Exploiting Core Working Sets to Filter the L1 Cache with Random Sampling

Yoav Etsion and Dror G. Feitelson

Abstract

Locality is often characterized by working sets, defined by Denning as the set of distinct addresses referenced within a certain window of time. This definition ignores the fact that dramatic differences exist between the usage patterns of frequently used data and transient data. We therefore propose to extend Denning's definition with that of *core* working sets, which identify blocks that are used most frequently and for the longest time.

The concept of a core motivates the design of dual cache structures that provide special treatment for the core. In particular, we present a probabilistic locality predictor for L1 caches that leverages the skewed popularity of blocks to distinguish transient cache insertions from more persistent ones.

We further present a dual L1 design that inserts only frequently used blocks into a low-latency, low-power, direct-mapped main cache, while serving others from a small fully-associative filter. To reduce the prohibitive cost of such a filter, we present a content addressable memory design that eliminates most of the costly lookups using a small auxiliary lookup table. The proposed design enables a 16K direct-mapped L1 cache, augmented with a small 2K filter, to outperform a 32K 4-way cache, while at the same time consume 70%–80% less dynamic power and 40% less static power.

Index Terms

Core Working Sets, Random Insertion Policy, Mass-Count Disparity, L1 Cache, Cache Insertion Policy, Dual Cache Design, Cache Filtering

I. Introduction

Memory system performance places a limit on computation rates due to the large gap between processor and memory speeds. The typical approach to alleviate this problem is to increase the capacity of L1 and L2 caches. This solution, however, also increases the power consumed by the caches. The problem is further exacerbated by the shift towards using chip multiprocessors and the ensuing replication of L1 caches for each core. These developments motivate attempts for better utilization of cache resources, through the design of more efficient caching structures. This in turn relies on extensive analysis of memory workloads and a deeper understanding of cache behavior.

The essence of caching is to identify and store those data items that will be most useful in the immediate future. Denning formalized this using the notion of a *working set*, defined to be those items that were accessed within a certain number of instructions [7]. We make the observation that memory workloads are highly skewed, with a small "core" of the working set servicing the majority of references. At the same time, many other cache blocks are only accessed a small number of times, in a bursty manner. This means that the success of caches can be based not only on locality in space and time, but also on exploiting the skewed *popularity* of different cache blocks.

The fact that memory references are highly skewed is well known. But our analysis reveals that the skew in block popularity is more extreme than previously thought, and exhibits the statistical phenomenon of *mass-count disparity*. This indicates that the majority of references are concentrated in the core, which

Y. Etsion is with the Electrical Engineering and Computer Science faculties, Technion - Israel Institute of Technology, Technion City. Haifa 32000, Israel. E-mail: yoav.etsion@gmail.com

D. G. Feitelson is with the School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel. E-mail: feit@cs.huji.ac.il

This work was done while Y. Etsion was with the Barcelona Supercomputing Center, Spain, and the Hebrew University of Jerusalem, Israel.

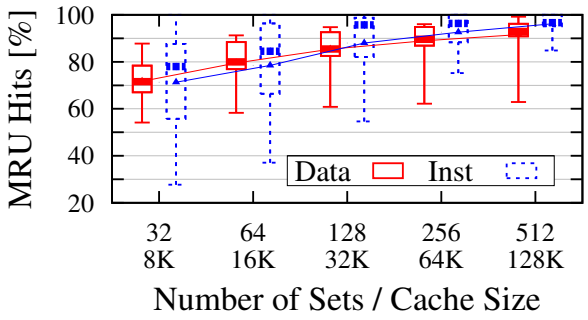

Fig. 1. *Fraction of accesses serviced by the most recently used (MRU) block in the cache sets for various cache sizes, with 4-way set associative caches and 64B cache lines. Boxes indicate the 25^{th} to 75^{th} percentiles over 20 SPEC benchmarks, with whiskers extending to the minimum and maximum.*

therefore may dominate system performance. It may therefore be beneficial to use separate structures to serve the core blocks in the most efficient manner possible.

The idea that blocks be classified according to access patterns suggests a need to maintain historical data. But with strong mass-count disparity we find that stateless random sampling suffices to make useful probabilistic classifications: since most memory references are serviced by a small fraction of the working set, randomly selected *references* will likely identify very popular core blocks.

Given such a sampling-based predictor, we design a dual[1]structure in which popular blocks are handled differently from the rest. Consider the data in Fig. 1. This shows that set-associative caches actually service most of the references — namely those to the highly popular blocks — from the cache sets' MRU position[2]. These caches thus effectively function as direct-mapped caches, while incurring set-associative latency and power consumption. But once we identify popular blocks, we can use a direct mapped structure explicitly, reducing latency and power consumption. The rest of the references are serviced from a small fully-associative auxiliary filter, thereby avoiding conflict misses in the direct-mapped cache. In addition, we introduce the *Wordline Look-aside Buffer* (WLB), a small lookup table that harnesses temporal locality to eliminate the vast majority of expensive lookups in the fully-associative filter, without affecting the fully-associative semantics. The result is a win-win design, which both improves overall performance and reduces power consumption.

This paper expands on the above ideas. Section II explains and quantifies mass-count disparity. Section III reviews motivation and related work on dual structures. Section IV provides details of the probabilistic predictor. Given all the above, Section V presents and evaluates our design for a filtered L1 cache. Lastly, Section VI reviews related work and Section VII presents our conclusions.

II. The Skewed Popularity of Memory Locations

Locality of reference is one of the best-known phenomena of computer workloads. This is usually divided into two types: spatial locality, namely accesses to addresses that are *near* an address that was just referenced, and temporal locality, which is *repeated references* to the same address. Temporal locality is actually the result of two distinct phenomena. One is the skewed popularity of different addresses, where some are continuously referenced over time, while others are only referenced a few times [15]. The other is correlation in time: accesses to the same address occur together in bursts of activity, rather than being distributed uniformly through time. While the intuition of what "temporal locality" means tends to the second of these, the first is actually the more important effect.

A. Cache Residency Length: A New Metric for Temporal Locality

The common *block popularity* metric rates memory blocks based on the number of times they are accessed during a reference window of predetermined size. The major caveat of this metric is that it considers *all* the references made to an address, whereas for caching studies references *clustered in time*

[1]We use the term *dual* cache to differentiate it from a *split* cache, used to split the data and instructions streams.

[2]Fig. 1 specifically emphasizes the effect of *temporal* locality, as repeated accesses to the same memory block were collapsed into a single memory access.

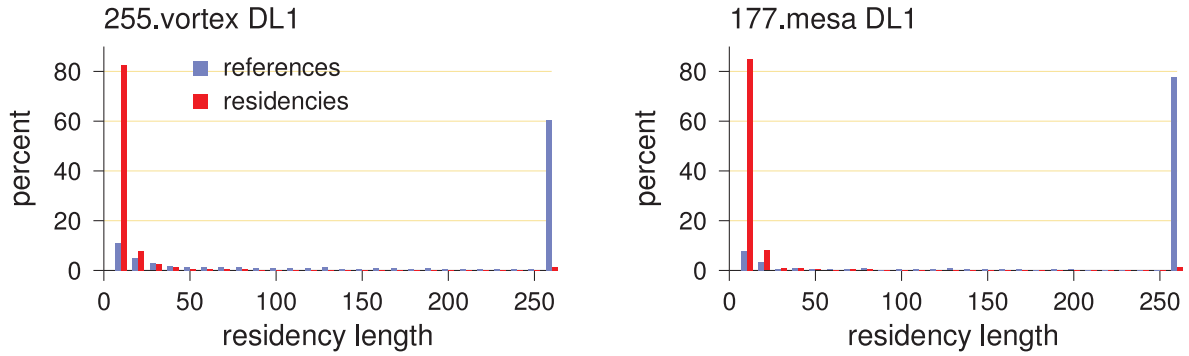

Fig. 2. *Histograms of residency lengths for select SPEC benchmarks (ref dataset).*

may be more important. In addition, the arbitrary sized windows may or may not be aligned with program phases, thus possibly amplifying inaccuracies.

We therefore propose not to use a predefined window, but rather to count the number of references made between a block's insertion into the cache and its subsequent eviction. This is denoted the *cache residency length*. Thus, if a certain block is referenced 100 times when it is brought into the cache, is then evicted, and then is referenced 200 times when brought into the cache again, we will consider this as two distinct cache residencies spanning 100 and 200 references, respectively.

One deficiency of the cache residency length metric is its dependence on the specific cache configuration. It is therefore important to remain consistent when comparing results. The results shown here are based on a 16K direct-mapped configuration. We also consistently define memory objects to be 64 bytes long, because this is the most common size for a cache line.

B. Mass-Count Disparity in L1 Workloads

Histograms of the distribution of residency lengths for a couple SPEC2000 benchmarks are shown in Fig. 2 (for 16KB direct-mapped caches as specified above). These show the distribution of residency lengths and the distribution of references to these residencies, up to residency lengths of 250 references, using buckets of size 10. Longer residencies (and references therein) are bunched together in the last bar on the right. This leads to characteristic bimodal distributions, where residencies are seen to be short (seldom more than 50 references), but most references belong to residencies that are longer than 250 references. While not universal, this pattern repeats in many cases.

A better quantification is possible using mass-count disparity plots (Fig. 3) [13]. These plots superimpose the cumulative distribution functions (CDF) of the two distributions in the above histograms. The first, called the *count* distribution, is the distribution of cache residency lengths, and $F_c(x)$ is the probability that a residency is has x references or less. The second, called the *mass* distribution, describes how the references are distributed among these residencies. Thus $F_m(x)$ represents the probability that a reference is *part of* a residency that has x references or less.

Mass-count disparity occurs when the two distributions diverge. a well-known example of this phenomenon is the distribution of wealth [19]: most people are relatively poor, and only a few are very rich, so the majority of wealth in the world belongs to a very small fraction of the population. Analogously, most residencies are short and only a few are very long, so the majority of references are targeted at a very small fraction of the residencies (and hence memory blocks).

