this way based on
a 16KB direct-mapped cache.

Returning to mass-count disparity plots, the disparity refers to the fact that the graphs of the count
and mass distributions are quite distinct. An example is shown in Fig. 1, using four of the SPEC 2000
benchmarks with the ref input. The divergence between the distributions can be quantified by the joint ratio
[7], which is a generalization of the proverbial 20/80 principle: This is the unique point in the graphs where
Figure 1: Mass-count disparity plots for memory accesses in select SPEC benchmarks, using the ref input (for each benchmark, the top plot shows the data stream, and the bottom one the instruction stream). The mass-count metrics are demonstrated on Vortex’s data stream: the left arrow shows the $W_{1/2}$ metric, the middle double-arrow shows the joint-ratio point, and the right arrow shows the location of the $N_{1/2}$ metric.

The sum of the two CDFs is 1. In the case of the vortex data stream graph for example, the joint ratio is approximately 13/87 (double-arrow at middle of plot). This means that 13% of the caching instances, and more specifically those instances that are highly referenced, get a full 87% of the references, whereas the remaining 87% of the instances get only 13% of the references. Thus a typical caching instance is only referenced a rather small number of times (up to about 10), whereas a typical reference is directed at a highly-accessed caching instance (one that is accessed from 100 to 10,000 times).

More important for our work are the $W_{1/2}$ and $N_{1/2}$ metrics [7]. The $W_{1/2}$ metric assesses the combined weight of the half of the caching instances that receive few references. For vortex, these 50% of the instances together get only 3% of the references (left down-pointing arrow). Thus these are instances of blocks that are inserted into the cache but hardly used, and should actually not be allowed to pollute the cache. Rather, the cache should be used preferentially to store heavily used instances, such as those that together account for 50% of the references. The number of highly-referenced instances needed to account for half the references is quantified by the $N_{1/2}$ metric; for vortex it is less than 1% (right up-pointing arrow).

The existence of significant mass-count disparity has important consequences regarding random sampling. Specifically, if you pick a block (or caching instance) at random, there is a good chance that it is seldom referenced. That is why random replacement is a reasonable eviction policy, as has been observed many times [26]. But if you pick a reference at random, there is a good chance that this reference refers to a block that is referenced very many times. Thus random sampling of references may be expected to identify those blocks that are most deserving to be inserted into the cache. Such an insertion policy is the focus of this paper; it is described in detail and analyzed in Sections 5 through 7.
Wire cache
micro-architecture
L1 design | split & filtered
---|---
DL1/IL1 size | 16/32 KB
DL1/IL1 line size | 64 B
DL1/IL1 set assoc. | 1 (DM)
DL1/IL1 latency | 1 cy.*

L2 cache
branch predictor
L2 design | unified
---|---
L2 size | 512 KB
L2 line size | 64 B
L2 set assoc. | 8
L2 latency | 16 cy.

memory
memory latency | 350 cy.

* 2 cycles in simulations of set-associative and fully-associative L1 caches.

Table 1: Basic configuration used in the out-of-order simulations.

3 Methodology

To evaluate the concepts presented in this paper we have used a modified version of the SimpleScalar toolset [2]. The modifications include replacing SimpleScalar’s cache module, as well as fixing the code of its out-of-order simulator (sim-outorder) to accommodate a non-random-access L1 cache model where a hit latency is not constant for all blocks — typically those residing in the cache vs. others residing in a small auxiliary filter (our cache model is described in Section 4).

We have used the SPEC2000 benchmarks suite [28] compiled for the Alpha AXP architectures [27]. An overall of 19 benchmarks were used — all benchmarks but seven: eon, gap, fma3d, and sixtrack failed, our copy of applu’s binary was corrupt, equake experienced too few L1 misses (under 0.02%) on both data and instruction streams, and mcf required a substantially reduced number of instructions.

All benchmarks were executed with the ref input set and were fast-forwarded 15 billion instructions to skip any initialization code (except for vpr whose full run is shorter), and were then executed for another 2 billion instructions. The out-of-order simulations consisted of an aggressive, 8 wide machine whose general parameters are detailed in Table 1. The same architectural model was used with various L1 cache sizes and associativity parameters in the evaluation of our design.

