Probabilistic Prediction of Temporal Locality

Yoav Etsion Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem, Israel
{etsman,feit}@cs.huji.ac.il

***Abstract*—The increasing gap between processor and memory speeds, as well as the introduction of multi-core CPUs, have exacerbated the dependency of CPU performance on the memory subsystem. This trend motivates the search for more efficient caching mechanisms, enabling both faster service of frequently used blocks and decreased power consumption. In this paper we describe a novel, random sampling based predictor that can distinguish transient cache insertions from non-transient ones. We show that this predictor can identify a small set of data cache resident blocks that service most of the memory references, thus serving as a building block for new cache designs and block replacement policies. Although we only discuss the L1 data cache, we have found this predictor to be efficient also when handling L1 instruction caches and shared L2 caches.**

I. Introduction

The increased dependence of modern processors on their memory system is driving a quest to find new methods to identify temporal locality, methods that are more accurate than the prevalent *stack depth* and its derivative mechanisms such as the *LRU* replacement policy.

In this paper we introduce the concept of a *core working set* — a small subset of memory blocks that service the majority of memory references — and describe a novel predictor that determines whether a cache-residing block is part of this core based on *independent random selections*. The independent selections eliminate the need to maintain any past-use information. The core working set concept and the predictor's design are based on analyzing L1 data memory references, and showing they can be characterized using a statistical phenomenon called *mass-count disparity* [5]. Specifically, this phenomenon stems from the known observation that memory usage is highly skewed, with most references directed at a relatively small subset of the address space; it is described in Sect. II.

The main metric we use is the number of references a block is likely to serve while in the cache, which is denoted as the *cache residency length*. The predictor classifies these into longer, non-transient residencies, and short, transient ones — corresponding to residencies of blocks that are part of the core working set, and residencies of non-core blocks. This is done using random selection of memory references, as described in Sect. III. It is compared with related work in Sect. IV.

The concepts presented in this paper were evaluated using cache traces of SPEC benchmarks, generated using the *SimpleScalar* toolset [1] for 16K, 4-way set-associative L1 data caches. All benchmarks were executed with the *ref* input set for 2×10^9 instructions, after fast-forwarding 15×10^9 instructions to skip any initialization code. Despite only describing results for data caches, the predictor was also found effective for instruction caches and shared caches.

II. The Skewed Distributions of Memory Accesses

Temporal locality of reference is one of the best-known phenomena in computer workloads [3]. But this is actually the result of two distinct properties: that references to the same address tend to come in batches, and that some addresses are much more popular than others [9]. These more popular addresses can be grouped together to form the *core working set* — a subset of the classic working set definition [3] whose cache residencies naturally serve the majority of references. Blocks that are accessed only a few times and are not part of this core will be called *transient*.

A good way to visualize skewed popularity is by using mass-count disparity plots [5]. These plots superimpose two distributions. The first, which is called the *count* distribution, is a distribution on blocks, and specifies how many times each block is referenced. Thus $F_c(x)$ represents the probability that a block is referenced x times or less. The second, called the *mass* distribution, is a distribution on references; it specifies the popularity of the block to which the reference pertains. Thus $F_m(x)$ represents the probability that a reference is directed at a block that is referenced x times or less.

A problem with the above definition is that it considers *all* the references to each block, throughout the duration of the run. But the relative popularity of different blocks may change in different phases of the computation, so the instantaneous popularity may be more important for caching studies. Our solution is therefore *not* to count all the references to each block, but to count only the number of references made during a single *cache residency*. Thus, if a certain block is referenced 100 times when it is brought into the cache for the first time, is then evicted, and finally is referenced again for 200 times when brought into the cache for the second time, we will consider this as two cache residencies containing 100 and 200 references, respectively, rather than as a single residency of 300 references.

