L1 Cache Filtering Through Random Selection of Memory References

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 fully-associative auxiliary cache structure. By inserting only frequently used blocks into the main cache structure, we can reduce the number of conflict 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 use a simple probabilistic filtering mechanism that uses random sampling to identify and select the frequently used blocks. Furthermore, by using a small direct-mapped lookup table to cache the most recently accessed blocks in the auxiliary cache, we eliminate the vast majority of the costly fully-associative lookups. Finally, we show that a 16K direct-mapped L1 cache, augmented with a fully-associative 4K filter, achieves on average 13% more instructions per cycle than a regular 16K, 4-way set-associative cache, and even ∼7% more IPC than a 32K, 4-way cache, while consuming 70%-80% 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, also increased the power consumed by the caches — estimated at more than 10% of the overall power consumed by a general purpose CPU [10], and up to 40% for embedded systems [3].

Today, as processor power consumption is also becoming a major concern, the power-performance tradeoff is ever more important [19]. This trend motivates researchers to design more efficient caches, that can deliver the required performance while maintaining the power budget.

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 [9]. However, they are more susceptible to conflict 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* [6]. Based on this observation, we design a simple random sampling L1 filtered cache that uses a simple coin toss to preferentially insert frequently used blocks into the cache proper, thus reducing the number of conflict 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 [11, 13, 14, 15, 16, 17, 20, 21, 25], 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 introducing the concept of random sampling of memory references (Sect. 3), we present our design for a random sampling L1 cache in Sect. 4. Following the description of our methodology in Sect. 5, we explore the effects of random sampling on the reference stream (Sect. 6), and the overall impact on power and performance (Sect. 7). Finally, we review previous work related to this study (Sect. 8), and conclude in Sect. 9.

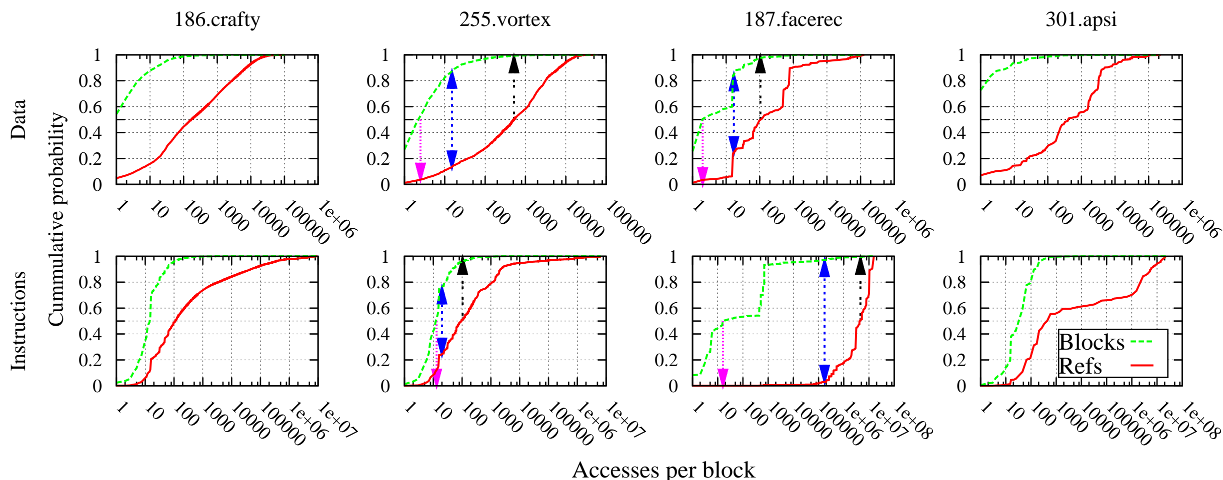

Figure 1. *Mass-count disparity plots for memory accesses in select SPEC2000 benchmarks (using the* ref *input set), superimposing the distributions of residency lengths and the number of references serviced by residencies of each length, for 16K direct-mapped L1 caches. (data stream on the top plots, instruction streams on the bottom). The mass-count metrics are demonstrated on the vortex and facerec plots: 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.*

2 On The Skewed Distributions of Memory Access Patterns

Locality of reference is one of the best-known phenomena in computer workloads [5]. Temporal locality in particular occurs because of two properties of reference streams: that some addresses are much more popular than others, and that accesses are batched rather than being random [12]. Importantly, references to blocks that are seldom accessed are also grouped together; we call such blocks *transient*.

A good way to visualize skewed popularity is by using mass-count disparity plots [6]. 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 between a single insertion of a block into the cache, and its corresponding eviction — denoted as the *cache residency length*. 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 distinct cache residencies spanning 100 and 200 references, respectively, rather than a single block with 300 references.

Fig. 1, shows the distributions of residency lengths and that of the references serviced by each residency length for 4 select SPEC2000 benchmarks using a 16K direct-mapped cache. The figure shows the distributions of both data and instruction streams.