Fig. 3 shows examples for the vortex and mesa benchmarks from the SPEC 2000 suite. The simplest metric for quantifying the disparity is the *joint ratio*, which is the unique point where the sum of the two CDFs is unity (if the CDFs have a discrete mode, as sometimes happens, the sum may be different) [13]. For example, in the case of the mesa benchmark data stream, the joint ratio is 10/90. This means that 10% of the cache residencies, and more specifically those that are highly referenced, service a full 90% of the references, whereas the remaining 90% of the residencies service only 10% of the references — a precise example of the proverbial 10/90 principle. Thus a typical residency is only referenced a small

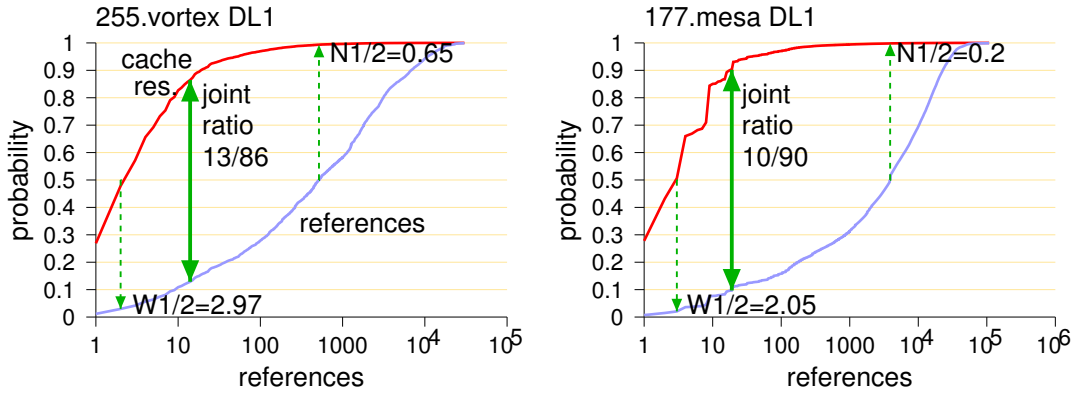

Fig. 3. *Mass-count disparity plots for memory accesses in select SPEC benchmarks (ref dataset).*

	Data						Instruction					
Benchmark	$W_{1/2}$	@	$N_{1/2}$	@	JR	@	$W_{1/2}$	@	$N_{1/2}$	@	JR	@
164.gzip	3.80	1	0.60	230	13 / 87	8	2.20	16	0.00	7.2 e5	10 / 90	123
175.vpr	7.60	2	1.71	72	20 / 80	8	1.43	16	0.00	2.5 e7	7 / 93	144
176.gcc	11.68	8	0.27	3826	21 / 79	8	2.74	13	0.05	5.4 e4	11 / 89	59
181.mcf	24.77	1	16.79	3	33 / 67	1	6.50	1.46 e8	10.69	1.0 e9	26 / 74	2.5 e8
186.crafty	4.39	1	0.69	169	14 / 86	8	11.21	15	3.72	112	24 / 76	24
197.parser	4.48	2	0.67	336	15 / 85	9	3.16	16	0.00	7.2 e5	11 / 89	48
253.perlbmk	2.29	3	0.93	731	12 / 88	25	5.91	12	1.48	858	16 / 84	30
255.vortex	3.25	3	0.65	517	13 / 87	15	11.33	13	4.34	96	24 / 76	20
256.bzip2	1.83	1	0.10	3247	10 / 90	24	0.19	7.5 e4	5.22	1.2 e7	18 / 82	4.3 e6
300.twolf	7.34	3	4.39	42	22 / 78	9	6.32	16	1.91	416	19 / 81	64
168.wupwise	3.59	8	1.07	804	16 / 84	32	0.65	16	0.00	2.6 e7	5 / 95	512
171.swim	38.98	10	37.48	10	44 / 56	10	0.01	11	0.46	3.5 e6	1 / 99	6.7 e5
172.mgrid	5.34	2	10.41	30	23 / 77	17	0.00	11	1.03	1.8 e6	5 / 95	2.0 e5
177.mesa	2.01	3	0.20	3886	10 / 90	20	4.65	11	0.02	3.0 e4	13 / 87	32
178.galgel	11.12	2	6.44	20	22 / 78	8	0.06	2.7 e5	5.33	1.6 e8	12 / 88	8.6 e6
179.art	21.69	2	16.52	3	33 / 67	2	0.00	16	0.04	1.1 e8	3 / 97	6.4 e4
187.facerec	3.41	2	2.32	104	20 / 80	16	0.01	16	0.28	4.6 e6	3 / 97	1.8 e5
188.ammp	5.88	3	1.85	96	19 / 81	12	1.02	16	0.00	1.3 e7	6 / 94	448
189.lucas	18.24	8	11.44	8	33 / 67	8	0.00	20	3.94	8.4 e6	20 / 80	4.2 e6
301.apsi	4.81	1	0.26	396	11 / 89	6	7.57	32	1.41	384	21 / 79	86
Average	9.33	3.3	5.74	726	20 / 80	12.3	3.25	7.3 e6	2.00	7.0 e7	13 / 87	1.7 e7
Median	5.08	2	1.39	137	21 / 81*	9	1.82	16	0.75	2.7 e6	12 / 89*	296

* Median Joint-Ratio values are independent and thus may not sum up to 100%.

TABLE I

$N_{1/2}$, $W_{1/2}$, and joint ratio metric values for both data and instruction streams of the SPEC2000 benchmarks used.

number of times (up to 10 or 20 in this case), whereas a typical reference is directed at a long residency (one that is referenced thousands of times).

Two other important metrics in the context of dual cache designs are $W_{1/2}$ and $N_{1/2}$. $W_{1/2}$ assesses the combined weight of the half of the residencies that receive the fewest references. For mesa, these 50% of the residencies together get only 2.05% of the references. These represent blocks that are inserted into the cache but hardly accessed, causing inefficient cache use. Instead, caches should preferably be used to store longer residencies, such as those that together account for 50% of the references. As quantified by the $N_{1/2}$ metric, in mesa these are just 0.2% of the residencies.

Mass-count disparity metric values are listed in Table I for all 20 benchmarks analyzed. The table also indicates the maximal residency length included in $W_{1/2}$, the minimal residency length included in $N_{1/2}$, and the residency length where the joint ratio occurs (marked by the @ values). These results generally indicate significant mass-count disparity. For data streams, the table reveals that half of the references are serviced by less than 2% of all residencies, in 12 of the 20 benchmarks. The disparity is less apparent

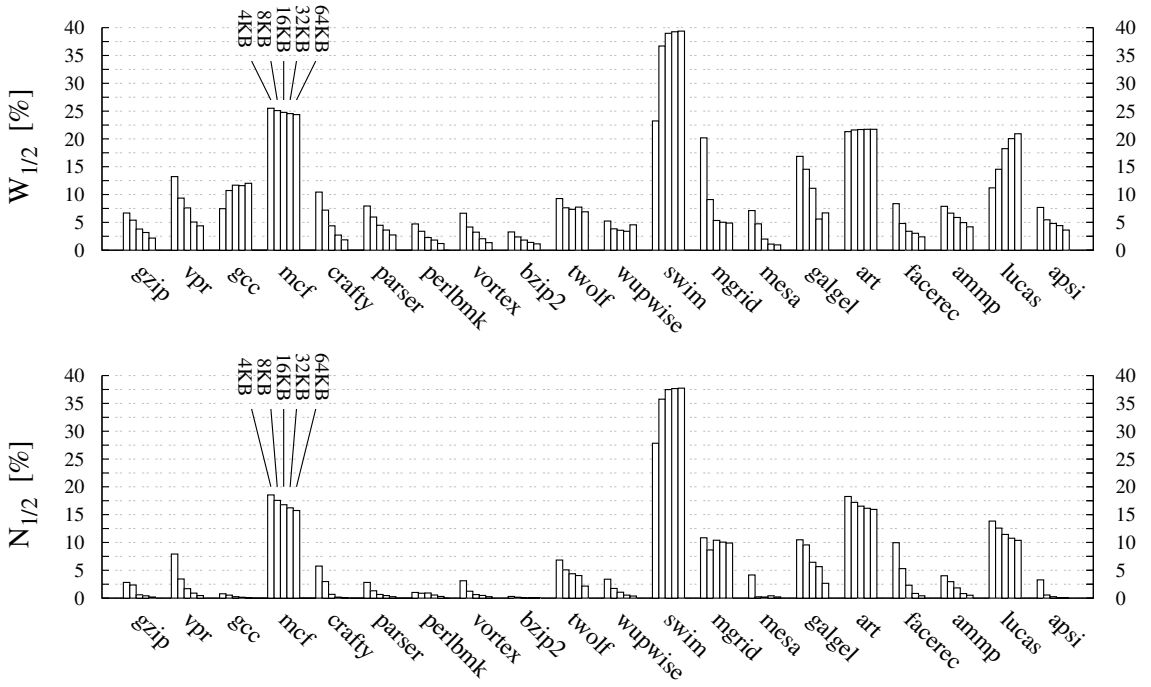

Fig. 4. *The effect of cache size, ranging from 4KB to 64KB, on the $W_{1/2}$ (top) and $N_{1/2}$ (bottom) statistics for the data streams of SPEC2000 benchmarks.*

in benchmarks known for their poor cache utilization, such as *mcf*, *art*, *swim*, and *lucas*. For example almost 96% of *mcf*'s residencies contain up to 5 references, but still they account for over 70% of the references. This leads to a joint ratio of 33/67, and relatively high $W_{1/2}$ and $N_{1/2}$ values — with half the residencies accounting for $\sim$25% of the mass, and half the mass centered in $\sim$17% of the residencies. But with the longest 3% of the residencies incorporating 30% of the mass, even *mcf* still exhibits some degree of disparity.

In the case of instruction streams the results tend to be more extreme than for data streams. The joint ratios are $\sim$26/74 and up, with several higher than 5/95. This is a manifestation of the rule-of-thumb that programs spend most of the time executing a small fraction of their code. The highest $W_{1/2}$ values are $\sim$11%, and most are much lower, indicating that the short residencies service only a small fraction of all references. The $N_{1/2}$ values are generally even lower, indicating the long residencies dominate the reference stream, and that the majority of references are directed at a small fraction of the working set. The only exception is *mcf* where $N_{1/2} = 10.7\%$. This stems from *mcf*'s exceptional code density, which results in a very small number of distinct instruction blocks accessed throughout the execution, so all residencies are longer than 10^5 references.

The above results may be expected to depend on cache size. Intuitively, larger caches will experience fewer evictions and thereby merge consucutive redidencies of the same block into a single, longer residency. The resulting increase in residency lengths will shift the mass distribution towards longer (core) residencies. Hence the fraction of the overall mass in the shorter 50% of the residencies ($W_{1/2}$) will decrease as caches get bigger, and so will the fraction of long residencies needed for 50% of the mass ($N_{1/2}$).

