On Identifying the Core of Working Sets

Yoav Etsion
Computer Sciences Department
Barcelona Supercomputing Center
08034 Barcelona, Spain

Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

Locality is often expressed using working sets, defined by Denning to be the set of distinct addresses referenced within a certain window of time. This definition puts all memory blocks in a working set on an equal footing. But in fact a dramatic difference exists between the usage patterns of frequently used data and those of lightly used data. We therefore propose to extend Denning's definition with that of *core* working sets, which identify the most important subset of blocks in a working set — those that are most frequently and for the longest time. We survey the motivation and ramifications of this concept. In particular, we use it as an underlying unifying principle for many dual cache structures that attempt to identify and provide special treatment for highly used data elements based on their access patterns.

1 Introduction

The notion of a memory hierarchy is one of the oldest and most profitable 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 [17]. While processor caches alleviate the speed gap between the CPU and memory, this gap nevertheless continues to grow. At the same time increasing on-chip parallelism threatens to stress caches more than ever before. These developments motivate attempts for better utilization of cache resources, through the design of more efficient caching structures. This design process relies on extensive analysis of memory workloads, and the development of new analysis tools enabling a deeper understanding of cache behavior.

The essence of caching is to identify and store those data items that will be most useful in the immediate future [1]. Caches predict future use of data based 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 [4]. 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.

Locality is usually regarded as a combination of two distinct properties — locality in time and locality in space — but is also a manifestation of the skewed distribution of the *popularity* of different memory blocks, where some blocks are accessed more frequently than others [8]. In fact, as we show 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 contrasts with 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 insight 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.

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 behaviors[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.

We start our discussion by showing that many benchmarks indeed have a relatively well-defined core (Section 2), and use this to define the core of a working set (Section 3). We then use the concept of a core to motivate dual cache structures based on cache bypass (Section 4), and show that various previous proposals for dual cache structures can be interpreted as attempts to implement improved support for caching the core working set (Section 5). Section 6 presents our conclusions.

2 The Skewed Popularity of Memory Locations

Locality of reference is one of the best-known phenomena of computer workloads. This is usually divided into two types: spatial locality, in which we see accesses to addresses that are *near* an address that was just referenced, and temporal locality, in which we see *repeated references* to the same address. Temporal locality is actually the result of two distinct phenomena. One is the skewed popularity of different addresses, where some are referenced a lot of times, while others are only referenced a few times [8]. The other is correlation in time: accesses to the same address are bunched together in a burst of activity, rather than being distributed uniformly throughout the execution. While the intuition of what "temporal locality" means tends to the second of these, the first is actually the more important effect.

While the skewed popularity of memory blocks is well-known, it has seldom been quantified. To do so we first need some definition. In the following we consistently define memory objects to be 64 bytes long, because this is the most common size for a cache line. Popularity is measured by the number of references to such a memory object in each *cache residency*, i.e. from the time

[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.

it is inserted into the cache until it is evicted. Thus if an object is referenced 100 times while in the cache, is evicted, and then is inserted again and referenced another 200 times, this is counted as two residencies with popularities of 100 and 200 references respectively. This characterization obviously depends on the cache design; the results shown here are for a 16 KB direct mapped cache.

Histograms of the distribution of residency lengths for select SPEC2000 benchmarks are shown in Fig. 1. These show the distribution of residency lengths and the distribution of references to these residencies, up to residency lengths of 250 references, using buckets of size 10. All residencies (and references therein) that are longer are bunched together in the last bar on the right. This leads to characteristic bimodal distributions, as examplified by the wupwise and mesa benchmarks. In them, residencies are seen to be short (typically up to 10 or 20 references, and seldom more than 50), but most references belong to residencies that are longer than 250 references. However, in some cases the patterns are a bit different. Benchmark swim is an example of cases were the residencies in the DL1 cache are all short, so the vast majority of references are also directed at short residencies. Benchmark bzip2 is an example of an even rarer phenomenon, where practically all residencies in the IL1 cache are long.

A more precise quantification is possible using mass-count disparity plots, as demonstrated in Fig. 2 [7]. These plots superimpose the CDFs of the same two distributions used in the histograms above. The first, which we call the *count* distribution, is the distribution of cache residencies, and specifies how many references each residency received. Thus $F_c(x)$ will represent the probability that a cache residency is referenced x times or less. The second, called the *mass* distribution, is the distribution of references; it specifies the popularity of the residency to which the reference pertains. Thus $F_m(x)$ will represent the probability that a reference is *part of* a residency that receives x references or less.