The mass-count disparity refers to the fact that the distributions of residency lengths (count) and the number of *references* serviced by each residency length (mass) are quite distinct, as shown in Fig. 1. The divergence between the distributions can be quantified by the joint ratio [6], which is a generalization of the proverbial 20/80 principle: This is the unique point in the graphs where 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 residencies, and more specifically the longest ones, get a full 87% of the references, whereas the remaining 87% of the residencies get only 13% of the references. Thus a typical *residency* is

only referenced a rather small number of times (up to about 10), whereas a typical *reference* is directed at a long cache residency (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 [6]. The $W_{1/2}$ metric assesses the combined weight of the half of the residencies that receive few references. For vortex, these 50% of the residencies 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 longer residencies, such as those that together account for 50% of the references. The number of highly-referenced residencies servicing half the references is quantified by the $N_{1/2}$ metric; for vortex it is less than 1% of all residencies (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 [22]. 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 for L1 caches is the focus of this paper.

We focus our investigation on L1 caches, since this is the level at which the mass-count disparity phenomenon is prevalent in its simplest form. More distant caches see a reference stream filtered down by the L1 level, thereby blurring the mass-count disparity.

3 Random Sampling of Memory References

The mass-count disparity implies that a small fraction of all L1 cache residencies service the majority of references, as described in Sect. 2. Servicing these residencies from a fast, low-power, direct-mapped cache, while using an auxiliary buffer for short, transient residencies, can potentially yield both performance and power gains, as the small number of long residencies will minimize the number of conflict misses — to which direct-mapped caches are so susceptible.

We therefore need to design a residency length predictor for identifying the longer residencies that should be inserted into the direct-mapped cache, distinguishing them from shorter ones that should be serviced by an auxiliary buffer, acting as a filter.

The naive approach to designing a residency predictor would simply be to count the number of references made to each block in the cache — i.e. the length of each residency — and classify it as *long* once it passes a certain threshold.

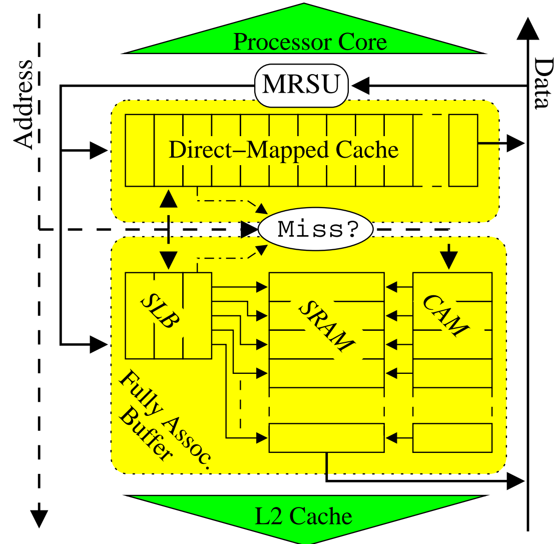

Figure 2. *The design of the random sampling filtered cache.*

However, this naive design is both susceptible to program phases changes, as well as requires maintaining an access counter for each cache line.

The alternative, based on the observations made in the previous section, is to use random sampling. If we sample references uniformly (a Bernoulli trial) with a relatively low probability P, short residencies will have a very low probability of being selected. But given that a single sample is enough to classify a residency as a long one, the probability that a residency is chosen after n references is $1-(1-P)^n$. This converges exponentially to 1 as n increases.

Importantly, implementing such a predictor does not require saving *any* state information for the blocks, since every random selection is independent of its predecessors. The only hardware required is a random number generator — a linear-feedback shift register [26], for example.

This random sampling mechanism serves as the base for our cache design.

4 Designing a Random Sampling Cache

Based on the principles described in the previous section we introduce an L1 cache design that uses Bernoulli trials to distinguish frequently used blocks from transient ones.

The proposed design, based on the dual cache paradigm, is depicted in Fig. 2. It consists of a direct-mapped cache preceded by a small, fully-associative filter. When a memory access occurs, the data is first searched in the cache proper, and only if that misses the filter is searched. If the filter misses as well, the request is sent to the next level

cache. In our experiments we have used 16K and 32K (common L1 sizes) for the direct-mapped cache, and a 4K fully-associative filter (all structures use 64B lines).

Each memory reference that is serviced by the filter or by the next level cache initiates a Bernoulli trial with a predetermined success probability P to decide whether in should be promoted into the cache proper. Note that this enables a block fetched from the next level cache to skip the filter altogether and jump directly into the cache. This decision is made by the *memory reference sampling unit* (MRSU) which performs the Bernoulli trials, and writes the block to the cache if selected. In case the block is not selected, and was not already present in the filter, the MRSU inserts it into the filter. The MRSU can in fact perform the sampling itself even before the data is fetched, enabling it to perform any necessary eviction (either from the cache proper of the filter) beforehand, thus overlapping the two operations. Sect. 6.1 explores the probabilistic design space for a suitable Bernoulli success probability.

