The Skewed Distribution of Working Sets:

Leveraging Randomness for Cache Design

Thesis submitted for the degree of

"*Doctor of Philosophy*"

by

Yoav Etsion

Submitted to the Senate of the Hebrew University

October / 2008

This work was carried out under the supervision of

Prof. Dror G. Feitelson

Acknowledgements

To my beloved Adina, for the loving support and never ending acceptance, and for never laughing out loud when I said I’m simply thinking about my research with my eyes closed.

To my advisor Dror Feitelson (who I still suspect was not in his right mind to take me as a student), for being a true role model and mentor, and for giving real meaning to the title of an academic father.

To my parents, for believing despite what all the teachers said.

And finally to my feisty son Yotam: although you only recently joined the team, your smile makes all the difference, redhead.

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.

This thesis explores the temporal locality phenomenon in an effort to devise such efficient caching mechanisms. Specifically, it is shown that while Denning's working sets model puts all memory blocks in a working set on an equal footing, a dramatic difference in fact exists between the usage patterns of frequently used data and those of lightly used data. This thesis therefore extends Denning's definition with the *core* working sets model, employing predicates to identify the most important subset of blocks in a working set.

This model forms the base for a probabilistic predictor that can distinguish transient cache insertions from non-transient ones. It is shown that this predictor can identify a small set of data cache resident blocks that service most of the memory references. This predictor is then used in the design of an L1 dual-cache that inserts only frequently used blocks into a low-latency, low-power direct-mapped main cache, while serving the rest of the blocks from a small fully-associative filter. The design further employs a novel, low-latency, low-power fully-associative element, that uses a small direct-mapped lookup table to cache recently accessed blocks in the filter — thereby eliminating most of the costly fully-associative lookups.

This L1 dual-cache design demonstrates that a 16K direct-mapped L1 cache, augmented with a fully-associative 2K filter, can outperform a 32K 4-way cache, while consuming 70%-80% less dynamic power and 40% less static power.

Contents

Chapter 1

Introduction

1.1 Background

The notion of a memory hierarchy is one of the oldest and most ubiquitous in computer design, dating back to the work of von Neumann and his associates in the 1940's. The idea is that a small and fast memory will cache the most useful items at any given time, with a larger but slower memory serving as a backing store [27, 32, 70].

As the essence of caching is to identify and store those data items that will be most useful in the immediate future [6], caches need to predict which items will be used in the immediate future. These predictions ubiquitously rely on the principle of locality, which states that at any given time only a small fraction of the whole address space is used, and that this used part changes relatively slowly [19]. Denning formalized this using the notion of a *working set*, defined to be those items that were accessed within a certain number of instructions. The goal of caching is thus effectively to keep the working set in the cache.

The effectiveness of caching thus relies on the existence of working sets, as these represent a subset of all memory blocks that are in current use. But working sets are not homogeneous, as some memory blocks are more popular than others [35]. This heterogeneous nature of working sets raises the question whether caches should employ a heterogeneous design rather than the common random-access paradigm, as processors can benefit if the more popular blocks in the working set (and in general) are treated beneficially by the cache by serving them faster. One way to give preferential treatment to the more important data elements is to use a *dual* cache structure. Such structures partition the cache into two parts, and use them for data elements that exhibit different access patterns [65][1]. In many cases, data elements can also move from one part to the other. For example, data may first be stored in a short-term buffer, and only data that is identified as important will be promoted into the long-term cache. The identification of a certain item as important can be done based on the references it received while in the short-term buffer: if it is referenced again and again, it is identified as part of the core and promoted. The concept of dual-cache designs has been extensively explored in the past, specifically in the context of filtering memory references [12, 33, 36, 37, 39, 40, 46, 62, 64, 65, 63, 79]. But while dual-cache designs offer flexibility, the filtering of memory blocks has proven difficult as it ordinarily requires maintaining information about memory blocks' previous usage patterns in order to predict future use. The mechanisms required to implement the filtering, and specifically those involved with maintaining past reuse information, have thus been deemed largely impractical, due to the number of transistors they require (with the corresponding die area) and the power they consume.

The increasing gap between processor and memory speeds witnessed in recent years has exacerbated the CPU's dependency on the memory system performance — and especially that of L1 caches with which the CPU interfaces directly. One result of this ongoing trend is the increase in the capacity of L1 and L2 caches, in an effort to bridge the memory-processor gap and improve overall system performance. This improvement, however, also increases the power consumed by the caches — estimated at more than 10% of the overall power consumed by a general purpose CPU [31], and up to 40% for embedded systems [9].

Recent years have seen a shift in processor design, as the increase in CPU clock speeds witnessed for some 30 years have largely ground to a halt, due to several inherent physical limitations [52, 61, 81]: example limitations include wire delays preventing the propagation of fast clock signals uniformly to all parts of the chip; shrinking Silicon features made possible by

[1] We differentiate this from a *split* cache structure, where one part is used for data and the other for instructions, but some authors use the terms interchangeably.

new process manufacturing improvements hindered transistor gate isolation and increase power leakage; and increased clock frequency drove designers to employ longer, more complicated pipelines, thwarting most of the theoretical performance increase made possible by the faster clock frequency.

These shifts in technology have increased processors' power density and have elevated processor power consumption into a major concern. Today, the power-performance tradeoff is ever more important. This trend motivates researchers to design more efficient caches, that can deliver performance while maintaining a power budget. Furthermore, despite predictions, transistor density continued to grow as predicted by Moore's Law [47, 48, 66]. The continued increase in transistor density and the limitations in increasing processor frequency caused the microprocessor industry to focus on on-chip parallelism, available by placing multiple processing cores on a single chip — also known as chip multiprocessors (CMP) [22, 25, 41]. This in turn, has made the power consumption of caches an even bigger concern, as multi-core CPUs typically replicates the L1 caches for each core to avoid the design complexities of multiple processors sharing an L1 cache [49].

The increasing concern regarding cache power consumption, together with the increased memory bandwidth requirements of multiple cores sharing a memory bus [10], have motivated a quest for improved utilization of cache resources through the design of more efficient caching structures. This quest solicits a revisit to existing ideas such as dual cache structures, specifically ones incorporating direct-mapped cache structures. Direct-mapped caches are very appealing in this context, as they are faster and consume less energy than set-associative caches typically used in L1 caches [28, 38]. However, they are more susceptible to conflict misses than set-associative caches, thus suffering higher miss-rates and achieving lower performance. This deficiency led to abandoning direct-mapped L1 caches in favor of set-associative ones in practically all but embedded processors.

The quest for more efficient caches relies on extensive analysis of memory workloads, and the development of new analysis tools enabling a deeper understanding of cache behavior. The rest of this chapter therefore introduces the workload analysis concepts explored in this work (Section 1.2), and gives a peek into the insights gained by the workload analysis, and the ensuing cache design incorporating these insights (Section 1.3).

1.2 The Mass-Count Disparity Phenomenon and Core-Working-Sets

It is well known that memory block popularity is skewed, and some blocks are more popular than others. However, little is known about the scale of this phenomenon and how extreme is the variation in the popularity of the various memory blocks. For caching purposes, the identification of the most popular blocks is of utmost importance as it allows caching mechanisms to make sure these blocks — servicing very many residencies — are cached. Moreover, the exact subset of popular blocks changes during program execution as it passes through different computing phases. Denning attempted to capture this changing subset of blocks in his definition of *working sets* [17].

The need to identify memory usage patterns, therefore, motivated an extensive analysis of memory workloads. This analysis, described in Chapter 3, reveals that the skew in block popularity is even more extreme than thought before, and experiences a statistical phenomenon called the *Mass-Count Disparity*. The phenomenon describes the relationship between the number of memory references serviced by each single memory block, and how *all* memory references are distributed between the different blocks. Effectively, it reveals that the vast majority of memory references are commonly serviced by a tiny fraction of all memory blocks. In addition, the analysis reveals that even the relatively unpopular blocks experience bursty access patterns.

These results suggests that locality, usually regarded as a combination of two distinct properties — locality in time and locality in space — is also a manifestation of the skewed distribution of the *popularity* of different memory blocks, where some blocks are accessed much more frequently than others. In fact, it may be possible to partition the working set into two sub-sets: those memory blocks that are very popular and are accessed at a very high rate, and those that are only accessed intermittently. This distinction is antithetical to Denning's definition which puts all items in a working set on an equal footing, and lies at the heart of our definition of the *core* of the working set.

The notion of a core leads to the realization that not all elements of the working set are equally important. As the elements in the working set are not accessed in a homogeneous manner, treating all the elements of the working set equally may lead to sub-optimal performance. Rather, it may be beneficial to try to identify the more important core elements, and give them preferential treatment.

The notion of a popular core leads to the formal definition of a *Core Working Set*, described in Chapter 4 as an extension to Denning's working set. By using logical predicates to identify this highly popular subset of the entire working set, core working sets devise a formal framework serving designers of caching mechanisms to explicitly express their notion of the working set's core, that is to be treated beneficially by their design. In addition, this framework enables designers to compare and contrast the mechanism's performance against the formal definition of their intended core.

1.3 Leveraging Randomness for L1 Cache Design

The existence of a small core that governs the majority of memory references is described by the mass-count disparity phenomenon. But the phenomenon also implies the opposite — that the majority of memory blocks only service a small fraction of all memory references. These two consequences of the mass-count disparity phenomenon lend themselves to the application of simple, stateless, random sampling in order to partition the reference stream. As most memory blocks are accessed a small number of times, picking a block at random will likely select a block that is rarely accessed. In fact, this is the reason why a random eviction policy yields fairly good results [70, 67]. But on the other hand, since most memory references are serviced by a small fraction of the working set, a randomly selected *memory reference* likely pertains to a very popular block.

This observation is the corner stone for the probabilistic popularity predictor presented in Chapter 5. In turn, the probabilistic predictor is used in the design a random sampling L1 filtered cache, described in Chapter 6, that uses simple coin tosses to preferentially insert only frequently used blocks into the cache — composed of a fast, low-power direct-mapped structure — that services the majority of memory reference. The rest of the references are serviced from the cache's filter — a small fully-associative auxiliary structure — thus reducing the number of conflict misses in the direct-mapped cache. This mechanism is shown to use a simple filtering strategy to overcome the direct-mapped susceptibility to conflict misses, thereby enabling to harness the speed and low power traits of direct-mapped caches to reduce the overall L1 power consumption, while still improving overall performance.

This is the first successful attempt that employs a simple statistical phenomenon to filter both L1 reference streams efficiently enough to use a direct-mapped structure for L1 caches, thus both reducing power consumption and improving performance.

1.4 List of Publications Composing this Work

The research presented in this thesis is described in the following publications:

- Yoav Etsion and Dror G. Feitelson, **Core Working Sets: Concept, Identification, and Use**.
  Submitted for Publication.
  Also published as *Technical Report 2008-64*, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Jul. 2008.

- Yoav Etsion and Dror G. Feitelson, **L1 Cache Filtering Through Random Selection of Memory References**.
  In *Parallel Architectures and Compilation Techniques (PACT)*, pages 235-244, Sep. 2007.

- Yoav Etsion and Dror G. Feitelson, **Probabilistic Prediction of Temporal Locality**.
  In *IEEE Computer Architecture Letters (CAL)*, 6(1), pages 17-20, May 2007.

- Yoav Etsion and Dror G. Feitelson, **Cache Insertion Policies to Reduce Bus Traffic and Cache Conflicts**.

  Technical Report 2006-4, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Feb. 2006

1.5 Main Contributions of this Work

For brevity, following is a list containing the main contributions of this research.

- **L1 Cache Workload Analysis and the Mass-Count Disparity Phenomenon**

  The foundation of this work is a detailed analysis of L1 cache workloads, and the resulting characterization of the mass-count disparity phenomenon in L1 caches (Chapter 3).

- **Core Working Sets**

  The definition of the predicate based *core working set* framework. This framework extends Denning's definition of working sets to accommodate their heterogeneous nature. The framework enables cache designers to explicitly express their perception of the important subset of the memory blocks (Chapter 4).

- **Probabilistic Block Popularity Predictor**

  A simple application of the mass-count disparity phenomenon offers the use of random sampling of memory reference to probabilistically identify popular memory blocks in a completely stateless fashion, without any need for maintaining past use information (Chapter 5).

- **Random Sampling L1 Filtered Cache**

  The proposed L1 cache design is based on the dual-cache paradigm and uses random sampling to filter out transient blocks and identify the small fraction of popular memory blocks. This partitioning of the working set enables the use of a fast, low-power direct-mapped structure to serve the majority of memory references, thereby improving overall performance and reducing the power consumed by L1 caches (Chapter 6).

- **Wordline-Lookaside-Buffer (WLB)**

  A problem with the fully-associative filter is its access time and power consumption. We alleviate this problem using a small lookup table that harnesses temporal locality to cache expensive fully-associative lookups in a small inexpensive direct-mapped table. The WLB therefore reduces both access times and power consumption of fully-associative caches, without affecting the fully-associative semantics (Chapter 6).

The methodology used throughout the research is discussed in Chapter 2. Following the description of the research itself (Chapters 3 through 6), the body of work in the field is reviewed in Chapter 7.

Chapter 2

Methodology

Extracting and analyzing memory and cache workloads, as well as the evaluation of alternative hardware designs, require software simulations of both existing and proposed architectures. The simulator that was used to carry out all simulations in this work is the *SimpleScalar* simulation toolset [4], simulating the *Alpha AXP* architecture [69].

Accounting for all aspects of the simulated architecture with a full simulation of all implementation details is an arduous task, resulting in excruciatingly slow simulations. The SimpleScalar toolset is therefore broken into several individual, yet co-dependent tools that simulate the underlying hardware with varying degrees of detail, thereby trading off simulation accuracy for speed. Of the different tools, the ones used were:

- **sim-fast** is a functional simulator that only simulates the ISA. By ignoring all implementation details of the underlying architecture it essentially simulates an optimal architecture that incurs no inherent latencies. It is therefore the fastest tool in the toolchain, but cannot be used to measure performance. In our case, *sim-fast* was augmented to collect raw workload statistics that are largely independent on architectural implementation details.

- **sim-cache** is a cache simulator. It can be seen as an extension of *sim-fast* that implements a full cache hierarchy to collect cache statistics such as hit-rate. Although *sim-cache* is oblivious of any statistical artifacts caused by full out-of-order execution, it was found accurate enough for collecting cache statistics. To evaluate cache performance of alternative cache designs, *sim-cache*'s cache module was completely rewritten to support the proposed cache design.

- **sim-outorder** simulates a detailed out-of-order processor. It is the most detailed simulation tool in the SimpleScalar toolset, and is therefore the slowest. *sim-outorder* was used to evaluate the effect of proposed caching mechanism on the overall architecture performance, and specifically on metrics such as IPC (instructions-per-cycle). The modifications to *sim-outorder* include replacing its cache module with one supporting the proposed cache design, and since this design is based on the dual-cache paradigm, its instruction scheduling algorithm was slightly modified to accommodate for variable L1 hit latencies. The original algorithm is based on the common practice in superscalar designs that schedule instructions based on the prediction that the L1 hits [67]. This modification was needed since L1 hits can now hit either in the cache proper or the auxiliary filter, where each has different hit latencies. The algorithm was simply extended to predict that the L1 hits, and the hit is services by the cache proper, with its associated latency — thus extending the common practice in an obvious manner.

The benchmarks used consisted on the *SPEC2000* benchmarks suite [72]. An overall of 20 benchmarks were used in order to accommodate as many different workloads [15]. This constitutes of all but six SPEC2000 benchmarks: *eon*, *gap*, *fma3d*, *sixtrack*, and *applu* failed to either compile or execute, and *equake* experienced too few L1 misses (under 0.02%) on both data and instruction streams to produce meaningful results. All benchmarks were executed with the *ref* input set and were fast-forwarded 15 billion (15×10^9) instructions to skip any initialization code (except for *vpr* whose full run is shorter), and were then executed for another 2 billion (2×10^9) instructions.

Power estimates were compiled using *CACTI*, an integrated cache and memory model that evaluates access time, cycle time, area, leakage, and dynamic power. The version used was 4.1, configured for a 70nm manufacturing process [75] (the finest feature size CACTI supports).

Throughout the thesis, whenever results are shown for all benchmarks, they are summarized using box-plots. The boxes show the 25th, 50th (median) and 75th percentiles over all the benchmarks' results. In addition, these plots include whiskers to show the minimum and maximum values, and a circle marking the average value.

Chapter 3

The Skewed Distribution of Memory Accesses and the Mass-Count Disparity Phenomenon

Mass-Count Disparity is a statistical phenomenon describing a situation where most items in a population are small, but a few are very large (also known as "mice & elephants" in networking [8]). The name *mass-count disparity* comes from the distinctly disjoint nature of two conjoining distributions — that of the sizes of individual elements (count distribution), and how the overall mass is distributed across elements of different sizes (mass distribution). Perhaps the most well-known example of mass-count disparity is an economic one, namely the distribution of wealth in the world [43]: with most people in the world being relatively poor, and only a tiny fraction of the entire population is very rich, the majority of world's wealth (the aggregate sum of all the world population's financial resources) is dominated by a very small fraction of the population.

Formally, given a finite sample space the mass-count disparity phenomenon refers to the interplay of two conjoined yet opposed distributions defined over that space. The first distribution — called the *count* distribution — is a distribution over the individual samples. Thus, $F_c(x)$ represents the probability that a sample has a mass smaller than *x*. Following the economic example, this represents the fraction of the world's population whose individual wealth is smaller than *x* currency units. The second distribution — called the *mass* distribution — is a distribution over the aggregate mass of all individual samples. $F_m(x)$ represents the probability that a mass unit is part of a sample whose total mass is smaller than *x*. In our economic example, this represents the probability that a currency unit belongs to a person whose wealth is smaller than *x* currency units.

The disparity between the two distributions exists when the count probability $F_c(x)$ is likely to be high, but the corresponding mass probability $F_m(x)$ is likely to be low. In our economic example, this is demonstrated in the fact that most of the world population is poor, therefore the count probability indicating the fraction of individuals whose total wealth is smaller than say $100 — $F_c(100)$ — is high. On the other hand, since most of the world's wealth is dominated by rich people, the probability that some arbitrary Dollar of all the money in the world belongs to an individual whose wealth is smaller than $100 — $F_m(100)$ — is very low.

The mass-count disparity phenomenon has interesting implications regarding the identification of the small fraction of samples dominating the mass. The rest of this chapter describes a novel caching efficiency metric — *Cache Residency Length* — and uses this metric to uncover a clear manifestation of the mass-count disparity phenomenon in both data and instruction memory reference streams.

3.1 Cache Residency Length: A New Metric for Rating Temporal Locality of Memory Blocks

Evaluating the relative importance of a memory block for caching purposes requires assessing both its momentary and global popularity. The naive solution would be counting the number of references made to each block, and rating the blocks by their sheer popularity. A possible refinement to this general popularity scaling can be achieved by using a specific window of memory references of a predetermined size, and rating blocks' importance based on their popularity within a window of references.

Either *Block-Popularity* metrics share a major caveat, as they consider *all* the references to

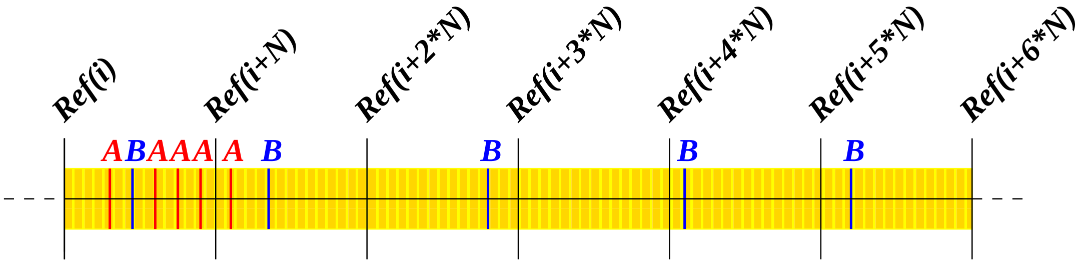

Figure 3.1: *Comparison of the* Cache Residency *metric vs. the more common* Block Popularity *metric. The figure shows the dispersion of accesses to blocks* A *and* B *throughout a window of* $6 \times N$ *memory references. Using the naive block popularity metric puts both blocks on an equal rating. On the other hand, the* Cache Residency Length *metric will separate the two blocks: Block* B *is sparsely used and therefore has a small impact on cache performance, which will likely manifest in a few short residencies. Block* A *is however densely used at the beginning of the window, and is therefore likely to have a long cache residency, corresponding with its bigger impact on cache performance.*

each address, made throughout an arbitrary sized reference window (or simply throughout the duration of the run). But the relative popularity of different addresses may change in different phases of the computation, so the instantaneous popularity may be more important for caching studies. Using arbitrary sized reference windows further amplifies this problem, since the size of the window may determine the outcome of the measurement, as it may or may not be aligned with program phases and instantaneous program memory load.

We therefore propose *not* to use a predetermined window of references, but rather to count the number of references made between a single insertion of a block into the cache, and its corresponding eviction. This is denoted as a *cache residency length*. Thus, if a certain block is referenced 100 times when it is brought into the cache for the first time, is then evicted, and finally is referenced again for 200 times when brought into the cache for the second time, we will consider this as two distinct cache residencies spanning 100 and 200 references, respectively, rather than a single block with 300 references.

Figure 3.1 demonstrates the difference between the two metrics: it shows the use of two blocks that are accessed over a window of $6 \times N$ references, where the references to each block are dispersed differently. While block *A* is densely used only at the beginning of the reference window, block *B* is referenced the same number of times, but these references are dispersed roughly evenly throughout the $6 \times N$ references in the window.

Block *A* reference density indicates it affects cache performance more substantially than block *B*. This is supported by Belady's optimal cache replacement algorithm which always replaces the block whose next use is furthest away in the future [6], indicating block *A*'s reuse frequency will increase its importance to cache performance during the period it is used. The block popularity metric on the other hand evaluates the entire reference window, and will therefore put both blocks on an equal footing with 5 references. In addition, even if we reduce the reference window size to N, the first window $[i \ldots i + N]$ will give a higher rating to block *A* — as it is used 4 times more than block *B* in that reference window — but, it will rate both blocks equally on the second $[i + N \ldots i + 2 \times N]$, although the access to block *A* during that period is likely to hit in the cache as it is a continuation of the sequence of references to it which started during the previous reference window. Such an inconsistency between the block popularity metric's evaluation of a block's importance and its actual impact on cache performance is caused by the arbitrariness of the reference window size and alignment, which does not reflect the program's memory phases.

This example demonstrates that arbitrary sized reference windows, as used by the block popularity metric, may cause incorrect evaluation of the memory workload. On the other hand, the *Cache Residency Length* will likely include all reference to block *A* as a single cache residency, while block *B* will show as multiple shorter residencies. In this manner, this metric incorporates momentary cache load and reference density, and is thus better suited to rate the temporal locality experienced by different blocks.

One deficiency of the cache residency length metric is the it depends on a specific cache configuration, and it is therefore important to remain consistent when comparing results. This thesis uses a 16K direct-mapped configuration as standard, and differing configurations are clearly marked.