Returning to mass-count disparity plots, the disparity refers to the fact that the graphs of the count and mass distributions are quite distinct. An example is shown in Fig. 1, showing the mass-count disparity for 4 SPEC 2000 benchmarks, one of which (mcf) is known for its poor cache utilization. The divergence between the distributions can be quantified by the joint ratio [5], 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, for example, the joint ratio is approximately 13/87 (double-arrow at middle of plot). This means that 13% of the cache residencies, and more specifically those instances that are highly referenced, service a full 87% of the references, whereas the remaining 87% of the residencies service only 13% of the

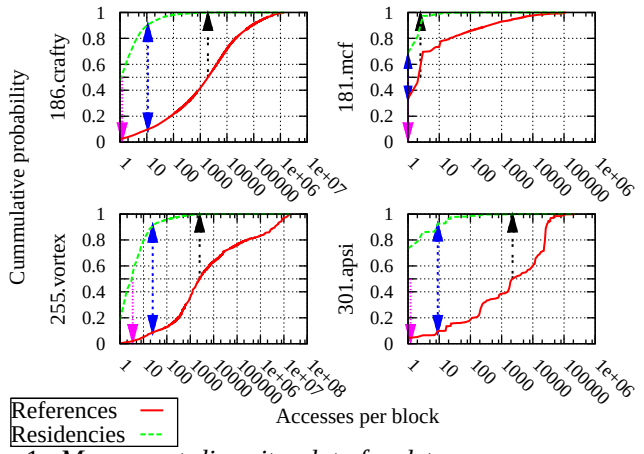

Fig. 1: *Mass-count disparity plots for data memory accesses in select SPEC benchmarks. The arrows demonstrate the $W_{1/2}$, joint-ratio, and $N_{1/2}$ metrics of mass-count disparity.*

TABLE I: *The $N_{1/2}$ and $W_{1/2}$ metrics values for L1 data streams for the 20 SPEC 2000 benchmarks used.*

Benchmark	$W_{1/2}$	$W_{1/2}$@	$N_{1/2}$	$N_{1/2}$@
gzip	3.22	1	0.01	6005
vpr	4.40	2	0.08	1606
mcf	24.47	1	15.99	2
crafty	2.49	1	0.13	1932
parser	3.21	3	0.10	3441
perlbmk	1.82	4	0.51	2277
vortex	1.75	3	0.19	2528
bzip2	1.35	1	0.00	52757
twolf	6.78	3	1.78	60
wupwise	4.44	15	0.00	11521092
swim	39.49	10	37.80	10
mgrid	11.51	9	14.55	31
mesa	1.50	15	0.36	13377
galgel	11.98	2	0.61	198
art	21.75	1	15.73	3
facerec	2.30	2	0.05	17534
ammp	5.17	3	0.24	444
lucas	21.91	7	10.24	15
apsi	3.04	1	0.06	2232
Average	9.25	4.6	4.92	582479
Median	4.44	3	0.24	2232

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 residency (one that is accessed from 100 to millions of times).

More important for our work are the $W_{1/2}$ and $N_{1/2}$ metrics [5]. 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 ∼1.7% 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 long residencies needed to account for half the references is quantified by the $N_{1/2}$ metric; for vortex it is less than 1% (right up-pointing arrow). Table I lists the measured $W_{1/2}$ and $N_{1/2}$ data for the 20 SPEC 2000 benchmarks used, along with the maximal residency length of the blocks accounting for $W_{1/2}$, and the minimal residency length of the blocks accounting for $N_{1/2}$ (marked by the @ value). For vortex, the table reveals that the 50% of the data cache residencies are accessed up to 3 times, and that 50% of vortex's references are serviced by ∼0.2% of the residencies, each accessed over 2500 times. All-in-all, the table reveals that half of the data references are serviced by less than 1% of all residencies, in 15 of the 20 benchmarks inspected.

On the other hand, the disparity is less apparent for 5 benchmarks including mcf: almost 98% of its residencies consist of no more than 5 references, but still compose over 70% of the references. This is manifested in a joint ratio of 33/66, and relatively high $W_{1/2}$ and $N_{1/2}$ values — the weight of half the residencies ($W_{1/2}$) is 25% of the mass, and the 16% longest residencies are required for half the mass ($N_{1/2}$). However, since the longest 2% of the residencies still compose 30% of the mass, mcf still exhibits some degree of disparity.