For a desired threshold probability P we pre-calculate a constant C_P such that $\frac{C_P}{2^K} \simeq P$. Given a source of random bits, the MRSU generates a random integer r in the range $\left[1 \ldots 2^K\right)$. Therefore, the result of the comparison $r \leq C_P$ yields *true* with probability $\sim P$.

Although such a mechanism is easy to implement (e.g. using a linear-feedback shift register [26]) and consumes negligible power, we also experimented with naive periodic sampling, using a period proportional to $\frac{1}{P}$. This achieved results similar to those of random sampling. We therefore only show the results for random sampling.

To reduce both time and power overheads associated with accessing a fully-associative structure, we have augmented the classic CAM / SRAM design [26] with a *set look-aside buffer* (SLB). The SLB consists of a direct-mapped structure, mapping tags of filter-resident blocks to their location in the fully-associative buffer's SRAM structure. The data contained in the SLB for each tag is a bitmap whose width is similar to the number of lines in the filter — 64 lines for a 4K filter. This allows for each SLB output bit to be directly connected to an SRAM word-line without a decoder, offering a fast, low-power caching of CAM results. In fact, the SLB structure is efficient enough to be accessed in parallel with the cache on every access, eliminating the need for a costly CAM lookup on most filter accesses. If the SLB misses, the CAM is accessed, and the result is fed back to the SLB during the ensuing SRAM access, hiding the SLB update latency. Furthermore, the number of entries in the SLB can be much smaller than the number of filter lines, because temporal locality also exists in the filter. Sect. 6.3 offers an analysis of the SLB performance to determine the number of entries it requires.

5 Methodology

To evaluate the concepts presented in this paper we have used a modified version of the *SimpleScalar* toolset [1]. 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 but rather depends on whether the target block was found in the filter or the cache proper.

We have used the *SPEC2000* benchmarks suite [23] compiled for the *Alpha AXP* architecture. 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.

Power estimates were compiled using *CACTI* 4.1 [24].

6 The Effects of Random Sampling

The number of references to frequently used blocks are numerous, but involve only a relatively small number of distinct blocks. This reduces the number of conflict misses, enabling the use of a low-latency, low-energy, direct-mapped cache structure. On the other hand, transient residencies compose the majority of residencies, but naturally have a shorter cache lifetime. Therefore, they can be served by a smaller, fully-associative (and expensive) structure.

However, aggressive filtering might be counter-productive: if too many blocks are serviced from the filter and not promoted to the cache proper, the filter can become a bottleneck and degrade performance.

This section is therefore dedicated to evaluate the effectiveness of probabilistic filtering, while exploring the statistical design space. The selected parameters are then used to evaluate performance and power consumption in Sect. 7.

6.1 Impact on Miss-Rate

First, we address the effects of filtering on the overall miss-rate in order to determine the Bernoulli probabilities that yields best cache performance. Fig. 3 shows the distributions of the miss-rate achieved by a filtered 16K, DM cache (both cache *and* filter misses) compared to that achieved by a regular 16K direct-mapped cache, for various Bernoulli success probabilities. Lower values are better, indicating decreased 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 range (25%–75%), average, median and min/max values. An ideal combination would yield maximal overall miss-rate reduction with a dense distribution, i.e. a small differences between the 25%–75% percentiles and min–max val-

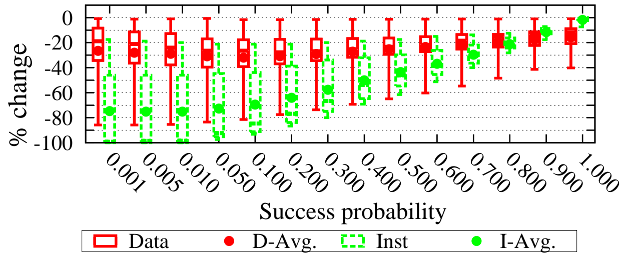

Figure 3. *Comparison of SPEC2000 instruction and data miss-rate distributions, using various sampling probabilities, for a 16K-DM cache. The boxes represent the 25%–75% percentile range, and the whiskers indicate the min/max values. Inside the box are the average (circle) and median (horizontal line).*

ues, as a denser distribution indicates more consistent results over all benchmarks.

The figure shows that the best average reduction in data miss-rate is ∼30%, and is achieved for P values of 0.05 to 0.2. Moreover, this average improvement is not the result of a single benchmark skewing the distribution: when comparing the center of these distributions — the 25%–75% box — we can see the entire distribution is moved downwards. The same can be said about the miss-rate reduction in the instruction stream, for which selection probabilities of 0.001 to 0.01 all achieve an average improvement of ∼75%. In this case as well the best averages are achieved for probabilities that shift the entire distribution downwards.

The fact that the a similar improvement is achieved over a range of probabilities, for both data and instruction, indicate that using a static selection probability is a reasonable choice, especially as it eliminates the need to add a dynamic tuning mechanism.

We therefore chose sampling probabilities of 0.2 and 0.01 for data and instruction stream, respectively, for the 16K cache configuration. In a similar manner, probabilities of 0.3 and 0.005 were selected for the data and instruction streams, respectively, for the 32K configuration.