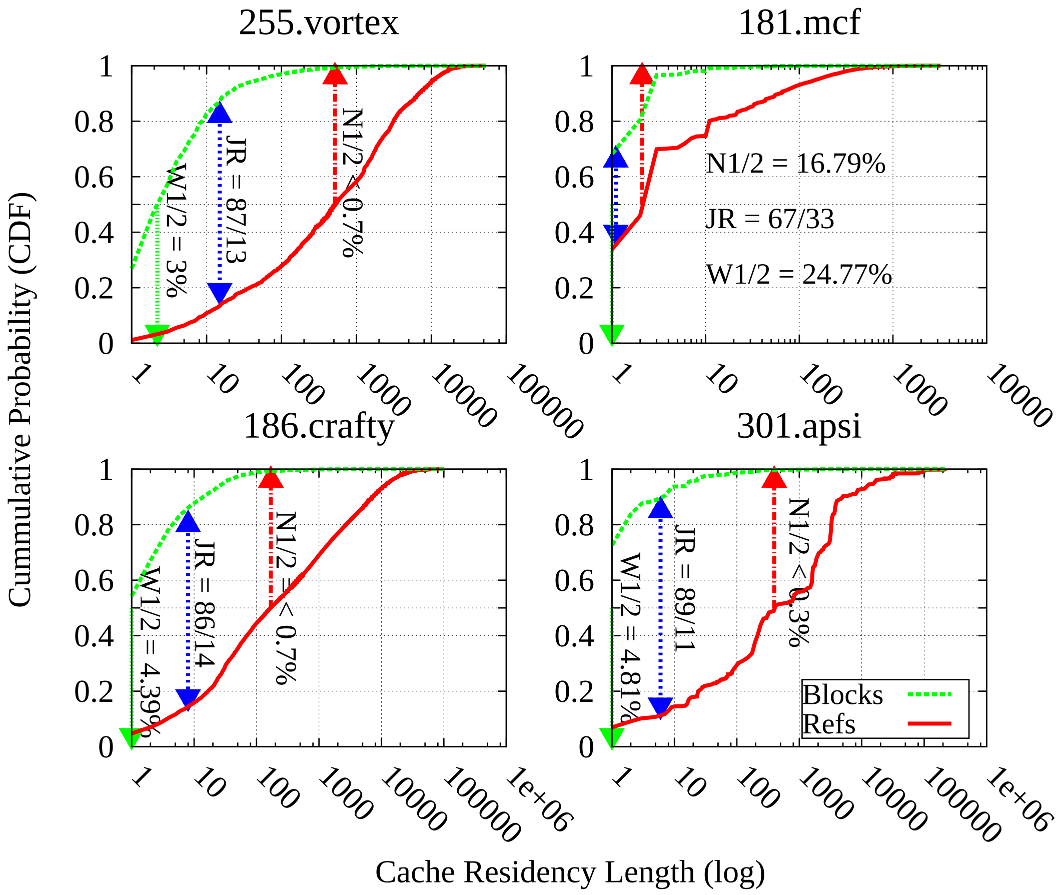

Figure 3.2: *Mass-count disparity plots for* ***data*** *memory accesses in select SPEC 2000 benchmarks. The arrows demonstrate the* $W_{1/2}$*, joint-ratio, and* $N_{1/2}$ *metrics of mass-count disparity.*

3.2 Mass-Count Disparity in L1 Workloads

The *Residency Length* metric enables us to demonstrate the existence of the mass-count disparity phenomenon of memory workloads. In our case, the count distribution $F_c(x)$ represents the probability that a block's residency length is composed of x references or less. The *mass* distribution on the other hand, 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 residency composed of x references or less. The disparity is visualized using mass-count disparity plots [20]. These plots superimpose the two distributions.

The mass-count disparity plots show that the graphs of the count and mass distributions are quite distinct. An example is shown in Figure 3.2, showing the mass-count disparity for

4 SPEC2000 benchmarks, one of which (*mcf*) is known for its poor cache utilization. The divergence between the distributions can be quantified by the joint ratio [20], 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 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 thousands of times).

More important for this work are the $W_{1/2}$ and $N_{1/2}$ metrics [20]. 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 $\sim$3% of the references (left down-pointing arrow). Thus these are instances of blocks that are inserted into the cache but hardly used, and should actually not be allowed to pollute the cache. Rather, the cache should ideally 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 3.1 lists the measured $W_{1/2}$, $N_{1/2}$ and joint-ratio data for the 20 SPEC2000 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 less than 1% of the residencies, each accessed over 500 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.

The disparity is less apparent in benchmarks that are well known for their poor cache utilization such as *mcf*, *art*, *swim* and *lucas*. For example almost 96% of *mcf*'s residencies consist of no more than 5 references, but still they account for over 70% of the references. This is man-

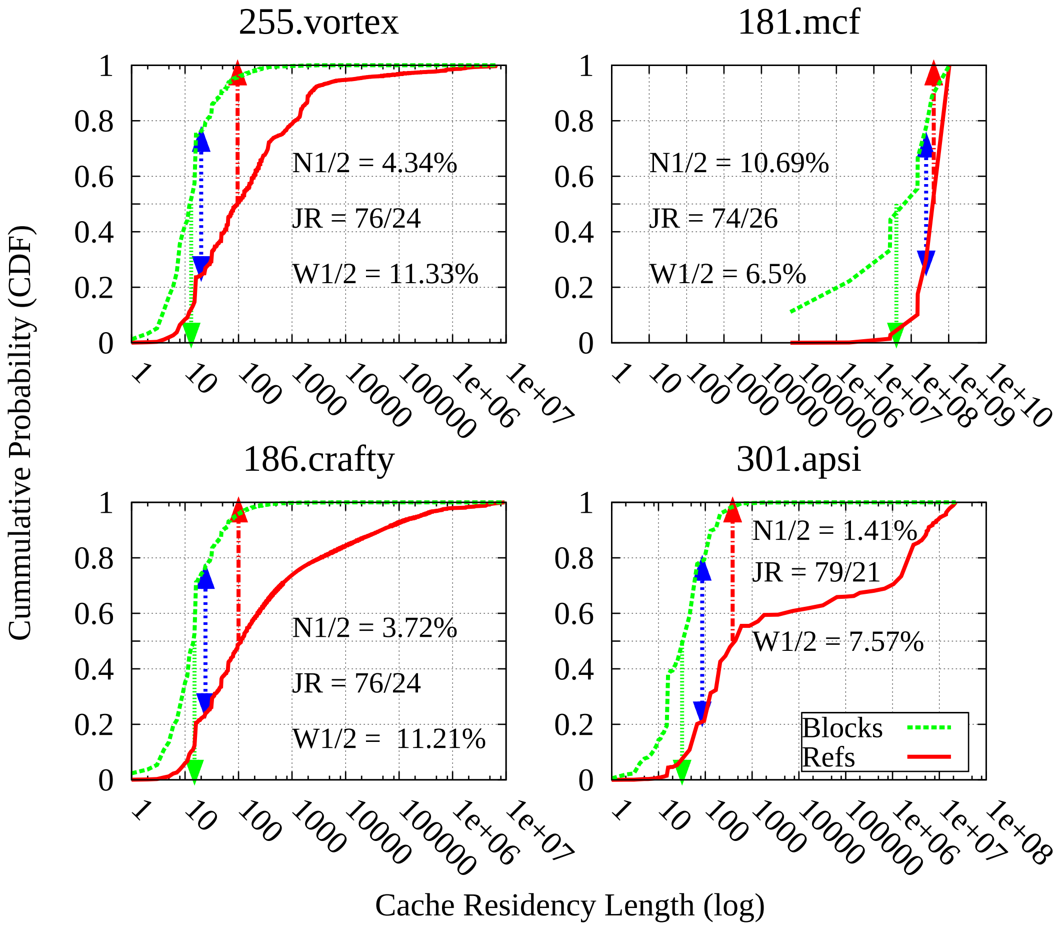

Figure 3.3: *Mass-count disparity plots for **instruction** memory accesses in select SPEC2000 benchmarks. The arrows demonstrate the $W_{1/2}$, joint-ratio, and $N_{1/2}$ metrics of mass-count disparity. Note that the* mcf *code is so dense that it has no residencies shorter than ~50000 references.*

ifested 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 ~17% longest residencies are required for half the mass ($N_{1/2}$). However, since the longest 3% of the residencies still compose 30% of the mass, *mcf* still exhibits some degree of disparity.

Evident in instruction streams as well, the mass-count disparity is not unique do data streams. Figure 3.3 shows the mass-count disparity plots for the instruction streams corresponding to Figure 3.2. We can see that the benchmarks still enjoy a $W_{1/2}$ values of ~11% and less, indicating that the short residencies only service a fraction of all references. Furthermore, the $N_{1/2}$ values are generally even lower, indicating the long residencies dominate the reference stream, and that the major bulk of the references are directed at a small fraction of the working set. The only exception to this is the $N_{1/2}$ value for *mcf* which stands at $\sim$10%. But this actually stems from *mcf*'s exceptional code density that results in a very small number of distinct instruction blocks accessed throughout the execution, which in turn yields a small number of very long residencies — the shortest of which is measured at $\sim 10^5$ references, as shown in Figure 3.3. Is is the small number of residencies that skews *mcf*'s statistics. All in all, the benchmarks enjoy joint ratios of $\sim$75/25 and up, suggesting that even these points — representing an equilibrium between short and long residencies — show a clear separation of the mass and the count distributions. The mass-count disparity exhibited by the instruction streams is in fact a manifestation of the well-known "rule-of-thumb" by which programs spend most of the time executing a small fraction of their code (described as the 90/10 rule by Hennessy and Patterson [27]).

These results demonstrate the existence of a mass-count disparity stemming from the skewed distributions of memory accesses, for both data and instruction streams. While only select benchmarks are discussed here individually, this phenomenon is indeed consistent for all 20 SPEC2000 benchmarks analyzed. For completeness, all data streams' mass-count disparity plots are shown in Figure 3.4, and are accompanied by Table 3.1 listing the corresponding values of the different mass-count metrics. Figure 3.5 and Table 3.2 show the same for the instruction streams.

Although only figures for residencies in a direct-mapped cache are discussed here, the mass-count disparity phenomenon is practically oblivious to cache associativity. Figure 3.6 and the corresponding metrics' values listed in Table 3.3 show the disparity observed for residencies in a 16K 4-way set-associative caches is very similar to that observed on similar size direct-mapped caches. This similarity repeats for the 4-way set-associative instruction stream results, as shown in Figure 3.7 and its corresponding metrics' values listed in Table 3.4.

Temporal locality of reference is one of the best-known phenomena in computer workloads [18, 32, 27], and is the foundation around which the computer's memory hierarchy is designed

[32, 27]. But mass-count disparity plots show that such locality 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 [35]. The popular blocks are manifested by the long residencies that service the majority of references. Opposite, most short residencies are not degenerate and still have a noticeable length (albeit of a small number of references) indicating that even accesses to a transient blocks are batched together as a burst of activity. Had these accesses not been bursty in nature, the block would have been evicted before it is reused and the residency length would have degenerated.

The existence of mass-count disparity demonstrates that the working set is not evenly used but is rather focused around a *core*. These more popular addresses can be grouped together to form the *core working set* — a subset of the Denning's classic working set definition [18] whose cache residencies naturally serve the majority of references. 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 [67, 70]. 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 the caching mechanism, and the nature of this core allows it to be identified using random sampling.

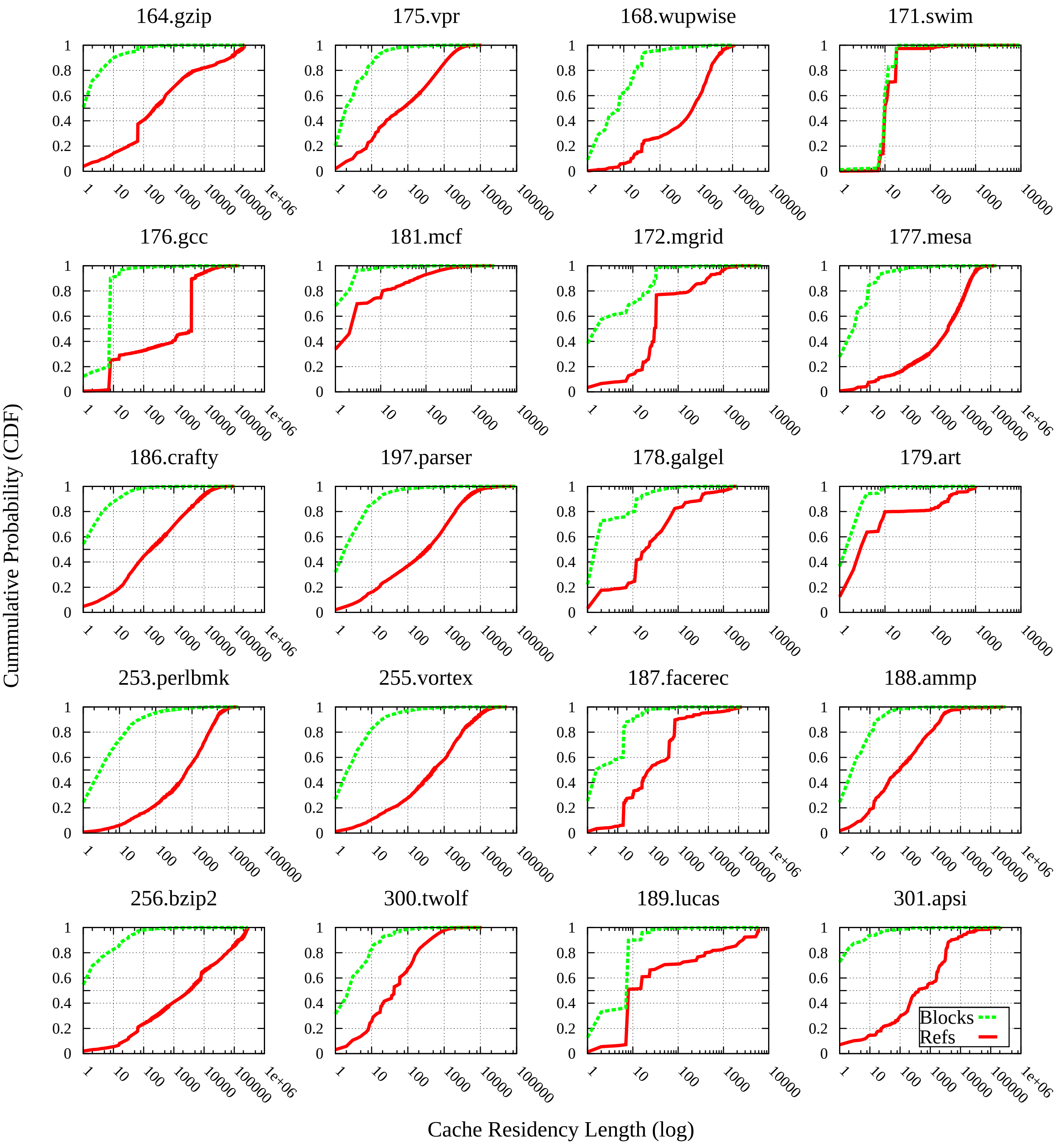

Figure 3.4: ***16KB Data cache, direct-mapped**: A general overview of the mass and count distributions for the **data** streams of all 20 SPEC2000 benchmarks reviewed. Together with the data from Table 3.1, this figure demonstrates that the mass-count disparity exists in practically all data streams.*

Table 3.1: ***16KB Data cache, direct-mapped****: The $N_{1/2}$ and $W_{1/2}$ metrics values for L1* ***data*** *streams of the 20 SPEC2000 benchmarks used.*

Benchmark	$W_{1/2}$	$W_{1/2}$@	$N_{1/2}$	$N_{1/2}$@	JR	JR@
164.gzip	3.80	1	0.60	230	87 / 13	8
175.vpr	7.60	2	1.71	72	80 / 20	8
176.gcc	11.68	8	0.27	3826	79 / 21	8
181.mcf	24.77	1	16.79	3	67 / 33	1
186.crafty	4.39	1	0.69	169	86 / 14	8
197.parser	4.48	2	0.67	336	85 / 15	9
253.perlbmk	2.29	3	0.93	731	88 / 12	25
255.vortex	3.25	3	0.65	517	87 / 13	15
256.bzip2	1.83	1	0.10	3247	90 / 10	24
300.twolf	7.34	3	4.39	42	78 / 22	9
168.wupwise	3.59	8	1.07	804	84 / 16	32
171.swim	38.98	10	37.48	10	56 / 44	10
172.mgrid	5.34	2	10.41	30	77 / 23	17
177.mesa	2.01	3	0.20	3886	90 / 10	20
178.galgel	11.12	2	6.44	20	78 / 22	8
179.art	21.69	2	16.52	3	67 / 33	2
187.facerec	3.41	2	2.32	104	80 / 20	16
188.ammp	5.88	3	1.85	96	81 / 19	12
189.lucas	18.24	8	11.44	8	67 / 33	8
301.apsi	4.81	1	0.26	396	89 / 11	6
Average	9.33	3.3	5.74	726	80 / 20	12.3
Median	5.34	2	1.71	169	81 / 20*	0

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

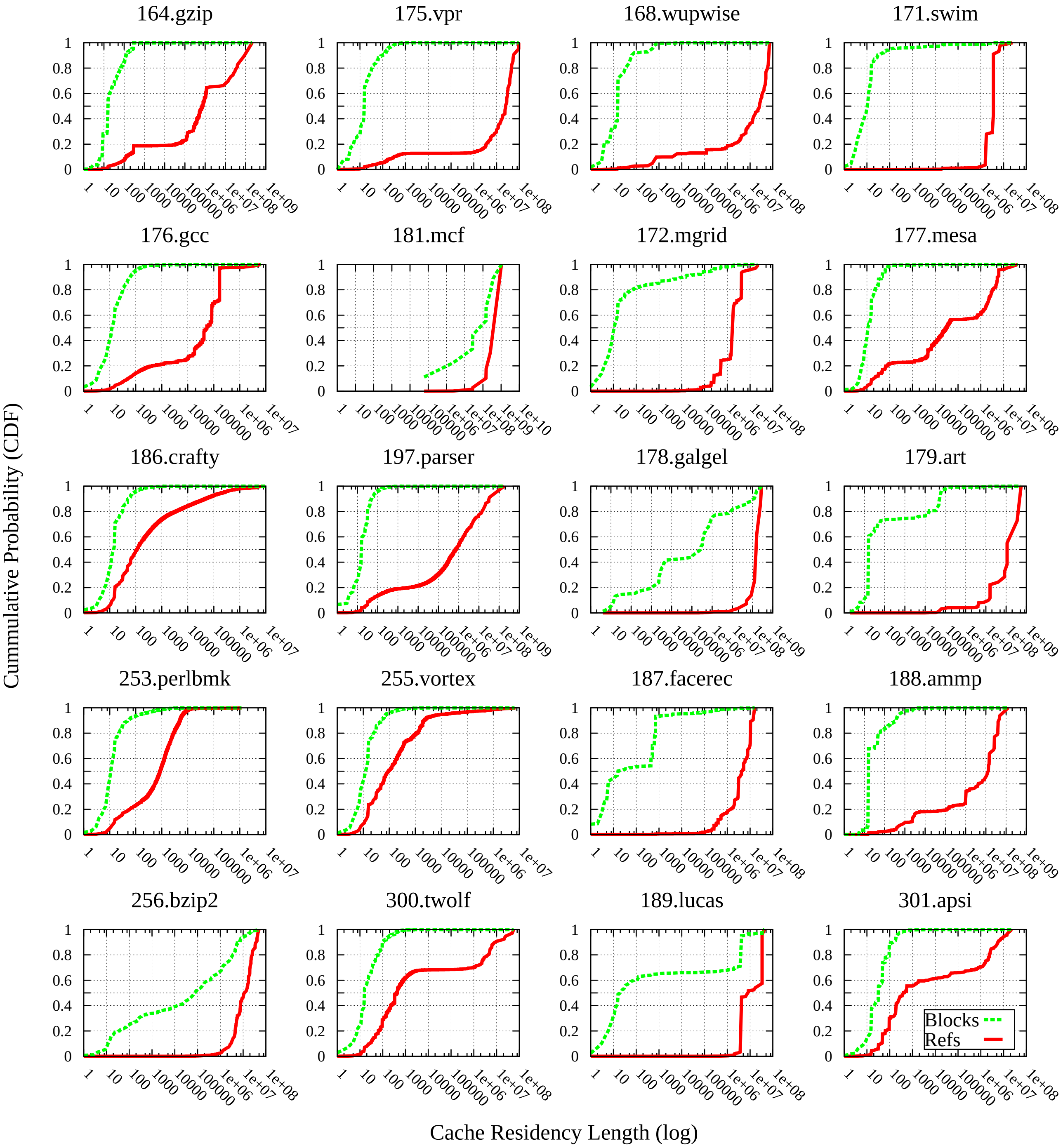

Figure 3.5: ***16KB Instruction cache, direct-mapped***: *A general overview of the mass and count distributions for the* ***instruction*** *streams of all 20 SPEC2000 benchmarks reviewed. Together with the data from Table 3.2, this figure demonstrates that the mass-count disparity exists in practically all data streams.*

Table 3.2: ***16KB Instruction cache, direct-mapped****: The* $N_{1/2}$ *and* $W_{1/2}$ *metrics values for L1* ***instruction*** *streams of the 20 SPEC2000 benchmarks used.*

Benchmark	$W_{1/2}$	$W_{1/2}$@	$N_{1/2}$	$N_{1/2}$@	JR	JR@
164.gzip	2.20	16	0.00	7.2 e+5	90 / 10	123
175.vpr	1.43	16	0.00	2.5 e+7	93 / 7	144
176.gcc	2.74	13	0.05	5.4 e+4	89 / 11	59
181.mcf	6.50	1.46 e+8	10.69	1.0 e+9	74 / 26	2.5 e+8
186.crafty	11.21	15	3.72	112	76 / 24	24
197.parser	3.16	16	0.00	7.2 e+5	89 / 11	48
253.perlbmk	5.91	12	1.48	858	84 / 16	30
255.vortex	11.33	13	4.34	96	76 / 24	20
256.bzip2	0.19	7.5 e+4	5.22	1.2 e+7	82 / 18	4.3 e+6
300.twolf	6.32	16	1.91	416	81 / 19	64
168.wupwise	0.65	16	0.00	2.6 e+7	95 / 5	512
171.swim	0.01	11	0.46	3.5 e+6	99 / 1	6.7 e+5
172.mgrid	0.00	11	1.03	1.8 e+6	95 / 5	2.0 e+5
177.mesa	4.65	11	0.02	3.0 e+4	87 / 13	32
178.galgel	0.06	2.7 e+5	5.33	1.6 e+8	88 / 12	8.6 e+6
179.art	0.00	16	0.04	1.1 e+8	97 / 3	6.4 e+4
187.facerec	0.01	16	0.28	4.6 e+6	97 / 3	1.8 e+5
188.ammp	1.02	16	0.00	1.3 e+7	94 / 6	448
189.lucas	0.00	20	3.94	8.4 e+6	80 / 20	4.2 e+6
301.apsi	7.57	32	1.41	384	79 / 21	86
Average	3.25	7.3 e+6	2.00	7.0 e+7	87 / 13	1.7 e+7
Median	2.20	16	1.03	3.5 e+6	89 / 12[*]	448

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

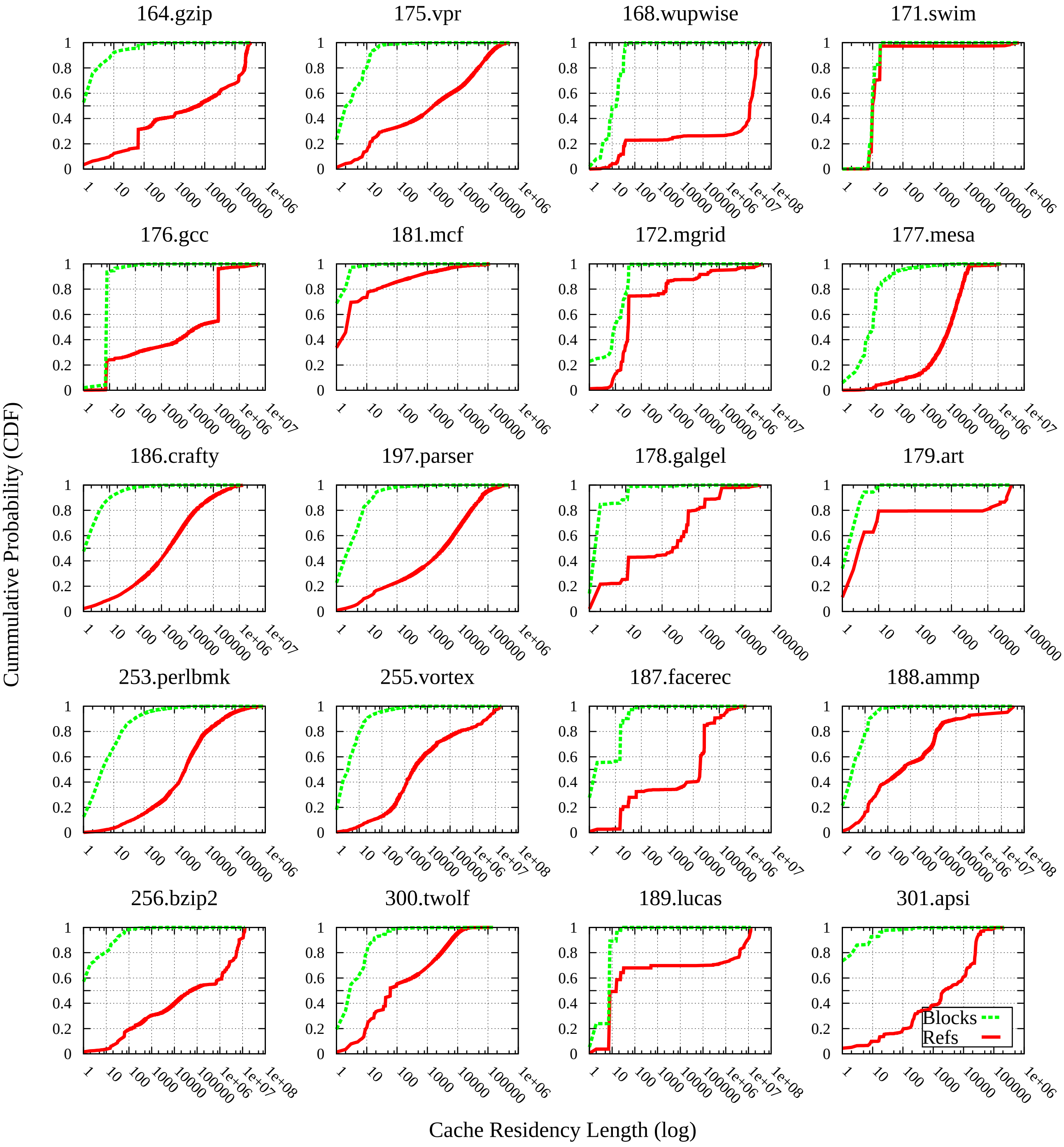

Figure 3.6: ***16KB Data cache, 4-way set-associative**: A general overview of the mass and count distributions for the **data** streams of all 20 SPEC2000 benchmarks reviewed. Together with the data from Table 3.3, this figure demonstrates that the mass-count disparity exists in practically all data streams.*