Mass-count disparity refers to situations where the two distributions diverge from each other. The figure shows examples for the wupwise and mesa benchmarks from the SPEC 2000 suite. The simplest metric for quantifying the disparity is the *joint ratio*, which is the unique point in the graphs where the sum of the two CDFs is unity (if the CDFs have a discrete mode, as sometimes happens, the sum may be different). For example, in the case of the mesa benchmark data stream, the joint ratio is 10/90. This means that 90% of the memory *references* are directed at only 10% of the *cache residencies*, whereas the remaining 90% of the residencies get only 10% of the references — a precise example of the proverbial 10/90 principle. Thus a typical residency is only referenced a rather small number of times (up to 10 or 20 in this case), whereas a typical reference is directed at a long residency (one that is referenced thousands of times).

Two other metrics that are especially important in the context of dual cache designs are $W_{1/2}$ and $N_{1/2}$. The $W_{1/2}$ metric assesses the combined weight of the half of the residencies that receive the fewest references. For mesa, these 50% of the residencies together get only 2.05% of the references. The $N_{1/2}$ metric characterizes the other end of the distribution: it gives the fraction of heavy-weight residencies needed to account for half of the total references. For mesa, just 0.2% of the residencies are enough.

Similar quantifications are possible for other SPEC2000 benchmarks [6]. For some benchmarks, the graphs are not well-formed, being dominated by a large discrete step. In those that are