Power and energy estimates were compiled using the CACTI tool [29].

4 Principles of Filtered Cache Design

Cache filtering is a special case of using a dual cache structure. This means that the cache is augmented with an auxiliary structure that is used in some special way. Well-known examples include the use of a victim cache or a prefetch buffer. Our work is compared and contrasted with such alternatives in Section 8.

Our filtered cache design — depicted in Figure 2 — consists of a regular cache preceded by a small, fully-associative filter into which blocks are initially fetched. When a memory access occurs, the data is
first searched in the cache. If the cache misses the filter is searched. If the filter misses as well, the request is sent to the next level cache, and when the block arrives it is inserted into the filter.

The filter is embedded with a reuse predictor, which, on every filter hit, issues a prediction whether the accessed block is a frequently used one. When the predictor asserts that the block is a frequently used one, the block is promoted into the cache, replacing the LRU block in the target set. Delivering the data to the CPU and promoting the block into the cache occur simultaneously (promotion consumes a single cycle latency). The replaced block is simply discarded (and written back to the next level cache if necessary).

Both the cache and the filter are managed using the LRU replacement policy. If a block hit in the filter is not promoted to the cache, it is nevertheless moved up to the MRU position. If a block reaches the LRU position of the filter without being promoted, it is discarded when the space is needed to insert a new block (again, not before being written back to the next level cache if necessary).

The concept of a filtered design with a reuse predictor is in fact a generalization of previously published L1 filtering designs [25, 21, 24, 3]. These are each based on different filtering metrics, which can be mapped to our general design using different reuse predictors.

An important issue in cache and filter design is the tradeoff between performance and power expenditure. Caches can either be accessed in a fast mode — in which the entire set of cache lines is brought to the edge of the data store in parallel to the tag lookup, and the lookup result is fed into a multiplexer to select the correct word from the set, or in a serial mode — in which the tag store is looked up first, and the result is used to access only the correct cache line in data store [32]. The first design offers greater speed but consumes more power, and is commonly used in L1 caches, where speed is more important as it lies on the processor’s critical path. The second design is more common in caches further away from the processor, that can tolerate a slight increase in latency [31, 19].

The same principles apply even more forcefully to the filter, which is fully associative. Our design uses a serially-accessed filter, and issues the filter’s tag lookup in parallel to the cache access. Since the tag lookup consumes little power compared to the cache power, the increase in overhead is minimal, and allows to decrease the filter access penalty to 2 cycles more than the cache access.

5 Filtering by Random Sampling

Given the framework for cache filtering presented above, the question is how to design the reuse predictor. As this predictor is supposed to identify blocks that enjoy relatively heavy use, an obvious candidate is to
simply count the number of accesses so far to each block (that is, since the block was inserted into the filter). When the count reaches a pre-selected threshold the block is declared to be heavily used and promoted into the cache.

The problem with this approach is that different applications require different thresholds. Thus a certain threshold value can be very good for one application, but useless for another (experimental results showing this are omitted due to space constraints). It is therefore better to use a more flexible approach, rather than a sharp boundary.

The disparity between the number of blocks referenced and the number of blocks servicing most references (see in Section 2) suggests a probabilistic method for finding frequently used blocks: as most references are serviced by a relatively small number of blocks, a randomly selected memory reference is very likely serviced by a frequently used block. For example, a data reference issued by the facerec SPEC benchmark (Figure 1) has a probability greater than 0.93 to be serviced by a block that is accessed more than 15 times while in the cache, even though a block inserted into the cache by the application has a probability of more than 0.50 to be evicted after 1 or 2 references.

Based on this observation we suggest to make promotion decisions based on random flips of a biased coin. In other words, when a block is hit in the filter promote it to the cache with some pre-selected probability \( P \). Note that, in its simplest form, this method does not store any explicit information about the history of the block. However, blocks that are referenced many times do in fact have a higher probability of being promoted, because they will be considered again upon each filter hit.