Fig. 4 shows the effect of cache size on $W_{1/2}$ and $N_{1/2}$ for the data streams of SPEC2000 benchmarks (instruction streams produced similar results). As expected, while the effect is not large, the metrics do tend to decrease as cache sizes increase. The exceptions to this rule are *gcc*, *swim*, and *lucas* for $W_{1/2}$, and *swim* for $N_{1/2}$. The reason for this is that these benchmars exhibit intrinsic access patterns that result in short residencies, regardless of the cache size. For example, *gcc*'s intrinsics lead to $\sim$90% of its residencies being shorter than 8 references. The increase in cache size and ensuing merging of multiple consecutive

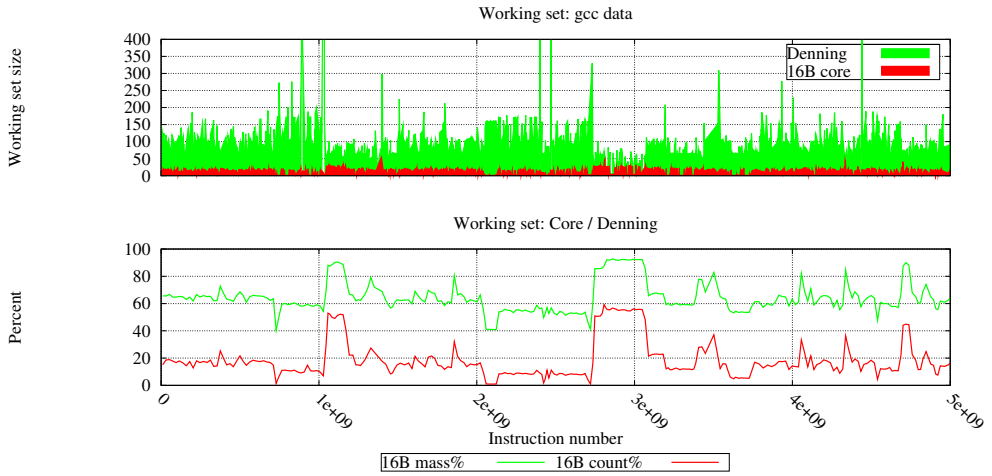

Fig. 5. *Examples of memory access patterns and the resulting Denning and core working sets.*

residencies only merges these clusters of 8 reference that were broken into multiple residencies into a single 8-reference residency. Therefore, residencies of length 8 constitute ∼89% of the residencies in a 4KB cache and ∼94% of the residencies in a 64KB cache. The end result is that the shorter 50% of the residencies in a 64KB cache constitute of more residencies of length 8, which increases their overall mass.

Fig. 4 demonstrates that our metrics are robust across multiple cache sizes and even capture the movement of residencies from *pollution* to *core*. In addition, while the results shown here focus on direct mapped caches, similar results are obtained for 4-way set-associative caches [10]. Together, they confirm the effectiveness of the metrics in providing a better understanding of cache workloads.

C. Definition of Core Working Sets

Denning's definition of working sets is based on the principle of locality, which he defined to include a slow change in the reference frequency to any given page [7], [8]. Our data, however, demonstrates the continued access to the same high-use memory objects, while much of the low-use data is only accessed for very short and intermittent time windows. In addition, transitions between phases of the computation may be expected to be sharp rather than gradual, and moreover, they will probably be correlated for multiple memory objects. This motivates a new definition that focuses on the persistent high-usage data in each phase. We thus define the *core working set* to be those blocks that appear in the working set and are reused "a significant number of times".

The simplest interpretation of this definition is based on counting the number of references to a block during a single cache residency. The number of references needed to qualify can be decided using data such as that presented in Figs. 2 and 3. For example, we can set the threshold so that for most benchmarks it will identify no more than 5% of the residencies, but more than 50% of the references. For typical residency length distributions, such a threshold would be 100–1000 references.

While the skewed distribution of popularity is a major contributor to temporal locality, one should nevertheless acknowledge the fact that references do display a bursty behavior. To study this, we looked at how many different blocks are referenced between successive references to a given block. The results indicate that the majority of inter-reference distances are indeed short. We can then define bursts to be sequences of references to a block that are separated by references to less than say 256 other blocks. Using this we can study the distribution of burst lengths, and find them to be generally short, ranging up to about 32 references for most benchmarks. However, they are long enough to discourage the use of a low threshold to identify blocks that belongs to the core working set with confidence. Again this points to a threshold of 100 or more.

The effect of the above definitions is illustrated in Fig. 5. The figure shows the Denning working set for a window of 1000 instructions, and the core working set as defined by a threshold of 16 references to a block (denoted 16B in the legend). The core working set is indeed much smaller, typically being just 10–20% of the Denning working set. Importantly, it eliminates all of the sharp peaks that appear in the Denning working set. Nevertheless, as shown in the bottom graph, it routinely captures about 60% of the memory references.

III. Cache Bypass and Dual Structures

We have established that memory blocks can be roughly divided into two groups: the core working set, which includes a relatively small number of blocks that are accessed a lot, and the rest, which are accessed few times in a bursty manner. The question is how this can be used to improve caching.

A. The Advantage of Cache Bypass

The principle behind optimal cache replacement is simply to replace the item that will not be used for the most time in the future (or never) [3]. In particular, it is certainly possible that the optimal algorithm will decide to replace the *last* item that was brought into the cache. This would indicate that this item was inserted into the cache only as part of the mechanism of performing the access, and not in order to retain it for future reuse.

Analyzing the reference streams of SPEC benchmarks indicates that this behavior does indeed occur in practice. For example, we found that if the references of the gcc benchmark were to be handled by a 16 KB fully-associative cache, 30% of insertions would belong to this class; in other benchmarks, we saw results ranging from 13% to a whopping 86%. Returning to gcc, if the cache is 4-way set associative the placement of new items is much more restricted, and a full 60% of insertions would be immediately removed by the optimal algorithm. These results imply that the conventional wisdom favoring the LRU replacement algorithm is debatable.

It is especially easy to visualize why LRU may fail by considering transient streaming data. When faced with such data, the optimal algorithm would dedicate a single cache line for all of it, and let the data stream flow through this cache line. All other cache lines would not be disturbed. Effectively, the optimal algorithm thus partitions the cache into the main cache (for core non-streaming data) and a cache bypass for the streaming component (non-core). The LRU algorithm, in contradistinction, would do the opposite and lose all the cache contents.

The advantage of a cache bypass mechanism can be formalized as follows, using a simple, specific example cache configuration. Assume a cache with $n^2 + n$ cache lines, and an address space partitioned into n equal-size disjoint partitions. The cache is organized in either of two ways:

Set associative: there are n sets of $n + 1$ cache lines each, and each serves a distinct partition of the address space. This is the commonly used approach.

Bypass: there are n sets of n cache lines each, used as above. The $n + 1$st set can accept any address and serves as a bypass.

These two designs expose a tradeoff. In the set associative design, each set is larger by one, reducing the danger of conflict misses. In the bypass design, the extra set is not tied to any specific address, increasing flexibility. However, it is relatively easy to see that the bypass design has the advantage. Formally this is shown by two claims.

Claim 1: The bypass design can simulate the set associative design.

Proof: While each cache line in the bypass set can hold any address from the address space, we are not required to use this functionality. We can limit each cache line to one of the partitions, so the effective space available for caching each partition becomes $n+1$, as in the set associative design. Thus the bypass design need never suffer more cache misses than the set associative design. ∎

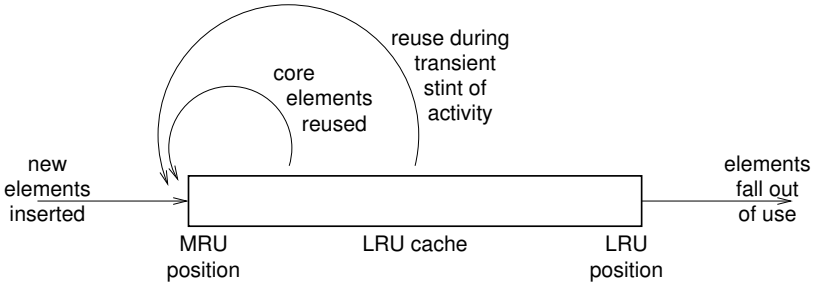

Fig. 6. *Operation of an LRU cache is implicitly based on the notion that core elements will be reused before they drop out of the cache.*

Claim 2: There exist access patterns that suffer arbitrarily more cache misses when served by the set associative design than when served by the bypass design.

Proof: An access pattern that provides such an example is the following: repeatedly access $2n$ addresses from any single partition in a cyclic manner m times. When using the set associative design, only a single set with n cache lines will be used. At best, an arbitrary subset of $n-1$ addresses will be cached, and the other $n+1$ will share the remaining cell, leading to a total of $O(nm)$ misses. When using the bypass design, on the other hand, all $2n$ addresses will be cached by using the original set and the bypass set. Therefore only the initial $2n$ compulsory misses will occur. By extending the length of this pattern (i.e. by increasing m) any arbitrary ratio can be achieved. ∎

More generally, the number of sets and the set sizes need not be the same. The size of the bypass set need also not be the same as that of all the other sets.

B. Related Work on Dual Structures

The skewed distribution of references seems to match the LRU cache replacement scheme. Core blocks are not evicted because they are referenced again and again, bumping them back to the top of the stack each time (Fig. 6). If the cache is big enough, transient blocks that enjoy a burst of activity can also be retained till this activity ends. Skew also explains random eviction (which may be used in set-associative caches [28], [29]), because if you select a cache residency at random, it is most probably a short residency. Long residencies are rarer, and therefore less likely to be evicted.

Nevertheless, core residencies may still be evicted by mistake. Although frequently accessed blocks will be quickly re-inserted into the cache, the first access after the eviction will incur a cache miss. The desire to reduce such mistakes is one of the motivations for using dual cache structures. Our design described in Section V is explicitly based on the skewed distribution of residency lengths, where one part of the cache is used for the core data, while the more transient data is filtered. This is a generalization of the cache bypass considered above.

Many similar schemes have been proposed in the literature [27]. Many of them are based on an attempt to identify and provide support for blocks that display temporal locality — in effect, the more popular blocks that are reused time and again. For example, Rivers and Davidson propose to tag cache lines with a temporal locality bit [26]. Initially, lines are stored in a small non-temporal buffer (in our terminology, this is the bypass area). If they are reused, the temporal bit is set indicating that, in our terminology, these lines should be considered as core elements. Later, when a line with the temporal bit set is fetched from memory, it is inserted into the larger temporal cache.

Park et al. also use a spatial buffer to observe usage [22]. However, they do so at different granularities: when a word is referenced, only a small sub-line including this word is promoted to the temporal cache. McFarling's dynamic exclusion cache augments cache lines with two state bits, the last-hit bit and the sticky bit [20]. The sticky bit is used to retain a desirable cache line rather than evicting it upon a conflict; the conflicting line is served directly to the processor without being cached. However, this approach is limited to instruction streams and specifically to cases where typically only two instructions conflict with

each other. A more extreme approach is the bypass mechanism of Johnson et al. [16]. This is based on a memory address table (MAT) which counts accesses to different areas of memory. Then, if a low-count access threatens to displace a cached high-count datum, it is simply loaded directly to the register file bypassing the cache. Another scheme is the Assist cache used in the HP PA 7200 CPU [5], which filters out streaming (spatial locality) data based on compiler hints.