Interestingly, 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 Fig. 1, 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. This difference is mainly attributed to the fact the instruction memory blocks are mostly read sequentially as blocks of instructions.

6.2 Impact on Reference Distribution

As noted above, random sampling is aimed at splitting the references stream into two components — one consisting of long cache residencies, and another consisting of short transient ones. In this section we conduct a qualitative analysis of the effectiveness of random sampling in splitting the distribution of memory references.

Fig. 4 compares distributions of reference masses — the fraction of references serviced by each residency length — of the filtered 16K cache and the original 16K direct-mapped cache. Results are shown for select SPEC benchmarks with Bernoulli success probabilities of 0.20 for data streams and 0.01 for instruction streams. These probabilities were chosen based on the results described in Sect. 6.1.

Each plot shows three lines: the distributions for the cache and filter for the filtered design, and the distribution for a conventional direct-mapped cache — which is the combination of the first two (this is the same distribution as the one shown in Fig. 1). The median value of each distribution is marked with a down pointing arrow. Invariably, the distributions show that the majority of references directed at the filter, are serviced by residencies much shorter than those serving the majority of the references directed at the cache proper.

To estimate the difference between the two resulting distributions we used two intuitive metrics: median ratio (marked with a horizontal double arrow) and false-$*$ equilibrium (marked with a vertical double arrow).

The first metric is the ratio between the median values of the cache and the filter: $ratio = \frac{median_c}{median_f}$. This metric is used to quantify the distinction between the two distributions, thereby evaluating the effectiveness of random selection to distinguish shorter residencies — which should stay in the filter — from longer ones that should be promoted into the cache proper.

In the benchmarks shown, the median ratios range from 100 to 10000, with the only exception of *facerec*'s instruction stream where the ratio is 2.5. In fact, the average medians ratio for all data streams is ∼440, with a ratio of ∼1440 for the instruction streams — indicating a clear distinction between residency lengths in the cache and the filter.

The second metric is denoted as the *false-$*$ equilibrium*, and is an estimate of false predictions: Any given residency length threshold we choose in hindsight will show up on the plot as a vertical line, with a fraction of the cache's distribution to its left indicating the false-positives (short residencies promoted to the cache), and a fraction of the filter's distribution to its right indicating the false-negatives (long residencies remaining in the filter). Obviously, choosing another threshold will either increases the fraction of false-positives *and* decrease the fraction of false-negatives (or vice versa). The false-$*$ equilibrium is a unique threshold

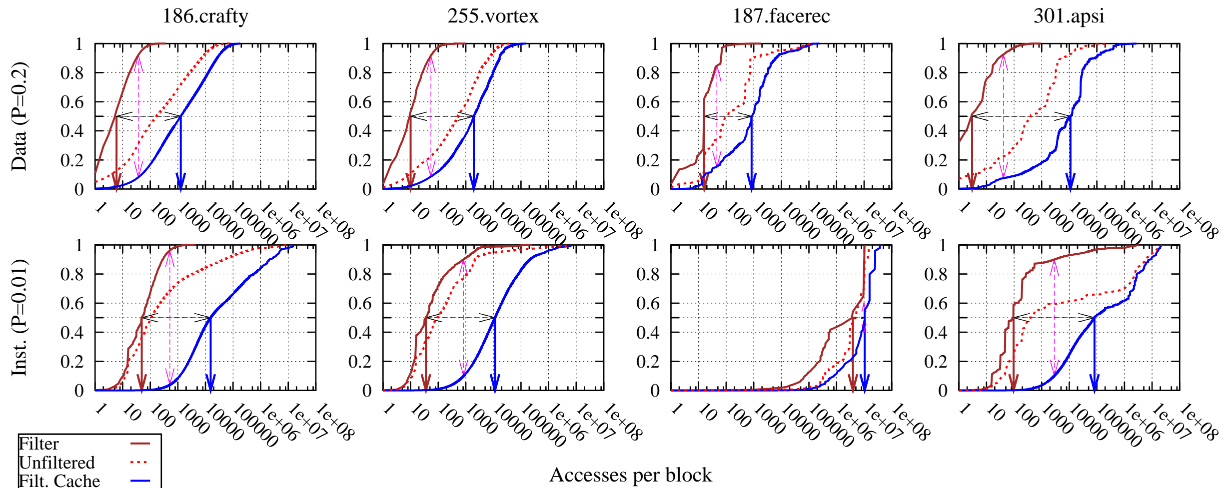

Figure 4. *Comparison of the data references' mass distributions in the filtered cache structure and the regular cache structure for select SPEC benchmarks using the ref input, for both data (top) and instruction (bottom). The horizontal double arrows show the median-to-median range, and the vertical double arrows show the false-∗ equilibrium point.*

that if chosen, generates equal percentages of false-positives and false-negatives, thereby serving as an upper bound for overall percentage of false predictions.