A possible concern regarding this scheme is the danger of false positives — blocks that are not accessed many times but happen to be lucky in the first coin toss. This motivates one of two approaches: reducing the probability of promotion \( P \), or requiring \( N \) successful flips before being promoted. This changes the distribution of hits till promotion from geometric to negative binomial. The next section will be devoted to studying the effect of these two parameters \((P \text{ and } N)\), and deciding which to use in the subsequent performance analysis.

The design of the random sampling filter requires a source of random bits generated for each filter reference, that can be used to determine the outcome of the ensuing Bernoulli trial. In most of the simulations reported below we simply used the \texttt{rand()} C library function. A possible design for a real implementation is based on a \( K \) bit linear-feedback shift register [32]. This is used to generate a number \( r \) in the range \([1 \ldots 2^K - 1]\). For the desired threshold probability \( P \) we then calculate a constant \( C_P \) such that \( \frac{C_P}{2^K} \approx P \). Therefore, the result of the comparison \( r \leq C_P \) would yield \textit{true} with probability \( \sim P \) and \textit{false} with probability \( \sim (1 - P) \).

Although the proposed pseudo random number generator is simple to implement and consumes negligible power, we have also experimented with naive periodic sampling. For a given probability \( P \), select the prime number closest to \( \frac{1}{P} \), and call it \( D \). Then simply promote the block associated with every \( D \)th hit in the filter. As shown in Figure 8 below, the performance results are nearly identical for true random sampling and for periodic sampling, indicating that a true source of randomness is not warranted.

### 6 The Effect of Random Filtering

Filtering can be viewed as the partitioning of the reference stream into two. Most of the references are to frequently accessed blocks, which should reside in the cache. The rest are to transient blocks that should remain in the filter. It is the filter’s goal to split the reference distribution into these two components.

The importance of this separation is that the two components are quite different from each other. The references to heavily used blocks in the cache are numerous, but involve only a relatively small number
of distinct blocks. This has the potential to reduce collision misses in the cache, and enable the use of a low-latency, low-energy, direct-mapped cache structure; in our simulations, this is typically 16KB. The references to the filter, on the other hand, require a smaller but relatively expensive structure. We used a realizable 4KB fully-associative basic filter (64 lines). As a result, aggressive filtering might be counterproductive: if too many blocks are serviced from the filter and not promoted to the cache proper, the filter can become the bottleneck and degrade performance.

With these goals in mind, we evaluated the effectiveness of probabilistic filtering, and its effect on the workload, while exploring the statistical design space. The selected parameters are then used to evaluate the performance and power consumption in Section 7.

6.1 Impact on Reference Distribution

As noted above, random filtering using a single trial to determine whether a block should be promoted or not is prone to false-positive results, since all it takes is a single unpredictable toss of the coin. It is therefore necessary to consider whether using several trials to separate the reference distribution into its two components can yield a better distinction.

Figures 3 and 4 (data and instructions, respectively) compare the effectiveness of using multiple Bernoulli trials in splitting the reference distribution for select SPEC2000 benchmarks. These figures use fixed Bernoulli success probabilities and a varying number of trials, and show the two resulting reference distributions, compared to the original unfiltered one. The probabilities used are 0.50 for data streams and 0.05 for instructions streams, based on the results described below in Section 6.2.

Each plot shows three lines: the distributions of references to blocks in the filter and in the cache for the filtered design, and the distribution for a conventional, unfiltered, direct-mapped cache. Invariably, the distribution for the filter shows that most references are serviced by blocks accessed only a few times, while the distribution for the filtered cache shows that most references are serviced by blocks accessed many times. The distribution for the unfiltered cache falls in between.

Figure 3: Comparison of the data references’ mass distributions in the filtered cache structure and the regular cache structure for select SPEC benchmarks, using the ref input. The filters used are $N = 1$ (top) and $N = 3$ (bottom), both with $P = 0.5$. Note that the distribution for the unfiltered cache does not depend on $N$. 
Figure 4: Comparison of mass distributions of instruction references, analogous to the distributions of data references shown in Fig. 3, but using $P = 0.05$.