The above schemes have the drawback of requiring historical information to be maintained for each cache lines. This is avoided by Walsh and Board, who propose a dual design with a direct-mapped main cache and a small fully associative filter [33]. Referenced data is first placed in the filter, and only if it is referenced again it is promoted to the main cache. This avoids polluting the cache with data that is only referenced once,but our data indicates that a much higher threshold is needed. Indeed, this is the basis for the design we propose below.

A somewhat different approach is provided by Jouppi's victim cache, which is a small auxiliary cache used to store lines that were evicted from the main cache [17]. This helps reduce the adverse effect of conflict misses, as the fully-associative victim buffer effectively increases the size of the most heavily used cache sets. In this case the added structure is not used to filter out transient data, but rather to recover core data that was accidentally displaced by transient data.

IV. Probabilistic Prediction of Temporal Locality

Mass-count disparity implies that the working set is not evenly used, but is rather focused around a core serving the majority of references. This has important consequences for random sampling. Specifically, if you pick a residency at random, there is a good chance that it is seldom referenced. 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 belonging to the core of the working set.

Identifying the core can be used to implement a *cache insertion policy* that will prevent transient blocks from being inserted into the cache and polluting it. The length of a cache residency can serve as a metric for cache efficiency, with longer residencies indicating better efficiency, since the initial block insertion overhead (latency and power) is amortized over many cache hits. The insertion policy should therefore be based on a residency length predictor, that will be used to predict whether inserting a block into the cache would be beneficial. As noted above, length is the very characteristic that allows the desired residencies to be identified using random sampling.

A. Identifying An Effective Subset of Blocks

Belady's optimal replacement policy [3] does not handle dual cache structures. Moreover, Brehob et al. have shown that optimal cache replacement is NP-Hard for dual-caches in which one component is fully-associative and the other is set-associative or direct-mapped [4]. Thus we cannot expect optimal algorithms to be implementable, and optimality cannot be used as an evaluation criterion.

An alternative approach is to evaluate cache filtering from a cost/gain perspective, trying to find the minimal core working set of residencies that will effectively maximize gain. In this framework, the *cost* can be defined as the fraction of all residencies included in the core working set, and the *gain* can be defined to be the fraction of all references subsumed by the residencies in the core. Alternatively, the cost of the core working set can be regarded as the size of the cache needed to accommodate it, and the gain as the hit-rate achieved.

The tradeoff between the cost and gain here is obvious. Inserting all residencies to the core will service all references from the core (100% gain), but will also require a cache of maximal size (100% cost). On the other hand, leaving the core empty will minimize the number of residencies included in the core (0% cost), but will serve no references from the core as well (0% gain). Balancing the two opposite goals requires a threshold parameter for the minimal residency length that should be considered part of the core. This can be done by finding a threshold that maximizes the average gain, i.e. $\frac{gain}{cost}$. But in the case

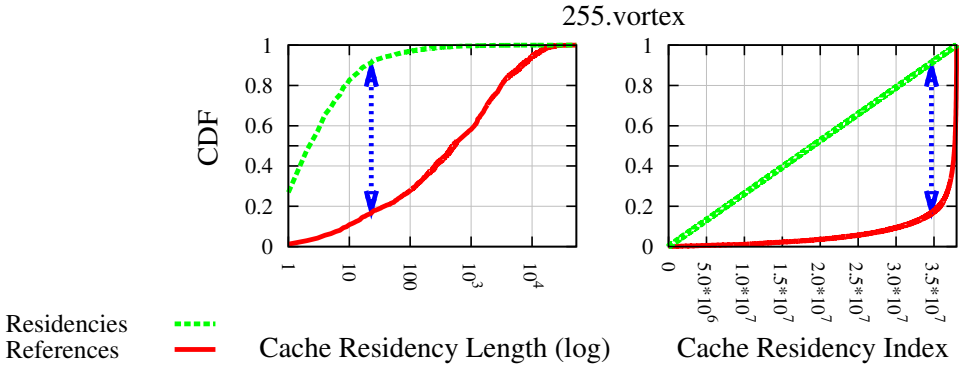

Fig. 7. *Demonstration of the maximal effectiveness threshold. Left: the mass-count plot for* vortex. *Right: same data plotted as a function of the residency index, where residencies are sorted by size. Double arrow indicates the point where the difference between the distributions is maximized.*

at hand, this goal translates directly to finding a threshold that maximizes the average residency, defined as $\frac{core\_references}{core\_residencies}$. Due to monotonicity, this is obviously maximized by only including the longest residency in the core working set.

The proposed strategy is therefore to use the threshold that maximizes the *difference* between the fraction of references handled by the core, and the fraction of residencies included in the core, namely $core\_references - core\_residencies$. All residencies whose length is longer than the threshold are considered part of the core working set. Interestingly, the desired threshold is very simple:

Claim 3: Setting the core threshold to the *average residency length* maximizes the target function $core\_references - core\_residencies$.

Proof: Given R the total number of references, and B the total number of residencies, we can sort the residencies according to length from the shortest to the longest. Let $len(i)$ denote the length of the ith shortest residency. The sorting guarantees that $len(i) \leq len(i+1)$ for all i. Mass-count disparity plots can then be plotted as a function of the residency indices, as shown in Fig. 7. In this plot style, the cumulative probabilities of the count and mass at the ith residency are

$$count(i) = \sum_{j=1}^{i} \frac{1}{B} = \frac{i}{B} \qquad \text{and} \qquad mass(i) = \sum_{j=1}^{i} \frac{len(j)}{R}$$

respectively. Hence the ith residency adds $\frac{1}{B}$ to the count distribution, and $\frac{len(i)}{R}$ to the mass. Therefore, given that $\frac{len(i)}{R}$ is monotonically non-decreasing, the gap between the two distributions grows monotonically while $\frac{len(i)}{R} < \frac{1}{B}$, narrows when $\frac{len(i)}{R} > \frac{1}{B}$, and peaks at the average residency length

$$threshold = \overline{len} = \frac{R}{B}$$

While this may not be integral, it can still serve as a threshold for the residency lengths. Furthermore, monotonicity of the gap dynamics assures that the threshold represents the global maximum. ∎

The average residency length thus represents a unique equilibrium point that maximizes the effectiveness of the core working set: any different proposed subset of residencies will inevitably replace a block in the core with one not belonging to the core, and will thus replace a long residency with shorter one — thereby reducing the number of references that will be served by the core. In the case of vortex, shown in Fig. 7, the threshold indicates that ∼9% of all residencies be in the core, and they serve ∼84% of all references.

Note also that using a threshold actually relaxes the requirements from the residency length predictor, as it does not have to predict the actual length of a residency, but rather produce a binary prediction stating whether the residency is likely to be longer than the threshold or not.

B. Probabilistic Residency Length Predictor

The probabilistic residency length predictor is based on the mass-count disparity phenomenon characteristic of the skewed distribution of block popularity. As noted above, selecting a memory reference at random by executing a Bernoulli trial on each memory reference is likely to identify a reference that belongs to a long residency. When this happens, the rest of the residency is considered to be part of the core working set.

The probability to use is the reciprocal of the desired residency length threshold. When sampling references with a low probability $P = \frac{1}{T}$, short residencies will have a very low probability of being selected. But given that a single hit is enough to induct a residency into the core, the probability that a residency is classified as core after n references is $1 - (1 - P)^n$. This converges exponentially to 1 for large n. In practice, the selection need not even be random, and we have verified that periodic selection achieves results similar to those obtained with random selection. For consistency, though, only results for random selection are shown.

Importantly, implementing such a predictor does not require saving *any* state information for the blocks, since every selection is independent of its predecessors. The hardware required to implement the selection mechanism is trivial and constitutes of a pseudo random number generator, which can be implemented using a simple linear-feedback shift register, whereas periodic selection requires only a saturating counter [34]. It can also share the random source with others, such as Qureshi et al. prediction of performance critical memory references [24] and Behar et al. [2] sampling-based predictor that reduces the power of trace caches by generating only selected traces.

C. Evaluating the Probabilistic Predictor

The evaluation of the probabilistic predictor is done against the threshold-based gain-maximization idea of Section IV-A. Since a sampling predictor with parameter P essentially tries to approximate a threshold of $\frac{1}{P}$, the evaluation focuses on the effectiveness of this approximation.

Fig. 8 compares the probabilistic runtime predictor with precise threshold-based selection for select benchmarks, with residencies generated using a 16KB direct-mapped cache. The figure shows the percentage of residencies classified as core (bottom lines) and the references they service (top lines) in both data (top row) and instruction (bottom row) streams. As a unified scale, the X-axis equates a sampling probability of P with a threshold of $\frac{1}{P}$. Focusing on the percentage of *data* references serviced by the predictor's selection, for example, we see a very good correlation to those serviced by the threshold-based classification, at least for $P \leq 0.01$. Thus for *vortex* using a selection probability of $P = 0.01$ in the sampling predictor covers over 60% of all data references, constituting over 90% of the number of references covered by threshold-based classification.

Table II lists the fraction of residencies classified as core by the probabilistic predictor on a 16KB direct-mapped cache, and the fraction of references they service, for both data and instruction streams, with $P = 0.01$ and $P = 0.001$. On average, sampling a mere 1% of the data references selects ∼7.39% of the residencies, and covers over 45% of the references. As the average is highly affected by benchmarks known for their poor temporal locality, like *swim*, *art*, and *mcf*, the median values are also shown, demonstrating a coverage of over 50% of the data references. Sampling probabilities can be an order of magnitude lower for *instruction* streams, as code density emphasizes locality. Thus sampling only 0.1% of the references selects an average of 20% of the residencies, covering some 77% of the references. Again, the medians emphasize the skewed averages, with selection reduced to 5.3% of the residencies but coverage growing to 89% of the references.