The existence of mass-count disparity demonstrate that the working set is not evenly used but is rather focused around a *core*. This has important consequences regarding random sampling. Specifically, if you pick a residency 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 [14]. 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 working set can improve the efficiency of caching mechanism, and the nature of this core allows it to be identified using random sampling. This observation is the focus of this paper.

III. Identifying the Core Working Set

The basic goal of a residency length predictor is to identify the residencies that are likely to be long. The optimal approach would be to simply count the number of references made to each block in the cache — i.e. the length of each residency — and classify the residency as *long* once it passes some threshold. Fig. 1 indicates that even an arbitrary threshold around 100 references-per-residency would suffice to identify a small subset of residencies that service the majority of references for most benchmarks. However, this naive design is costly as it maintains a 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 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 belonging to the core (at least until the corresponding block is evicted), 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.

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 pseudo random number generator — a simple linear-feedback shift register, for example.

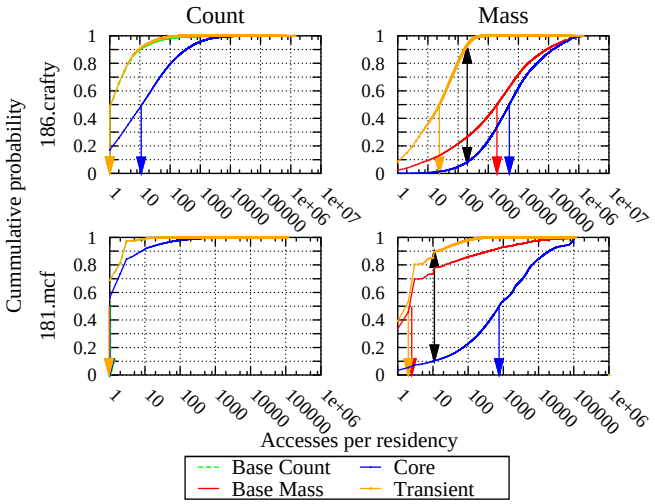

Fig. 2: *The distributions characterizing selected and non-selected L1 data cache residencies, with selection probability* $P = 0.01$. *The downward arrows indicate the median points.*

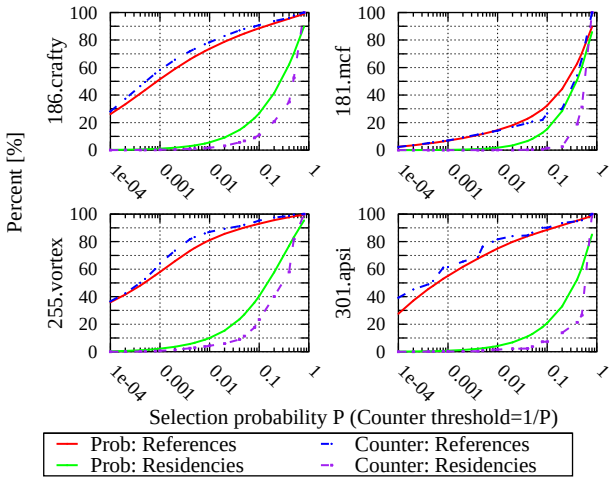

Fig. 3: *Fraction of blocks sampled by the probabilistic predictor and the percent of memory references they service, compared to those of the optimal counter-based predictor.*

The predictor divides the set of all residencies into two: those that are classified as core, and those that retain the transient label. In effect, the probabilistic classification also splits every core residency in two, representing the references made to the residency *before* and *after* the random selection. One way to analyze the predictor is by comparing the distributions of references made to residencies in the two groups (where the transient group includes both residencies that are not selected and the pre-selection part of those that are).