The figures show how increasing the number of successful Bernoulli trials a block is required to pass in order be inserted into the cache increases the distance between the distributions. In a nutshell, increasing the number of Bernoulli trials prevents infrequently used blocks from entering the cache, thus moving the distribution of references to cached blocks towards the higher values. For example, when looking at vortex’s data stream (Figure 3) we see how the median values change: while the median number of references to a cached block in a conventional cache is $\sim 500$, a single Bernoulli trial separates this into just 3 references for blocks that remain in the filter, as opposed to $\sim 1200$ for blocks that are promoted to the cache. Using two trials increases the separation to a median of 7 for the filter and more than 2300 for the cache (not shown), and three trials leads to 10 for the filter and $\sim 10K$ for the cache. This is a clear result of the mass-count disparity, where most of the blocks enjoy only few references, and increasing the number of Bernoulli trials simply retains more of them in the filter. This result is even more distinct in the apsi benchmark’s data stream, and is also manifested in the instructions streams (Figure 4).

The only exception is facerec’s instructions stream, where the two distributions remain similar to the original unfiltered distribution. This is caused by an extremely uneven distribution, in which nearly half the blocks enjoy 500–1000 references when cached, but this only accounts for less than 1% of the total references. The result is that even though more than 98% of the filter’s blocks are accessed less than 1000 times (as opposed to $\sim 10\%$ of the cache’s), the false-negatives — a small number of frequently used blocks that remain in the filter — are sufficient to skew the distribution. This demonstrates the problem of requiring too many successful Bernoulli trials: they reduce the number of false-positive, but tend to increase the number of false-negatives, tilting the filter’s reference distribution back in favor of the frequently used blocks.

Figure 5(a) shows the percentage of references serviced by the cache, compared with the percentage of blocks promoted from the filter into the cache, for various Bernoulli probabilities (with $N = 2$). Considering the mass-count disparity we would expect that promoting frequently accessed blocks into the cache would yield a more substantial increase in the number of references serviced by the cache, while promoting less frequently used blocks into the cache would have a smaller impact on the number references serviced by the cache. This is indeed evident in Figure 5(a): When increasing the success probabilities we see a distinctive
increase in the number of references serviced by the cache, until some level — indicated by the horizontal line — where this increase slows dramatically and promoting more blocks into the cache proper yields very little increase in the cache’s hit-rate. For example, when using two Bernoulli trials this saturation occurs at $P = 0.6$ for the data streams and $P = 0.4$ for the instruction stream. Beyond these probabilities the promoted blocks are mostly transient blocks and we experience the law of diminishing returns.

Figure 5(b) demonstrates how increasing the number of Bernoulli trials affects cache behavior. It is clear that increasing the number of Bernoulli trials results in a more aggressive filter, reducing the number of blocks promoted into the cache, thus reducing the number of references serviced from the cache. Interestingly enough, the figure shows that $N = 2$ is a saturation point for the instruction cache, and reducing the number of mandated Bernoulli successes only increases the number of blocks promoted but not the cache’s hit-rate. This result indicates efficient filtering of the instruction stream requires a more aggressive filter: either using $N \geq 2$ or $P \leq 0.05$. The exact opposite can be said about the data stream: increasing the number of trials reduces both the number of references serviced by the cache and the number of blocks promoted into it, indicating that using $P = 0.5$ and $N \geq 2$ may be too aggressive.

### 6.2 Impact on Miss-Rate

Next, we address the effects of filtering on the overall miss-rate in order to determine which combination of filter parameters yields the best cache performance. Figure 6 shows the distributions of the miss-rate achieved by a filtered 16K, DM cache (cache and filter misses) compared to that achieved by a regular 16K direct-mapped cache, for various combinations of success probability and number of Bernoulli trials. Lower values are better, as they indicate a bigger decrease in the miss rate. The data shown for each combination are a summary of the observed change in miss rate over all benchmarks simulated: the distribution’s middle
Our first observation is that when more trials are used, the best performance (biggest decrease in miss rate) is achieved with higher success probabilities for each trial. The reason is that with a single trial we cannot afford high probabilities, as they will lead to too many false positives; with multiple trials, we need a higher success probability to prevent too many false negatives.