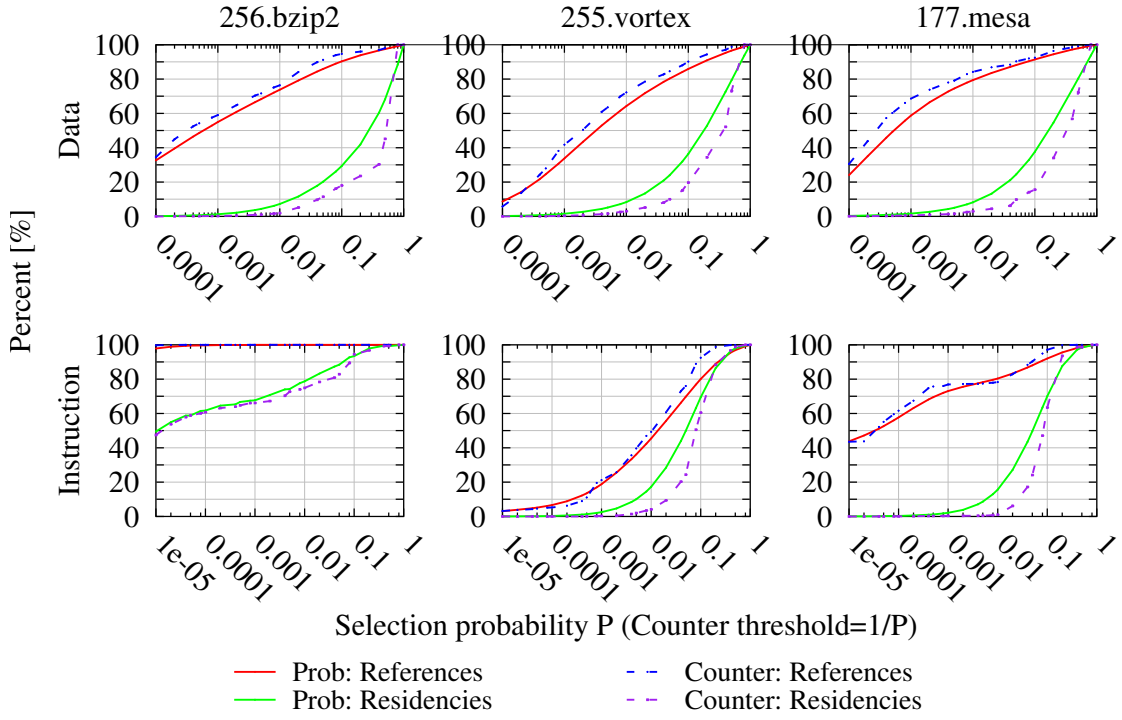

Fig. 8. *Fraction of blocks selected by the probabilistic predictor, and the fraction of references they service, compared with actually classifying residencies by their lengths.*

Overall, these results imply that executing Bernoulli trials with success probabilities of $P = 0.01$ for data streams and $P = 0.001$ for instruction streams are good operating points for all benchmarks analyzed. Similar results were also achieved for 4-way set-associative data and instruction caches [10]. The next section describes a dual-cache design based on the probabilistic predictor proposed above, and details a thorough exploration of specific probabilities suitable for this design.

V. A Random Sampling L1 Cache Design

This section introduces a novel dual L1 cache design that uses reference sampling to distinguish long residencies from short, transient ones. It then employs a direct-mapped structure to serve blocks belonging to the core working set, and a fully-associative structure acting as filter to serve the transient residencies. It is shown that such a design offers both better performance as well as reduced power consumption compared to common cache structures.

Fig. 9 depicts the proposed design. On each memory access, the data is first searched in the direct-mapped main cache. If this misses the fully-associative filter is searched. If the filter misses as well the request is sent to the next level cache. In our experiments we used 16K and 32K (common L1 sizes) for the direct-mapped cache, and a 2K fully-associative filter. All structures use 64B lines.

Each memory reference that is serviced by either the filter or the next level cache triggers a Bernoulli trial with a predetermined success probability P, to decide whether it 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). In case the block is not selected, and was not already present in the filter, the MRSU inserts it into the filter, typically evicting the LRU line there in order to make space. Section V-A explores the probabilistic design space for a suitable Bernoulli success probability.

To reduce both time and power overheads associated with accessing the fully-associative filter, we have augmented the classic CAM + SRAM design [34] with a *wordline look-aside buffer* (WLB) that caches recent lookups in the CAM. This is a small direct-mapped structure mapping block tags directly to the

	Data				Instructions			
	$P = 0.001$		$P = 0.01$		$P = 0.001$		$P = 0.01$	
Benchmark	%Resds	%Refs	%Resds	%Refs	%Resds	%Refs	%Resds	%Refs
164.gzip	0.90	31.91	5.77	56.63	4.50	82.34	28.23	89.06
175.vpr	0.90	13.86	6.24	41.09	3.49	88.40	21.98	92.77
176.gcc	1.30	45.53	8.38	65.26	2.86	77.19	18.89	85.06
181.mcf	0.20	1.32	1.87	8.12	100.00	100.00	100.00	100.00
186.crafty	0.83	27.34	5.49	52.29	2.63	26.64	18.42	49.17
197.parser	1.11	27.15	6.62	56.86	2.80	80.93	20.20	86.34
253.perlbmk	2.39	35.55	11.40	69.55	3.80	36.18	17.90	70.17
255.vortex	1.53	33.80	8.36	64.28	2.54	18.84	17.27	45.44
256.bzip2	1.23	54.96	7.22	73.83	67.70	99.97	78.71	100.00
300.twolf	0.94	5.98	7.22	28.79	4.56	41.09	26.51	66.08
168.wupwise	2.63	33.60	13.28	66.83	5.49	91.69	22.64	96.61
171.swim	1.13	0.88	10.62	7.49	5.12	99.87	15.21	99.96
172.mgrid	1.08	5.72	8.72	25.08	16.03	99.74	27.17	99.96
177.mesa	1.63	58.75	8.21	79.39	2.13	73.08	15.66	80.39
178.galgel	0.67	4.36	5.61	20.54	80.95	100.00	89.29	100.00
179.art	0.28	3.19	2.53	14.51	27.01	99.91	38.87	99.99
187.facerec	1.81	17.56	11.43	48.20	24.49	99.51	49.91	99.90
188.ammp	1.16	17.60	7.90	44.56	8.22	88.96	30.93	95.88
189.lucas	0.84	13.97	6.99	28.73	36.24	99.99	44.83	100.00
301.apsi	0.68	34.82	3.90	62.80	5.54	46.07	33.79	67.31
Average	1.16	23.39	7.39	45.74	20.30	77.52	35.82	86.20
Median	1.10	22.38	7.22	50.25	5.31	88.68	26.84	94.33

TABLE II

Percents of residencies classified as core and the references they service, for $P = 0.001$ and $P = 0.01$ (16KB direct-mapped data and instructions caches).

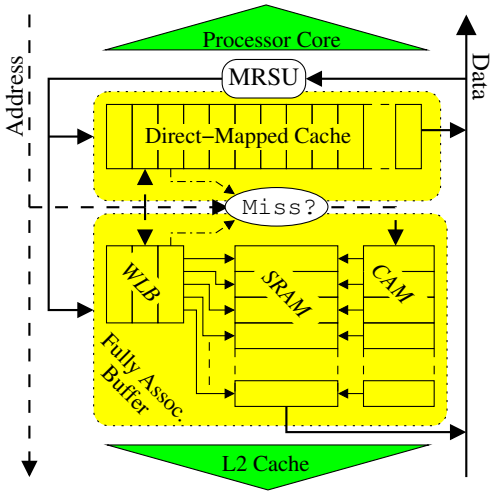

Fig. 9. *Design of a random sampling filtered cache.*

filter's SRAM based data store, so hits in the WLB avoid the majority of the costly CAM lookups, while still maintaining fully-associative semantics. Section V-B offers a detailed description of the WLB and an analysis of its design space.

A. The Effects of Random Sampling

We use random sampling of memory references to partition the reference stream into long residencies and short transient residencies. The number of references to long residencies is large, but they 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 naturally have a shorter cache lifetime, but there are many of them. Therefore, they are better served by a smaller, fully-associative structure.

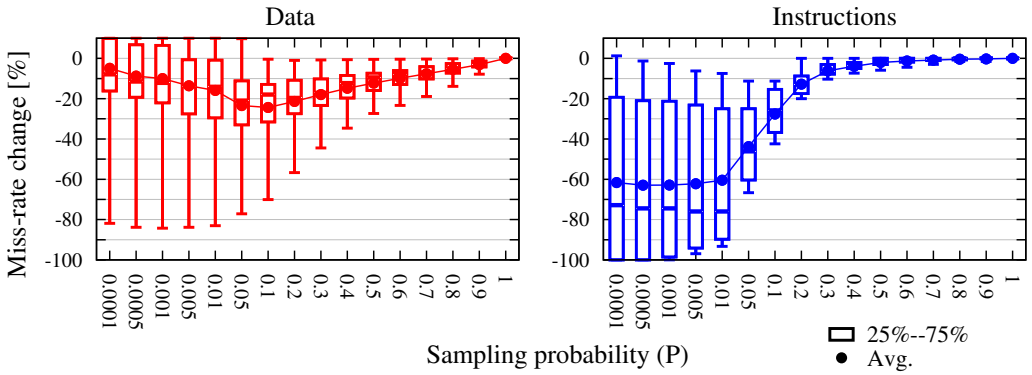

Fig. 10. *Distributions of changes in miss-rate when using various sampling probabilities, as applied to SPEC2000 benchmarks, for a 16K-DM cache.*

The sampling probability (and by implication the filtering rate) is therefore a delicate tuning knob: aggressive filtering might be counter-productive, since too many blocks may end up being served by the filter and not promoted to the cache proper, overwhelming the filter and causing capacity misses. In contrast, permissive filtering may promote too many blocks to the main cache, thus increasing the number of conflict misses, and degrading the performance.

This section is therefore dedicated to evaluating the effect of the filtering probability on the partitioning of references. The selected parameters are then used to evaluate performance and power consumption in Section V-C.

1) Impact on Miss-Rate: First, we address the effects of filtering on the overall miss-rate (fraction of blocks missed by both the cache *and* the filter) in order to determine the sampling probabilities that yield best cache performance. Fig. 10 shows the distributions of changes in the miss-rate, when adding filtering to a 16KB direct-mapped cache (for example, -40 means that the miss rate dropped by 40%). The data shown for each probability are a summary of the observed changes in miss rate over all benchmarks simulated. Good results should combine a large overall reduction in miss-rate with a dense distribution, i.e. a small differences between the 25%–75% percentiles and min–max values, as a denser distribution indicates more consistent results over all benchmarks.

The figure shows that the best average reduction in data miss-rate is $\sim$25%, achieved for P values of 0.05 to 0.1. Moreover, this average improvement is not the result of a single benchmark skewing the distribution, but rather the entire distribution — as represented by the 25%–75% box — is moved downwards. The same happens with the instruction stream, where selection probabilities of 0.01 to 0.0001 all achieve an average improvement of $\sim$60%.

The fact that a similar improvement is achieved over a range of probabilities, for both data and instruction, indicates 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 0.05 and 0.0005 for the data and instruction streams, respectively, of a 16KB configuration. In a similar manner, probabilities of 0.1 and 0.0005 were selected for the data and instruction streams, respectively, of a 32KB configuration.