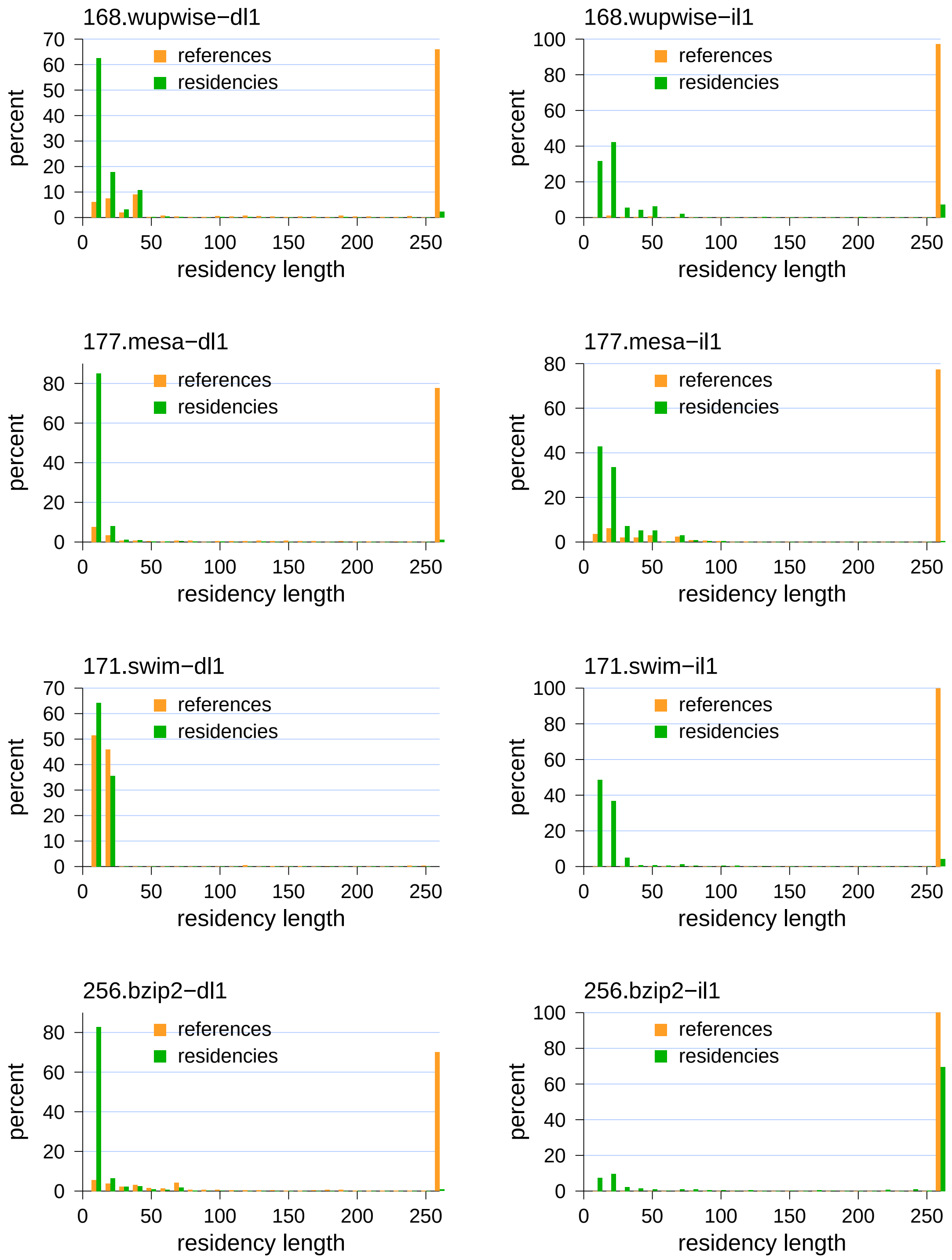
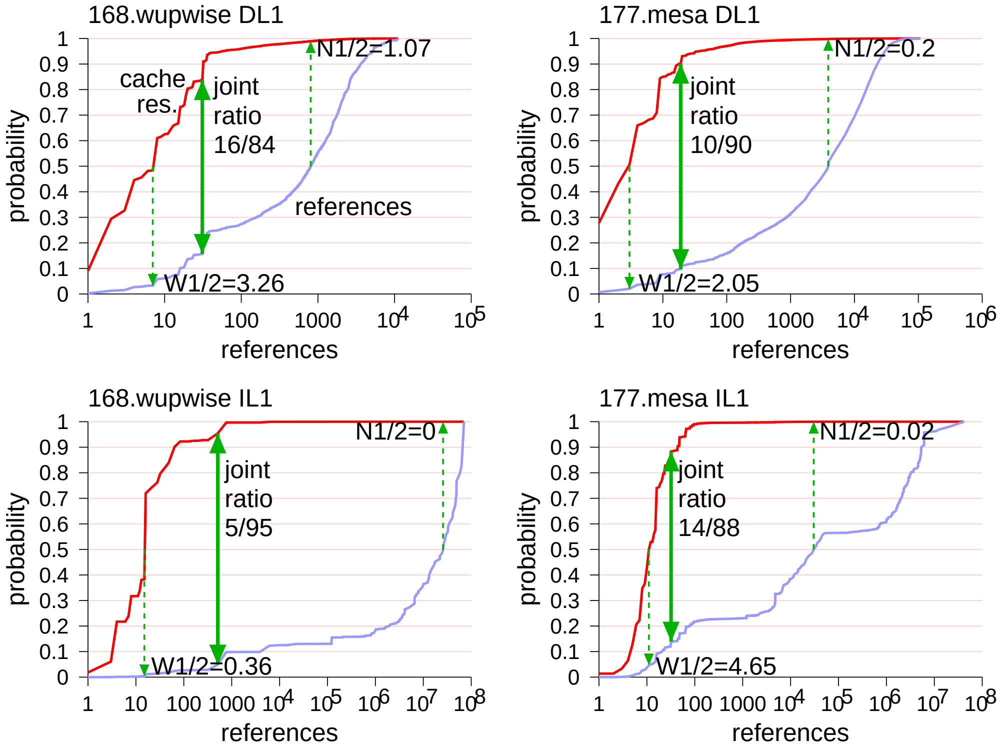


Figure 1: *Histograms of residency lengths for select SPEC benchmarks, using the ref input.*


168.wupwise DL1

177.mesa DL1

168.wupwise IL1

177.mesa IL1

Figure 2: *Mass-count disparity plots for memory accesses in select SPEC benchmarks, using the ref input.*

well-formed, the actual values observed for the joint ratio were in the range 10/90 to 33/67 for the data stream, and 1/99 to 24/76 for the instruction stream. A few cases are dominated by uniform access (that is, a very large fraction of the blocks are all accessed in the same way) and then there was naturally little if any mass-count disparity.