Somewhat surprisingly, using a single trial seems to be the best choice. The first reason is that increasing the number of trials results in more disperse distributions (this is more obvious for the instruction stream, but is also apparent in the data stream). For example, combinations $N = 1, P = 0.2$ and $N = 3, P = 0.5$ achieve similar averages and medians for the data stream, but the first combination has a denser distribution: min–max difference of 76% vs. 81%, and 25–75 percentile range of 19% vs. 25%, respectively.

A second reason, at least for the instruction stream, is that less trials actually achieve better results: the biggest median decrease is $\sim 83\%$ for $N = 1$, $\sim 92\%$ for $N = 2$, and $\sim 99\%$ for $N = 3$, with a similar trend for the averages. For the data stream, however, the differences are much smaller.

A third reason is not related to cache performance but to implementation cost: using a single trial saves
Table 2: Recommended filter parameters for our two main cache configurations.

<table>
<thead>
<tr>
<th>Cache</th>
<th>Data</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$P$</td>
<td>$N$</td>
</tr>
<tr>
<td>16K, DM</td>
<td>0.20</td>
<td>1</td>
</tr>
<tr>
<td>32K, DM</td>
<td>0.30</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 7: IPC improvement for DM filtered caches (with 4K filter) over similar sized 4-way caches.

the need to maintain data regarding previous trials, which simplifies the implementation and saves energy.

Given the choice of $N = 1$, we still need to select the optimal $P$. In all cases, we find that there are several values that achieve essentially the same miss-rates. Thus the best configurations are not singular points capturing some workload eccentricity, but rather moderate extrema points manifesting a real parametric trend. Using the best median improvement as a guide coupled with the lowest possible dispersion leads to the configurations summarized in Table 2. The table also shows parameters for a 32K direct-mapped filtered cache, and the period if using periodic sampling in addition to the random sampling parameters.

The most striking result in this choice of parameters is that the data and instruction stream require different Bernoulli success probabilities — with an order of magnitude difference! The reason for this is the fact that the instruction memory blocks are usually accessed over an order of magnitude more times compared to data blocks. In the benchmarks shown in Figure 50% of the data memory blocks are accessed 1–2 times while in the cache, whereas the same percentile of instruction blocks are accessed 10–15 times. Obviously, this difference is mainly attributed to the fact the instruction memory blocks are mostly read sequentially as blocks of instructions.

7 Impact on Power and Performance

A filtered cache combines two different mechanisms to improve performance — direct-mapped caches offer lower access latency, and block filtering reduces the overall cache miss rate. Furthermore, as direct-mapped caches consume less power than set-associative ones, such a combination can potentially offer reduced power consumption by the L1 caches, as well as improved performance. Using the SimpleScalar toolset [2] we have compared the performance achieved by direct-mapped filtered caches against various set-associative caches. In this experiment, the hit latency incurred by the direct-mapped L1 cache was set to 1 cycle for a cache hit and 3 cycles for a filter hit. The hit latency incurred by set-associative caches was set to 2 cycles.
Figure 7 shows the IPC improvement achieved by a filtered direct-mapped cache compared to a similar size 4-way associative cache, for the SPEC 2000 benchmarks. The figure shows consistent improvements (up to ~46% for a 16K filtered cache and ~32% for 32K filtered cache), with an average overall IPC improvement of almost 12% for 16K and ~10% for 32K. While the results are consistent, it is clear that benchmarks suffering from collision misses — most notably *crafty* and *apsi* — enjoy better performance gains. This claim is supported by the fact that doubling the cache size from 16K to 32K, which is less susceptible to collisions, decreases the performance gains for these benchmarks, while the gain experienced by other benchmarks remain largely unchanged.