For example, if we examine *vortex*'s data stream we see that the false-* equilibrium point stands at a residency length of ∼50 and generates ∼8% false predictions (∼10% for the instruction stream). The false prediction rate for *facerec*'s data stream was found to be ∼15%, with a very poor ∼40% false prediction for the instruction stream. The overall average percentage of false predictions for the data streams was found to be ∼14%, with ∼17% for the instruction streams — a fairly good upper bound considering it is based on true random sampling.

Another aspect of the reference distributions is the number of references accounting each distribution, compared with the number of residencies served by the cache and the filter. Fig. 5 shows the percentage of references serviced by the cache, compared with the percentage of blocks promoted into the cache, for various probabilities. Considering the mass-count disparity we expect that promoting frequently accessed *blocks* into the cache will result in a substantial increase in the number of *references* it will service, and that promoting not-so-frequently used blocks have a smaller impact on the number references serviced by the cache. This is indeed evident in Fig. 5: 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 hardly increases the cache's hit-rate. In our case this saturation occurs at $P = 0.6$ for the data and $P = 0.4$ for the instructions. Beyond these probabilities the promoted blocks are mostly transient blocks and we start experiencing diminishing returns.

In summary, we see that random sampling is very effective in splitting the distribution of references into two distinct components — one composed mainly of frequently used blocks, and the other composed mainly of transient blocks.

6.3 The Set Look-aside Buffer

A fully-associative filter introduces longer access latencies and increased power consumption. We therefore suggest a *set look-aside buffer* (SLB) to cache recent lookup results. The SLB is a small direct-mapped cache structure mapping block tags directly to the filter's SRAM based data store, thus avoiding the majority of the costly CAM lookups, while still maintain fully-associative semantics. This section explores the SLB design space.

Fig. 6 shows the stack depths distributions of filter accesses, for the different SPEC benchmarks, as well as the average distribution over all benchmarks (the various benchmarks are not individually marked as only the clustering of distributions matters in this context). Clearly, the vast majority of accesses are to recently used blocks — in fact, on average over 90% of accesses are to stack depths of 8 or less, out of a total of 64 lines in the filter. In our experiments, we have found that using an 8 entry SLB achieves an average of ∼78% hit-rate for the data stream (∼82% median) and over 98% for the instruction stream (∼99% median) for a 4K filter. Doubling the SLB size to 16 entries

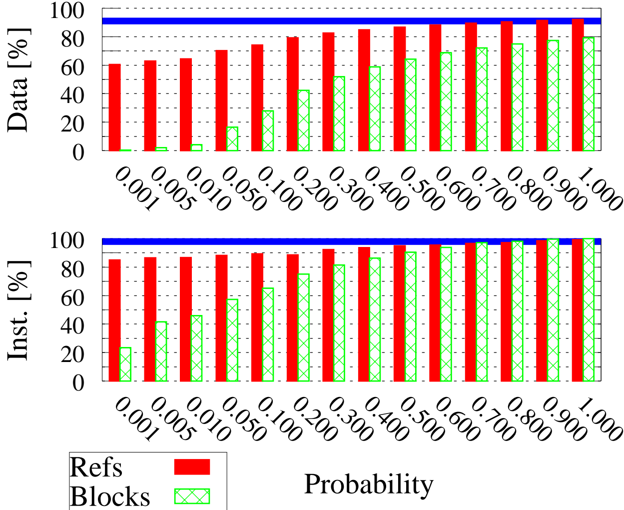

Figure 5. *Percent of references serviced by the cache vs. the percent of blocks transferred from the filter into the cache for varying sampling probabilities (averages over all benchmarks). The horizontal lines near the top figure indicate the asymptotic maximum of references serviced by the cache. Cache size was 16K, with a 4K filter.*

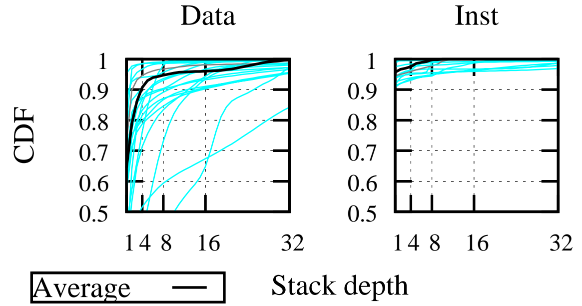

Figure 6. *Distributions of filter access depth for all SPEC2000 benchmarks, and the average distribution. Note that the vast majority of accesses are focused around the MRU position (the specific behavior for each benchmarks is irrelevant in this context, and are thus not individually marked).*

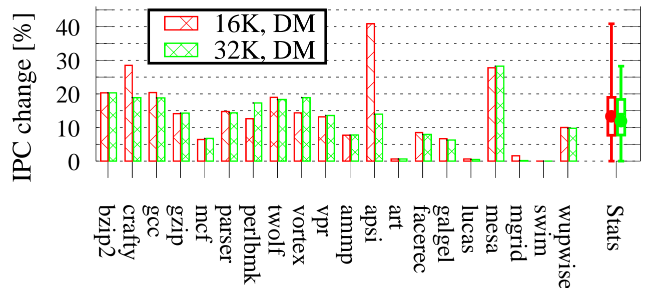

Figure 7. *IPC improvement for DM random sampling caches (4K filt.) over similar size 4-way caches.*

only improved the average data hit-rate to ∼84% (∼88% median) and ∼99% for the instruction stream, but increased the dynamic power consumption by ∼10% and the leakage by ∼50% (with similar results for the 32K configuration).