3 Definition of Core Working Sets

Denning's definition of working sets [3] is based on the principle of locality, which he defined to include three components [4]: a nonuniform popularity of different addresses, a slow change in the reference frequency to any given page, and a correlation between the immediate past and 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 [3]. We augment this definition by defining the *core working set* to be those blocks that appear in the working set and are reused "a significant number of times". The simplest interpretation of this definition is based on counting the number of references to a block during a single cache residency. The number of references needed to qualify can be decided upon based on data such as that presented in Figs. 1 and 2. For example, we can set the threshold so that for most benchmarks it will identify no more than 5% of the residencies, but more than 50% of the references. Given the typically skewed distribution of residency lengths, such a threshold should be in the range between 100 and 1000 references.

In fact, the highly-skewed distributions imply a partitioning of the residencies into two distinct groups: very many residencies that together receive only a small fraction of the references, and a small group of residencies that together account for the vast majority of references — the ones we call the core working set. Many dual cache structures attempt to capture this division. The motivation is straightforward. The lightly used residencies do not benefit very much from the caching, and should not be allowed to pollute the cache. Rather, the caches should be used preferentially to store heavily used data items, such as the small number of blocks that together account for half of all references. The dual structure helps in identifying and handling the two types correctly.

While the skewed distribution of popularity is a major contributor to temporal locality, one should nevertheless acknowledge the fact that references do display a bursty behavior. To study this, we looked at how many different blocks are referenced between successive references to a given block. The results indicate that the majority of inter-reference distances are indeed short. We can then define bursts to be sequences of references to a block that are separated by references to less than say 256 other blocks. Using this we can study the distribution of burst lengths, and find them to be generally short, ranging up to about 32 references for most benchmarks. However, they are long enough to prohibit the use of a low threshold to identify blocks that belongs to the core working set with confidence. The core members, in turn, exhibit extremely long bursts; these are actually blocks that are used continuously, and therefore do not have long gaps between successive accesses, so all their accesses will seem to be one long burst.

The effect of the above definitions is illustrated in Fig. 3. 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 for a window of 1000 instructions, and the core working set as defined by a threshold of 16 references to a block (denoted 16B in the legend). 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 Cache Bypass

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.