Interestingly, the data and instruction streams require widely different Bernoulli success probabilities. The reason for this is that the instruction memory blocks are usually accessed an order of magnitude more times compared to data blocks. This difference is attributed to the fact that instructions tend to be executed sequentially, so instruction memory blocks are mostly read sequentially.

2) Impact on Reference Distribution: Filtering effectively splits the reference stream into two components: one supposedly consisting of long residencies, and another consisting of short transient ones. We now evaluates this separation.

Fig. 11 shows *vortex*'s distributions of references (the fraction of references serviced by residencies of up to a certain length) observed in both parts of the filtered 16K cache, compared with that of a regular

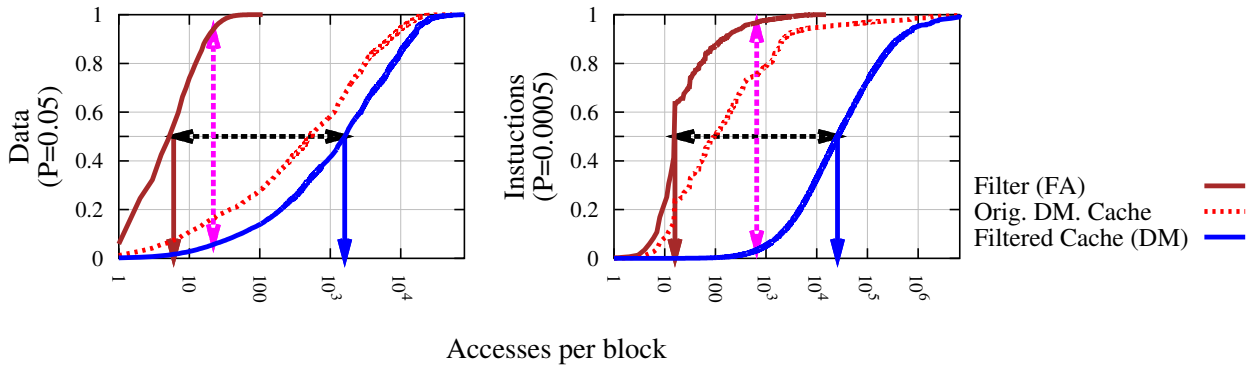

Fig. 11. *The mass distributions of residencies for vortex in both the cache and the filter, compared with the original mass distribution. The horizontal arrows show the median-to-median range, and the vertical double arrow shows the false-∗ equilibrium point.*

16K DM cache. Table III lists the results for all SPEC2000 benchmarks. Residencies that are promoted from the filter to the cache are split appropriately, with references before the promotion counted as a filter residency and those after it as a cache residency.

For the filtered cache, the difference between the resulting distributions is quantified by two metrics: the median-median ratio (horizontal double arrow in Fig. 11) and the false-∗ equilibrium (vertical double arrow). The first metric is the ratio between the median values of the cache and filter distributions (the median values are marked with down-pointing arrows). This quantifies the distance between the two distributions. Invariably, the results show that filter-based residencies are much shorter than those of blocks promoted to the cache proper, which in turn serve the majority of the references. The ratios for data streams are at 50–1000, and average ∼320 (∼50,000 for instruction streams), suggesting random sampling is effective in splitting the reference distribution.

The second metric, called the *false-∗ equilibrium*, quantifies the fraction of false predictions. Any residency length threshold will show up on the plot as a vertical line, with the fraction of the cache's distribution to its *left* indicating the false-positives (short residencies promoted to the cache), and the fraction of the filter's distribution to its *right* indicating the false-negatives (long residencies remaining in the filter). The false-∗ equilibrium is the unique threshold that equalizes the percentages of false-positives and false-negatives.

For example, the false-∗ equilibrium point for *vortex* stands at a residency length of ∼20 and generates ∼6% false predictions. On the other hand, the data stream of the cache unfriendly *mcf* experiences a false prediction rate of ∼22%, which is among the highest values observed. This is caused by the large number of short residencies which swamp the filter leading to a dual effect: they push blocks whose residencies can potentially grow to be long out of the filter, and some are erroneously selected by the random sampling due to their sheer number. Nevertheless, the average false prediction rate is as low as ∼12% for data streams (∼2% for instruction streams), indicating the effectiveness of the proposed technique.

Finally, Fig. 12 compares the percentage of references serviced by the cache proper with the percentage of promoted blocks, for various probabilities. As expected, for low probabilities the fraction of serviced references is much higher than the fraction of promoted blocks, because only frequently accessed blocks are promoted. Higher probabilities also promote shorter residencies, and at some point the promoted residencies are too short to affect cache's hit-rate (indicated by the horizontal line). In our case this saturation occurs around $P = 0.2$ for the data and $P = 0.05$ for the instructions. The probabilities suggested above ($P = 0.05$ for data and 0.0005 for instructions) insert an average of only ∼33% of the data blocks into the cache proper, servicing ∼77% of the data references (∼35% and ∼92% for the instruction streams).

In summary, random sampling indeed effectively splits the distribution of references into two distinct components — one representing frequently used blocks, the other transient ones.

	Data			Instructions		
Benchmark	Medians ratio	False-∗ Equ.	False-∗ @	Medians ratio	False-∗ Equ.	False-∗ @
164.gzip	840	6 / 94	23	5.5 e3	3 / 97	9715
175.vpr	90	12 / 88	12	1.4 e4	2 / 98	10298
176.gcc	480	11 / 89	8	2.2 e4	3 / 97	2438
181.mcf	26	22 / 78	2	8 e5	0 / 100	29946
186.crafty	350	5 / 95	19	1.7 e3	1 / 99	320
197.parser	240	8 / 92	16	4.3 e2	6 / 94	7995
253.perlbmk	240	5 / 95	24	6.3 e2	6 / 94	1327
255.vortex	270	6 / 94	22	1.6 e3	3 / 97	655
256.bzip2	1900	5 / 95	25	7.3 e3	0 / 100	9591
300.twolf	29	14 / 86	11	1.7 e3	2 / 98	1429
168.wupwise	180	8 / 92	20	9.5 e3	1 / 99	11865
171.swim	1.1	43 / 57	8	1.6 e3	0 / 100	19332
172.mgrid	3.6	12 / 88	14	2.1 e3	0 / 100	13748
177.mesa	600	5 / 95	32	2.1 e3	9 / 91	5918
178.galgel	57	15 / 85	10	6.5 e4	0 / 100	23124
179.art	140	23 / 77	3	4.6 e4	0 / 100	11266
187.facerec	48	15 / 85	16	4.4 e3	0 / 100	12051
188.ammp	58	12 / 88	14	5.9 e3	6 / 94	8462
189.lucas	4.1	26 / 74	8	2.7 e3	0 / 100	31174
301.apsi	840	6 / 94	16	1.8 e3	3 / 97	1307
Average	320	12 / 87*	15	5 e4	2 / 97*	10598
Median	160	12 / 89*	15	3.5 e3	2 / 99*	9653

* False-∗ values calculated independently and thus may not sum up to 100%.

TABLE III

Median-median ratios and false-∗ equilibrium values for all benchmarks reviewed, using sampling probabilities $P = 0.05$ and $P = 0.0005$ for data and instruction streams, respectively.

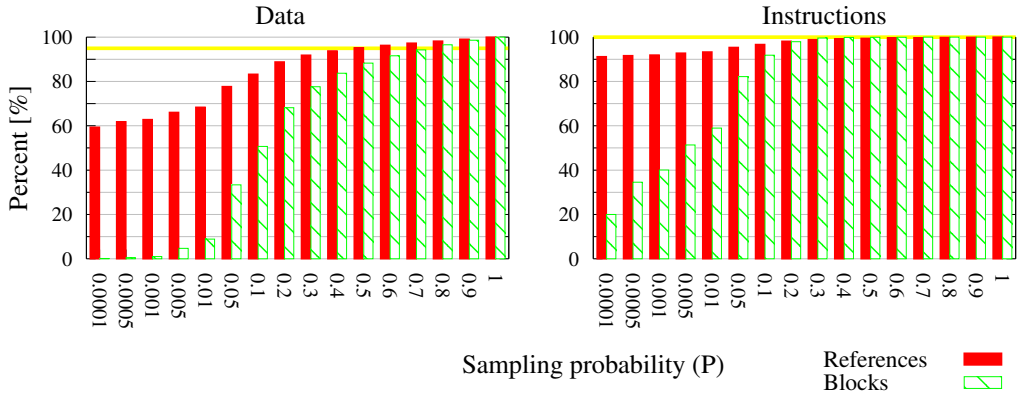

Fig. 12. *Percent of references serviced by the cache vs. the percent of blocks transferred from the filter into the cache for varying sampling probabilities (averages over all benchmarks).*

B. The Wordline Look-aside Buffer

The filter in our dual cache design is a fully-associative caching element. Such elements introduce long access latencies and increased power consumption due to the fully-associative lookups. As shown in Fig. 13 (left), the common implementation uses content-addressable-memory (CAM) for the tag-store, with the wordlines connected to the wordlines of an SRAM block serving as the data-store [34]. But temporal locality suggests the expensive fully-associative lookups may be frequently repeated for a specific block. We therefore propose a *wordline look-aside buffer* (WLB) to cache recent lookup results. The resulting design is shown in Fig. 13 (right).

The WLB consists of a direct-mapped structure, mapping tags of filter-resident blocks to their location in the SRAM data store. The data contained in the WLB for each tag is a bitmap whose width is the

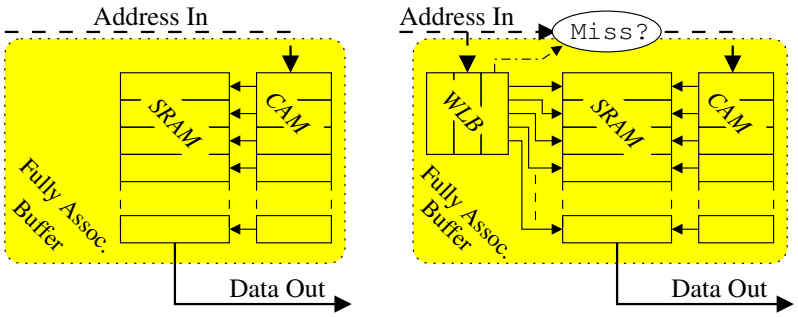

Fig. 13. *Left: common fully associative buffer with a CAM tag-store and an SRAM data-store. Right: using a WLB to cache mappings of recent lookups.*

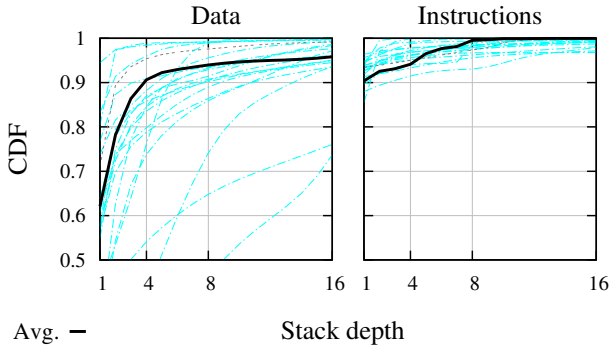

Fig. 14. *Distributions of filter access depth for all SPEC2000 benchmarks, and the average distribution. The vast majority of accesses are focused around the MRU position.*

number of lines in the filter — 32 lines for a 2K filter. This allows for each WLB output bit to be directly connected to an SRAM wordline without a decoder, offering fast, low-power caching of CAM results. In fact, the WLB structure is efficient enough to be accessed in parallel with the cache on every access, totally eliminating the lookup latency on most filter accesses. If the WLB misses, the CAM is accessed, and the result is fed back to the WLB during the ensuing SRAM access, hiding the WLB update latency.