Figure 8 compares the average performance of 16K and 32K direct-mapped filtered caches relative to other, more conventional cache structures. It shows that a direct-mapped filtered cache achieves significantly better performance than a similar size set-associative cache. Moreover, a filtered cache can even gain better overall performance than larger, more expensive caches. For example, the IPC of a 16K-DM filtered cache is more than 4% higher than that of a 32K-4way cache, and more than 2% higher than a 32K fully associative cache; a 32K-DM filtered cache is more than 6% and 5% higher than 64K-4way and 64K-FA caches, respectively (and note that the 4-way and FA caches were simulated with parallel tag lookup and data retrieval, whereas the filter is serial). Likewise, using the extra 4K for a filter yields better performance than using them as a victim buffer, indicating that even such a relatively large victim buffer may be swamped by transient blocks. Obviously, part of the improvement is attributed to the lower latency of the direct-mapped cache. However, our experiments show that even a 2-cycle direct-mapped cache still outperforms a set-associative cache, while maintaining its full energetic advantage.

Interestingly, the IPC improvement is similar when comparing the 16K-DM filtered cache to both a regular 16K-DM cache and a 16K-4way set-associative cache, indicating similar performance achieved by the latter two (0.36% in favor of the direct-mapped cache). For the larger 32K and 64K caches the direct-mapped prevails. This is caused by the fact that using larger IL1 caches yields extremely small miss-rates (~0.3% for 32K-DM and ~0.08% for 64K-DM). With such a small miss-rate, the reduced latency of direct-mapped caches allows the 8-wide out-of-order machine — that is very dependent on its instructions supply — to enjoy a better instructions supply rate, thus counteracting the miss-rate degradation in the DL1 cache.

Next, we compare the power consumption of the direct-mapped filtered cache with that of the other configurations. As periodic sampling yields similar performance improvement as random sampling, we use it in our implementation as its overhead is negligible both in terms of circuitry and in terms of power.
Table 3: Dynamic read energy consumed by a single read operation for common cache structures configured with 1 R/W port, 1 exclusive read port, and using a 70nm process. Estimates are based on CACTI 4.2 [29].

<table>
<thead>
<tr>
<th>Cache</th>
<th>Dyn. read Energy</th>
<th>Dyn. write energy</th>
<th>Leakage power</th>
</tr>
</thead>
<tbody>
<tr>
<td>16K, DM</td>
<td>17.2 pJ</td>
<td>2.8 pJ</td>
<td>120.2 mW</td>
</tr>
<tr>
<td>16K, 4way</td>
<td>63.6 pJ</td>
<td>4.0 pJ</td>
<td>135.7 mW</td>
</tr>
<tr>
<td>16K, FA</td>
<td>283.7 pJ</td>
<td>55.2 pJ</td>
<td>603.5 mW</td>
</tr>
<tr>
<td>32K, DM</td>
<td>27.9 pJ</td>
<td>4.0 pJ</td>
<td>233.8 mW</td>
</tr>
<tr>
<td>32K, 4way</td>
<td>97.8 pJ</td>
<td>5.5 pJ</td>
<td>252.9 mW</td>
</tr>
<tr>
<td>32K, FA</td>
<td>522.9 pJ</td>
<td>98.7 pJ</td>
<td>1181.0 mW</td>
</tr>
<tr>
<td>64K, DM</td>
<td>31.6 pJ</td>
<td>5.3 pJ</td>
<td>467.4 mW</td>
</tr>
<tr>
<td>64K, 4way</td>
<td>113.9 pJ</td>
<td>8.8 pJ</td>
<td>490.6 mW</td>
</tr>
<tr>
<td>64K, FA</td>
<td>1033.0 pJ</td>
<td>204.3 pJ</td>
<td>2370.0 mW</td>
</tr>
<tr>
<td>4K, FA</td>
<td>42.9 pJ (tag lookup 4.6 pJ)</td>
<td>30.5 pJ</td>
<td>19.3 mW</td>
</tr>
</tbody>
</table>