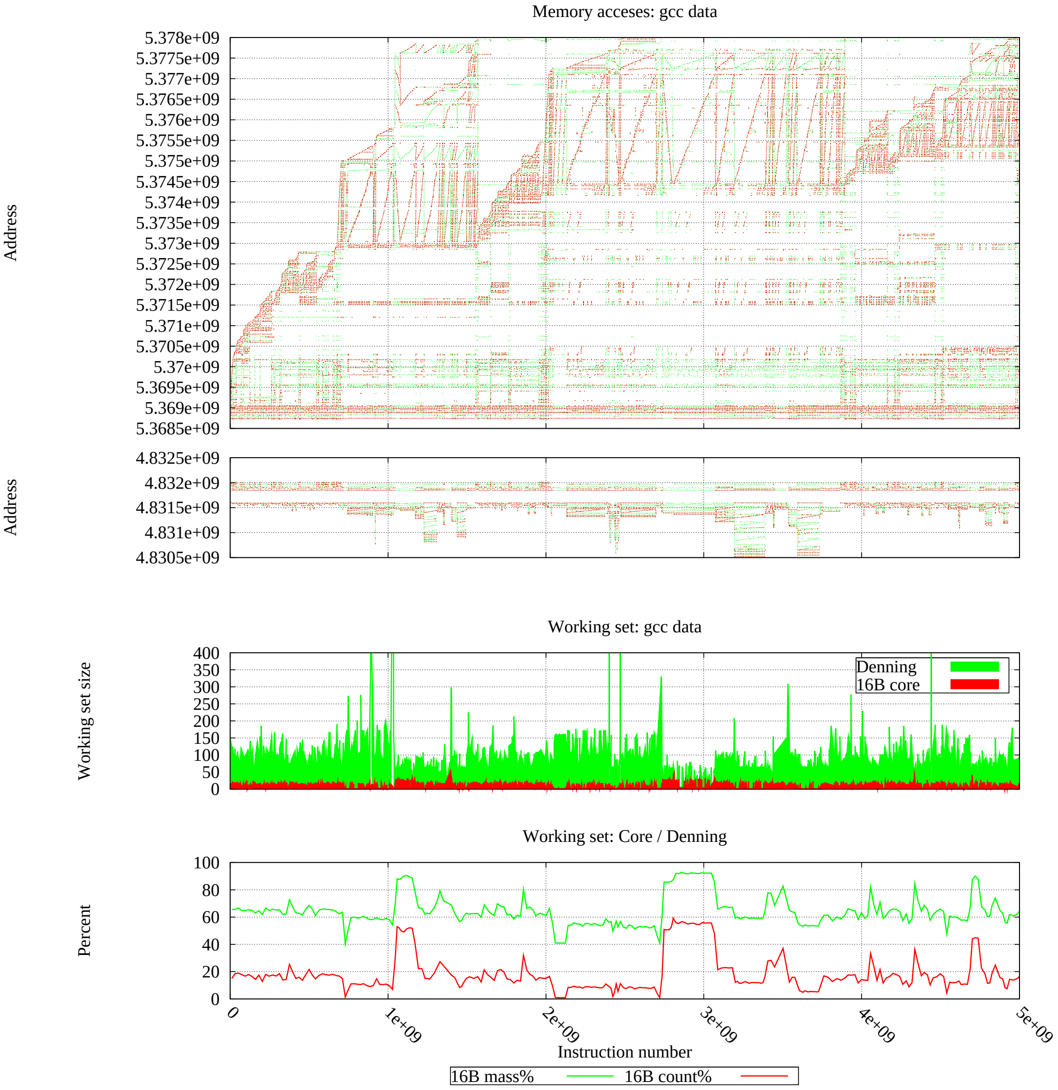


Figure 3: *Examples of memory access patterns and the resulting Denning and core working sets.*

The principle behind optimal cache replacement is very simple: when space is needed, replace the item that will not be used for the most time in the future (or never) [1]. 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 this 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.

By analyzing the reference streams of SPEC 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 debatable.

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

The advantage of a cache bypass mechanism can be formalized as follows, using a simple, specific example cache configuration. Assume a cache with $n^2 + n$ cache lines, organized into either n or $n+1$ equal sets. 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 $n + 1$st set can accept any address and serves as a bypass.

These two designs expose a tradeoff: in the set associative design, each set is larger by one, reducing the danger of conflict misses. In the bypass design, the extra set is not tied to any specific address, increasing flexibility.

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 bypass 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

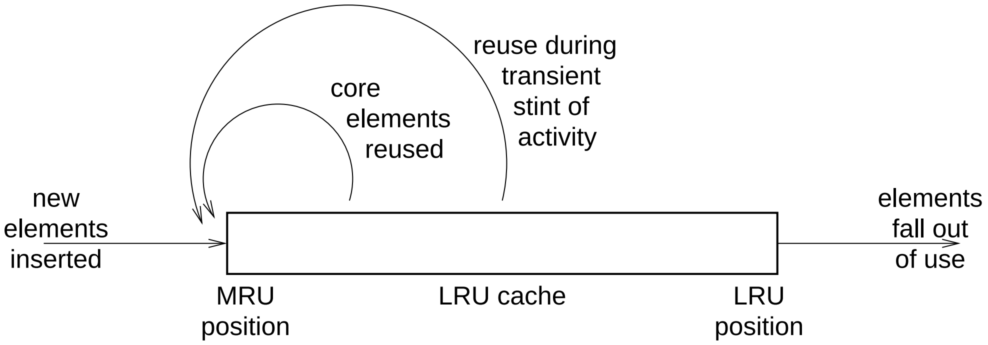


Figure 4: *Operation of an LRU cache is implicitly based on the notion that core elements will be reused before they drop out of the cache.*

a total of $O(nm)$ misses. When using the bypass design, on the other hand, all $2n$ addresses will be cached by using the original set and the bypass set. Therefore only the initial $2n$ compulsory misses will occur. 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. ∎

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

5 Dual Cache Structures

The design of caches is implicitly based on the skewed distribution of references. The well-known LRU scheme inserts raferenced blocks at the top of a logical stack, and evicts the block from the bottom of the stack to make room when needed. Core blocks are not evicted because they are referenced again and again, buming them back to the top of the stack each time (Fig. 4). If the cache is big enough, transient blocks that enjoy a burst of activity can also be retained till this activity ends.

LRU over the whole cache is not practical in processor caches. Rather, set-associative designs are used, with typical set sizes of 4 or 8. Here either LRU can be used on each set, or the eviction can be random. Random eviction works because if you select a cache residency at random, it is most probably a short residency. Long residencies are rarer, so have a smaller probability of being evicted.