Table 3.3: ***16KB Data cache, 4-way set-associative**: The $N_{1/2}$ and $W_{1/2}$ metrics values for L1* ***data** streams of the 20 SPEC2000 benchmarks used.*

Benchmark	$W_{1/2}$	$W_{1/2}$@	$N_{1/2}$	$N_{1/2}$@	JR	JR@
164.gzip	3.22	1	0.02	6005	89 / 11	8
175.vpr	4.41	3	0.09	1606	84 / 16	11
176.gcc	12.40	8	0.01	24085	80 / 20	8
181.mcf	24.47	1	16.00	3	67 / 33	1
186.crafty	2.49	2	0.14	1932	90 / 10	11
197.parser	3.21	3	0.10	3441	88 / 12	14
253.perlbmk	1.83	5	0.52	2277	90 / 10	45
255.vortex	1.75	4	0.20	2528	91 / 9	26
256.bzip2	1.35	1	0.00	52757	91 / 9	32
300.twolf	6.79	3	1.79	60	79 / 21	10
168.wupwise	4.45	16	0.00	11565917	85 / 15	32
171.swim	39.50	10	37.81	10	56 / 44	10
172.mgrid	11.51	10	14.55	32	71 / 29	21
177.mesa	1.51	16	0.36	13377	93 / 7	124
178.galgel	11.99	2	0.61	198	80 / 20	2
179.art	21.75	2	15.74	3	67 / 33	3
187.facerec	2.30	2	0.05	17920	83 / 17	16
188.ammp	5.17	3	0.25	444	82 / 18	14
189.lucas	21.92	8	10.24	16	67 / 33	8
301.apsi	3.04	1	0.07	2232	91 / 9	9
Average	9.25	5	4.93	584742	81 / 19	20
Median	4.45	3	0.25	2232	84 / 17	11

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

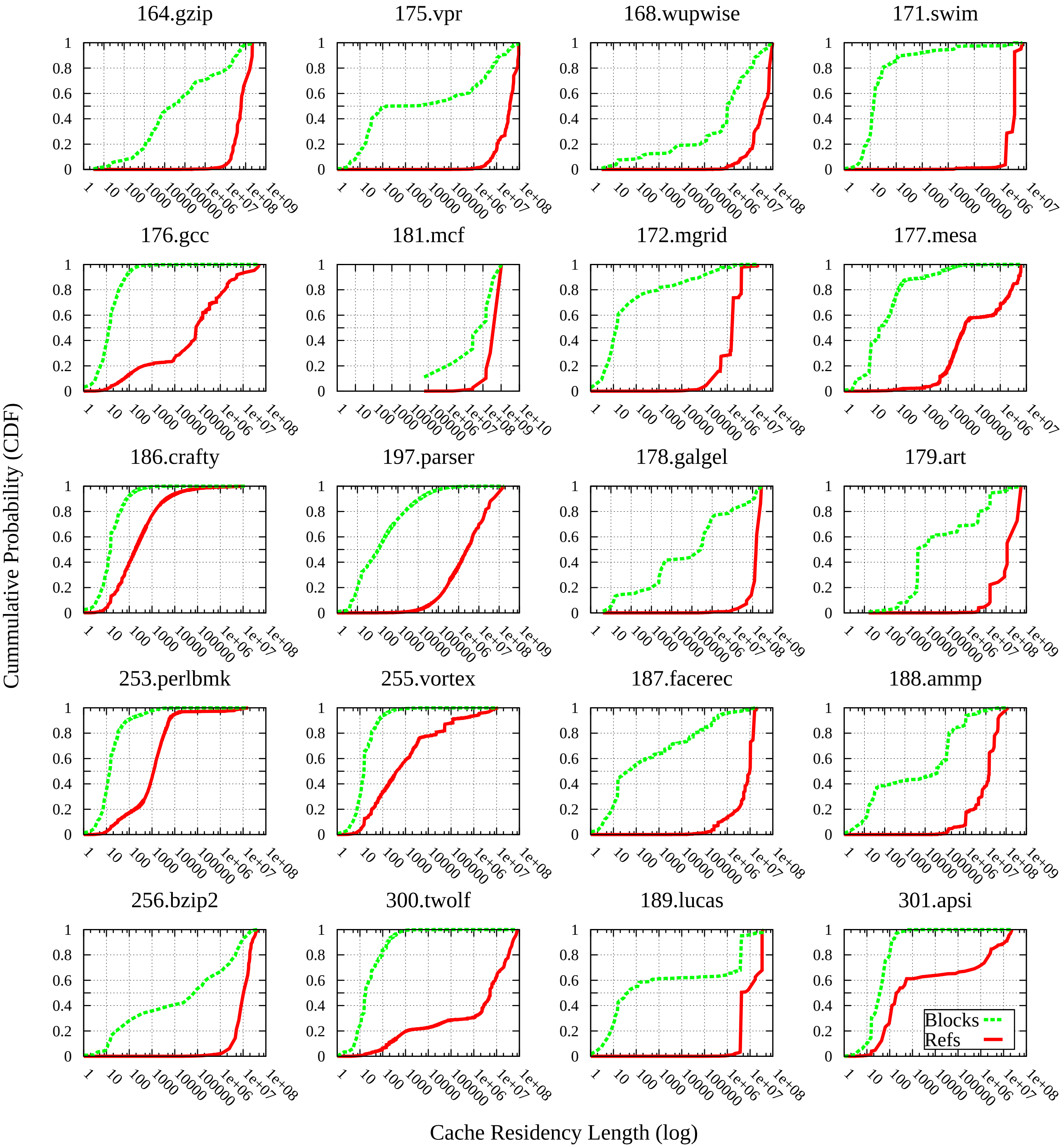

Figure 3.7: ***Instruction cache, 4-way set-associative**: A general overview of the mass and count distributions for the **instruction** streams of all 20 SPEC2000 benchmarks reviewed. Together with the data from Table 3.4, this figure demonstrates that the mass-count disparity exists in practically all data streams.*

Table 3.4: ***16KB Instruction cache, 4-way set-associative****: The $N_{1/2}$ and $W_{1/2}$ metrics values for L1* ***instruction*** *streams of the 20 SPEC2000 benchmarks used.*

Benchmark	$W_{1/2}$	$W_{1/2}$@	$N_{1/2}$	$N_{1/2}$@	JR	JR@
164.gzip	0.02	2.5 e+4	4.58	6.0 e+7	85 / 15	2.3 e+7
175.vpr	0.00	144	4.73	3.8 e+7	85 / 15	9.9 e+6
176.gcc	2.63	14	0.02	8.4 e+4	89 / 11	65
181.mcf	6.50	146 e+8	10.69	1.0 e+9	74 / 26	2.5 e+8
186.crafty	8.81	16	3.83	190	78 / 22	36
197.parser	0.05	108	0.14	2.6 e+6	94 / 6	3.1 e+4
253.perlbmk	4.32	14	1.81	1.2 e+3	86 / 14	52
255.vortex	8.08	16	1.29	405	80 / 20	32
256.bzip2	0.15	6.9 e+4	7.04	1.0 e+7	82 / 18	4.9 e+6
300.twolf	1.68	18	0.00	5.2 e+6	91 / 9	176
168.wupwise	2.45	1.0 e+6	5.66	4.2 e+7	82 / 18	1.3 e+7
171.swim	0.01	14	0.89	3.5 e+6	98 / 2	1.1 e+6
172.mgrid	0.00	14	1.39	1.8 e+6	93 / 7	2.0 e+5
177.mesa	0.30	27	0.35	4.2 e+4	93 / 7	4.5 e+3
178.galgel	0.06	2.7 e+5	5.33	1.6 e+8	88 / 12	8.6 e+7
179.art	0.01	4.4 e+3	1.75	1.1 e+8	87 / 13	1.6 e+7
187.facerec	0.00	45	1.24	9.8 e+6	92 / 8	3.9 e+5
188.ammp	0.19	3.8 e+4	1.50	1.5 e+7	89 / 11	1.0 e+6
189.lucas	0.00	42	5.14	4.2 e+6	78 / 22	4.2 e+6
301.apsi	10.17	32	2.93	208	76 / 24	72
Average	2.27	7.4 e+6	3.02	7.5 e+7	86 / 14	2.1 e+7
Median	0.19	45	1.81	5.2 e+6	87 / 14	1.0 e+6

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

Chapter 4

Core Working Sets: Capturing the Dual Nature of Memory Workloads

Locality is usually regarded as a combination of two distinct properties — locality in time and locality in space. But it is also a manifestation of the skewed distribution of the *popularity* of different memory blocks, where some blocks are accessed more frequently than others. In fact, as shown below, it may be possible to partition the working set into two sub-sets: those data items that are very popular and accessed at a very high rate, and those that are only accessed intermittently. This distinction is antithetical to Denning's definition which puts all items in a working set on an equal footing, and lies at the heart of the definition of the *core* of the working set.

The notion of a core leads to the realization that not all elements of the working set are equally important. The elements in the working set are not accessed in a homogeneous manner. Thus treating all the elements of the working set equally may lead to sub-optimal performance. Rather, it may be beneficial to try to identify the more important core elements, and give them preferential treatment.

A striking manifestation of a "hot" core within the working set can be seen when exploring the stack depths accessed in a set-associative cache's sets, and more specifically as the fraction of references serviced by the cache sets' most recently used (MRU) blocks. Figure 4.1 presents

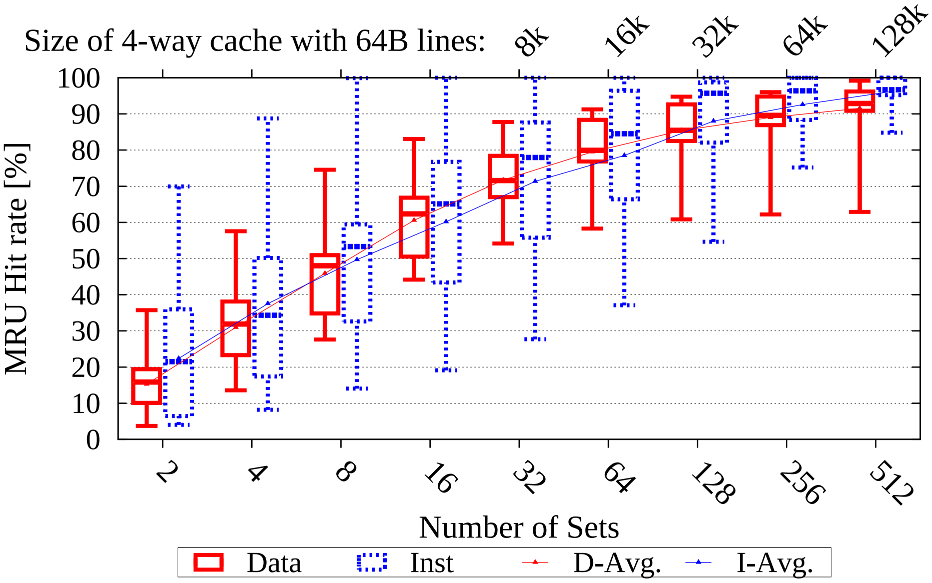

Figure 4.1: *Fraction of accesses serviced by the most recently used (MRU) block in a cache's sets for various cache configurations. The top axis show the size of 4-way set-associative cache (with 64B cache lines) corresponding to each number of sets. Results are shown as an average over all SPEC2000 benchmarks surveyed.*

the fraction of memory references serviced by the MRU blocks in the cache sets for various cache configurations. This is achieved by simply varying the numbers of cache sets examined (the metric is oblivious to the cache's exact associativity as it only examines MRU accesses). The X axis in the Figure indicates a number of sets in the cache, and the box-plots represent the MRU access statistics for all SPEC2000 benchmarks given a specific number of cache sets — where each box-plot shows the 25-75 percentiles, minimum and maximum values, median and the average (marked with a circle) over all benchmarks. The Figure shows that as the number of cache sets increases, more references are serviced by the MRU block in the referenced set: for caches with at least 128 sets an average of ∼85% of all memory references (data and instructions) are serviced by the MRU blocks, with even the 25th percentile crossing the 80% threshold. When the number of sets doubles to 256 the MRU hits reach ∼90% of all references. The reason for the dependence on the number of sets in the cache is the small set of blocks servicing the majority of references, and how it is distributed among the cache sets: as the number of cache sets increases the number of mapping conflicts between the core blocks decreases, thus enabling this small set of blocks — the core of the working set — to be evenly distributed between the different sets. When the number of set suffices, each set contains on average at most a single block from the working set's core. As the core is accessed repeatedly, its blocks maintain their MRU positions.

The Figure also shows the size of a 4-way set-associative cache corresponding to the number of cache sets displayed. Interestingly, modern processors such as the Intel Core product series [29] and AMD Phenom series [1] use 32K and 64K L1 caches, respectively, thereby serving an average of $\sim$90% of all memory references from the sets' MRU positions. In fact, since the AMD Phenom uses a 64K 2-way set-associative cache, it has double the number of sets than a similar size 4-way set-associative cache — thus enjoying an even higher $\sim$95% MRU hits.

This chapter introduces a formal framework that extends and refines Denning's definition of a working set, enabling designers to explicitly express their perception of which blocks in the working set are considered important. This framework uses logical predicates to distinguish between the important subset — the core — and the remaining blocks. An example of a predicate that can be used to identify the core is "the block is accessed at least 16 times when brought into the cache". The extraction of an explicit predicate enables qualitative comparison between different caching mechanisms and implementations. In particular, it decouples the *notion* of the working set's core from the actual *caching mechanism* used to implement it.

While the core working set framework is aimed for use with any caching mechanism, this exploration is focused on the synergy between the skewed distribution of memory references and dual cache structures. Defining the core based on the intensity of memory references naturally leads to a dual design, where one part of the cache is used for the core data, while the more transient data is served by another part. In effect this filters non-core data and prevents them from polluting the cache structure used for core data.

4.1 Definition of Core Working Sets

Denning's definition of working sets [17] is based on the principle of locality, which he defined to include three components [19]: a nonuniform popularity of different addresses, a slow change in the reference frequency to any given page, and a correlation between the immediate past the near future. Our data strongly supports the first component, that of non-uniform access. But it casts a doubt on the other two, by demonstrating 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, namely the core working set.

The definition of a working set by Denning is the set of *all* distinct blocks that were accessed within a window of T instructions [17]. This set will denoted as $D_T(t)$, to mean "the Denning working set at time t using a window size of T". Our findings imply that this definition is deficient in the sense that it does not distinguish between the heavily used items and the lightly used ones.

As an alternative, we define the *core working set* to be those blocks that appear in the working set and are reused a significant number of times. This will be denoted $C_{T,P}(t)$, where the extra parameter P reflects a predictor used to identify core members; the predictor will be expressed as a predicate that evaluates to "true" for core members, and "false" for other blocks. This is a generalization of the Denning working set, which can simply be expressed as the core working set with a predicate that is always true:

$$D_T(t) \equiv C_{T,\text{true}}(t)$$

The predicate P is meant to capture reuse of memory. In the context of virtual memory, temporal locality has been used to justify page replacement algorithms such as LRU or the clock algorithm. In particular, Belady emphasized the importance of use bits to identify recently used data that should be retained [6]. Our reuse predictors can be seen as an extension of this practice. The generality of core working sets can also be demonstrated by its applicability to block prefetchers: at any time t, a prefetcher would estimate the core at a future time $t + n$. Therefore, the prefetcher's core can be described as $C_{T,P}(t+n)$, where P represents the predicate best describing the prefetcher designer's perception of the important subset of blocks.

The simplest reuse predictor is based on counting the number of references to a given block (or the number of references during a residency). Let B represent a block of k words. Let w_i, $i = 1, \ldots, k$ be the words in block B. Let $r(w)$ be the number of references to word w within the window of interest. Using this, we can define the predicate nB that evaluates to true if block B was referenced n times or more:

$$n\text{B} \equiv \sum_{i=1}^{k} r(w_i) \geq n$$

For example, the predicate 3B identifies those blocks that were referenced a total of 3 times or more.

The nB predicates are meant to identify a combination of spatial and/or temporal locality, without requiring either type explicitly. Alternatively, we can write a temporal-locality predicate that requires that some specific word w in block B was referenced n times or more:

$$n\text{W} \equiv \exists w \in B \quad \text{s.t.} \quad r(w) \geq n$$

We can also write a predicate that requires a certain number of distinct words to be referenced, to express spatial locality.

An example of a more complicated predicate is the $n \times ST$ predicate, which is meant to identify a non-uniform strided reference pattern with reuse. This predicate is designed to filter out memory scans that use strided access, even if they include up to n accesses to the same memory location within the scan. This is done by tabulating the last few accesses, as illustrated by the following pseudocode (where addr is the address accessed last):

```
if (prev_addr == addr) {repeat++;}
else {prev_stride = stride; stride = addr - prev_addr; repeat = 0;}
prev_addr = addr;
```

using this data, a block is considered in the core if it was accessed with inconsistent strides, or if a single word was referenced more than n times in a row. Formally, this is written as

$$n \times ST \equiv (repeat > n) \vee (stride \neq prev\_stride)$$

These examples only demonstrate the richness of the predicates' design space. But given the rich set of possible predicates, the question is how to select one that captures the notion of a core working set. Based on the discussion about the bursty nature of access patterns (Section 3.2), it seems advisable to require a significant number of references. In particular, we have found 16B to be a promising predicate.

The effect of this definition is illustrated in Figure 4.2. Using the SPEC *gcc* benchmark as an example, the top graph simply shows the access pattern to the data. Below it we show the Denning working set $D_{1000}(t)$ (i.e. for a window of 1000 instructions) and the core working set $C_{1000,16B}(t)$. As we can easily see, 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.

4.2 Core Working Sets and Dual Cache 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 only a few times in a bursty manner. The question then is how this can be put to use to improve caching.

The principle behind optimal cache replacement is very simple: when space is needed,

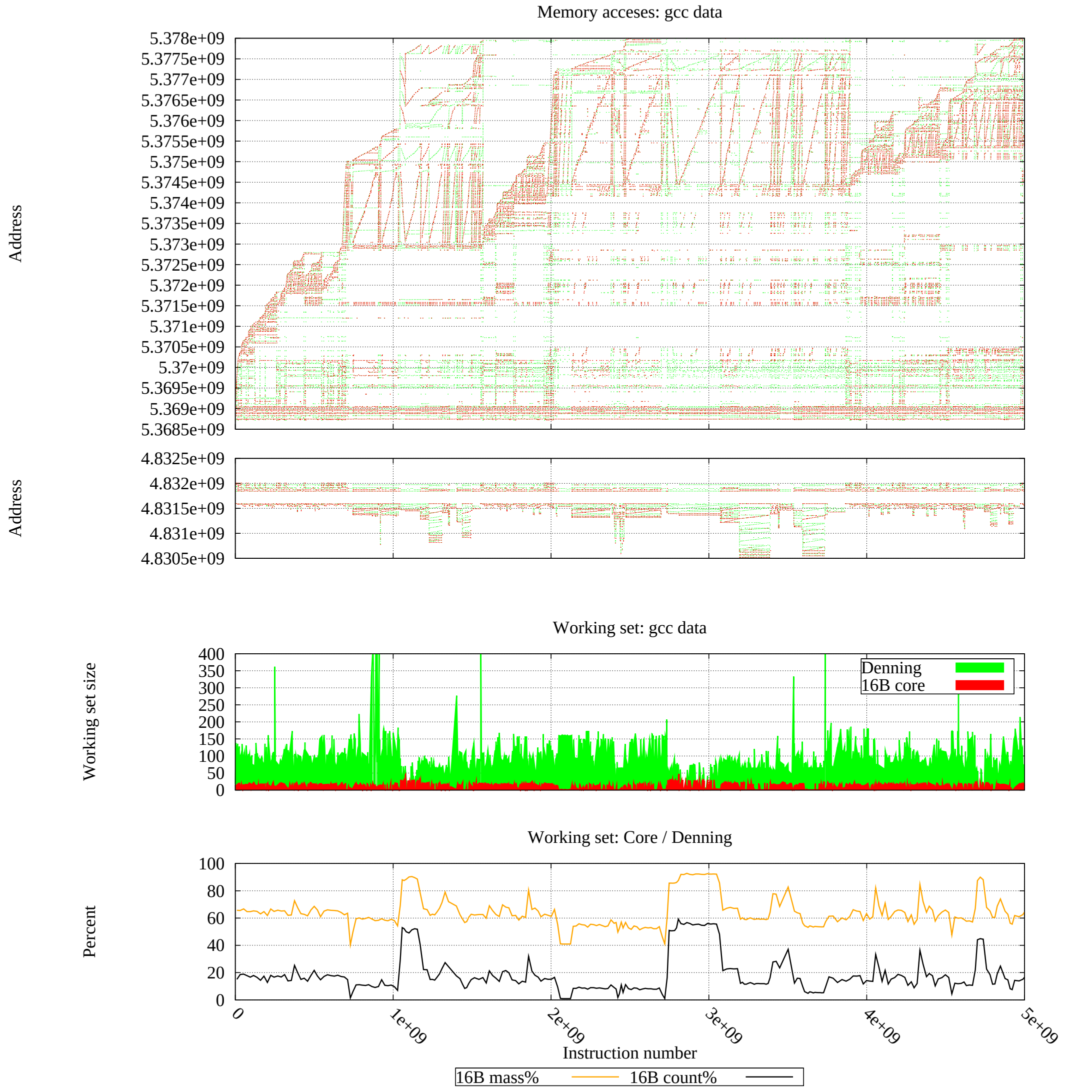

Figure 4.2: *Examples of data memory access patterns and the resulting Denning and core working sets. The bottom plot shows that while the 16B core working set is composed of only ∼20% of the blocks in Denning's working set, it still services some ∼60% of its references.*

replace the item that will not be used for the most time in the future (or never) [6]. In particular, it should be noticed that it is certainly possible that the optimal algorithm will decide to replace the *last* item that was brought into the cache, if all other items will be accessed before this item is accessed again. This would indicate that the item was only inserted into the cache as part of the mechanism of performing the access; it was not inserted into the cache in order to retain it for future reuse. Such blocks were appropriately described as dead-on-arrival (DOA) by Qureshi et al. [54].

By analyzing the reference streams of SPEC2000 benchmarks it is possible to see that this sort of 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 of questionable merit.

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, by contradistinction, would do the opposite and lose all the cache contents.

The aforementioned benefits of using a cache bypass can be demonstrated formally using a simple, specific example cache configuration. Assume a cache with $n^2 + n$ cache lines, organized into n sets whose size is either n or $n+1$ cache lines each. In either case, the address space is partitioned into n equal-size disjoint partitions (assuming n is a power of 2) using the memory address bits. The two organizations are used as follows.

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, and each serves a distinct partition of the address space, as in the conventional approach. The remaining n cache lines are grouped as a disjoint set (which we will call the "extra" set), and can accept any address and serves as a bypass. [1]

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.

Considering these two options, 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 extra set can hold any address from the address space, we are not required to use this functionality. Instead, we can limit each cache line to one of the partitions in the address space. Thus the effective space available for caching each partition becomes $n + 1$, just like in the set associative design. ∎

The conclusion from this claim is that the bypass design need never suffer more cache misses than the set associative design. At the same time, we have the following claim that establishes that it actually has an advantage.