The main design question is the size of the WLB. Fig. 14 shows the stack depth distributions of filter accesses for all benchmarks, as well as the average distribution. The three benchmarks that fall below the main cluster are *swim*, *art*, and *crafty*. While the first two benchmarks are known for their poor temporal locality, it seems the filtered design is very effective at splitting *crafty*'s workload so the vast majority of blocks in the filter exhibit very poor locality (*crafty*'s miss-rate is in fact reduced by ∼50%). Still, it is clear that the vast majority of accesses pertain to recently used blocks: on average ∼94% of data accesses are to stack depths of 8 or less, out of a total of 32 lines in the filter.

Because of the its susceptibility to conflict misses, a WLB consisting of N entries can only *approximate* a stack of depth N. We have therefore explored WLB sizes of 8 and 16 entries. In our experiments, we have found that using an 8 entry WLB achieves an average of ∼78% hit-rate for the data stream (∼83% median) and over 97% for the instruction stream (∼97% median) for a 2K filter. Doubling the WLB size to 16 entries was found to improve its performance by ∼5%, but substantially increases its power consumption. We therefore use an 8 entry WLB in our power and performance evaluation. Given that the hit-rate for the main cache stands at almost 80% for the data stream and over 90% for the instruction stream (Section V-A2), these results indicate that on average only ∼4% of the data references and ∼0.1% of the instruction references still initiate expensive fully-associative lookups.

The WLB demonstrates that temporal locality can be used to greatly reduce a CAM-based filter's power consumption and improve its performance, without losing the fully-associative property.

C. 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 consump-

DL1/IL1 cache		micro-architecture	
size	16/32 KB	fetch / issue / decode	4
line size	64 B	functional units	4
associativity	DM	window size	128
latency	1 cy.*	Load/Store queue	64
filter		branch predictor	
entries	32	meta-predictor with 64K-entry	
associativity	full	bimodal and gshare, and a	
max total latency	5 cy.	similar size meta table.	
CAM latency	3 cy.	4K branch target buffer (BTB).	
SRAM latency	1 cy.	L2 cache	
WLB latency	1 cy.	design	unified
WLB entries	8	size	512 KB
WLB line	32 b	line size	64 B
MRSU data prob. (16K)	0.05	associativity	8
MRSU inst. prob. (16K)	0.0005	latency	16 cy.
MRSU data prob. (32K)	0.1	memory	
MRSU inst. prob. (32K)	0.0005	latency	350 cy.

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

TABLE IV
micro-architecture and cache configurations used in the out-of-order simulations.

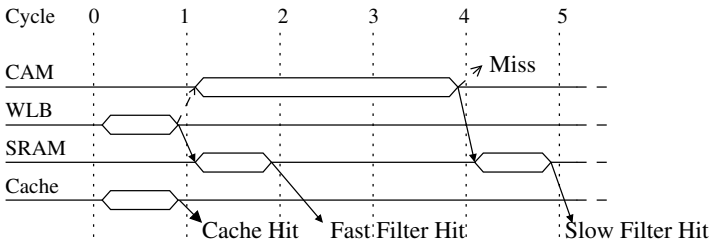

Fig. 15. *Timing diagram of the cache design, based on the μarch parameters listed in Table IV. Down pointing arrows indicate lookup hits, while up pointing arrows indicate misses.*

tion. Augmenting the fully-associative filter with a WLB reduces the overhead incurred by the filter, further improving efficiency. Using the SimpleScalar toolset [1] for out-of-order simulations and CACTI 4.1 [32] for power estimates, we have compared the performance achieved by direct-mapped filtered caches against various set-associative caches. The parameters of the superscalar design used in the simulations are listed in Table IV. CACTI was configured for a 70nm manufacturing process (the finest supported feature size). The *SPEC2000* benchmark suite was used [30] with the *ref* input sets. Benchmarks were fast-forwarded 15 billion instructions to skip any initialization code, and were then executed for 2 billion instructions.

Fig. 15 shows a timing diagram of the different components in the proposed cache design. The main cache and WLB are searched in parallel in the first cycle. If the main cache misses, the result of the WLB lookup determines the filter lookup path: if the requested block is found in the WLB then no CAM lookup is necessary, enabling direct access to the SRAM, and resulting in a 2 cycles total filter latency; this is the normal case. Only if both the direct-mapped main cache *and* the WLB miss, the filter's CAM looked is used (taking 3 additional cycles). The MRSU does not add any latency as it can perform the random sampling even before the data is fetched, enabling it to perform any necessary eviction (either from the cache proper or the filter) beforehand.

Fig. 16 shows the IPC improvement achieved by a random sampling cache over a similar sized 4-way associative cache. The figure shows consistent improvements (up to $\sim$35% for a 16K configuration and $\sim$28% for a 32K one), with an average IPC improvement of just over 10% for both sizes. In no case did the filtered design cause a degradation in performance. While the results are consistent, it is clear that

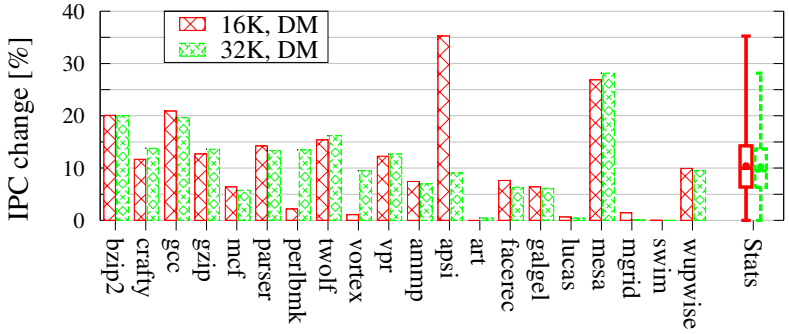

Fig. 16. *IPC improvement for direct-mapped random sampling caches (using a 2K filter) over similar sized 4-way caches.*

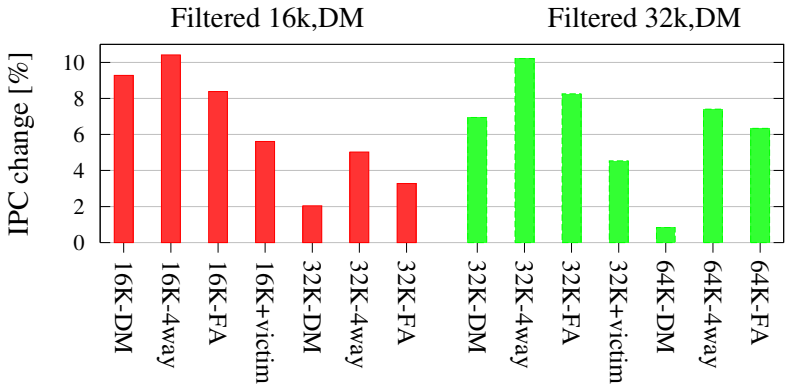

Fig. 17. *Average IPC improvement for 16K and 32K direct-mapped filtered caches over common cache configurations.*

benchmarks suffering from conflict misses enjoy better performance gains. This is most pronounced for *apsi*, where over 70% of the residencies consist of a single reference. Indeed, doubling the cache size to 32K — thus increasing the number of sets and reducing conflicts — decreases its performance gains, while other benchmarks remain largely unaffected.

Fig. 17 compares the average performance achieved by 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 not only compared with similar size set-associative caches, but also compared with larger, more expensive caches: a 16K-DM random sampling cache yields ∼5% higher IPC than a 32K-4way cache, and a 32K configuration outperforms a 64K-4way by over 7%. Likewise, using the extra 2K for a filter yields better performance than using it as a victim buffer, indicating that even 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 is that the direct-mapped cache's low access latency compensates for its higher miss-rate. This is even more evident when considering the larger 32K and 64K caches, where the doubling of the number of cache sets reduces the number of conflicts, thus allowing the direct-mapped cache's lower latency to prevail.

Next, we evaluate the proposed design's power consumption. Using independent random sampling eliminates the need to maintain any previous reuse information, reducing the power consumption calculation to summing the energies consumed by the direct-mapped cache, the fully-associative filter, and the small, direct-mapped WLB, each weighted by the number of accesses it serves.

Fig. 18 shows both dynamic read energy and leakage power consumed by the random sampling cache, compared to common cache configurations (same as those in Fig. 17). Obviously, the power consumed by the random sampling cache is higher than that of a simple direct-mapped cache, because of the filter: up to ∼30% more dynamic energy and ∼15% excess leakage power for a 16K random sampling cache,

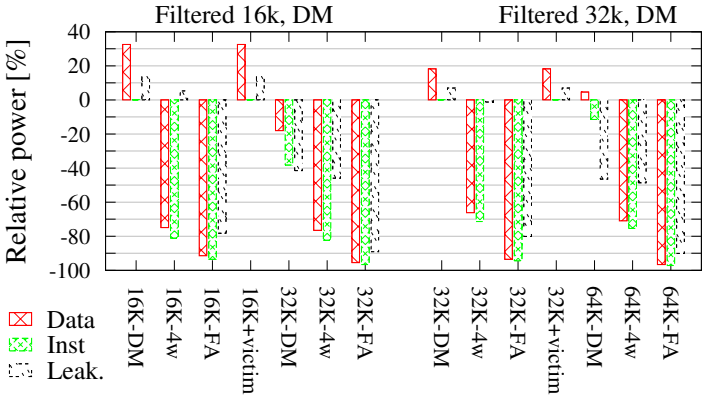

Fig. 18. *Relative power consumption of the random sampling cache, compared to common cache designs (lower is better), for a 70nm process.*

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 consumes 70%–80% less dynamic energy, with only ∼5% more leakage power. The 32K configuration yields 60%-70% reduction in dynamic energy, with no increase in leakage.