Fig. 2 shows the distributions of residencies (count) and the number of references they service (mass) for each class, using $P = 0.01$, for two benchmarks. The distributions and their median values are compared to the base distributions of residencies and references from Fig. 1. Note that the base distribution of residencies practically overlaps that of the residencies classified as transient, whereas the base distribution of references resembles that of the residencies classified as core (at least for cache-friendly benchmarks). This is another manifestation of the mass-count disparity phenomenon.

The resulting distributions show a good correlation between the residency's length and whether it was classified as core, with residencies classified as transient likely to be shorter than those marked core (left of figure): less than 10% percent of *crafty*'s residencies that are classified as transient consist of more than 10 references, as opposed to over 50% of residencies classified as core. Furthermore, some 92% of *crafty*'s references that are serviced by residencies classified as core are indeed served by residencies longer than 200 references (middle double-arrow). In contradistinction, only $\sim$7% of the references serviced by residencies that are classified as transient actually reference residencies longer than 200.

Random sampling even yields reasonable results for the cache-unfriendly mcf benchmark: Although 90% of the residencies classified as core are shorter than 10 references, they only account for 10% of the core's mass. The other 90% of the mass is composed of residencies longer than 10 references. These 90% of the core references in fact cover over 60% of mcf's overall reference that target residencies longer than 10 references. Furthermore, the average residency length for mcf's entire stream is $\sim$2, and under 1.8 for the transient residencies — as oppose to 16.8 for the core residencies. Thus random sampling was effective at identifying the core working set to the degree that such a core exists at all.

Fig. 3 compares the probabilistic predictor to the optimal (naive) counting approach, by showing the percentage of residencies classified as core and the references they service. The X-axis equates a sampling probability of P with a counting threshold of $\frac{1}{P}$. When analyzing the percent of references serviced we see a very good correlation to those serviced by the optimal predictor, at least for P up to 0.01. For example, with $P = 0.01$ the sampling predictor covers over 93% of the number of references covered by the optimal predictor for crafty. This good correlation stems from the fact that both predictors only select a very small percentage of the residencies, usually just a few percents. But when P is relatively high, we get too many false positives where transient residencies are classified as core (residencies shorter than 15 references constitutes some 90% of all residencies in the benchmarks shown in Fig. 1). These residencies are also the reason why the probabilistic sampling predictor sometimes seems to serve more references than the optimal predictor (this is true for mcf, where the coverage at $P = 0.01$ is almost 103%). This implies that $P = 0.01$ is a good operating point, a result that was consistent for all benchmarks analyzed.

Summing over all the residencies, Table II shows how many are classified as core and how many references they service. By sampling only 0.1% of the references we select on average $\sim$1.3% of the residencies, and cover over 50% of the references. As the average is highly affected by benchmarks known for their poor temporal locality, such as *swim*, *art*, and *mcf*, we also show the median values, demonstrating a coverage of over 60% of the references.

In conclusion, the probabilistic predictor is shown to be very effective in distinguishing between transient and core cache residencies, thus approximating the optimal counting predictor.

TABLE II: *Percents of residencies (insertions) classified as core and the references they service, for $P = 0.001$ and $P = 0.01$.*

	$P = 0.001$		$P = 0.01$	
Benchmark	%Ins	%Refs	%Ins	%Refs
gzip	0.69	60.21	4.93	75.47
vpr	0.87	57.33	6.17	70.58
gcc	1.12	66.63	9.04	73.78
mcf	0.19	9.29	1.76	18.09
crafty	1.02	62.68	5.60	81.25
parser	1.14	65.66	6.90	79.72
perlbmk	3.05	68.01	13.66	87.22
vortex	2.19	69.16	9.96	88.33
bzip2	1.14	73.32	7.41	85.38
twolf	1.05	35.10	7.66	55.46
wupwise	1.64	77.54	14.45	82.01
swim	1.12	3.89	10.68	13.95
mgrid	1.56	21.55	13.00	40.77
mesa	4.72	87.24	19.19	94.28
galgel	0.58	29.04	3.86	57.06
art	0.24	20.82	2.38	23.53
facerec	1.17	66.80	9.43	74.19
ammp	1.00	49.24	7.39	65.20
lucas	0.79	31.72	7.44	39.24
apsi	0.73	64.93	4.14	82.29
Average	1.30	51.01	8.25	64.39
Median	1.12	62.68	7.44	74.19

IV. Related Work

Predicting temporal locality — and specifically identifying the core working set — is an integral part of every block replacement policy and cache filtering mechanism. Either implicitly or explicitly, several predictors have been discussed as part of the research in cache design.