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 address space 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 $2n$

[1] For simplicity, the claims assume the bypass buffer contains the same number of ways as sets in the main cache — but obviously these numbers need not be the same.

addresses will be cached by using the original set and the extra set. Therefore only the initial $2n$ compulsory misses will occur. In this sense, a bypass mechanism can potentially relieve pressure on specific cache sets resulting from bursty conflict misses. By extending the length of this pattern (i.e. by increasing m) any arbitrary ratio can be achieved. ∎

An example of a dual-cache design that extends a simple set-associative cache is the *Victim Cache* proposed by Jouppi [37]. The victim cache includes a small fully-associative buffer into which all memory blocks evicted from the direct-mapped main cache (the "victims") are inserted. Jouppi showed that many of the blocks evicted from the main cache will be requested again within a short period of time, and should therefore be kept in an auxiliary cache and be given a chance for re-insertion. In that sense, the auxiliary buffer — called the *victim buffer* — serves as an extension to the main cache. The myriad of dual-cache structures proposed in the literature are discussed in Chapter 7.

The definition of *core working sets* thus extends the classic working set definition by Denning to capture the dual nature of memory workloads, and directly corresponds with the dual-cache paradigm. While the paradigm itself is not new, the formalization of core working sets is novel. Core working sets thus enable cache designers to formally describe, compare and contrast dual-cache designs in a natural way — thereby adding a useful tool to the cache designer's toolbox.

Chapter 5

Cache Filtering Through Probabilistic Prediction of Temporal Locality

The existence of lengthy cache residencies is a direct result of temporal locality — as discussed in Section 3.1 — since frequent accesses to a block prevent it from being replaced in a standard LRU based set-associative cache. But even long residencies are sometimes terminated in favor of shorter residencies, when a more popular block is evicted in favor of a less popular one. This is commonly caused by changes in a program's memory workload that increase the number of cache misses (and therefore cache insertions) momentarily, and mainly affects caches with a relatively low degree of associativity that cannot effectively sustain a sudden burst of cache insertions. Although frequently accessed blocks will be quickly re-inserted into the cache, the first access after the eviction will incur a high latency cache miss. This is in fact the rationale behind the *least-frequently-used* (LFU) replacement policy.

Alternatively, these inefficient replacements can be avoided by employing a *cache insertion policy* that will prevent transient blocks from being inserted to cache in the first place — because such blocks effectively just pollute the cache. The length of a cache residency can thus serve as a metric for cache efficiency, with longer residencies indicating better efficiency, as the initial block insertion overhead (latency and power) is amortized over many cache hits. A simple filter can therefore be based on a residency length predictor that will be used to predict whether inserting a block into the cache would be beneficial. If the insertion is not beneficial, references to the block can be served from an auxiliary buffer, or bypass the cache altogether (these two alternatives are effectively equivalent since the latter can be seen as using an auxiliary buffer containing a single block).

However, the definition of a "beneficial insertion" is not trivial, and is tightly coupled with the specifics of the cache design. Therefore, the discussion on residency length prediction (or rather whether a residency should be characterized as a long one) is separated from the proposed cache design (discussed in Chapter 6).

5.1 On the Non-Feasibility of an Optimal Insertion Policy for Dual-Cache Structures

Caches are commonly evaluated using their achievable miss-rate (or hit-rate), which, for simple associative cache structures, can be minimized using Belady's optimal replacement policy [6]. Using an insertion policy however, implies a dual-cache structure since the block selection can be effectively viewed as making a decision whether to insert the block into the cache itself, or rather into an auxiliary buffer. Although in general the auxiliary buffer can contain only one block — thus effectively implementing a cache bypass — it is assumed here that the auxiliary buffer contains more than one block.

Dual-cache structures are not addressed by Belady's optimal algorithm which only accounts for a replacement policy, and not a cache/buffer arbitration policy. Moreover, Brehob et al. showed that optimal cache replacement is NP-Hard for dual-caches in which one component is fully-associative and the other is either set-associative or direct-mapped [7]. This result can be extended to show that even if both caches are set-associative but with different numbers of sets, the problem is still NP-Hard — thereby removing the requirement that one of the components is fully-associative:

Claim 3 *An optimal replacement algorithm for a dual-cache, in which both components are* *set-associative with different number of sets, is NP-Hard if*[1]*:*

- *the numbers of sets in both components are powers of 2.*
- *the address bits used to map addresses to sets in the component containing the smaller number of sets, is a subset of the mapping bits in the other component.*

Proof: Consider a dual-cache containing two set-associative components, where component 1 is of size B_1 blocks and associativity of A_1 (such that it is not fully-associative, i.e. $A_1 < B_1$), and where component 2 is of size B_2 and associativity A_2 (such that $A_2 < B_2$). Component 1 therefore contains $S_1 = \frac{B_1}{A_1}$ disjoint sets, and component 2 contains $S_2 = \frac{B_2}{A_2}$ sets. Based on the precondition, $S_1 \neq S_2$.

Now, let us assume (without loss of generality) that component 2 contains more sets than component 1, i.e. $S_2 > S_1$. We can therefore divide the entire reference stream into S_1 disjoint reference sub-streams $R_1 \cdots R_{S_1}$, where each sub-stream R_i is mapped to, and uniquely serviced by, a single set in component 1. In addition, because the address bits used in mapping addresses to cache sets in component 1 are said to be subset of those used in component 2, R_i is also mapped to, and uniquely serviced by, exactly $\frac{S_2}{S_1}$ sets in component 2. Furthermore, based on the precondition that S_1 and S_2 are distinct powers of 2, it is guaranteed that $\frac{S_2}{S_1}$ is a natural number.

The entire dual-cache can thus be regarded as S_1 disjoint dual-caches $C_1 \cdots C_{S_1}$, each containing a single set from component 1 and $\frac{S_2}{S_1}$ sets from component 2 — with each C_i servicing only part of the entire reference stream, namely the R_i sub-stream. But each C_i is effectively a dual-cache composed of a fully-associative structure (the single set from component 1), and a set-associative cache of degree A_2 and size $A_2 \times \frac{S_2}{S_1}$ — a design for which an optimal replacement policy is NP-Hard according to Brehob et al.

The result is that no feasible optimal replacement algorithm exists for each of the sub-streams, based on the proof by Brehob et al. [7]. But because the sub-streams use disjoint sub-structures of the entire dual-cache, a block from one sub-stream cannot replace a cached

[1] Since the two preconditions are in fact common practices in cache design, they do not effectively hinder the breadth of the claim.

block from another sub-stream. Thus, a *global* optimal replacement algorithm for the entire dual-cache must be NP-Hard as well. ∎

This lack of a feasible optimal cache insertion policy suggests a need for a different strategy to evaluate such policies. Moreover, using a strategy that completely ignores the underlying caching mechanism is impossible when dealing with cache residencies, since the definition of a residency depends on the cache parameters — such as the cache size and its eviction policy.

5.2 Identifying An Effective Subset of Blocks

The lack of an optimal replacement algorithm for dual-caches suggests a different strategy should be used to evaluate cache filtering. It is therefore suggested to approach this problem from a cost/gain perspective, trying to create a minimal core working set of residencies that will effectively maximize gain — where the *cost* is defined as the fraction of all residencies included in the core working set, and the *gain* defined to be the fraction of all references composing the residencies in the core. Effectively, the cost of the core working set can also be regarded as the size of the cache needed to accommodate all the residencies selected, and the gain as the hit-rate achieved.

The conflict between the *cost* and *gain* here is obvious: on one hand, inserting all residencies to the core will service all references from the core, but will also represent a cache of infinite size. On the other hand, leaving the core empty will minimize the number of residencies included in the core (none), but will serve no residencies from the core as well. The need to balance the two opposite goals indicates the strategy should employ a threshold parameter determining the minimal length of a residency that should be considered part of the core. Although maximizing the gain in lieu of a cost metric is traditionally addressed by finding a threshold that maximizes the average gain $\frac{gain}{cost}$, this strategy will not work here. In the case at hand, this method would try to identify a residency length threshold that would maximize the average core residency defined as $\frac{core\text{-}references}{core\text{-}residencies}$ — which would simply set the threshold to be the length of the longest residency, thus inserting only that single residency into the core

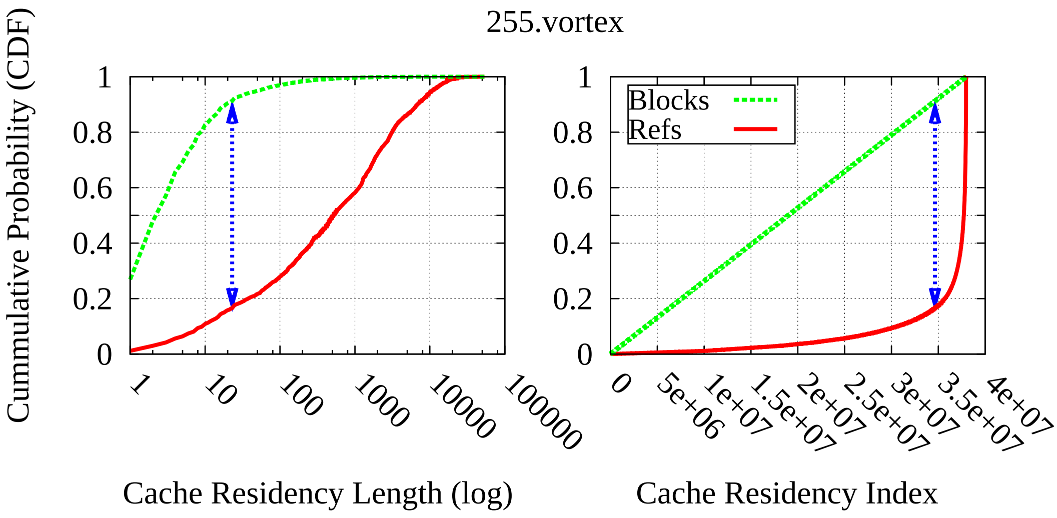

Figure 5.1: *Demonstration of the maximal effectiveness point for* vortex. *On the left is the original mass-count plot, and on the right is the same data plotted as a function of the residency index in the sorted list of all residencies. The double arrow demonstrate the point where the difference between the mass and count is maximal in which* ~*9% of all residencies serve* ~*84% of all references. It occurs for a residency length of 23 references, and requires only* $\sim 4 \times 10^6$ *residencies out of a total of* $\sim 38 \times 10^6$ *residencies.*

working set.

The strategy proposed is therefore to identify a residency length threshold (the minimal length of core residencies) that maximizes the *difference* between the fraction of all references composing the core, and the fraction of all residencies included in the core — namely $core\text{-}references - core\text{-}residencies$. All residencies whose length is longer than the threshold are considered part of the core working set.

Claim 4 *The threshold that maximizes the difference between the fraction of all references composing the core, and the fraction of all residencies included in the core, is the average residency length.*

Proof: Given R the total number of references, B the total number of residencies, we can sort the residencies according to length, such that each residency is attributed with an index i in the range $[1 \ldots B]$. $len(i)$ is defined as length of the ith shortest residency, and the sorted residencies indicate that $len(i) \leq len(i+1)$ for all i. Mass-count disparity plots can then be plotted as a function of the residencies' indices, as shown in Figure 5.1. In this plot style, the count cumulative probability value at the ith residency is

$$count(i) = \sum_{j=1}^{i} \frac{1}{B} = \frac{i}{B}$$

and its mass cumulative probability is

$$mass(i) = \sum_{j=1}^{i} \frac{len(j)}{R}$$

This means that 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 (as $len(i) \leq len(i+1)$), the gap between the two distributions grows while $\frac{len(i)}{R} < \frac{1}{B}$, narrows when $\frac{len(i)}{R} > \frac{1}{B}$, and peaks at

$$\frac{len(i)}{R} \simeq \frac{1}{B} \quad \Rightarrow \quad len(i) \simeq \frac{R}{B}$$

which is the average residency length. But since the average is not necessarily integral, whereas a residency length is integral by definition, the feasible threshold is simply the integral part of the average:

$$threshold = \lfloor \frac{R}{B} \rfloor$$

Furthermore, because the residencies' lengths are monotonically non-decreasing $len(i) \leq len(i+1)$, the slope of the mass distribution's cumulative function is guaranteed to never decrease, and the gap between the mass and the count will never decrease as well. This assures that the threshold does not represent a local maxima, but rather the global one. ∎

The floor value of the average residency length thus represents 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. Since the number of references for a specific stream is constant, this in turn will reduce the difference between the cost and gain, thus reducing the effectiveness of the core.

Selecting a threshold requires future knowledge about the number of references issued by the program and the number of residencies exhibited by the cache (obtained by pre-executing the program at hand). Once a threshold is selected though, blocks can be theoretically classified at runtime by maintaining a reference counter for each block. This selection algorithm is equivalent to the definition of a core working set with a predicate nB, described in Section 4.1. The predicate evaluates to *true* for each residency longer than n references, which in this case will be the predetermined threshold.

In practice, the counter-based approach cannot be implemented in hardware because it requires maintaining a counter per memory block. A viable predictor can therefore only approximate this counter-based approach using a feasible cache design.

Interestingly, the counter-based algorithm presented here relaxes the requirements from a 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. Therefore, rather than focusing on a specific threshold, a generic evaluation of residency length predictor focuses on the ability to approximate any given threshold.

The following section thus presents a probabilistic residency length predictor that uses a probabilistic parameter P. This predictor is then evaluated in Section 5.4 by exploring the relationship between the probabilistic parameter P and the threshold parameter used by the counter-based algorithm.

5.3 Probabilistic Residency Length Predictor

The probabilistic residency predictor harnesses the skewed distributions characterized by the mass-count disparity phenomenon (discussed in Chapter 3). The phenomenon shows that while most residencies are short, most references are serviced by a long residency. Therefore, selecting a memory reference at random by executing a Bernoulli trial on each memory reference is likely to identify a reference that is part of a long residency. Thus, if the trial's outcome is *true*, the rest of the residency is considered to be part of the core working set. When sampling 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, 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 periodic selection achieves results similar to those obtained with random selection. For consistency though, only results for random selection are shown.

The definition of a core working set presented in Chapter 4 requires the formalization of a predicate capturing the core blocks. Given B the entire working set of a program and $block(i)$ the block accessed in the ith memory reference, and given a uniformly distributed random bit stream with a success probability P whose ith bit is $rand_P(i)$ (bit values are referred to here by their boolean equivalents, such that bit value 1 corresponds to boolean "true", and bit value 0 corresponds to boolean "false"), the predicate $randomcore_P$ describing the core blocks is:

$$randomcore_P \equiv (\; b \in B \mid \exists i \;\; s.t. \;\; block(i) = b \wedge rand_P(i) = true \;)$$

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 — random selection requires a pseudo random number generator, which can be implemented using a simple linear-feedback shift register, whereas periodic selection simply requires a saturating counter [81]. This also enables easy integration with other predictor types, such as those addressing memory level parallelism and the criticality of specific references for performance [55].

The core working set $C_{T,randomcore_P}(t)$ thus represents the core selected using the random sampling algorithm presented above. Note that the formal definition of the predicate describes the set of blocks that should be treated preferentially according to the algorithm, but does not mandate any specific implementation. This is a striking example of how core working sets offer the decoupling of a cache designer's perception of the important blocks, from any caching mechanism or implementation based on this perception

5.4 Evaluating the Probabilistic Predictor

The evaluation of the probabilistic predictor is done against the counter-based algorithm presented in Section 5.1. In order for the results to be oblivious to any specific residency length threshold, the evaluation focuses on the relationship between the parameters of the two methods — the residency length threshold used in the counter-based approach, and the Bernoulli trial success probability used in the probabilistic predictor.

The counter-based approach essentially relies on achieving hysteresis of N levels (using a $\lg(N)$-bit saturating counter), which can be approximated using a series of Bernoulli trials with a success probability $\frac{1}{N/K}$, where K is the number of successful trials needed to approximate counter saturation [58]. The correlation sought is therefore between the residency length threshold N and the Bernoulli success probability P. Specifically, as the simplicity of the predictor is crucial for the feasibility of its implementation, the predictor can only be viable if a single Bernoulli trial is sufficient for block selection. The goal is therefore to evaluate relative performance of probabilistic selection with probability P and counter-based selection with a target count of $N = \frac{1}{P}$.

Figures 5.2 and 5.4 compare the probabilistic runtime predictor with the counter-based selection, for residencies generated using direct-mapped and 4-way set-associative 16K L1 data caches, respectively. The figures show the percentage of residencies classified as core (bottom lines) and the references they service (top lines). Complementing results for the instruction streams are shown in Figures 5.3 and 5.5 (direct-mapped and 4-way set-associative, respectively). As a unified scale, the X-axis equates a sampling probability of P with a counting threshold of $\frac{1}{P}$. When analyzing the percent of references serviced by the predictor's selected core, we see a very good correlation to those serviced by the counter-based predictor, at least for P up to 0.01. For example, when running *crafty* with a direct-mapped data cache (Figure 5.2) and using a selection probability of $P = 0.01$, the sampling predictor covers some

52% of all memory references, constituting over 90% of the number of references covered by the counter-based predictor. This result is fairly consistent for all benchmarks, using residency traces generated on both direct-mapped and 4-way set-associative caches, with the only exceptions being *swim art* and *mcf* benchmarks, notorious for their poor temporal locality.

When observing the number of residencies selected by both predictors, we see that the probabilistic predictor may select more residencies than the counter-based one, but for probabilities lower than $P = 0.01$ for data streams and $P = 0.001$ for instruction streams, the difference is up to a few percents. 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, too many false positives — or transient residencies — are classified as core (residencies shorter than 15 references constitutes some 90% of all residencies in the benchmarks shown in Figure 3.2). These extra residencies are also the reason why the probabilistic sampling predictor sometimes seems to serve more references than the counter-based predictor.

Tables 5.1 through 5.4 show how many residencies are classified as core and how many references they service: the first two tables show the statistics for 16K direct-mapped data and instruction streams, respectively, and the other two tables show the corresponding results for 4-way set-associative residencies' traces. When summing over all the residencies experienced by a 16K direct-mapped data cache, sampling only 0.01% of the data references selects an average $\sim$7.39% of the residencies, while covering over 45% of the references ($\sim$8% of residencies and $\sim$59% of the references for 4-way set-associative data caches). As the average is highly affected by benchmarks known for their poor temporal locality, such as *swim*, *art*, and *mcf*, the median values are shown as well, demonstrating a coverage of over 50% of direct-mapped data references (over 70% for 4-way set-associative data caches). The coverage is even better for both direct-mapped and 4-way set-associative instruction cache (Tables 5.2 and 5.4, respectively).

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 — a result that is consistent for all benchmarks analyzed. The following chapter describes a

Table 5.1: ***Data cache, 16KB direct-mapped**: Percents of residencies (insertions) classified as core and the references they service, for $P = 0.001$, $P = 0.01$ and $P = 0.05$.*

	$P = 0.001$		$P = 0.01$		$P = 0.05$	
Benchmark	%Ins	%Refs	%Ins	%Refs	%Ins	%Refs
164.gzip	0.90	31.91	5.77	56.63	16.89	75.64
175.vpr	0.90	13.86	6.24	41.09	21.35	61.33
176.gcc	1.30	45.53	8.38	65.26	31.77	74.72
181.mcf	0.20	1.32	1.87	8.12	8.43	20.68
186.crafty	0.83	27.34	5.49	52.29	17.39	70.98
197.parser	1.11	27.15	6.62	56.86	20.99	73.73
253.perlbmk	2.39	35.55	11.40	69.55	29.57	84.86
255.vortex	1.53	33.80	8.36	64.28	24.01	80.32
256.bzip2	1.23	54.96	7.22	73.83	20.22	85.94
300.twolf	0.94	5.98	7.22	28.79	23.38	55.70
168.wupwise	2.63	33.60	13.28	66.83	37.70	81.92
171.swim	1.13	0.88	10.62	7.49	42.60	28.77
172.mgrid	1.08	5.72	8.72	25.08	28.56	52.86
177.mesa	1.63	58.75	8.21	79.39	24.47	88.20
178.galgel	0.67	4.36	5.61	20.54	20.06	45.51
179.art	0.28	3.19	2.53	14.51	11.24	26.93
187.facerec	1.81	17.56	11.43	48.20	33.02	71.29
188.ammp	1.16	17.60	7.90	44.56	25.40	65.81
189.lucas	0.84	13.97	6.99	28.73	27.81	45.62
301.apsi	0.68	34.82	3.90	62.80	12.01	78.04
Average	1.16	23.39	7.39	45.74	23.84	63.44
Median	1.11	27.15	7.22	52.29	24.01	71.29

dual-cache design based on the probabilistic predictor proposed in this chapter, and details a thorough exploration of specific probabilities suitable for the design.

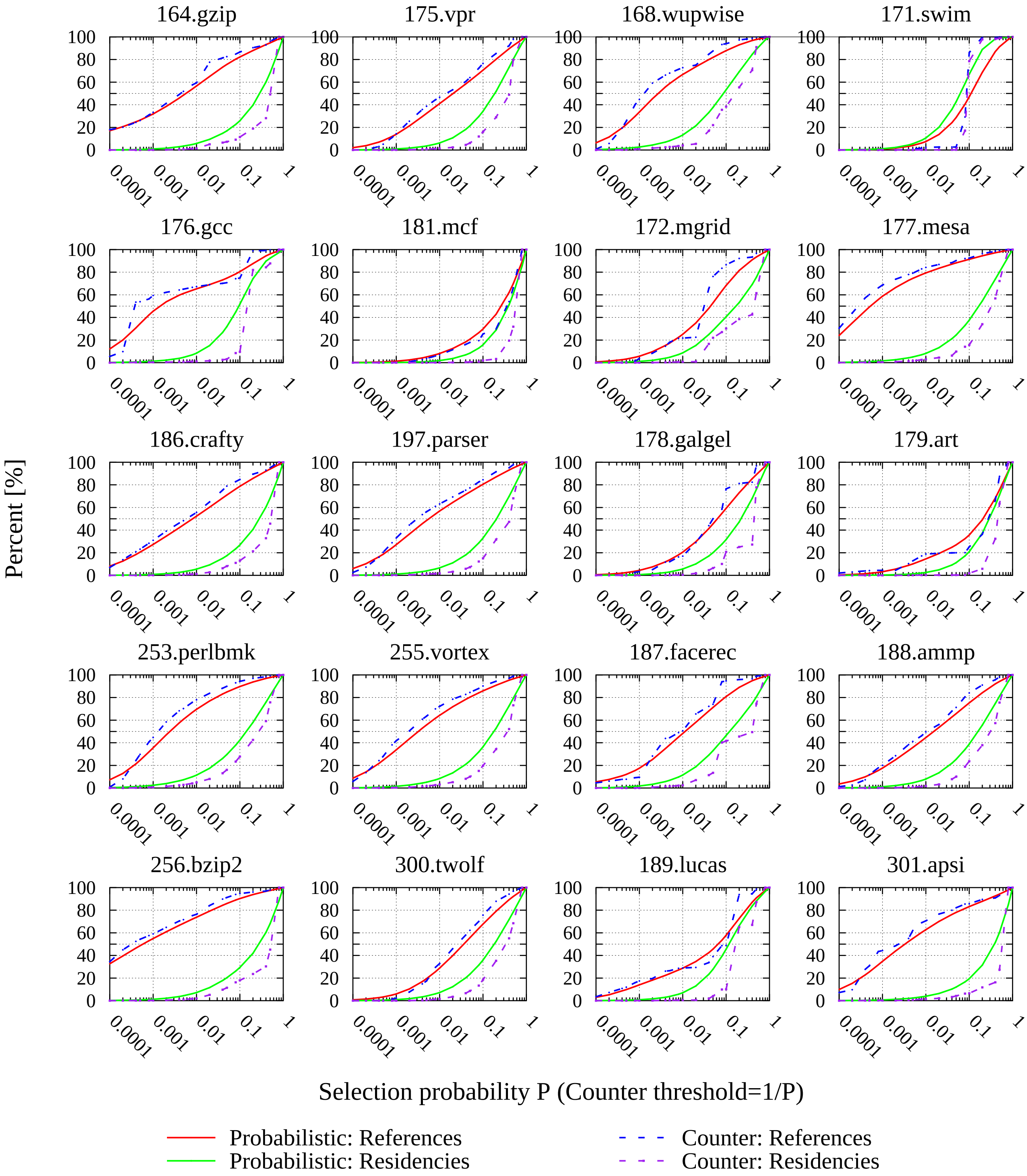

Figure 5.2: ***Data cache, 16KB direct-mapped**: Fraction of blocks sampled by the probabilistic predictor and the percent memory references they service, compared to those of the counter-based predictor.*

Table 5.2: ***Instructions cache, 16KB direct-mapped***: *Percents of residencies (insertions) classified as core and the references they service, for* $P = 0.001$, $P = 0.01$ *and* $P = 0.05$.