Figure 9 shows both the dynamic read energy and leakage power consumed by the direct-mapped filtered cache, compared to other cache configurations (same as those in Figure 8). Obviously, the power consumed by the filtered cache is higher than that of a simple direct-mapped cache, because of the fully-associative filter: ~50%-70% more dynamic energy and ~10% more leakage power for a 16K filtered cache (but much less for a 32K cache). However, when comparing a filtered direct-mapped cache to the more common 4-way
associative cache of a similar size, the 16K-DM filtered cache design consumes almost 60% less dynamic energy, with similar leakage power consumption, and the 32K version even does a little better. Furthermore, a filtered cache consumes \(~70\%\) less dynamic energy and \(~50\%\) less leakage power than a double size, 4-way associative regular cache, at the same time providing 4–6\% improvement in IPC.

Admitted, the dynamic energy consumed by a filtered cache on a miss is likely to increase as it involves writing the fetched data to the fully-associative filter rather than to a set-associative cache. However, this increase is likely to be minor (according to Table 3) relative to the dynamic read energy spent reading the data from the L2 caches or even the main memory. Furthermore, reading from more distant caches involves more routing energy and handling in bus contention, especially for modern, multi-core, shared cache architectures — further reducing the relative L1 overhead.

In summary, we show that direct-mapped filtered caches offer performance superior to that of same and double sized set-associative caches, while consuming much less power — both dynamic and static. With the ubiquitous use of set-associative caches for L1 caches in modern, general purpose processors [12, 1, 11] we believe these results can potentially contribute to general processor design and implementation.

8 Related Work

Early auxiliary structures designed to improve L1 cache performance are the victim cache and stream buffers suggested by Jouppi [17]. A similar structure has even been included in a commercial microprocessor: the assist cache of the HP PA 7220 CPU [4]. The function of this assist cache is to compensate for the fact that the main cache is direct mapped, thus making it vulnerable to address conflicts. Its size (64 lines of 32 bytes, fully associative) serves as a guideline for what can be implemented in practice.

The observation that memory access patterns may display different types of locality, and that these may warrant different types of caching policies, has been made before and motivated studies that tried to identify the frequently used blocks.

Tyson et al. show that a small fraction of memory access instructions have a disproportionally large effect on the miss rate, as they generate the majority of misses [30]. They therefore suggest that these instructions be identified and marked so that they will not be cached at all. Their conclusions are that this
can significantly reduce the memory bus traffic.

González at al. suggest that the cache be partitioned into two parts, one each for handling data that exhibit spatial and temporal locality [8]. This is managed by a locality prediction table, which stores data about the history of the most recently executed load/store instructions. The predictions attempt to classify accesses as scalar or vector, in which case they also attempt to identify the stride and vector size.

The work of Sahuquillo and Pont involves a filter used to optimize the hit ratio of the cache [25]. Their design also focuses on identifying the most heavily used items, and the filter itself (not the cache) is used to store those items that have been accessed the most times. This comes at a price of having to maintain an access counter for each item. Moreover, their main motivation is to reduce the bus utilization in multiprocessor systems. A similar mechanism is proposed by Rivers and Davidson, who also base the caching on an access count [22].

Karlsson and Hagersten use a slightly different method by using the filter to audit whether a block would have been replaced before its next access [18]. If the next access is close enough, the block is promoted to the cache. This mechanism requires keeping a last-accessed-timestamp for every block in the cache, and comparing it on every replacement.

Kin et al. also use filtering before the L1 cache [20], in an attempt to reduce energy consumption, while incurring some performance loss. The idea is that the filter should catch most of the memory references by acting as an L0 cache, allowing the L1 cache to remain in a low-power standby mode. However, this power saving comes at the cost of a certain performance degradation, because the L1 cache is only accessed after the filter fails, rather than in parallel to accessing the filter. In a followup work by Memik and Mangione-Smith, the filter is placed in front of the L2 cache [21].