Variations of the optimal predictor were used by Sahuquillo and Pont [13] and by Rivers and Davidson [12], at a price of maintaining an access counter for each cache line. González at al. used a stride predictor to identify spatial and temporal locality, for a dual-cache structure [6]. Karlsson and Hagersten found that the number of cache replacements between a block's last use and its eviction is fairly stable, and checked whether a block's reuse distance is smaller [10]. Jalminger and Stenström targeted similar goals using a structure similar to a two level branch predictor [7]. In a different approach, Tyson et al. implemented non-transiency prediction by identifying load/store operations that are likely to cause a cache miss, using a cache bypass for the fetched data [15]. Temporal locality predictors are also used in buffer-caches, with relaxed resource consumption limitations thereby making them infeasible for use in hardware caches (for example, the LIRS predictor is based on an inter-reference recency metric [8]).

The only study that used some form of sampling was done by Behar et al. [2]. As programs are known to spend most of the time executing a small portion of the code, they observed that generating only 1 in 10 traces for the trace cache does not hinder performance and reduces the power consumption of the trace generating unit. This 90/10 effect indicates that mass-count disparity is also common in trace generation. Naturally, they do not address data caches.

All but the last study rely on maintaining block usage history information, requiring additional storage with its implications of power and timing. In contradistinction, our design relies on a simple probabilistic phenomenon observed on reference streams, enabling it to utilize pure random sampling.

V. Conclusions

In this paper we explore the mass-count disparity of memory references, where the vast majority of *references* are serviced by a very small fraction of all cache residencies, and the majority of *residencies* serve only very few references. This even applies to cache-unfriendly benchmarks.

Harnessing this phenomenon, we have designed a predictor that classifies cache residencies based on their expected length. The predictor uses independent random selection of references with a low probabilty (e.g. $\frac{1}{100}$), thereby mostly selecting long residencies (in 20 SPEC benchmarks, it selected an average of $\sim$8% of the residencies that service $\sim$64% of all references). The use of independent selection eliminates the need to maintain any past-use information. This also enables easy integration with other predictor types, such as those addressing memory level parallelism and the criticality of specific references for performance [11].

Extending this work, we have successfully used random sampling to preferentially insert core residencies into the cache [4]. The proposed design services core residencies from a direct-mapped cache, and transient ones from a small filter. By utilizing the direct-mapped cache's low-latency and low-power traits, while eliminating most conflict misses, this design achieves better performance and consumes less power than an equal size set-associative cache.

In future work we intend to explore the use of random sampling for the design of shared L2 caches, attempting to reduce cache pollution caused by transient residencies. Preliminary experiments show promising results.

References

[1] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

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

[3] P. J. Denning, "The locality principle," *Comm. ACM*, vol. 48, no. 7, pp. 19–24, Jul 2005.

[4] Y. Etsion and D. G. Feitelson, "Probabilistic cache filtering," School of Comp. Sci. and Eng., Hebrew University, TR-2007-12, Apr 2007.

[5] D. G. Feitelson, "Metrics for mass-count disparity," in *Modeling, Anal. & Simulation of Comput. & Telecomm. Systems*, Sep 2006, pp. 61–68.

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

[7] J. Jalminger and P. Stenström, "A novel approach to cache block reuse predictions," *Intl. Conf. on Parallel Processing*, p. 294, 2003.

[8] S. Jiang and X. Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Intl. Conf. on Meas. & Model. of Computer Systems*, Jun 2002, pp. 31–42.

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

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

[11] 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*. Jun 2006, pp. 167–178.

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

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

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

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