	$P = 0.001$		$P = 0.01$		$P = 0.05$	
Benchmark	%Ins	%Refs	%Ins	%Refs	%Ins	%Refs
164.gzip	4.50	82.34	28.23	89.06	63.50	95.27
175.vpr	3.49	88.40	21.98	92.77	54.96	96.53
176.gcc	2.86	77.19	18.89	85.06	51.61	92.17
181.mcf	100.00	100.00	100.00	100.00	100.00	100.00
186.crafty	2.63	26.64	18.42	49.17	52.56	72.16
197.parser	2.80	80.93	20.20	86.34	56.79	92.63
253.perlbmk	3.80	36.18	17.90	70.17	48.61	84.44
255.vortex	2.54	18.84	17.27	45.44	49.92	69.73
256.bzip2	67.70	99.97	78.71	100.00	88.40	100.00
300.twolf	4.56	41.09	26.51	66.08	61.12	84.99
168.wupwise	5.49	91.69	22.64	96.61	54.45	98.43
171.swim	5.12	99.87	15.21	99.96	42.76	99.98
172.mgrid	16.03	99.74	27.17	99.96	51.02	99.98
177.mesa	2.13	73.08	15.66	80.39	49.67	88.07
178.galgel	80.95	100.00	89.29	100.00	92.86	100.00
179.art	27.01	99.91	38.87	99.99	67.99	100.00
187.facerec	24.49	99.51	49.91	99.90	60.68	99.98
188.ammp	8.22	88.96	30.93	95.88	67.30	98.28
189.lucas	36.24	99.99	44.83	100.00	65.27	100.00
301.apsi	5.54	46.07	33.79	67.31	72.45	86.56
Average	20.30	77.52	35.82	86.20	62.60	92.96
Median	5.49	88.96	27.17	95.88	60.68	98.28

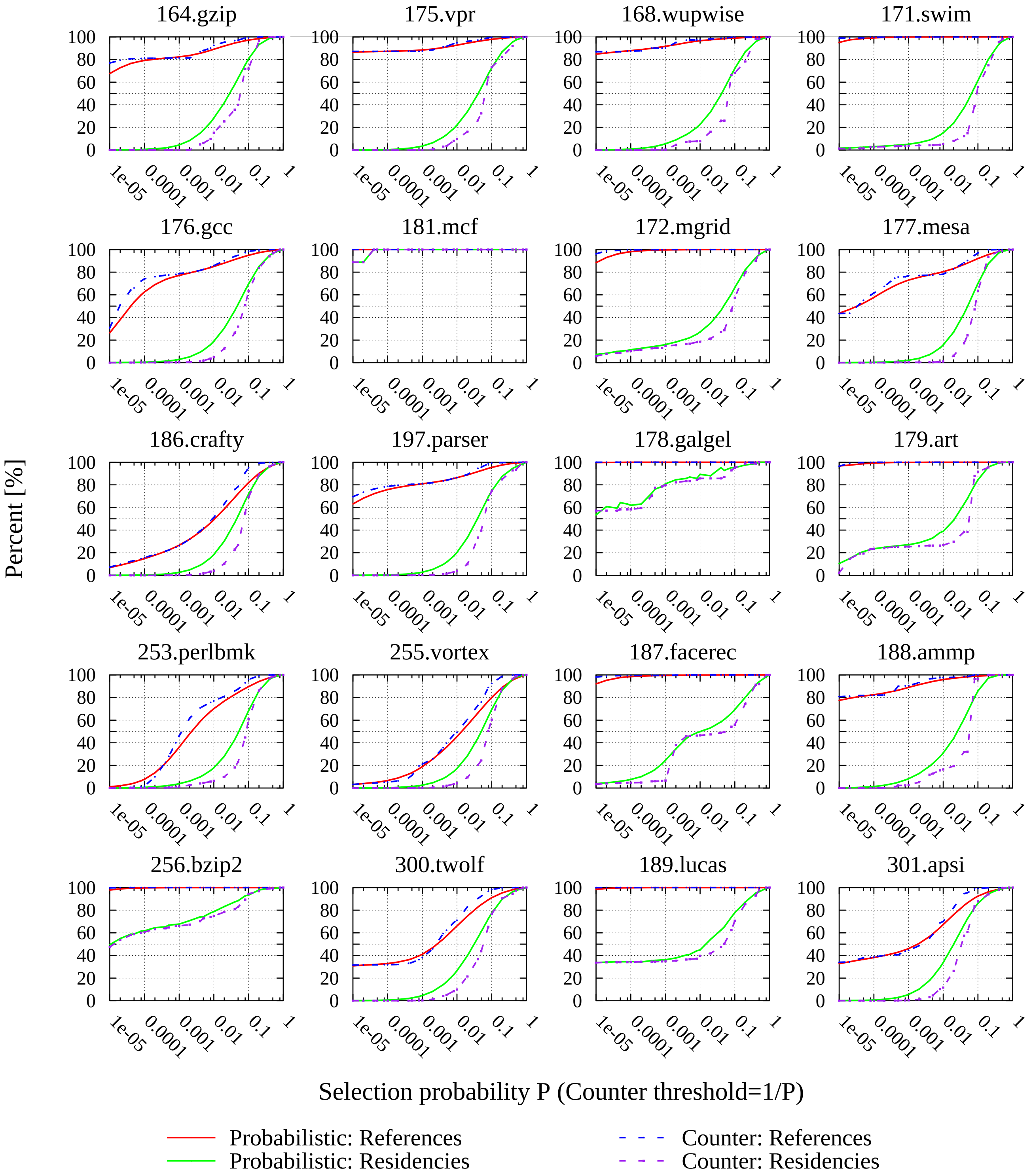

Figure 5.3: ***Instruction cache, 16KB direct-mapped:*** *Fraction of blocks sampled by the probabilistic predictor and the percent memory references they service, compared to those of the counter-based predictor.*

Table 5.3: ***Data cache, 16KB 4-way set associative****: Percents of residencies (insertions) classified as core and the references they service, for* $P = 0.001$*,* $P = 0.01$ *and* $P = 0.05$.

	$P = 0.001$		$P = 0.01$		$P = 0.05$	
Benchmark	%Ins	%Refs	%Ins	%Refs	%Ins	%Refs
164.gzip	0.69	55.30	4.94	68.51	15.29	81.28
175.vpr	0.88	49.99	6.18	65.10	22.66	75.45
176.gcc	1.13	62.95	9.04	70.49	35.14	78.00
181.mcf	0.19	6.86	1.76	14.51	8.10	24.67
186.crafty	1.02	51.57	5.60	73.68	17.03	84.66
197.parser	1.15	55.95	6.90	73.81	23.04	83.23
253.perlbmk	3.05	53.74	13.67	79.43	34.79	89.96
255.vortex	2.20	57.92	9.97	81.15	27.01	90.20
256.bzip2	1.15	68.91	7.41	80.12	20.91	89.24
300.twolf	1.05	25.88	7.67	46.52	26.20	64.89
168.wupwise	1.64	76.67	14.45	79.70	49.03	86.79
171.swim	1.12	3.34	10.68	8.91	43.20	29.28
172.mgrid	1.57	17.30	12.99	32.08	43.71	56.16
177.mesa	4.73	77.62	19.20	90.99	49.67	95.53
178.galgel	0.58	19.16	3.86	46.61	14.57	61.35
179.art	0.24	19.89	2.38	22.35	11.27	29.53
187.facerec	1.17	62.60	9.45	70.22	31.92	80.67
188.ammp	1.01	42.49	7.40	58.11	25.60	72.17
189.lucas	0.79	31.02	7.44	35.72	30.82	48.91
301.apsi	0.74	55.15	4.15	74.99	12.88	85.12
Average	1.31	44.72	8.26	58.65	27.14	70.35
Median	1.12	53.74	7.44	70.22	26.20	80.67

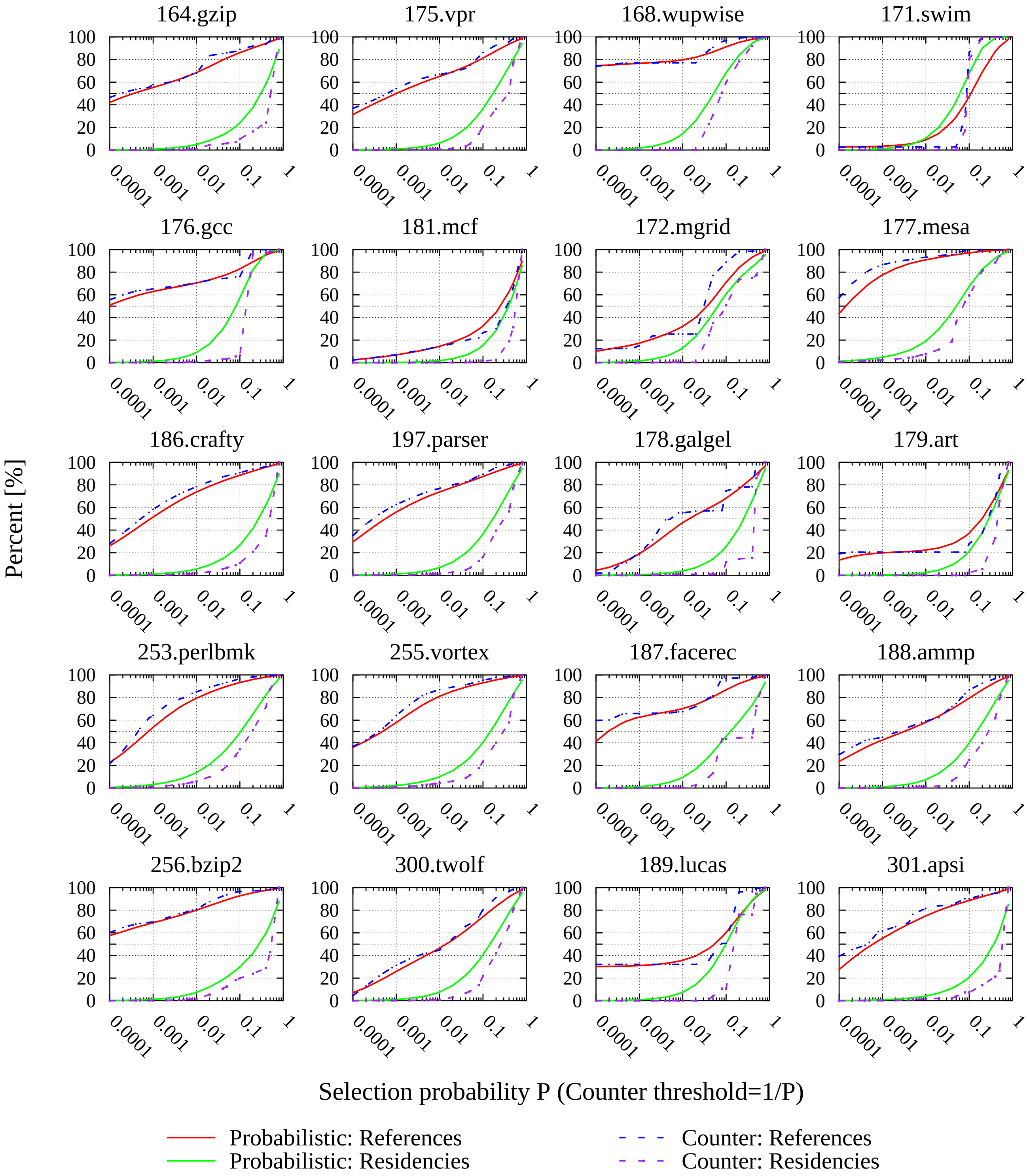

Figure 5.4: ***Data cache, 16KB 4-way set-associative****: Fraction of blocks sampled by the probabilistic predictor and the percent memory references they service, compared to those of the counter-based predictor.*

Table 5.4: ***Instructions cache, 16KB 4-way set associative**: Percents of residencies (insertions) classified as core and the references they service, for $P = 0.001$, $P = 0.01$ and $P = 0.05$.*

	$P = 0.001$		$P = 0.01$		$P = 0.05$	
Benchmark	%Ins	%Refs	%Ins	%Refs	%Ins	%Refs
164.gzip	86.56	99.99	94.62	100.00	96.77	100.00
175.vpr	51.50	99.99	64.97	100.00	79.04	100.00
176.gcc	3.09	77.80	20.25	85.53	53.46	92.69
181.mcf	100.00	100.00	100.00	100.00	100.00	100.00
186.crafty	3.67	23.10	22.13	53.96	56.07	77.61
197.parser	32.11	98.79	57.35	99.79	77.05	99.94
253.perlbmk	5.46	42.20	22.31	76.55	54.87	88.94
255.vortex	3.45	38.35	21.06	62.73	56.15	80.93
256.bzip2	65.94	99.97	75.96	100.00	88.56	100.00
300.twolf	5.93	81.32	30.44	90.49	64.16	96.16
168.wupwise	87.60	99.99	92.56	100.00	95.45	100.00
171.swim	10.35	99.85	24.90	99.96	56.13	99.98
172.mgrid	21.62	99.71	34.67	99.95	57.71	99.99
177.mesa	14.24	92.29	37.99	97.97	68.36	99.30
178.galgel	80.95	100.00	86.90	100.00	91.67	100.00
179.art	93.98	99.99	98.80	100.00	99.40	100.00
187.facerec	38.34	99.88	52.35	99.98	70.79	100.00
188.ammp	59.06	99.92	67.65	99.99	82.37	100.00
189.lucas	40.97	99.98	49.82	100.00	71.15	100.00
301.apsi	5.35	41.85	34.95	62.32	75.86	84.33
Average	40.51	84.75	54.48	91.46	74.75	95.99
Median	38.34	99.88	52.35	99.98	75.86	100.00

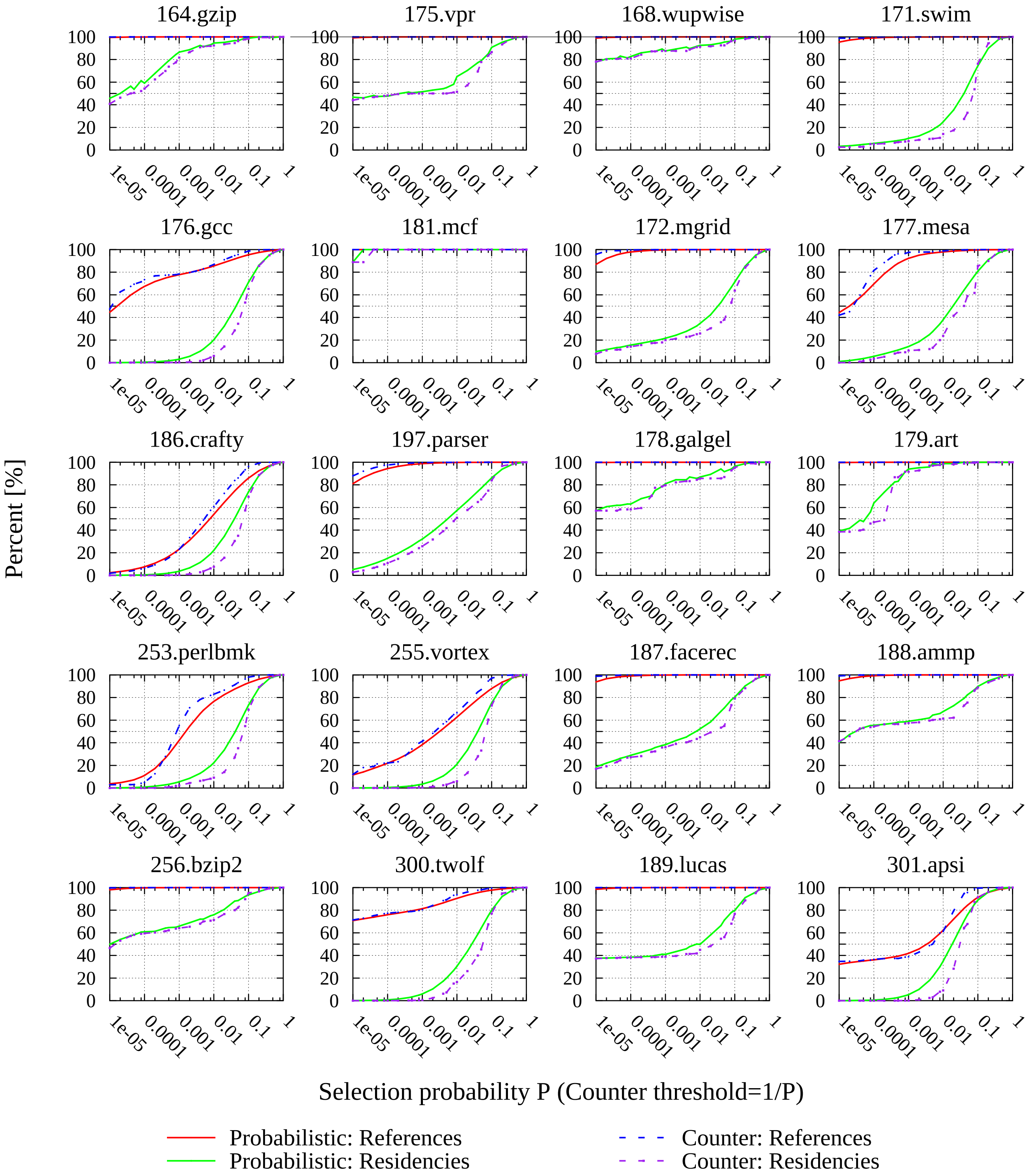

Figure 5.5: ***Instruction cache, 16KB 4-way set-associative***: *Fraction of blocks sampled by the probabilistic predictor and the percent memory references they service, compared to those of the counter-based predictor.*

Chapter 6

A Random Sampling L1 Cache Design

Based on the principles described in the previous chapters, this chapter introduces a novel L1 cache design that uses Bernoulli trials to distinguish long residencies from short, transient ones. As the long residencies represent a small subset of the working set that service the majority of references, by identifying these residencies and servicing them using power-efficient, direct-mapped L1 caches, we can potentially increase CPU performance and at the same time reduce the power consumption.

Direct-mapped caches are faster and consume less energy than set-associative caches typically used in L1 caches [28, 38]. However, they are more susceptible to conflict misses than set-associative caches, thus suffering higher miss-rates and achieving lower performance [32]. This deficiency led to abandoning direct-mapped L1 caches in favor of set-associative ones in practically all but embedded processors. The ability to partition the reference stream into long and short residencies, enables to serve only the small set of long residencies from the direct-mapped cache, thus harnessing its power and performance traits, while dramatically reducing the number of cache conflicts.

The use of a direct-mapped cache for servicing the core is supported by Figure 4.1, which demonstrates how modern set-associative caches service most of the references from the cache sets' MRU position, thus acting as de-facto direct-mapped caches, but with set-associative access times and power consumption. This phenomenon was even exploited by Flautner et al.

[21] to reduce the power consumption of set-associative cache by putting the non-MRU line into a low power / longer latency state.

The rest of this chapter proposes a design for a random sampling cache based on dual-cache paradigm, that employs a direct-mapped structure to serve the core working set, and a fully-associative structure acting as filter serving 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.

6.1 Proposed Design

The proposed design, based on the dual cache paradigm, is depicted in Figure 6.1. It consists of a direct-mapped cache preceded by a small, fully-associative filter. When a memory access occurs, the data is first searched in the cache proper, and only if that misses the filter is searched. If the filter misses as well, the request is sent to the next level cache. In our experiments we have used 16K and 32K (common L1 sizes) for the direct-mapped cache, and a 2K fully-associative filter (all structures use 64B lines).

Each memory reference that is serviced either by the filter or by the next level cache initiates 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) which performs the Bernoulli trials, and writes the block to the cache if selected. In case the block is not selected, and was not already present in the filter, the MRSU inserts it into the filter.

The MRSU can in fact perform the sampling itself even before the data is fetched, enabling it to perform any necessary eviction (either from the cache proper of the filter) beforehand, thus overlapping the two operations. Section 6.2.1 explores the probabilistic design space for a suitable Bernoulli success probability.

For a desired threshold probability P we pre-calculate a constant C_P such that $\frac{C_P}{2^K} \simeq P$.

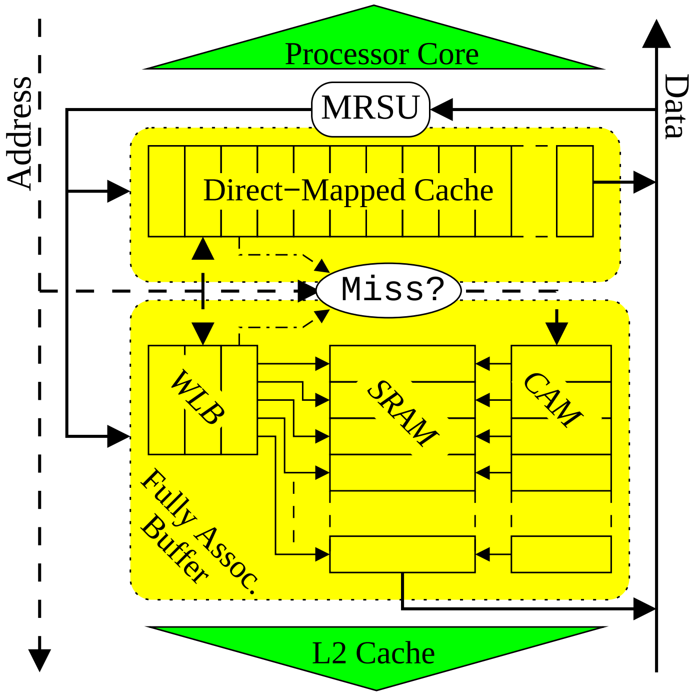

Figure 6.1: *Design of a random sampling filtered cache.*

Given a source of random bits, the MRSU generates a random integer r in the range $\left[1 \ldots 2^K\right)$. Therefore, the result of the comparison $r \leq C_P$ yields *true* with probability $\sim P$.

Although such a mechanism is easy to implement (e.g. using a linear-feedback shift register [81]) and consumes negligible power, we also experimented with naive periodic sampling, using a period proportional to $\frac{1}{P}$ (possibly implemented using a $log_2\left(\frac{1}{P}\right)$-bit saturating counter). This achieved results similar to those of random sampling. We therefore only show the results for random sampling.

To reduce both time and power overheads associated with accessing the fully-associative filter, we have augmented the classic CAM / SRAM design [14, 81] with a *wordline look-aside buffer* (WLB) that caches recent lookups to the fully-associative buffer, thus eliminating repeating fully-associative lookups and saving the power/performance cost associated with them. The WLB is a small direct-mapped cache structure mapping block tags directly to the filter's SRAM based data store, thus avoiding the majority of the costly CAM lookups, while still maintaining fully-associative semantics. Section 6.3 offers a detailed description of the WLB design, and an analysis of the WLB performance to determine the number of entries it requires.

Dual caches, and specifically filtered caches have been proposed in various contexts in the past, using various filtering metrics [37, 64, 63, 46, 62, 5]. However, none of the aforementioned studies offer such a simple categorization mechanism to distinguish between frequently used and transient memory blocks using random sampling, eliminating the need to maintain any per-block reuse history information. This work is compared and contrasted with such alternatives in Chapter 7.

6.2 The Effects of Random Sampling

Random sampling of memory references can be viewed as the partitioning of the reference stream into two components — one consisting of long residencies, and the other consisting of short transient residencies. This partitioning enables treating each component of the workload using a special caching structure that is better suited to service the blocks (and thereby residencies) composing that component. The number of references to frequently used blocks are numerous, but involve only a relatively small number of distinct blocks. This reduces the number of conflict misses, enabling the use of a low-latency, low-energy, direct-mapped cache structure. On the other hand, transient residencies compose the majority of residencies, but naturally have a shorter cache lifetime. Therefore, they can be served by a smaller, fully-associative (and costly) structure.

The filtering rate and probability therefore poses 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, making the filter a bottleneck and degrade performance. On the other hand, loose filtering may promote too many blocks to the main direct-mapped cache, thereby saturating the direct-mapped structure, increasing the number of conflict misses and degrading its performance.

This section is therefore dedicated to evaluate the effectiveness of probabilistic filtering, while exploring the statistical design space. The selected parameters are then used to evaluate performance and power consumption in Section 6.4.