Nevertheless, core residencies may indeed be evicted by mistake. The desire to reduce such mistakes is one of the motivations for using dual cache structures. Again, this is based on the skewed distribution of residency lengths. 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. This is a generalization of the cache bypass considered above.

Many similar schemes have been proposed in the literature [16]. Many of them are based on an attempt to identify and provide support for blocks that display temporal locality — in effect, the more popular blocks that are reused time and again. For example, Rivers and Davidson propose to tag cache lines with a temporal locality bit [14]. Initially, lines are stored in a small non-temporal buffer (in our terminology, this is the bypass area). If they are reused, the temporal bit is set indicating that, in our terminology, these lines should be considered as core elements. Later, when a line with the temporal bit set is fetched from memory, it is inserted into the larger temporal cache.

Park et al. also use a spatial buffer to observe usage [13]. However, they do so at different granularities: when a word is referenced, only a small sub-line including this word is promoted to the temporal cache. A more extreme approach is the bypass mechanism of Johnson et al. [9]. This is based on a memory address table (MAT) which counts accesses to different areas of memory. Then, if a low-count access threatens to displace a cached high-count datum, it is simply loaded directly to the register file and bypasses the cache altogether. Another scheme is the Assist cache used in the HP PA 7200 CPU [2], which filters out streaming (spatial locality) data based on compiler hints.

A minimalistic, bypass-only approach, is McFarling's dynamic exclusion cache [12]. 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.

The above schemes have the drawback of requiring historical information to be maintained for each cache lines. But filtering can also be done without resorting to the use of such data. For example, Walsh and Board propose a dual design with a direct-mapped main cache and a small fully associative filter [18]. Referenced data is first placed in the filter, and only if it is referenced again it is promoted to the main cache. This avoids polluting the cache with data that is only referenced once; however it is limited to distinguishing only between blocks that are used once or more, and our data indicates that a much higher threshold may be needed. Filtering with higher effective thresholds may be achieved by using randomized sampling, based on the skewed popularity distributions described above [5]. The idea is that every reference to data in the filter is sampled with a low probability, and only memory blocks that come up in the sampling are promoted to the main cache. Due to the mass-count disparity phenomenon, this effectively identifies those memory blocks that are accessed a very large number of times — but without requiring historical data to be maintained.

A somewhat different approach is provided by Jouppi's victim cache, which works on evicted blocks rather than on newly accessed blocks. Specifically, The victim cache is a small auxiliary cache used to store cache lines that were evicted from the main cache [10]. 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. Blocks that are referenced while in the victim cache are simply returned to the main cache.

Interestingly, dual structures are not used only to improve performance. Sahuquillo et al. [15] proposed a filter cache, in which a relatively small buffer is used for the most highly accessed elements, in order to reduce bus traffic in multiprocessor systems. A similar design by Kin at al. [11] was proposed in order to reduce energy consumption, by allowing the main cache to remain in power save mode most of the time. Some of the dual structures described above also reduce power consumption, by virtue of using a direct-mapped design for part of the cache [13, 5]. Thus they can lead to a win-win situation, where both performance and power characteristics are improved, instead of having to trade them off against each other.

6 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. We have 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. We therefore propose the core working set framework as an extension and refinement of Denning's working set. This framework makes a distinction between the more important (that is, more heavily used) subset of the data and the rest. Such a distinction, in turn, motivates dual cache structures that handle core and non-core data differently. By matching the handling to the access pattern, one can even achieve a win-win situation, which provides both performance improvements and power reduction.

References

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

[2] 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.

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

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

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

[6] Y. Etsion and D. G. Feitelson. Probabilistic prediction of temporal locality. *IEEE Comput. Arch. Let.*, 6(1):17–20, Jan-Jun 2007.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] S. McFarling. Cache replacement with dynamic exclusion. In *Intl. Symp. on Computer Architecture*, pages 191–200, New York, NY, USA, May 1992. ACM Press.

[13] 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. Springer Verlag, Jun 2000.

[14] 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.

[15] 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.

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

[17] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, Sep 1982.

[18] S. J. Walsh and J. A. Board. Pollution control caching. In *Intl. Conf. on Computer Design*, pages 300–306, Washington, DC, USA, Oct 1995. IEEE Computer Society.