We have therefore used an 8 entry SLB in our power and performance evaluation, eliminating almost 80% of the costly filter CAM lookups.

7 Impact on Power and Performance

The reduced miss-rate achieved by the random sampling design, combined with a low-latency, low-power, direct-mapped cache potentially offers both improved performance and reduced power consumption. Augmenting the fully-associative filter with an SLB can reduce the overheads incurred by the filter, further improving efficiency.

Using the SimpleScalar toolset [1] for out-of-order simulations we have compared the performance achieved by direct-mapped filtered caches against various set-associative caches. Our micro-architecture consisted of a 4-wide superscalar design, whose parameters are listed in Table 1. 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. If the request block is found in the SLB then no CAM lookup is necessary — enabling direct SRAM access and a total 2 cycles filter latency. The hit latency incurred by set-associative caches was set to 2 cycles. For fully-associative caches we used an unrealistically fast 2 cycle latency — same as set-associative — placing both on a similar baseline thus focusing on the reduced miss-rates achieved by fully-associative caches.

Fig. 7 shows the IPC improvement achieved by a random sampling cache over a similar size 4-way associative cache, for the SPEC benchmarks. The figure shows consistent improvements (up to ∼40% for a 16K random sampling cache and ∼28% for 32K one), with an average overall IPC improvement of ∼13% for a 16K random sampling cache and ∼12% for a 32K cache). While the results are consistent, it is clear that benchmarks suffering from conflict misses enjoy better performance gains. This is most pronounced for *crafty* and *apsi* that include a large portion of short residencies (Fig. 1). Supporting this is the fact that doubling the cache size to 32K — thus reducing conflicts by increasing the number of sets — decreases the performance gains for these benchmarks, while other benchmarks remain largely unaffected.

Fig. 8 compares the average performance achieved with

IL1/DL1 cache		micro-architecture	
size	16/32 K	fetch / issue / decode	4
line size	64 B	functional units	4
assoc.	DM	window size	128
latency	1 cy.*	Load/Store queue	64
filter		**branch predictor**	
entries	64	meta-predictor with	
assoc.	full	64K-entry bimodal and	
latency	3 cy.	gshare, and a similar	
CAM lat.	2 cy.	size meta table. 4K BTB	
SRAM lat.	1 cy.	**L2 cache**	
SLB lat.	1 cy.	design	unified
SLB entries	8	size	512K
SLB line	64 b	line size	64B
memory		assoc.	8
latency	350 cy.	latency	16cy.

Bernoulli probabilities		
Size	Data	Instruction
16K	$P = 0.2$	$P = 0.01$
32K	$P = 0.3$	$P = 0.005$

* L1 latency is 2 cycles for set-associative and fully-associative caches

Table 1. *micro-architecture and cache configurations used in the out-of-order simulations.*

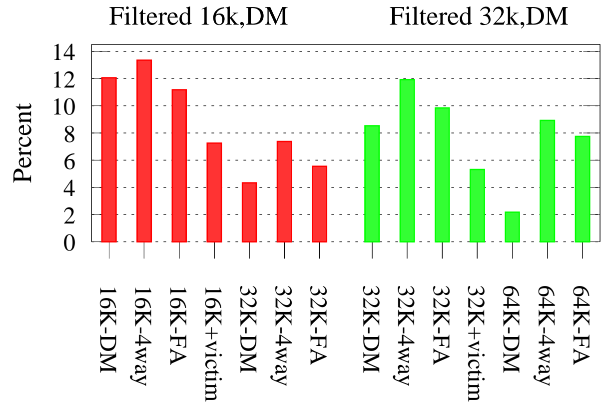

Figure 8. *Average IPC improvement for 16K and 32K direct-mapped filtered caches over common cache configurations.*

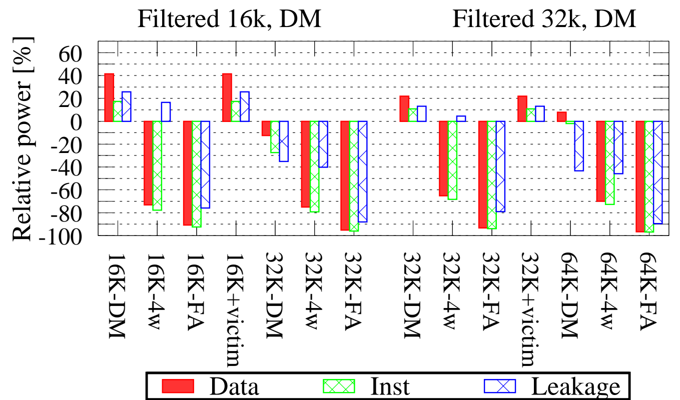

Figure 9. *Relative power consumption of the random sampling cache, compared to common cache designs (lower is better), for a 70nm process.*