6.2.1 Impact on Miss-Rate

First, we address the effects of filtering on the overall miss-rate in order to determine the Bernoulli probabilities that yields best cache performance. Figure 6.2 shows the distributions of the miss-rate achieved by a filtered 16K direct-mapped cache (fraction of blocks missed by both the cache *and* the filter) compared to that achieved by a regular 16K direct-mapped cache, for various Bernoulli success probabilities (lower values indicate a decreased miss rate). The data shown for each combination are a summary of the observed change in miss rate over all benchmarks simulated: the distribution's middle range (25%–75%), average, median and min/max values. An ideal combination would yield maximal overall miss-rate reduction with a dense distribution, i.e. a small differences between the 25%–75% percentiles and min–max 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%, and is 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: when comparing the center of these distributions — the 25%–75% box — we can see the entire distribution is moved downwards. The same can be said about the miss-rate reduction in the instruction stream, for which selection probabilities of 0.01 to 0.0001 all achieve an average improvement of $\sim$60%. In this case as well the best averages are achieved for probabilities that shift the entire distribution downwards.

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, for the 16K cache configuration. In a similar manner, probabilities of 0.1 and 0.0005 were selected for the data and instruction streams, respectively, for the 32K configuration.

Interestingly, the data and instruction stream require different Bernoulli success probabilities — with two orders of magnitude difference! The reason for this is the fact that the instruction memory blocks are usually accessed over an order of magnitude more times compared

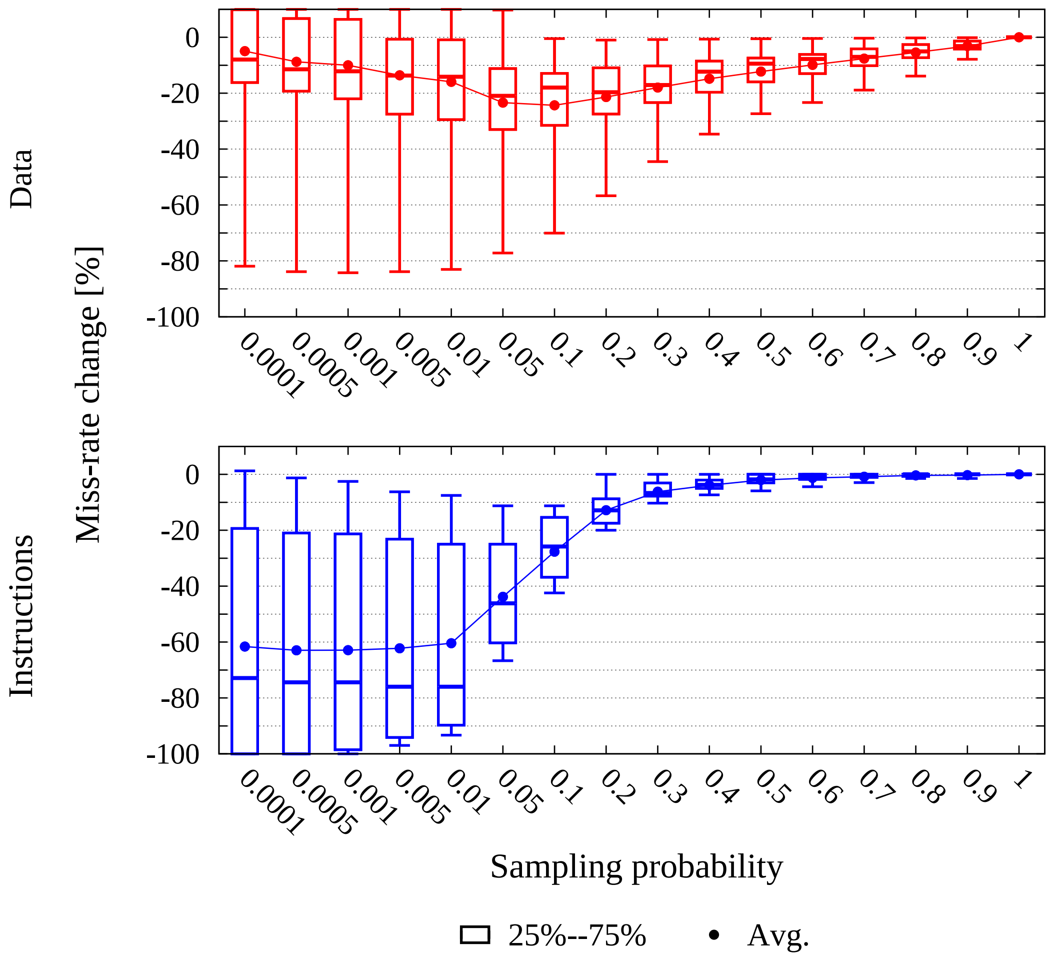

Figure 6.2: *Comparison of SPEC2000 instruction and data miss-rate distributions, using various sampling probabilities, for a 16K-DM cache. The boxes represent the 25%–75% percentile range, and the whiskers indicate the min/max values. Inside the box are the average (circle) and median (horizontal line).*

to data blocks. In the benchmarks shown in Figure 3.2, 50% of the data memory blocks are accessed 1–2 times while in the cache, whereas the same percentile of instruction blocks are accessed 10–15 times. This difference is mainly attributed to the fact that instruction memory blocks are mostly read sequentially as blocks of instructions.

6.2.2 Impact on Reference Distribution

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

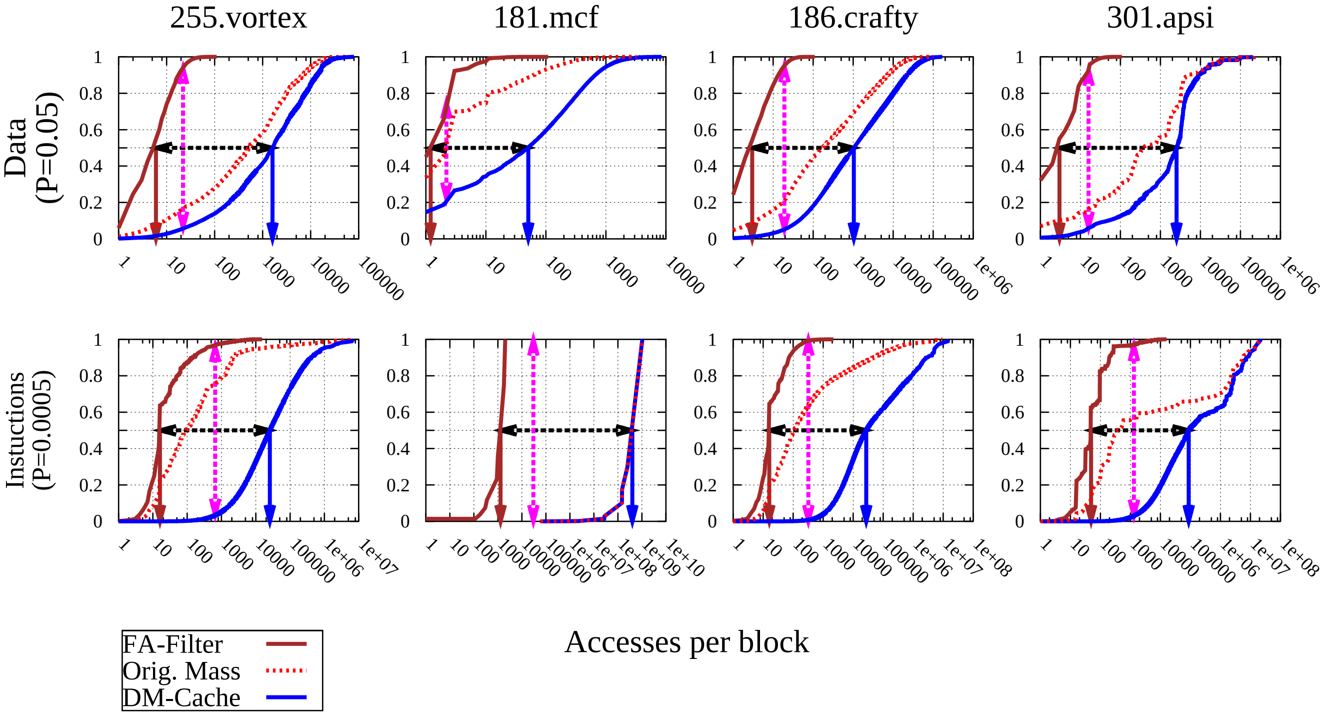

Figure 6.3: *Comparison of the data references' mass distributions in the filtered cache structure and the regular cache structure for select SPEC2000 benchmarks using the ref input, for both data (top) and instruction (bottom). The horizontal arrows show the median-to-median range, and the vertical arrows show the false-$*$ equilibrium point.*

in splitting the distribution of memory references.

Figure 6.3 compares distributions of reference masses — the fraction of references serviced by each residency length — of the filtered 16K cache and the original 16K direct-mapped cache. Results are shown for select SPEC2000 benchmarks with Bernoulli probabilities of 0.05 for data streams and 0.0005 for instruction streams. The probability selection is based on the analysis described in Section 6.2.1.

Each plot shows three lines: the distributions for the direct-mapped cache and fully-associative filter for the filtered design, and the original distribution for a conventional direct-mapped cache — which is the combination of the first two (this is the same distribution as the one shown in Figure 3.2). The distributions for the filter and the cache account for residencies that are split because the block was promoted to the cache — references to the block serviced while the block was in the filter are counted as an individual filter residency, whereas the references serviced from the cache itself after the block was promoted are counted as a separate cache residency. The median value of the two filtered distributions is marked with a down pointing arrow. Invariably, the distributions show that the majority of references directed at the filter are part of residencies much shorter than those in the direct-mapped cache proper, which in turn serve the majority of the references. (Figures 6.4 and 6.5 display the naked data and instruction distributions, respectively, for all SPEC2000 benchmarks' reviewed).

The qualitative difference between the two resulting distributions is estimated using two intuitive metrics: median ratio (marked with a horizontal double arrow) and false-$*$ equilibrium (marked with a vertical double arrow). Tables 6.1 and 6.2 list the two metrics' values for all SPEC2000 benchmarks data and instruction streams, respectively.

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

The median ratios for all benchmarks' data streams are measured at $50 - 10^4$, with an average ratio of $\sim$320 (median $\sim$180), with the instruction streams' median ratio averaging at $\sim$50,000 (median $\sim$4,400). In practice, this result indicates that the median residency in the direct-mapped cache is several orders of magnitude longer than the median residency in its corresponding fully-associative for filter, demonstrating the effectiveness of the design in splitting the original reference (mass) distribution.

The second metric is denoted as the *false-$*$ equilibrium*, and is an estimate of false predictions: Any given residency length threshold we choose in hindsight will show up on the plot as a vertical line, with a fraction of the cache's distribution to its left indicating the false-positives (short residencies promoted to the cache), and a fraction of the filter's distribution to its right indicating the false-negatives (long residencies remaining in the filter). Obviously, choosing another threshold will either increase the fraction of false-positives *and* decrease the fraction of false-negatives, or vice versa. The false-$\ast$ equilibrium is a unique threshold that if chosen, generates equal percentages of false-positives and false-negatives, thereby serving as an upper bound for the overall percentage of false predictions.

For example, if we examine *vortex*'s data stream we see that the false-* equilibrium point stands at a residency length of $\sim$20 and generates $\sim$6% false predictions ($\sim$3% for the instruction stream). The false prediction rate for the cache unfriendly *mcf* data stream was found to be $\sim$22%, which although tolerable was among the highest values observed. This is caused by the large number of short residencies in the original reference stream — which is actually dominated by these residencies. These short residencies swamp the filter causing a dual effect: the short residencies push blocks whose residencies can potentially grow to be long out of the filter, while the random sampling algorithm statistically selects more short residencies because of their sheer number. The result is consistent with other cache unfriendly benchmarks such as *swim* (over 40% false predictions) and *art* (over 20% false predictions). Still, the overall average percentage of false predictions for the data streams was found to be $\sim$13%, with $\sim$2% for the instruction streams — a fairly good upper bound considering it is based on stateless random sampling.

Table 6.1: *L1* ***data*** *cache, selection probability* $P = 0.05$*: Median/Median ratios and false-*$*$ *equilibrium values for all SPEC2000 benchmarks reviewed.*

Benchmark	Med-to-Med ratio	False-$*$ Equ.	False-$*$ @
164.gzip	840	6 / 94	23
175.vpr	90	12 / 88	12
176.gcc	480	11 / 89	8
181.mcf	26	22 / 78	2
186.crafty	350	5 / 95	19
197.parser	240	8 / 92	16
253.perlbmk	240	5 / 95	24
255.vortex	270	6 / 94	22
256.bzip2	1900	5 / 95	25
300.twolf	29	14 / 86	11
168.wupwise	180	8 / 92	20
171.swim	1.1	43 / 57	8
172.mgrid	3.6	12 / 88	14
177.mesa	600	5 / 95	32
178.galgel	57	15 / 85	10
179.art	140	23 / 77	3
187.facerec	48	15 / 85	16
188.ammp	58	12 / 88	14
189.lucas	4.1	26 / 74	8
301.apsi	840	6 / 94	16
Average	320	12 / 87*	15
Median	180	12 / 89*	16

* Average and median False-$*$ equilibria values are calculated independently and thus may not sum up to 100%.

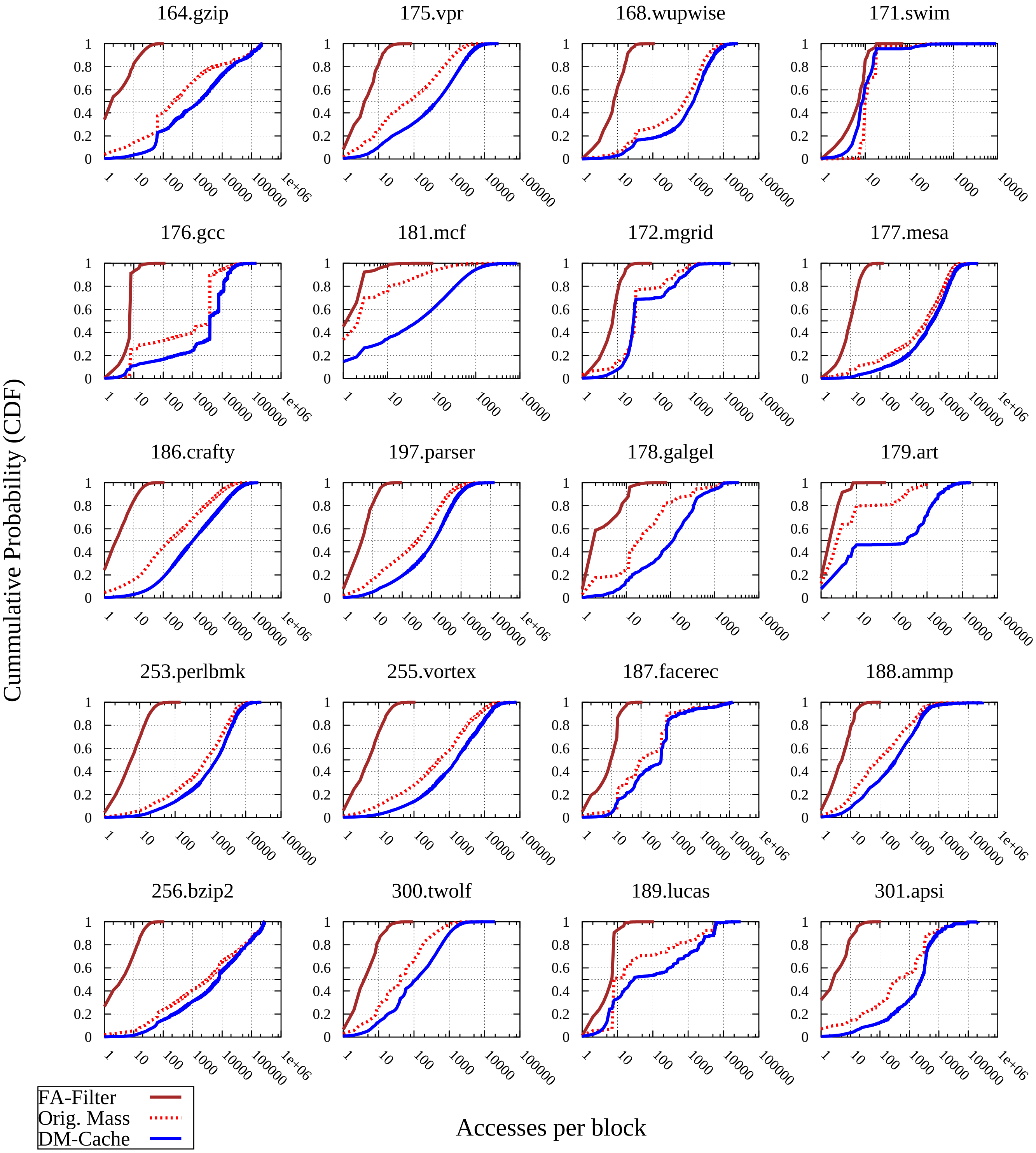

Figure 6.4: *Comparison of the* ***data*** *references' mass distributions in the filtered cache structure and the regular cache structure for all SPEC2000 benchmarks.*

Table 6.2: *L1* ***instruction*** *cache, selection probability $P = 0.0005$: Median/Median ratios and false-$*$ equilibrium values for all SPEC2000 benchmarks reviewed.*

Benchmark	Med-to-Med ratio	False-$*$ Equ.	False-$*$ @
164.gzip	5.5 e+3	3 / 97	9715
175.vpr	1.4 e+4	2 / 98	10298
176.gcc	2.2 e+4	3 / 97	2438
181.mcf	8 e+5	0 / 100	29946
186.crafty	1.7 e+3	1 / 99	320
197.parser	4.3 e+2	6 / 94	7995
253.perlbmk	6.3 e+2	6 / 94	1327
255.vortex	1.6 e+3	3 / 97	655
256.bzip2	7.3 e+3	0 / 100	9591
300.twolf	1.7 e+3	2 / 98	1429
168.wupwise	9.5 e+3	1 / 99	11865
171.swim	1.6 e+3	0 / 100	19332
172.mgrid	2.1 e+3	0 / 100	13748
177.mesa	2.1 e+3	9 / 91	5918
178.galgel	6.5 e+4	0 / 100	23124
179.art	4.6 e+4	0 / 100	11266
187.facerec	4.4 e+3	0 / 100	12051
188.ammp	5.9 e+3	6 / 94	8462
189.lucas	2.7 e+3	0 / 100	31174
301.apsi	1.8 e+3	3 / 97	1307
Average	5 e+4	2 / 97	10598
Median	4.4 e+3	2 / 99	9715

* Average and median False-$*$ equilibria values are calculated independently and thus may not sum up to 100%.

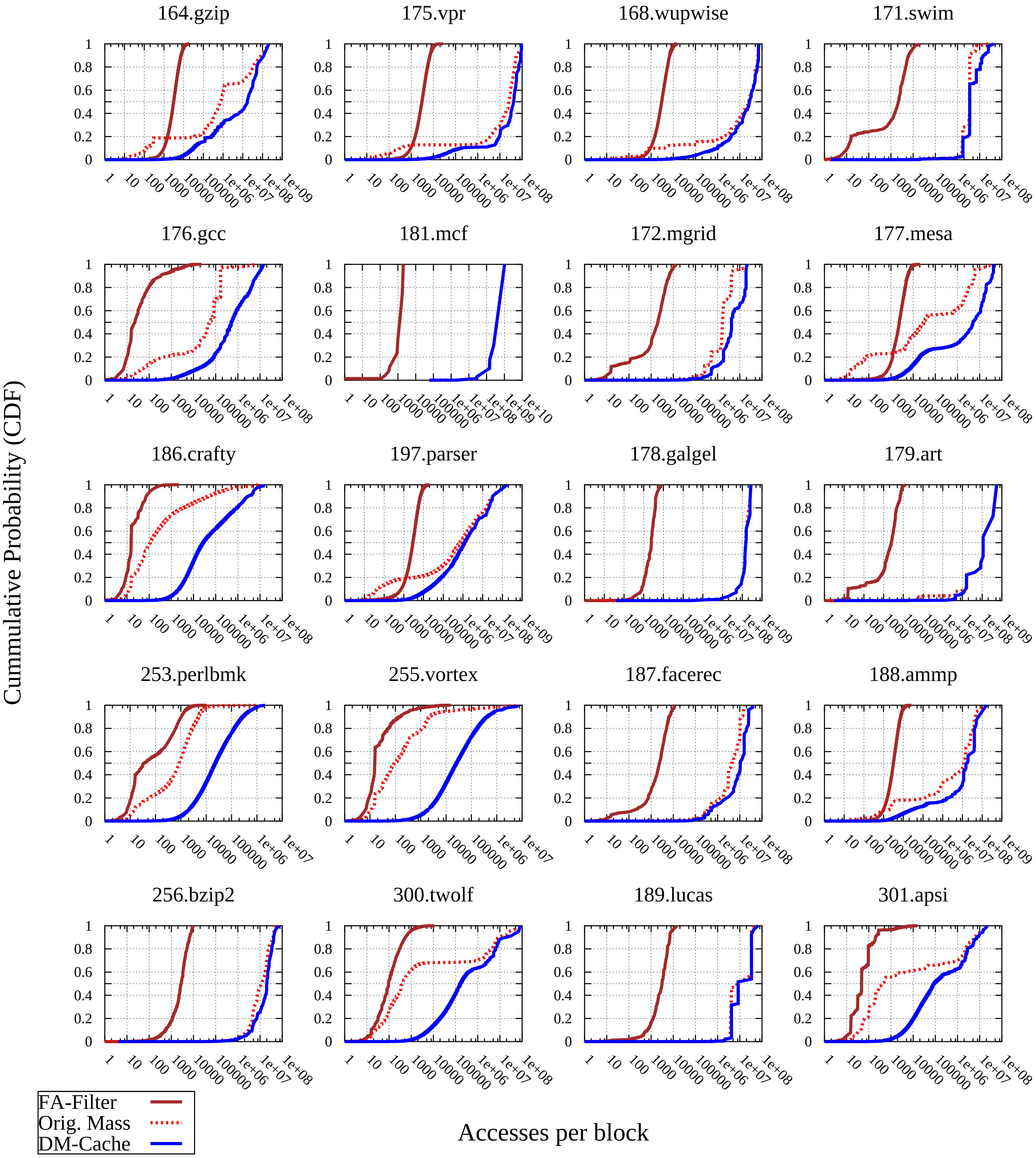

Figure 6.5: *Comparison of the* ***instruction*** *references' mass distributions in the filtered cache structure and the regular cache structure for all SPEC2000 benchmarks.*

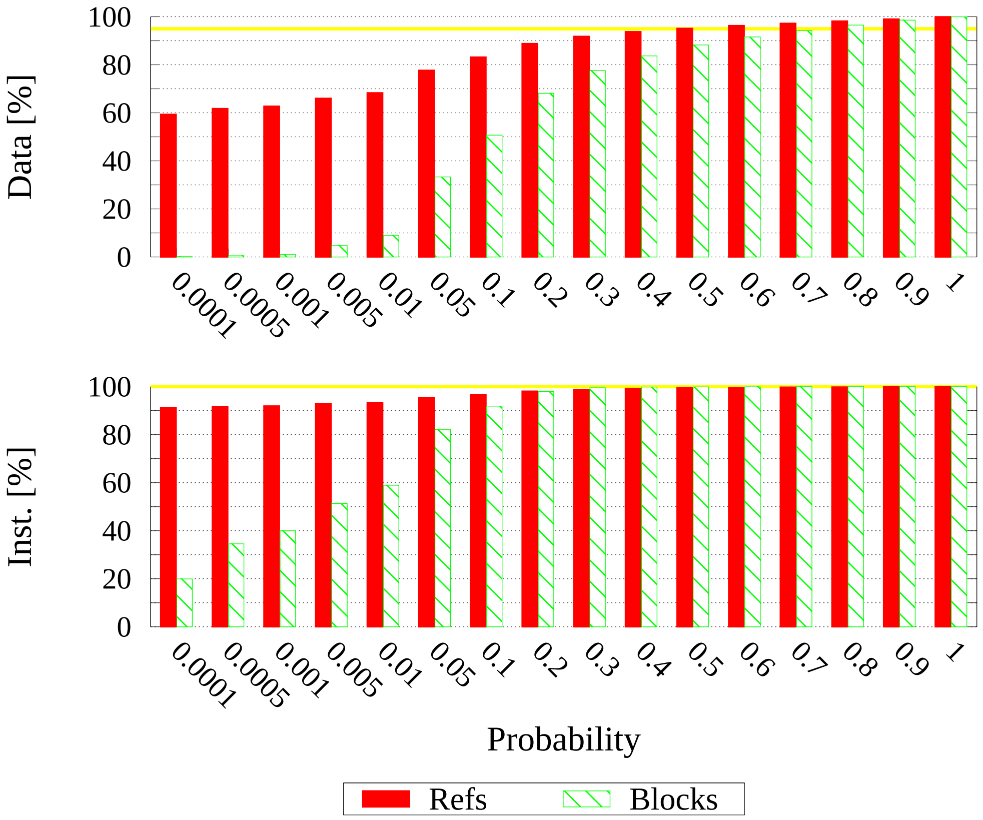

Figure 6.6: *Percent of references serviced by the cache vs. the percent of blocks transferred from the filter into the cache for varying sampling probabilities (averages over all benchmarks). The horizontal lines near the top of the figure indicate the asymptotic maximum of references serviced by the cache. Cache size was 16K, with a 2K filter.*

Another aspect of the reference distributions is the number of references in each distribution, compared with the number of residencies served by the cache and the filter. Figure 6.6 shows the percentage of references serviced by the cache, compared with the percentage of blocks promoted into the cache, for various probabilities. Considering the mass-count disparity we expect that promoting frequently accessed *blocks* into the cache will result in a substantial increase in the number of *references* it will service, and that promoting not-so-frequently used blocks have a smaller impact on the number of references serviced by the cache. This is indeed evident in Figure 6.6: when increasing the success probabilities we see a distinctive increase in the number of references serviced by the cache, until some level — indicated by the horizontal line — where this increase slows dramatically and promoting more blocks into the cache hardly increases the cache's hit-rate. In our case this saturation occurs at $P = 0.2$ for the data and $P = 0.05$ for the instructions. Beyond these probabilities the promoted blocks are mostly transient blocks and we start experiencing diminishing returns. Specifically, the chosen probabilities insert an average of only $\sim$33% of the data blocks into the cache proper, servicing $\sim$77% of the data references ($\sim$35% / $\sim$92% for instructions streams).