The same principle has also been applied to trace caches, either trying to filter out infrequently used traces, or simply avoid generating them in the first place. Rosner et al. used a fully associative filter to only insert frequently used traces into the trace cache [24]. In their paper, the authors explored several filtering techniques which rely on past block usage to predict whether it would be beneficial to promote it into the trace cache. Behar et al. improved this design using a probabilistic filter [3]. Based on the observation that most of the execution time is spent executing a small number of traces (the proverbial 90/10 rule for instructions traces [9]) the authors have sampled every Nth observed trace, thus reducing the power spent on generating infrequently used traces. The 90/10 effect described by the authors only demonstrates that the mass-count disparity is also common in trace generation.

Johnson and Hwu grouped contiguous memory blocks whose cache behavior is expected to be consistent into macro-blocks, and kept track of the macro-blocks’ reuse frequency using a special Memory Access Table (MAT) [16]. All blocks are inserted into a bypass buffer, and are only allowed in the cache if the victim block was accessed less frequency than the new candidate block. This approach is quite costly in hardware, requiring both a bypass buffer and a second cache structure to implement the MAT. Jalminger and Stenström tried to achieve similar goals using a structure based on a two level branch predictor [14], and also show a reduction in L1 miss-rate.

All the structures described in the above studies require maintaining per-block information, thus complicating the filtering hardware. Furthermore, none except victim demonstrate an effective enough filtering technique enabling the use of a direct-mapped cache as the main structure. In contradistinction, our filtering algorithm is purely probabilistic and does not require any state per block other than its mere presence in either the cache or the filter, demonstrating an efficient use of a fast, low-power, direct-mapped structure in the L1 caches enabling both performance improvement and reduction of power consumed.
9 Conclusions

In this paper we have explored the skewed nature of memory references, in which the vast majority of references are serviced by a very small fraction of memory blocks, and the vast majority of memory blocks service only a small fraction of the references — a phenomenon called mass-count disparity. Based on this phenomenon we have designed a probabilistic filter which uses the reference distribution to split the block distribution into its two components — frequently used blocks that should be served from the cache, and transient blocks which should be prevented from polluting the cache and causing conflict misses.

The design of the probabilistic filter is simple: the L1 cache is preceded by a small fully-associative filter, into which all blocks are inserted. Using a simple Bernoulli trial we sample memory references and mark their designated blocks. A block that has been marked a predefined number of times is then promoted from the filter into the cache. This design is so effective that it enables the use of fast, low-power, direct-mapped caches to serve the frequently used blocks, dramatically reducing the latency and overall power consumed by the L1 caches.

The probabilistic filter has two parameters: $P$, the success probability of each Bernoulli trial, and $N$, the number of successful trials a block is required to pass before it is confirmed as a frequently used block. After examining the design space of these two parameters we have found that using constant $P$ and $N$ values for a specific cache configuration is very effective for most benchmarks, with no need for adaptive tuning. For example, our simulations show that when using a 16K direct-mapped cache and a 4K fully-associative filter, the values $P = 0.2$, $N = 1$ and $P = 0.01$, $N = 1$ are the best choices for the data and instruction streams, respectively. It is important to note that using $N = 1$ eliminates the need to add state bits to the filter, further simplifying it.

Using the Bernoulli filter we were able to effectively utilize a 16K direct-mapped structure for both L1 caches yielding up to $\sim$46% improvement in IPC, with an average of $\sim$12% over all benchmarks — better than a double size, 4-way set-associative conventional cache! Furthermore, this design dramatically reduces the power consumption of the L1 caches, reducing the dynamic consumption by $\sim$60%--70% for 16K and 32K caches, and $\sim$50% less leakage power compared a performance-wise inferior double sized cache.

The only overhead incurred by the Bernoulli filter is the power and die-area consumed by the filter structure itself — a 4K fully-associative cache. However, it is shown that despite the power overhead the overall power consumption is reduced thanks to the use of a low-power direct-mapped cache as the main structure. Moreover, recent advances in Silicon feature sizes has increased the number of transistors per die, making the die-area overhead negligible.

In future work, we intend to explore the effectiveness of probabilistic filtering for L2 caches, for which the filter cannot be used as-is since the L2 caches are oblivious to most of the reference stream preventing them from experiencing the mass-count disparity to the same degree.

References