An even bigger advantage of random sampling caches is apparent when compared to a set-associative cache double its size: both the 16K and 32K random sampling caches consume 70%-80% less dynamic energy, and 40%-50% less leakage, than 32K and 64K 4-way set-associative caches, respectively, while still offering better performance, as shown in Fig. 17. This suggests that adding just a small buffer and a trivial insertion policy is more efficient than blindly doubling cache size.

VI. Related Work

Several studies have considered the notion of separating memory reference streams into *core* and *transient* groups, and designing mechanisms that serve each class using a different caching policy.

Megiddo and Modha [21] designed the ARC policy for operating systems' buffer caches. New blocks are managed using an LRU list, and moved to an LFU list on the second access. ARC thus favors frequently accessed blocks, yet still dedicates at least half the cache memory to transient data.

In L2 caches, Qureshi et al. [24] describe the LIN replacement policy that takes the cost of a cache miss into account, prioritizing isolated memory accesses that stall the processor. Interestingly, this policy serves some benchmark very well and some very poorly. They therefore propose SBAR, a set dueling policy which maintains dual LRU/LIN tag-stores for a subset of the cache sets, and uses their performance to determine the overall replacement policy. Adaptively chosing among dueling L2 replacement policies was also suggested by Subramanian et al. [31].

In a later study, Qureshi et al. [23] observe that most L2 blocks are never reused, and propose a bimodal insertion policy (BIP) that occasionally inserts blocks to the MRU position rather than to the LRU position. They then again use set dueling to dynamically select the best performing policy.

Rajan and Ramaswamy [25] attempted to approximate optimal replacement by tracking the reference distance of cached blocks. Each L2 cache set is partitioned into a Main Cache (MC) and a Shepherd Cache (SC) that uses FIFO replacement. New blocks are inserted into the SC where the inter-block access distances are tracked. On block evictions, the cache selects the block that is expected to be accessed furthest away from the oldest SC block. A somewhat similar approach was taken by Jaleel et al. [14] in the Re-Reference Interval Predictions (RRIP) eviction policy, which evicts the cache block that will likely be used in the most distant future.

Cho et al. [6] observed that a large portion of memory references target local variables stored on the stack (as was also observed much earlier by Ditzel and McLellan [9]). They then propose a compiler-assisted mechanism that splits the memory stream into local and global accesses, allowing the pipeline to serve local variables from a small local variable cache (LVC).

Lee et al. [18] also target stack references. They show that stack references account for 56% of all memory reference in SPECint2000 benchmarks, and that the vast majority of stack references are to offsets smaller than 8KB of the top-of-stack. They therefore propose to use a dedicated stack value file (SVF), which caches all stack references and thereby reduces memory access latencies, cache accesses, and memory traffic.

In summary, most of the studies described above target non-L1 caches and involve mechanisms that are too complex to be efficiently implemented in L1 caches (Shepherd Cache [25], RRIP [14], set dueling [24], [31]) or rely on the fact that the L2 only views a subset of the memory stream that is filtered by the L1 cache (ARC [21] and LIP/BIP [23]). The few studies that do target L1 cache performance specifically address stack references. In contrast, we present a generic mechanism based on a thorough analysis of L1 memory references, characterization of the mass-count disparity in L1 reference streams, and the definition of the *core working sets* concept, where a small subset of memory blocks dominate the reference stream.

VII. Conclusions

The growing size and power consumption of processor caches, and the replication of caches in multicore designs, motivate renewed efforts to improve cache characteristics. One way to do so is by taking cues from workload patterns. In this vein, the main contributions of this paper are the characterization of the mass-count disparity phenomenon prevalent in L1 cache workloads, and the design of a random sampling filtered cache that employs this statistical phenomenon to filter L1 reference streams. In addition, we introduce the Wordline Look-aside Buffer (WLB) to eliminate the vast majority of expensive fully associative lookups, leading to a win-win design with both improved overall performance and reduced power consumption.

The skewed distribution of memory references is well-known. Our contribution is in the quantification of this effect using metrics for mass-count disparity. This quantification reveals that random sampling of references can be used to effectively identify the core of an application's working set. This mechanism is crucial for our design, as it avoids the need for maintaining historical information that has plagued previous dual cache designs.

Our dual cache design employs a fast, low-power direct-mapped component for the heavily accessed core, and a smaller fully-associative filter for the more transient data. Specifically, we suggest augmenting a 16KB cache with a 2KB filter. The sampling probability used to promote a cache line from the filter to the cache is $P = 0.05$ for the data stream and $P = 0.0005$ for instructions. In implementing the filter, a WLB with only 8 entries was sufficient to avoid $\sim$80% of the lookups in the costly CAM. This design yields up to $\sim$35% improvement in IPC, with an average of $\sim$10% over all benchmarks — better than a double size, 4-way set-associative conventional cache. Moreover, it dramatically reduces the overall power consumption: dynamic power consumption is reduced by $\sim$70%–80% relative to the double-size cache, and leakage by over 40%.

While the various components need to be balanced correctly to achieve good performance, our analysis indicates that the design is not overly sensitive to the design parameters. However, additional research is needed in order to fully capitalize on these ideas. In particular, effectively integrating caches with variable response times into the instruction pipeline is an interesting challenge.

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* (PACT), pp. 255–266, Sep 2005.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] M. Brehob, S. Wagner, E. Torng, and R. Enbody. Optimal replacement is NP-Hard for nonstandard caches. *IEEE Trans. on Computers*, 53(1):73–76, 2004.

[5] 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.

[6] S. Cho, P. C. Yew, and G. Lee. Decoupling local variable accesses in a wide-issue superscalar processor. In *Intl. Symp. on Computer Architecture* (ISCA), pages 100–110, 1999.

[7] P. J. Denning. The working set model for program behavior. *Comm. ACM*, 11(5):323–333, May 1968.

[8] P. J. Denning and S. C. Schwartz. Properties of the working-set model. *Comm. ACM*, 15(3):191–198, Mar 1972.

[9] D. R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. In *Intl. Conf. on Arch. Support for Programming Languages & Operating Systems* (ASPLOS), pages 48–56, 1982.

[10] Y. Etsion. *The Skewed Distribution of Working Sets: Leveraging Randomness for Cache Design*. PhD thesis, School of Computer Science and Engineering, Hebrew University, Oct 2008.

[11] Y. Etsion and D. G. Feitelson. L1 cache filtering through random selection of memory references. In *Intl. Conf. on Parallel Arch. and Compilation Techniques* (PACT), pages 235–244, Sep 2007.

[12] Y. Etsion and D. G. Feitelson. Probabilistic prediction of temporal locality. *IEEE Computer Architecture Letters*, 6(1):17–20, Jan–Jun 2007.

[13] D. G. Feitelson. Metrics for mass-count disparity. In *Modeling, Anal. & Simulation of Comput. & Telecomm. Systems* (MASCOTS), pages 61–68, Sep 2006.

[14] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Intl. Symp. on Computer Architecture* (ISCA), pages 60–71, 2010.

[15] S. Jin and A. Bestavros. Sources and characteristics of web temporal locality. In *Modeling, Anal. & Simulation of Comput. & Telecomm. Systems* (MASCOTS), pages 28–35, Aug 2000.

[16] T. L. Johnson, D. A. Connors, M. C. Merten, and W. mei W. Hwu. Run-time cache bypassing. *IEEE Trans. on Computers*, 48(12):1338–1354, 1999.

[17] 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* (ISCA), pages 364–373, 1990.

[18] H. H. S. Lee, M. Smelyanskiy, G. S. Tyson, and C. J. Newburn. Stack Value File: Custom microarchitecture for the stack. In *Symp. on High-Performance Computer Architecture* (HPCA), pages 5–14, 2001.

[19] M. O. Lorenz. Methods of measuring the concentration of wealth. *Publications of the American Statistical Association.*, 9(70):209–219, Jun 1905.

[20] S. McFarling. Cache replacement with dynamic exclusion. In *Intl. Symp. on Computer Architecture* (ISCA), pages 191–200, 1992.

[21] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Technologies* (FAST), 2003.

[22] G.-H. Park, K.-W. Lee, J.-H. Lee, T.-D. Han, and S.-D. Kim. A power efficient cache structure for embedded processors based on the dual cache structure. In *Workshop Languages, Compilers, and Tools for Embedded Systems* (LCTES), pages 162–177. Springer Verlag, 2000.

[23] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Intl. Symp. on Computer Architecture* (ISCA), pages 381–391, 2007.

[24] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *Intl. Symp. on Computer Architecture* (ISCA), pages 167–178, Jun 2006.

[25] K. Rajan and G. Ramaswamy. Emulating optimal replacement with a shepherd cache. In *Intl. Symp. on Microarchitecture*, pages 445–454, 2007.

[26] J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Intl. Conf. on Parallel Processing* (ICPP), volume 1, pages 154–163, 1996.

[27] J. Sahuquillo and A. Pont. Splitting the data cache: A survey. *IEEE Concurrency*, 8(3):30–35, Jul–Sep 2000.

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

[29] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[30] Standard Performance Evaluation Corporation. SPEC2000 benchmark suite. http://www.spec.org.

[31] R. Subramanian, Y. Smaragdakis, and G. H. Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *Intl. Symp. on Microarchitecture*, pages 385–396, 2006.

[32] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, HP Laboratories, Palo Alto, June 2006. http://quid.hpl.hp.com:9081/cacti/.

[33] S. J. Walsh and J. A. Board. Pollution control caching. In *Intl. Conf. on Computer Design* (ICCD), pages 300–306, 1995.

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

BIOGRAPHIES

Yoav Etsion is an Assistant Professor (Senior Lecturer) at the Technion - Israel Institute of Technology, where he is a member of both Electrical Engineering and Computer Science faculties. He received his PhD in Computer Science from the Hebrew University of Jerusalem in 2009, and then served as a Senior Researcher at the Barcelona Supercomputing Center (BSC-CNS). His research interests include high-performance computer architectures, computer systems, and HW/SW interoperability.

Dror Feitelson received the PhD in Computer Science from the Hebrew University of Jerusalem in 1991, and then worked on parallel systems at the IBM T.J. Watson Research Center. He joined the faculty of the School of Engineering and Computer Science at Hebrew University in 1995, and heads the experimental systems lab. His research interests include performance evaluation, and especially the

effect of workloads on system performance, and software evolution in open-source systems. He is the founding co-organizer of JSSPP, the annual series of workshops on Job Scheduling Strategies for Parallel Processing, and maintains the Parallel Workloads Archive.