16K and 32K random sampling caches to that of common cache structures. It shows that a direct-mapped random sampling filtered cache achieves significantly better performance than a similar size set-associative cache. Moreover, a random sampling cache can even gain better overall performance than larger, more expensive caches. For example, the IPC of a 16K-DM random sampling cache is more than 7% higher than that of a 32K-4way cache, and more than 5% higher than a 32K fully associative cache; a 32K-DM random sampling cache is more than 9% and 7% higher than 64K-4way and 64K-FA caches, respectively. 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.

Interestingly, the IPC improvement is similar when comparing the 16K-DM random sampling cache to both a regular 16K-DM cache and a 16K-4way set-associative cache, indicating similar performance achieved by the latter two. The reason for this similarity is that while the direct-mapped cache suffers from a higher miss-rate compared to the 4-way set-associative cache, it compensates with its lower access latency. This is even more evident when considering the larger 32K and 64K caches, where the direct-mapped configuration takes the lead. When doubling the cache size from the 32K to 64K the number of cache sets doubles, thus reducing the number of conflicts and allowing the direct-mapped cache's lower latency to prevail.

Next, we compare the power consumption of the random sampling cache with that of the other configurations. Using independent random sampling eliminates the need to maintain any previous reuse information, reducing the power consumption calculation to averaging the energies consumed by the combination of a direct-mapped cache, a fully-associative filter, and a small, direct-mapped SLB. All power consumption estimates are based on the CACTI 4.1 power model [24].

The average dynamic energy consumption is simply aggregate energy — the sum of $numberofaccesses \times accessenergy$ for each component — divided by the overall number of hits. Even simpler, the leakage power consumed by the random sampling cache is the sum of leakage power consumed by all components.

Fig. 9 shows both dynamic read energy and leakage

power consumed by the random sampling cache, compared to common cache configurations (same as those in Fig. 8). Obviously, the power consumed by the random sampling cache is higher than that of a simple direct-mapped cache, because of the fully-associative filter: $\sim$20%-40% more dynamic energy and $\sim$25% more leakage power for a 16K random sampling cache (and just over half that for a 32K cache). However, when comparing a random sampling cache to a more common 4-way associative cache of a similar size, the 16K random sampling cache design consumes almost 80% less dynamic energy, with only 15% more leakage power. The 32K configuration yields 60%-70% reduction in dynamic energy, at only 5% increase in leakage.

However, the main contribution of a random sampling cache is apparent when compared to a set-associative cache double its size: both the 16K and 32K random sampling caches consume 60%-70% less dynamic energy, and 40%-50% less leakage than 32K and 64K 4-way set-associative caches, respectively — while yielding better performance as shown in Fig. 8.

In summary, this section shows that a random sampling direct-mapped cache offers performance superior to that of a double sized set-associative cache, while consuming considerably less power — both dynamic and static.

8 Related Work

Early auxiliary structures designed to improve L1 cache performance are the victim cache and stream buffers suggested by Jouppi [14]. 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, possibly warranting different types of caching policies, has already motivated studies that tried to identify the frequently used blocks.

Tyson et al. show that a small fraction of memory access instructions generate the majority of misses [25]. They therefore propose to avoid caching memory locations when accessed by these instructions.

González at al. suggest that the cache be partitioned into two parts, one each for handling data that exhibit spatial and temporal locality [7]. Based on previous reuse information, their predictor classifies memory accesses to either scalars (temporal locality) or vectors (spatial locality).

The work of Sahuquillo and Pont involves a filter used to optimize the hit ratio of the cache [21]. The authors associate a reference counter with each cache line promoting the most heavily accessed lines into a small L0 cache. A similar mechanism is proposed by Rivers and Davidson, who also base the caching on a reference count [18].

Kin et al. also used an L0 design and maintained the L1 in low-power mode thus reducing energy consumption [16]. The power reduction is in fact traded off for performance as the L1 has to be re-powered on every access. In a followup work by Memik and Mangione-Smith, the filter is placed in front of the L2 cache [17].

Karlsson and Hagersten use a filter to audit whether a block would have been replaced before its next access [15]. If the reuse distance is short 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.

The same principle has also been applied to trace caches, either to filter out infrequently used traces, or to avoid generating them in the first place. Rosner et al. explored several trace filtering techniques which rely on past block usage to predict whether it would be beneficial to promote a trace from a fully-associative filter into the trace cache proper [20]. Behar et al. reduced power spent on trace generation by using periodic trace sampling [2]. This was based on the observation that the majority of execution time is spent executing a small number of traces (the proverbial 90/10 rule for instructions traces [8]). The 90/10 effect described by the authors only demonstrates that the mass-count disparity is also common in trace generation.