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

6.3 The Wordline Look-aside Buffer

A fully-associative caching element like the filter, introduces long access latencies and increased power consumption that are mostly caused by the fully-associative block lookup. Such an element is commonly implemented using content-addressable-memory (CAM) serving as a tag-store, and whose wordline are connected to the wordlines of an SRAM block, serving as the data-store [14, 81], as shown in Figure 6.7 (left). Temporal locality only aggravates its impact on performance as it 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 design is shown in Figure 6.7 (right).

The WLB consists of a direct-mapped structure, mapping tags of filter-resident blocks to their location in the fully-associative buffer's SRAM structure. The data contained in the WLB for each tag is a bitmap whose width is similar to the 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 word-line without a decoder, offering a 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, eliminating the need for a costly CAM lookup 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. Furthermore, the number of entries in the WLB can be much smaller than the number of filter lines, as temporal locality also exists in the filter. This section

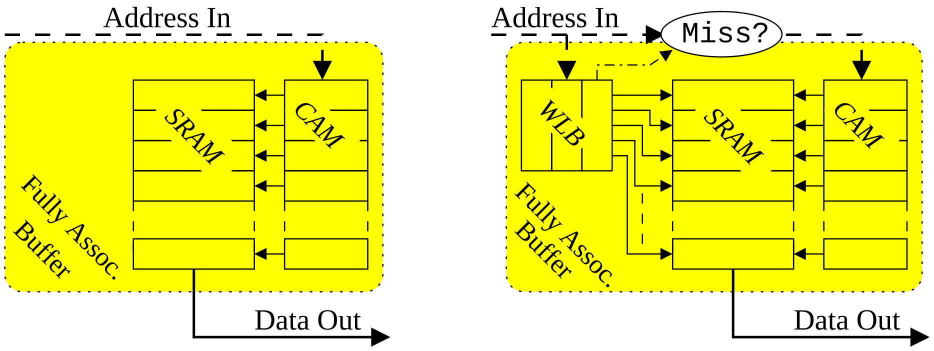

Figure 6.7: *A common design of a fully associative buffer using a CAM based tag-store and a SRAM data-store (left), and the design augmented with a WLB (right) to cache the mappings of recent lookups.*

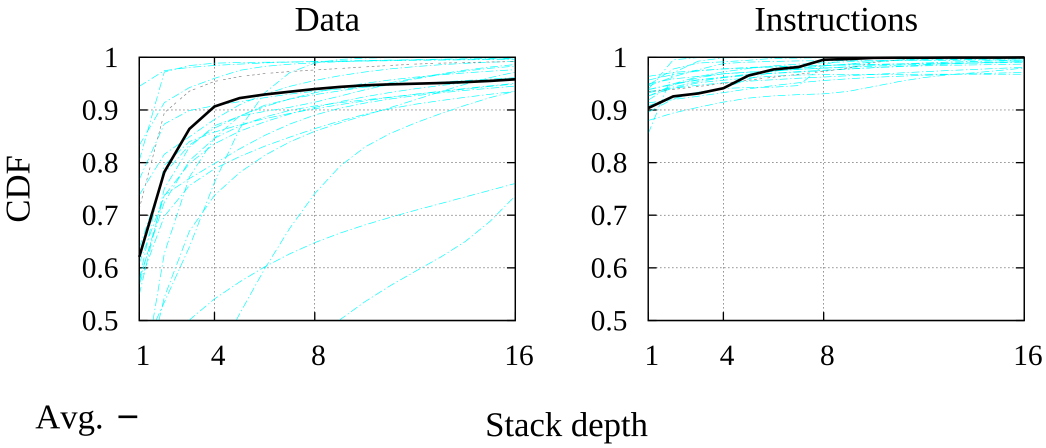

Figure 6.8: *Distributions of filter access depth for all SPEC2000 benchmarks, and the average distribution. Note that the vast majority of accesses are focused around the MRU position. The benchmarks are not individually marked as their specific behavior is irrelevant in this context, except for the three unclustered plots which represent (top to bottom)* swim*,* crafty*, and* art.

explores the WLB design space.

A basic question when exploring the design space of the WLB is to determine what may be an effective size for the buffer. Figure 6.8 shows the stack depth distributions of filter accesses, for the different SPEC2000 benchmarks, as well as the average distribution over all benchmarks. The various benchmarks are not individually marked as only the clustering of distributions matters in this context, but it is interesting to note that 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 (the design in fact reduces *crafty*'s miss-rate by $\sim$50%). Still, it is clear that overall the vast majority of accesses pertain to recently used blocks — in fact, on average $\sim$94% of data accesses are to stack depths of 8 or less, out of a total of 32 lines in the filter. But a WLB consisting of 8 entries may not be sufficient to simulate a stack depth of 8 accesses, because the WLB is a direct-mapped structure, and is thus susceptible to conflicts. A WLB consisting of N entries can therefore only *approximate* a stack of depth N. For this reason we have explored WLB sizes of 8 and 16 entries. In our experiments, we have found that using an 8 entry WLB achieves an average of $\sim$78% hit-rate for the data stream ($\sim$83% median) and over 97% for the instruction stream ($\sim$97% median) for a 2K filter. Doubling the WLB size to 16 entries only improves the average data hit-rate to $\sim$84% ($\sim$89% median) and $\sim$99% for the instruction stream ($\sim$99% median), but increased the dynamic power consumption by $\sim$10% and the leakage by $\sim$50% (with similar results for the 32K configuration). The conclusion was that although an 8-entry WLB loses performance to cache conflicts, its hit-rate is still good, especially given that doubling the size of the WLB to 16 entries only increases its hit-rate by a few percents.

We have therefore used an 8 entry WLB in our power and performance evaluation, eliminating almost 80% of the costly filter CAM lookups for the data cache, and 98% of those in the instruction cache. 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 6.2.2), these results yield that on average only $\sim$4% of the data references and $\sim$0.1% of the instruction references still initiate expensive fully-associative lookups. Furthermore, the small size of the WLB results in a negligible power consumption, and since its access time is shorter than that of the main direct-mapped cache, the WLB can be accessed in parallel to the main cache, reducing the average fully-associative filter's latency even further.

The WLB thus demonstrates that harnessing temporal locality we can dramatically reduce a fully-associative filter's power consumption, while improving its performance — without losing the fully-associative semantics.

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

Bernoulli probabilities		
Size	Data	Instruction
16K	$P = 0.05$	$P = 0.0005$
32K	$P = 0.1$	$P = 0.0005$

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

Table 6.3: *micro-architecture and cache configurations used in the out-of-order simulations.*

6.4 Impact on Power and Performance

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

Using the SimpleScalar toolset [4] for out-of-order simulations we have compared the performance achieved by direct-mapped filtered caches against various set-associative caches. Our micro-architecture consisted of a 4-wide superscalar design, whose parameters are listed in Table 6.3. Timing estimates are based on the CACTI 4.1 timing model: the direct-mapped cache's hit latency was set to 1 cycle, as is the lookup latency in the WLB, as well as the SRAM access time. The latency of a CAM lookup is estimated at 3 cycles.

Figure 6.9 shows the timing diagram of the different components in the proposed cache design. The main cache and WLB are searched in parallel during the first cycle. If the main cache misses, the result of the WLB lookup determines the filter lookup path: if the requested

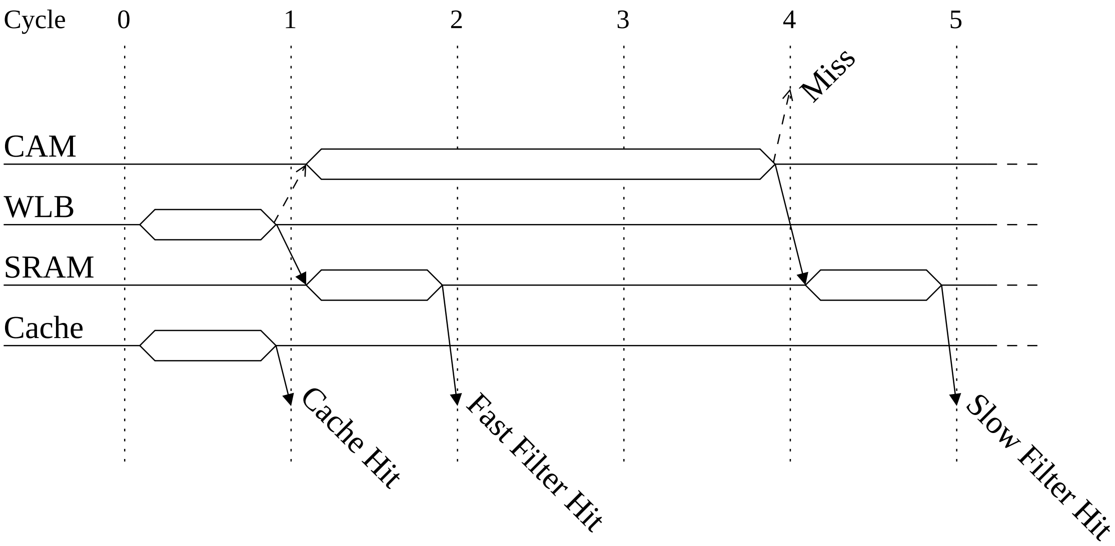

Figure 6.9: *Timing diagram of the cache design, based on the microarchitectural parameters listed in Table 6.3. Down pointing arrows indicate lookup hits, while up pointing arrows indicate lookup misses. Based on the results described in Section 6.3, almost 96% of all L1 hits are resolved within 2 cycles.*

block is found in the WLB then no CAM lookup is necessary, thus enabling a fast path by accessing the SRAM directly, and resulting in a 2 cycles total filter latency. Only if both the direct-mapped main cache *and* the WLB miss, the filter's slow path is needed and the CAM is looked up.

In addition, the hit latency incurred by set-associative caches was set to 2 cycles. For fully-associative caches we used an unrealistically fast 2 cycle latency — same as set-associative — placing both on a similar baseline, thus focusing on the reduced miss-rates achieved by fully-associative caches.

Figure 6.10 shows the IPC improvement achieved by a random sampling cache over a similar size 4-way associative cache, for the SPEC2000 benchmarks. The figure shows consistent improvements (up to ∼35% for a 16K configuration and ∼28% for 32K one), with an average IPC improvement of just over 10% for both 16K and 32K configuration.

While the results are consistent, it is clear that benchmarks suffering from conflict misses enjoy better performance gains. This is most pronounced for *apsi* that includes a large portion of short residencies — over 70% of all residencies consist of a single reference, as shown in Figure 3.4. Supporting this is the fact that doubling the cache size to 32K — thus reducing con-

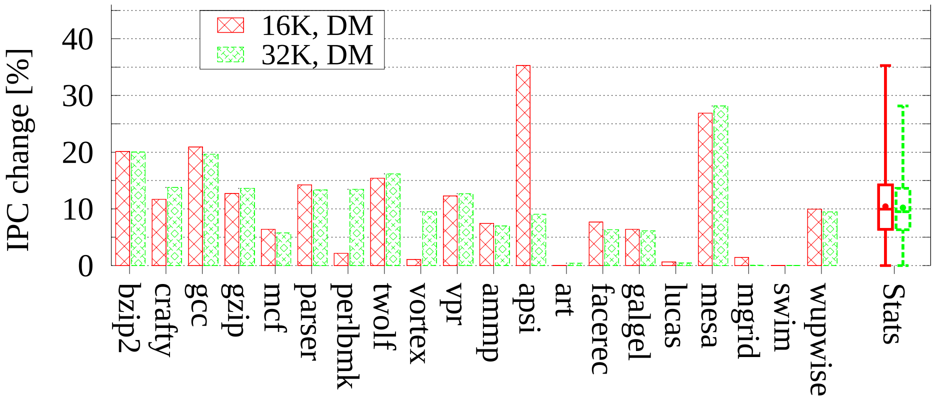

Figure 6.10: *IPC improvement for direct-mapped random sampling caches (using a 2K filter) over similar size 4-way caches.*

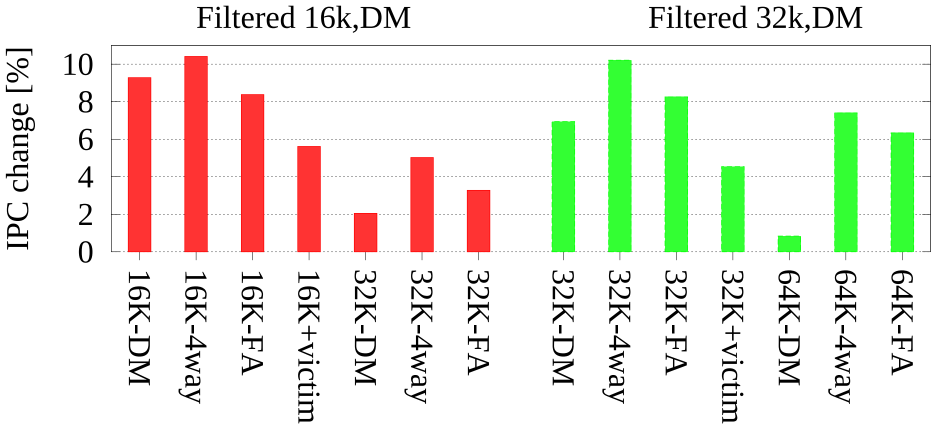

Figure 6.11: *Average IPC improvement for 16K and 32K direct-mapped filtered caches over common cache configurations.*

flicts by increasing the number of sets — decreases the performance gains for this benchmark, while other benchmarks remain largely unaffected.

Figure 6.11 compares the average performance achieved with 16K and 32K random sampling caches to that of common cache structures. It shows that a direct-mapped random sampling filtered cache achieves significantly better performance than a similar size set-associative cache. Moreover, a random sampling cache can even gain better overall performance than larger, more expensive caches: 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 them as a victim buffer, indicating that even such a relatively large victim buffer may be swamped by transient blocks.

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

Next, we compare the power consumption of the random sampling cache with that of the other configurations. Using independent random sampling eliminates the need to maintain any previous reuse information, reducing the power consumption calculation to averaging the energies consumed by the combination of a direct-mapped cache, a fully-associative filter, and a small, direct-mapped WLB. All power consumption estimates are based on the CACTI 4.1 power model [75] (which models CAMs for fully-associative structures).

The average dynamic energy consumption is simply the aggregate energy — the sum of

$$number\ of\ accesses \times access\ energy$$

for each component — divided by the overall number of hits. Even simpler, the leakage power consumed by the random sampling cache is the sum of leakage power consumed by all components.

Figure 6.12 shows both dynamic read energy and leakage power consumed by the random sampling cache, compared to common cache configurations (same as those in Figure 6.11). Obviously, the power consumed by the random sampling cache is higher than that of a simple direct-mapped cache, because of the fully-associative filter: up to $\sim$30% more dynamic energy and under 15% excess leakage power for a 16K random sampling cache (and just over half that for a 32K cache). However, when comparing a random sampling cache to a more common

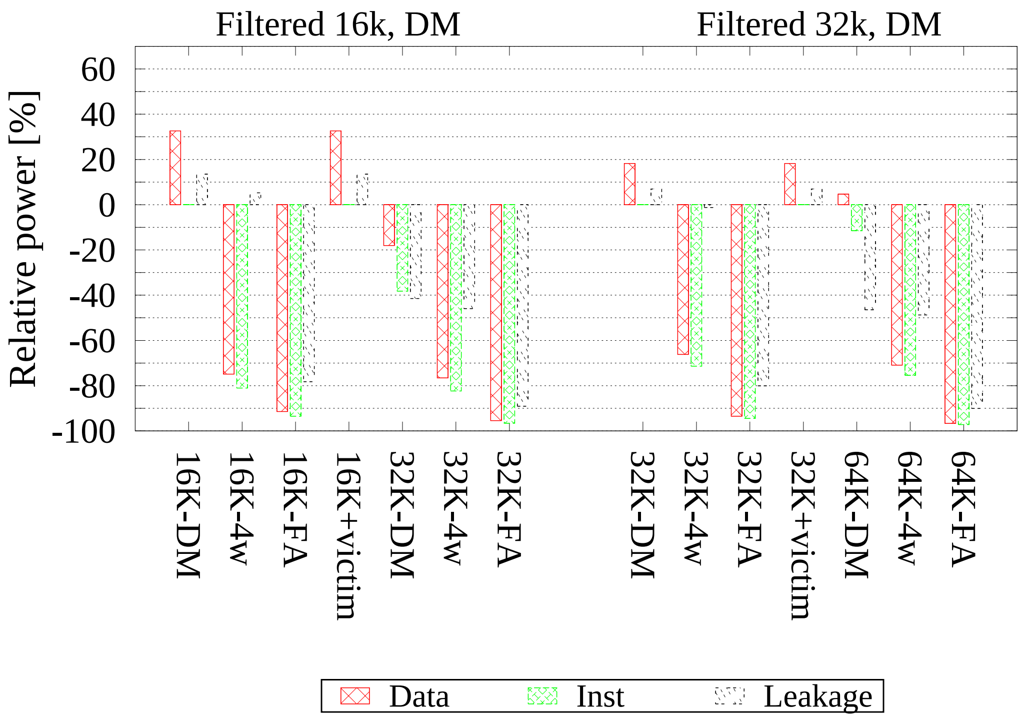

Figure 6.12: *Relative power consumption of the random sampling cache, compared to common cache designs (lower is better), for a 70nm process.*

4-way associative cache of a similar size, the 16K random sampling cache design 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.

However, the main contribution of a random sampling cache is apparent when compared to a set-associative cache double its size: both the 16K and 32K random sampling caches consume 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 Figure 6.11.

In summary, this chapter shows that a random sampling direct-mapped cache backed up by a fully-associative filter, offers performance superior to that of a double sized set-associative cache, while consuming considerably less power — both dynamic and static. This suggests that adding just a small buffer and a trivial insertion policy is more efficient than blindly doubling cache size.

Chapter 7

Related Work

The concepts presented in this work cover a variety of research topics covered by the different subsections in this review. The detailed review is preceded by a general overview of the work and its related topics.

The fundamental base of the research presented in this thesis is the identification of the *Mass-Count Disparity* phenomenon in L1 workloads, and its implications — the skewed distributions of memory access patterns, and specifically the partitioning of the reference stream into two parts, with the minor part addressing short cache residencies, and the major part addressing long cache residencies. But the phenomenon is not unique to cache workloads, and in fact present in different scientific disciplines, discussed in Section 7.1.

The partitioning of the reference stream into frequently used blocks (long cache residencies) and transient blocks that experience short term bursty access patterns (short cache residencies) suggests that reference streams commonly experience the duality of frequency vs. recency. The observation that memory access patterns may display different types of locality, possibly warranting different types of caching policies, has already motivated studies that tried to identify the frequently used blocks. This duality is manifested by the co-existence of two disjoint replacement policies, namely the *least-recently-used* (LRU) and the *least-frequently-used* (LFU) policies [32]. Attempts to combine characteristics from multiple replacement policies normally yield more complex policies, but a recent study showed a simple method to combine multiple separate replacement policies into a single meta-policy that chooses the best single policy for a momentary workload. Qureshi et al. explored several variations on *Cache Set Dueling*, where a small number of cache sets are allocated as placebo groups, each managed by a different replacement policy [55, 56, 54]. The cache controller tracks how each of the placebo groups performs, and switches the remaining bulk of cache sets to operate using the best performing policy. Although this adaptive mechanisms is currently only suitable for L2 caches because of its associated overheads, such meta-policies holds promise for future cache designs.

Interestingly, combining different replacement policies has been attempted in other caching disciplines, such as the software-based caching that is used in operating systems' buffer-caches [68]. Such software-based designs have the advantage of being able to incur a much more substantial runtime overhead. This advantage in fact makes buffer-cache studies an interesting source through which one can gain new insights and perspectives about caching. Section 7.2 reviews several buffer cache replacement algorithms.

The extremely skewed access distributions described above enables probabilistic sampling to be successfully employed in this work as an easy-to-implement mechanism that can successfully separate short and long term cache residencies, allowing them to be served by different cache structures using the dual-cache paradigm. Using probabilistic sampling for cache filtering is a new idea, and two studies published in recent years are reviewed in Section 7.3.

But probabilistic filtering is only one possible method to partition the memory reference stream. A myriad of techniques to partition the memory reference stream have been proposed in the past, tuned to different types of locality — with most targeting dual-cache designs [65]. Different partitioning methods are reviewed in Section 7.4. However, it is important to note that all the reviewed methods rely on maintaining block reuse information, which in turn warrants designing a special hardware mechanism to store that information, thereby incurring both energy and space overheads. In contradistinction, the probabilistic partitioning proposed here is completely stateless, and does not depend on maintaining any reuse information.

Finally, the cache design presented above makes use of both direct-mapped and fully-associative structures. Direct-mapped caches are known for their low-latency and low-power characteristic, but also for their susceptibility to conflict misses. The opposite can be said about fully-associative caches, which suffer no conflicts (and hence lower miss-rates) at the cost of increased latency and power overheads. These combinations motivated many researchers to overcome the two structures' inherent deficiencies in order to get the best of both worlds. These designs are reviewed in Section 7.5.

7.1 Mass Count Disparity

The *mass-count disparity* phenomenon identified above in L1 cache workloads is surprisingly not uncommon, and is an artifact of heavy-tail distributions. Evidence of inverse correlations between the popularity of object sizes and their dominance in the aggregate size of the sample space have been reported in various fields of study. One of the first reports was made by Lorenz in 1905, when he described inequalities in the distribution of wealth [43], in which most people are poor, but the rich govern the majority of the wealth. This (now) well-known economical phenomenon was thus used to explain mass-count disparity in Chapter 3. More relevant to this research, inverse correlations were reported in several aspects of computer systems. Among others, Irlam showed most files on UNIX file systems to be small, with most disk space occupied by a small number of very large files [30]; Broido et al. reported several different inverse correlations in IP traffic, such as that only a fraction of all Internet service providers (ISPs) are involved in up to 90% of the routes observed, and that a fraction of all autonomous administrative systems may contribute up to 95% of the traffic on a link [8]; Harchol-Balter and Downey found UNIX processes to be mostly short, but that most of the CPU time goes to a small number of long-lasting processes [26].