Johnson and Hwu used a bypass buffer for all blocks only allowing most frequently used blocks into the cache proper [13]. A Memory Access Table (MAT) is used to group contiguous memory blocks experiencing similar cache behavior. This is quite costly in hardware as it requires maintaining access frequency information for all macro-blocks. Jalminger and Stenström used a two level branch predictor design to achieve similar goals [11].

All the structures described above require maintaining reuse 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, the random sampling cache is purely probabilistic and does not require any per-block information other than its mere presence in either the cache or the filter. It demonstrates 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 memory *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 present a random sampling design, that uses the *reference* distribution to split the *block* distribution into its two components — frequently used blocks that should be served from a fast, low-power, direct-mapped cache, and transient blocks that should be served by a fully-associative filter, thus preventing them from polluting the cache and causing conflict misses.

Filtering is done by performing a simple Bernoulli trial on each memory reference, and promoting the accessed block into the cache proper if the trial succeeds. The probability for a block to be selected for promotion therefore grows exponentially with each access.

After examining the design space we have found that using a constant Bernoulli success probability P per specific cache configuration is very effective for most benchmarks, with no need for adaptive tuning. For example, when using a 16K direct-mapped cache and a 4K filter, the values $P = 0.2$ and $P = 0.01$ are found to be best choices for the data and instruction streams, respectively.

To reduce the added overheads of using a fully-associative buffer, we show that most fully-associative CAM lookups can be avoided by using a direct mapped *set look-aside buffer* (SLB) that caches recent fully-associative lookups. An SLB consisting of only 8 entries was sufficient to avoid $\sim$80% of the lookups for a 64 entry CAM.

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. Moreover, our L1 design dramatically reduce the overall power consumption — both 16K and 32K shown to perform better than 32K and 64K caches, respectively, while reducing the dynamic consumption by $\sim$60%–70% with over 40% reduction in leakage power. With the ubiquitous use of set-associative L1 caches in modern processors we believe these results can contribute to future processor design and implementation.

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

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[2] M. Behar, A. Mendelson, and A. Kolodny. Trace cache sampling filter. *Intl. Conf. on Parallel Arch. and Compilation Techniques*, pp. 255–266, 2005.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Intl. Symp. on Computer Architecture*, pp. 83–94, 2000.

[4] K. K. Chan, C. C. Hay, J. R. Keller, G. P. Kurpanek, F. X. Schumacher, , and J. Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal*, 47(1), Feb 1996.

[5] P. J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, Jul 2005.

[6] D. G. Feitelson. Metrics for mass-count disparity. In *Modeling, Anal. & Simulation of Comput. & Telecomm. Sys.*, pp. 61–68, Sep 2006.

[7] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ACM Intl. Conf. on Supercomputing*, pp. 338–347, 1995.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative approach*. Morgan Kaufmann, 3rd ed., 2003.

[9] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, 1988.

[10] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Intl. Symp. on Microarchitecture*, pp. 93–104, 2003.

[11] J. Jalminger and P. Stenström. A novel approach to cache block reuse predictions. *Intl. Conf. on Parallel Processing*, 2003.

[12] S. Jin and A. Bestavros. Sources and characteristics of web temporal locality. In *Modeling, Anal. & Simulation of Comput. & Telecomm. Sys.*, pp. 28–35, Aug 2000.

[13] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Intl. Symp. on Computer Architecture*, pp. 315–326, 1997.

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Intl. Symp. on Computer Architecture*, pp. 364–373, 1990.

[15] M. Karlsson and E. Hagersten. Timestamp-based selective cache allocation. In *Workshop on Memory Performance Issues*, Jun 2001.

[16] J. Kin, M. Gupta, and W. H. Mangione-Smith. Filtering memory references to increase energy efficiency. *IEEE Trans. on Computers*, 49(1):1–15, Jan 2000.

[17] G. Memik and W. H. Mangione-Smith. Increasing power efficiency of multi-core network processors through data filtering. In *Intl. Conf. on Compilers, Arch., and Synth. for Embedded Sys.*, pp. 108–116, 2002.

[18] J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Intl. Conf. on Parallel Processing*, vol. 1, pp. 154–163, 1996.

[19] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. P. Shen. Coming challenges in microarchitecture and architecture. *Proc. of the IEEE*, 89(3):325–340, Mar 2001.

[20] R. Rosner, A. Mendelson, and R. Ronen. Filtering techniques to improve trace-cache efficiency. In *Intl. Conf. on Parallel Arch. and Compilation Techniques*, pp. 37–48, 2001.

[21] J. Sahuquillo and A. Pont. The filter cache: A run-time cache management approach. In *EUROMICRO*, pp. 1424–1431, 1999.

[22] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Mcgraw-Hill, Jul 2004.

[23] Standard Performance Evaluation Corporation. SPEC2000. http://www.spec.org.

[24] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, HP Laboratories, Palo Alto, June 2006.

[25] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *28th Intl. Symp. on Microarchitecture*, pp. 93–103, Nov 1995.

[26] N. H. E. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison Wesley, 3rd edition, 2005.