Somewhat surprisingly, it was not until recent years that these supposedly disjoint occurrences of inverse correlations were characterized and generalized as a single statistical phenomenon. The two seminal papers describing the *Mass-Count Disparity* phenomenon were published by Crovella [16] and Feitelson [20]. Crovella identified that several supposedly disjoint examples of inverse correlations are in fact occurrences of a single statistical phenomenon, and coined the term *Mass-Count Disparity* to describe it. Feitelson developed this concept by introducing quantitative metrics to evaluate the disparity. These metrics, described in Chapter 3 include the $W_{1/2}$ metric that evaluates the aggregate mass of the smaller half of the samples; the $N_{1/2}$ metric evaluating the fraction of samples dominating half of the aggregate mass; and the *Joint-Ratio* metric as a generalization of the proverbial 80/20 rule that estimates the equilibrium between the count and mass distributions (both Irlam [30] and Broido et al. [8] in fact used a similar metric, referred to in the latter as the *crossover point*).

7.2 Strategies for Operating Systems' Buffer Caches

An operating system's buffer cache [68] shares similar goals with a hardware cache, e.g. to cache items that are likely to be used in the near future, in order to eliminate long-latency accesses to a secondary storage. Despite different storage facilities, buffer cache replacement policies can inspire hardware cache replacement policies, especially given their relaxed timing requirements. Therefore, reviewing some of the buffer cache replacement policies proposed is useful to gain some insights about caching in general.

Most notable of these is the *adaptive replacement cache* (ARC) by Megiddo and Modha [45]. ARC combines the frequency and recency of reuse metrics — used in the LFU and LRU eviction policies, respectively, and common in cache design — into a single mechanism. By managing two lists, one for each metric, this approach yields a dynamic partitioning of the buffer cache between LRU and LFU.

Lee et al. show the correlation between frequency and recency and the probability that a buffer is re-accessed, and build an analytical model for buffer replacement, using a weight function to weigh each buffer's reuse probability [42]. The observations on which this policy is based are that a buffer might have been used frequently in the recent past, but a change in the working set deems it unnecessary, thus making the LFU metric counter-productive. On the other hand, a buffer might have been accessed recently, but only once and is not to be accessed again, in contradiction with the rationale behind LRU. The proposed weight function thus utilize a combined recency and frequency metric that weighs each previous access to a buffer with its age. The authors show that this weight function can model both LFU and LRU, and a variety of algorithms in between.

Tomkins et al. advocate informed buffer prefetching by using programmer hints to know which pages to prefetch [77]. The authors try to combine both TIP2 [51] and LRU-SP [11] algorithms and overcome what seems to be the major drawbacks of each — the aggressive prefetching of LRU-SP that sometimes fetches pages that won't be needed, and the limited prefetching horizon of TIP2 which limits the number of prefetched pages to the number of ”hits” spent by a single ”miss”.

Jiang and Jhang tried to balance frequency and recency in the *Least Inter-Reference Set* (LIRS) buffer cache policy [34]. LIRS tries to improve the common LRU by accounting for access frequency. This is achieved by measuring how many times other buffers were accessed between each two consecutive accesses to a certain buffer — defined as the *inter-reference recency* (IRR) metric. The buffers that were accessed more frequently — i.e. ones with a low IRR count — are not considered for eviction.

As noted above, although these policies target a software based cache whose requirements are much more relaxed than a hardware based cache. The improved performance achieved by these policies demonstrates the promise of combining caching policies tuned to different types of locality, and thus motivate further exploration aimed towards lowering their overheads, making them feasible for use in processor caches.

7.3 Probabilistic Filtering

Skewed distributions, such as those described by mass-count disparity, naturally lend themselves to the design of probabilistic sampling algorithms. Namely, when the distribution is skewed, randomly selecting elements from the sample space will quickly identify the element groups dominating the distribution. Although the simplicity of such designs is appealing, only two other cases are known to the author in which sampling was used to enhance performance of hardware caches.

Behar et al. employed this same principle to reduce the power spent on trace generation by using periodic trace sampling [5]. This was based on the observation that the majority of execution time is spent executing a small number of traces (the proverbial 90/10 rule for instructions traces [27]), thereby generating only every Nth trace is sufficient to quickly find the most useful traces. The 90/10 effect described by the authors only demonstrates that the mass-count disparity is also common in trace generation.

Qureshi et al. observed that most L2 blocks are never reused, and suggested inserting most blocks into the LRU position (rather then to the MRU as commonly practiced), and infrequently select a random block insertion to be inserted into the MRU position [54].

Skewed distributions can therefore be used in different aspects of hardware caching to improve performance and save power. The application of probabilistic sampling to filter a workload that exhibits mass-count disparity — presented in this research — was in fact inspired by the study conducted by Behar et al. Nevertheless, the use of sampling is the only commonality between the two studies, as the current study focuses on a different workload and emphasizes the reasoning behind the effectiveness of sampling — with the combination of the two traits warranting a completely different design. Furthermore, as known examples for the use of sampling in processor caches have only been published in recent years, it is the author's hope that this principle will gain more popularity in the coming years. Anecdotally, probabilistic sampling has already been used in software-based web caching strategies: Starobinski and Tse use probability to decide whether to promote or demote a document in the cache [73].

7.4 Partitioning the Reference Stream

Dual-caches have been extensively studied in the past as a design concept that can be used to accommodate different types of locality. Such designs differ mainly in the types of locality examined (temporal vs. spacial, popular blocks vs. transient ones, etc.), and the corresponding criteria used to partition the reference stream [65]. This section therefore surveys several such designs.

González at al. suggested that the cache be partitioned into two parts, one each for handling data that exhibit spatial and temporal locality [23]. Based on previous reuse information, their predictor classifies memory accesses to either scalars references (temporal locality) or vectors references (spatial locality).

The work of Sahuquillo and Pont involves a filter used to optimize the hit ratio of the cache [64, 63]. The authors associate a reference counter with each cache line promoting the most popular blocks into a small L0 cache. A similar mechanism is proposed by Rivers and Davidson in their *non-temporal streaming* (NTS) cache, which also base caching decisions in a dual-cache structure on a reference count [59].

Kin et al. focused on reducing cache power consumption, and used an L0 structure to accommodate the most popular blocks while maintaining the L1 in low-power mode [40]. The power reduction is in fact traded off for performance as the L1 has to be re-powered on every access. In a followup work by Memik and Mangione-Smith, the filtering takes place between the L1 and L2 caches [46].

Karlsson and Hagersten use a filter to audit whether a block would have been replaced before its next access [39]. If the reuse distance is short enough, the block is promoted to the cache. This mechanism requires keeping a last-accessed-timestamp for every block in the cache, and comparing it on every replacement.

Johnson and Hwu used a bypass buffer for all blocks only allowing most frequently used blocks into the cache proper [36]. A *Memory Access Table* (MAT) is used to group contiguous memory blocks experiencing similar cache behavior. This is quite costly in hardware as it requires maintaining access frequency information for all macro-blocks. Jalminger and Stenström followed the same rationale, but tracked memory access patterns using a design inspired by a two level branch predictor [33].

Chang et al. effectively designed an dual-cache mechanism by partitioning the cache of the *System/370* CPU into an on-chip and off-chip parts, with emphasize on reducing the hit latency of the MRU block in a cache set to 1 cycle [13]. They do so by storing information in the TLB and accessing it in parallel with the cache lookup.

Rosner et al. employed the dual-cache paradigm in the design of trace caches, either to filter out infrequently used traces, or to avoid generating them in the first place [62]. In this paper, the authors explore several trace filtering techniques which rely on past block usage to predict whether it would be beneficial to promote a trace from a fully-associative filter into the trace cache proper.

Unlike the previous methods that characterize a memory block based on its address, some studies used the program counter (PC) of the accessing instruction as a means to characterize the memory access. Tyson et al. showed that a small fraction of memory access instructions generate the majority of misses [79]. They therefore proposed to avoid caching memory locations when accessed by these instructions. Rivers et al. extended this work to create the *program counter selective* (PCS) cache [60]. In this extension, the PC based prediction is used to decide into which part of a dual-cache structure a memory block should be inserted.

In a comparative study, Tam et al. compared several approaches to the reference stream partitioning (required for dual-cache designs), in order to assess their potential [74]. The techniques compared were NTS [59], PCS [60], MAT [36] and the *pseudo-opt* near-optimal replacement scheme for dual-caches [71]. The conclusion from this comparison is that effective address based approaches such as NTS outperform macroblock grouping (MAT) and PC-based (PCS) designs. Still, the authors showed that there is still much room for improvement as the near optimal algorithm outperform all others by a large margin.

Some partitioning approaches focus on partitioning the memory blocks themselves, based on the observation that not all words in a block necessarily exhibit the same types of locality. Fetching only partial blocks into the cache can therefore save both precious cache and memory bandwidth resources.

Pujara and Aggarwal noticed that despite the prevalent method of caching memory blocks as a whole to employ spacial locality, the common access pattern only uses a small number of the words in a block [53]. This effect dramatically reduces both L1 cache and bus utilization — defined as the ratio between the number of memory words used by the processor and the number of words brought into the cache. The authors have thus devised a mechanism that predicts which words in a memory block are likely to be used, and only fetches these into the L1 cache. This predictor tracks accesses to individual words in the cache, and based on the words accessed in recently evicted blocks predicts which words should be brought into the cache.

Park et al. used a spatial buffer to observe usage at different granularities [50]. Then when a word is referenced, only a small sub-line including this word is promoted to the temporal cache.

Qureshi et al. addressed a similar phenomenon in L2 caches, and have taken an opposite approach [57]. Rather than preventing the fetching of unused words, they introduce a mechanism called “Line Distillation” that identifies unused words in the cache and evicts them to save cache space. Their approach however is less appropriate for L1 caches at it still wastes precious bus bandwidth to fetch the full memory block.

The main problem when evaluating a dual-cache design is the lack of an optimal eviction policy with which a design can be compared. In an effort to develop an optimal replacement policy, Srinivasan and Davidson developed a near optimal policy, named *pseudo-opt* [71]. The policy is based on Belady’s optimal replacement for simple caches and tries to move the blocks that will be used farthest in the future from the main cache to the auxiliary buffer (exchanging blocks between the two structures if needed). The authors describe cases in which such a policy is sub-optimal, but were not able to describe an optimal policy. A few years later, Brehob et al. showed that an optimal replacement policy for dual caches where one of the components is fully-associative and the other is not is indeed NP-Hard [7].

As described above, all the structures reviewed require maintaining reuse information, thus complicating the filtering hardware. In contradistinction, the random sampling cache proposed in this study is purely probabilistic and is therefore stateless, thus not requiring any per-block information other than its mere presence in either the cache or the filter.

7.5 Efficient Use of Direct-Mapped and Fully-associative Caches

Direct-mapped caches are faster and consume less energy than set-associative caches typically used in L1 caches [28, 38]. However, they are more susceptible to conflict misses than set-associative caches, thus suffering higher miss-rates and achieving lower performance [32]. On the other hand, fully-associative caches offer lower miss-rates but do so with increased timing and power overheads caused by the fully-associative lookup, commonly performed using *content addressable memory* (CAM) [81]. These issues motivate researchers to augment the classic designs in order to overcome their deficiencies and enjoy the inherent advantages of these diverse structures.

In one of the first attempts to overcome direct-mapped caches' susceptibility to conflicts, Jouppi presented the the victim cache and stream buffers [37]. The design includes a small auxiliary cache used to store cache lines that were evicted from the main cache. This helps reduce the adverse effects of conflict misses, because the victim buffer is fully associative and therefore 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. By virtue of being applied after lines are evicted, this too avoids the need to maintain historical data. Walsh and Board extended Jouppi's victim cache [80]. They also proposed a dual design with a direct-mapped main cache and a small fully-associative filter, but in their design the referenced data is first placed in the filter, and only if it is referenced again it is promoted to the direct-mapped cache — thus avoiding polluting the cache with data that is only referenced once. Walsh and Board's design can be regarded as a degenerate version of the design presented in this thesis (Chapter 6): our design both allows for a block to be inserted into the cache after the first memory reference, as well as allowing blocks to be served from the filter for more than a single memory reference.

Column-associative caches presented by Agarwal and Pudar improve the hit-rate of a direct-mapped cache by using two different hash functions for mapping blocks into the cache [3]. Specifically, if a conflict exists in the original mapping, it uses an alternative mapping function and stores the block in the alternative location. Because of the double hashing, this design is also known as a hash-rehash cache. While eliminating most conflicts, this approach incurs a timing overhead as the cache has to be accessed in two phases when a conflict occurs. Topham et al. used a slightly different approach that uses probabilistic mapping rather than double hashing to achieve the same goal [78].

Theobald et al. generalized some of the aforementioned mechanisms that partition the cache in order to reduce the number of conflict misses into a general framework named *half-and-half caches* [76]. The authors specifically addressed *victim caches* [37], *column-associative / hash-rehash caches* [2, 3] and MRU caches [13].

Another mechanism employing the dual-cache paradigm to overcome conflict misses was the *assist cache* presented by Chan et al. [12]. The function of the assist cache is to compensate for the fact that the main cache is direct mapped, thus making it vulnerable to address conflicts. Unlike the victim cache, the fully-associative assist cache is placed between the direct-mapped cache and the bus such that blocks are inserted into the assist cache before they are moved into the cache itself, rather than the post-eviction approach used in victim caches. This mechanism was included in the commercial HP-PA 7220 CPU, and therefore serves as a guideline for what can be implemented in practice (fully associative buffer containing 64 lines of 32 bytes each).

A different approach to overcome conflict misses in direct-mapped caches was made by Mcfarling [44], in his minimalistic, bypass-only approach, dynamic exclusion cache. Here cache lines are augmented with just two state bits, the last-hit bit and the sticky bit. In particular, 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 recent study by Zhang presented the *B-Cache*, addressing cache conflicts by accessing the direct-mapped cache using two decoders, one of which is programmable [82]. The replacement algorithms utilizes the programmable decoder to alleviate excess replacements on any single cache set and distribute the load evenly among the different sets, thereby preventing most conflicts. The use of a programmable decoder eliminates the need for accessing the cache in two phases in the case of a conflict, as is required by the hash-rehash approach.

Except for the B-Cache, all studies trying to overcome conflict misses in direct-mapped cache were performed over a decade ago, and have never really become mainstream. But recent concerns about processors' power consumption, aggravated by the shift towards on-chip parallelism and the resulting replication of caches, brings renewed attention to power consumed by caches relative to that consumed by the entire processor. In this context, direct-mapped caches' low-power (and low-latency) characteristics makes them a very attractive solution. Given an effective partitioning of the reference stream, only the most popular blocks can be inserted into the direct-mapped cache. The mass-count disparity exhibited in L1 reference stream suggest such designs will only insert a fraction of all memory blocks into the direct-mapped cache, but will still enjoy the low-power and low-latency traits in most memory references.

A similar problem exists on the opposite side of the cache design spectrum, where fully-associative caches yield fewer misses than other designs (as they are not susceptible to conflicts at all), but do so with increased timing and power overheads. The research presented in this thesis addresses the issue using the wordline-lookaside-buffer (WLB) that harnesses temporal locality to avoid costly fully-associative lookups. Another interesting study addressed this issue by combining both hardware and software to speed up accesses to a fully-associative cache. Hallnor and Reinhardt proposed using the hardware cache's tag-store to hold the mappings only for cache resident block that are likely to be referenced in the near future, whereas the rest of the mappings are stored in a list oriented, software managed map [24]. When data is accessed, its location in the cache is first searched in the hardware tag-store, and if that fails the lookup continues in the software based tag-store.

The main advantages of the WLB over Hallnor and Reinhardt's scheme is that recent accesses do not require set-associative lookups at all (as the WLB is direct-mapped), thus consuming considerably less power. In addition, if the looked up block has not been accessed recently, the WLB scheme falls back to normal fully-associative lookup, whereas Hallnor and Reinhardt's scheme falls back to a much slower serial software lookup.

Chapter 8

Conclusions

Processor caches have been an area of active research for decades. Nevertheless, additional work is still important due to the continuing gap between processors and memory. In fact, the problem is expected to intensify with the advent of multicore processors, due to the replication of L1 caches for each core and the increased pressure on shared L2 caches.

One way to continue and improve is by taking cues from workload patterns. This research has shown that memory references display mass-count disparity, with a relatively small fraction of memory blocks receiving a relatively large fraction of the references. But this skewed distribution is at odds with the classic homogeneous definition of working sets, that puts all memory blocks in the working set on an equal footing. The author therefore proposes the core working set framework as an extension and refinement of Denning's working set. This formal framework uses logical predicates to distinguish between the more important subset of the data and the rest. Such a distinction, in turn, motivates dual cache structures that handle core and non-core data differently.

Harnessing the mass-count disparity phenomenon, this research describes the design of a reuse predictor that classifies cache residencies based on their expected length. The predictor uses independent random selection of references with a low success probability, thereby mostly selecting long residencies. 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 [55].

The reuse predictor is then used in the design of a random sampling cache, that uses probabilistic sampling of the *reference* distribution to split the *block* distribution into its two components — frequently used blocks that are to be served from a fast, low-power, direct-mapped cache, and transient blocks served by a fully-associative filter, thus preventing them from polluting the cache and causing conflict misses.

After examining the design space it was found that using a constant Bernoulli success probability P per specific cache configuration is very effective for most benchmarks, with no need for adaptive tuning. For example, when using a 16K direct-mapped cache and a 2K filter, the values $P = 0.05$ and $P = 0.0005$ are found to be best choices for the data and instruction streams, respectively.

To reduce the added overheads of using a fully-associative buffer, it is shown that most fully-associative CAM lookups can be avoided by using a direct mapped *wordline look-aside buffer* (WLB) that caches recent fully-associative lookups. A WLB with only 8 entries was sufficient to avoid $\sim$80% of the lookups for a 32 entry CAM.

The random sampling cache design was able to effectively utilize a 16K direct-mapped structure for both L1 caches yielding 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, this L1 design dramatically reduces the overall power consumption — both 16K and 32K caches were shown to perform better than 32K and 64K caches, respectively, while reducing the dynamic power consumption by $\sim$70%–80% and the leakage by over 40%. With the ubiquitous use of set-associative L1 caches in modern processors the author believes these results can contribute to future processor design and implementation as the proposed design offers a win-win situation — achieving better performance while consuming less power.

Bibliography

[1] Advanced Micro Devices, Inc. *AMD Phenom Processor Product Data Sheet*, Nov 2007.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Trans. on Computer Systems*, 6(4):393–431, Nov 1988.

[3] A. Agarwal and S. D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *Intl. Symp. on Computer Architecture*, pages 179–190, May 1993.

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

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

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

[7] 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, Jan 2004.

[8] A. Broido, Y. Hyun, R. Gao, and K. Claffy. Their share: Diversity and disparity in IP traffic. In *Intl. Workshop Passive & Active Network Measurement*, pages 113–125. Springer Verlag, Apr 2004. Lect. Notes Comput. Sci. vol. 3015.

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

[10] D. Burger, J. R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Intl. Symp. on Computer Architecture*, pages 78–89, May 1996.

[11] P. Cao, E. W. Felten, and K. Li. Application controlled file caching policies. In *Usenix Annual Technical Conf. (Summer)*, pages 171–182, Jun 1994.

[12] 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.

[13] J. H. Chang, H. Chao, and K. So. Cache design of a sub-micron CMOS System/370. In *Intl. Symp. on Computer Architecture*, pages 208–213, Jun 1987.

[14] L. Chisvin and R. J. Duckworth. Content-addressable and associative memory: Alternatives to the ubiquitous RAM. *IEEE Computer*, 22(7):51–64, Jul 1989.

[15] D. Citron. MisSPECulation: partial and misleading use of SPEC CPU2000 in computer architecture conferences. In *Intl. Symp. on Computer Architecture*, pages 52–61, Jun 2003.

[16] M. E. Crovella. Performance evaluation with heavy tailed distributions. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–10. Springer-Verlag, Jun 2001. Lect. Notes Comput. Sci. vol. 2221.

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

[18] P. J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, Jul 2005.

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

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

[21] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Intl. Symp. on Computer Architecture*, pages 148–157, May 2002.

[22] W. W. Gibbs. A split at the core. *Scientific American*, 291(5):96–101, Nov 2004.

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

[24] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *Intl. Symp. on Computer Architecture*, pages 107–116, Jun 2000.

[25] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, Sep 1997.

[26] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Trans. on Computer Systems*, 15(3):253–285, Aug 1997.

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

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

[29] Intel Corp. *Intel 64 and IA-32 Architecture Software Developr's Manual. Vol. 1: Basic Architecture*, Nov 2007.

[30] G. Irlam. Unix file size survey. URL http://www.gordoni.com/ufs93.html, 1993.

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

[32] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[33] J. Jalminger and P. Stenström. A novel approach to cache block reuse predictions. *Intl. Conf. on Parallel Processing*, 294–303, Oct 2003.

[34] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Intl. Conf. on Measurement & Modeling of Computer Systems*, pages 31–42, Jun 2002.

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

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

[37] 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*, pages 364–373, May 1990.

[38] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Intl. Symp. on Computer Architecture*, pages 34–45, Apr 1994.

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

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

[41] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. A multi-core approach to addressing the energy-complexity problem in microprocessors. In *Workshop on Complexity-Effective Design (WCED)*, Jun 2003.

[42] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Intl. Conf. on Measurement & Modeling of Computer Systems*, pages 134–143, May 1999.

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

[44] S. McFarling. Cache replacement with dynamic exclusion. In *Intl. Symp. on Computer Architecture*, pages 191–200, May 1992.

[45] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *USENIX Conf. on File and Storage Technologies*, Mar 2003.

[46] G. Memik and W. H. Mangione-Smith. Increasing power efficiency of multi-core network processors through data filtering. In *Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 108–116, Oct 2002.

[47] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[48] G. E. Moore. Progress in digital electronics. In *Technical Digest of the Intl. Electron Devices Meeting*, page 13. 1975.

[49] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Intl. Symp. on Computer Architecture*, pages 67–77, May 1996.

[50] 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*, pages 162–177. Jun 2000.

[51] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *ACM Symp. on Operating Systems Principles*, pages 79–95, Dec 1995.

[52] F. J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies. *Intl. Symp. on Microarchitecture*, page 2, Nov 1999.

[53] P. Pujara and A. Aggarwal. Cache noise prediction. *IEEE Trans. on Computers*, 57(10):1372–1386, Oct 2008.

[54] 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*, pages 381–391, Jun 2007.

[55] 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*, pages 167–178, Jun 2006.

[56] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *Intl. Symp. on Microarchitecture*, 423–432, Dec 2006.

[57] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. *Symp. on High-Performance Computer Architecture*, 250–259, Feb 2007.

[58] N. Riley and C. Zilles. Probabilistic counter updates for predictor hysteresis and bias. *IEEE Computer Architecture Letters*, 5(1):17–20, Jan–Jun 2006.

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

[60] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *ACM Intl. Conf. on Supercomputing*, pages 449–456, Jul 1998.

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

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

[63] J. Sahuquillo, S. Petit, A. Pont, and V. Milutinović. Exploring the performance of split data cache schemes on superscalar processors and symmetric multiprocessors. *Journal of Systems Architecture*, 51(8):451–469, Aug 2005.

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

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

[66] R. R. Schaller. Moore's Law: Past, present, and future. *IEEE Spectrum*, 34(6):52–59, Jun 1997.

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

[68] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Addison Wesley, Reading, MA, USA, 7th edition, Dec 2004.

[69] R. L. Sites. Alpha AXP architecture. *Commun. ACM*, 36(2):33–44, Feb 1993.

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

[71] V. Srinivasan and E. Davidson. Improving performance of an L1 cache with an associated buffer. Technical Report CSE-TR-361-98, Univ. of Michigan-Ann Arbor, Mar 1998.

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

[73] D. Starobinski and D. Tse. Probabilistic methods for web caching. *Performance Evaluation*, 46(2–3):125–137, Oct 2001.

[74] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, and E. S. Davidson. Active management of data caches by exploiting reuse information. *IEEE Trans. on Computers*, 48(11):1244–1259, Nov 1999.

[75] 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/.

[76] K. B. Theobald, H. H. Hum, and G. R. Gao. A design framework for hybrid-access caches. In *Symp. on High-Performance Computer Architecture*, pages 144–153, Jan 1995.

[77] A. Tomkins, R. H. Patterson, and G. Gibson. Informed multi-process prefetching and caching. In *Intl. Conf. on Measurement & Modeling of Computer Systems*, pages 100–114, Jun 1997.

[78] N. Topham, A. González, and J. González. The design and performance of a conflict-avoiding cache. In *Intl. Symp. on Microarchitecture,* pages 71–80, Dec 1997.

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

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

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

[82] C. Zhang. Balanced cache: Reducing conflict misses of direct-mapped caches. In *Intl. Symp. on Computer Architecture*, pages 155–166, Jun 2006.