Cache Insertion Policies to Reduce Bus Traffic and Cache Conflicts

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

Abstract

The distribution of the number of references to different memory words is highly skewed, with most addresses being referenced only a very small number of times, while others are referenced often. This motivates the definition of *cache insertion policies*, which will identify cache lines that are likely to be reused, by examining a small window of the most recent memory accesses. Such a framework can filter out those accesses that only pollute the cache. We show that a 64-entry fully associative filter can be very effective for both DL1 and IL1 caches, and leads to substantial reductions in the miss rate, the average memory access time, and the traffic between the L1 and L2 caches. Specifically, when using the SPEC 2000 benchmarks, we reduce the DL1 misses by a maximum of 57%, and 9% when averaging on all the benchmarks. Furthermore, we reduce the traffic between the L1 and L2 cache by a maximum of 30% and 7% on average. This is especially important in emerging multi-core processors in which the cores share the L2 cache.

1 Introduction

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. This structure appears in countless contexts, from computer architecture, to operating systems, to the world wide web. Perhaps the most basic example of caching occurs between the CPU and the main memory [17], where it alleviates the increasing gap between CPU and memory speeds.

The basic framework for caching that has materialized over the years includes several components. In general caching is done by demand, with the possible use of prefetching in anticipation of future requests. The main parameters of the basic design are the block sizes and the associativity of the cache, and possibly some auxiliary structures such as a victim cache [9]. The main algorithmic issue is the eviction policy. The design then rests on an evaluation of the tradeoffs involved in using various combinations of design options, in terms of the achieved performance and the cost as reflected by space and energy consumption.

The success of caching is based on the principle of locality [4]. However, there are different types of locality, such as the well-known distinction between spatial and temporal locality. In particular, recent work has focused on streaming data and its possible interaction with the memory hierarchy (e.g. [11]). We extend this analysis to the level of the processor cache, and propose a distinction between the conventional working set of a computation and its *core* working set, defined to be those memory items that are really used many times (Section 3). Based on this distinction, we suggest that only the core data be cached, whereas the more transient data be filtered out and prevented from entering the cache and polluting it. Our main contribution is the formalization of the idea of an *insertion policy*, and identifying it as a new major component in the design space of caching systems.

An insertion policy based on filtering has two main components. First is the filtering algorithm, i.e. how to decide on-line whether a data item should be cached or not. Obviously the data needs to be stored in some interim location until a decision can be reached. This leads to the partitioning of the cache into two: the filter and the cache proper. The second part of the design is the parameters of the filter, e.g. its block size and associativity, which need not be the same as for the cache itself. These issues are elaborated in Section 4, and the different options are evaluated in Section 5.

The idea of a filter cache is not completely new. Previous related work is surveyed in Section 2. However, these efforts were mainly ad-hoc in nature, and sometimes led to overly complicated designs. Our contribution is to identify the insertion policy as a separate topic worthy of detailed study, based on an analysis of common workloads, and leading to efficient designs and better results. However, this is obviously not the final word on the matter. Issues that are left for future work are listed in Section 7, together with our conclusions.

2 Related Work

The observation that memory access patterns may display different types of locality, and that these may warrant different types of caching policies, has been made before. Tyson et al. show that a small fraction of memory access instructions have a disproportionally large effect on the miss rate, as they generate the majority of misses [20]. They therefore suggest that these instructions be identified and marked so that they will not be cached at all. Their conclusions are that this can significantly reduce the memory bus traffic.

González at al. suggest that the cache be partitioned into two parts, one each for handling data that exhibit spatial and temporal locality [8]. This is managed by a locality prediction table, which stores data about the history of the most recently executed load/store instructions. The predictions attempt to classify accesses as scalar or as belonging to a vector, in which case they also attempt to identify the stride and vector size. Scalars are cached in the temporal sub-cache. Vectors with small stride are cached in the spatial sub-cache, which has a larger line size. In some cases, a datum may be cached in both sub-caches, wasting precious cache resources. Our filter design is much simpler, as it just looks at repeated accesses to the same block, and does not attempt to adjust dynamically.

The work of Sahuquillo and Pont is even closer to ours in their design, which involves a filter used to optimize the hit ratio of the cache [14]. However, their main motivation is to reduce the bus utilization in multiprocessor systems. Their design therefore focuses on identifying the most heavily used items, and the filter is used to store those items that have been accessed the most times. This comes at a price of having to maintain an access counter for each item. A similar mechanism is proposed by Rivers and Davidson, who also base the caching on an access count [13]. In our scheme this is implicit in the promotion algorithm that moves data items from the filter to the main cache.

Kin et al. also use filtering before the L1 cache [10]. However, their motivation is not to improve performance but rather to reduce energy consumption. The idea is that the filter should catch most of the memory references, allowing the L1 cache to remain in a low-power standby mode. However, this power saving comes at the cost of a certain performance degradation, because the L1 cache is only accessed after the filter fails, rather than in parallel to accessing the filter. In a followup work by Memik and Mangione-Smith, the filter is placed in front of the L2 cache [12].

In all the above studies, the filter is a small auxiliary structure, typically fully associative, designed to assist the larger and more expensive cache. As such they are similar to the victim cache and stream buffers suggested earlier by Jouppi [9]. A similar structure has even been included in a commercial microprocessor: the assist cache of the HP PA 7220 CPU [2]. The function of this assist cache is to compensate for the fact that the main cache is direct mapped, thus making it vulnerable to address conflicts. Its size (64 lines of 32 bytes, fully associative) serves as a guideline for what can be implemented in practice.

3 Filtering the Reference Stream

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 often assumed to imply a correlation in time: that accesses to the same address are bunched together in a certain part of the execution, rather than being distributed uniformly throughout the execution. But another important phenomenon is the skewed popularity of different addresses: some are referenced a lot of times, while others are only referenced once or a few times.

3.1 Motivation

The skewed popularity can be demonstrated using mass-count disparity plots. These plots superimpose two distributions. The first, which we call the *count* distribution, is a distribution on addresses, and specifies how many times each address is referenced. Thus $F_c(x)$ will represent the probability that an address is referenced x times or less. The second, called the *mass* distribution, is a distribution on references; it specifies the popularity of the address to which the reference pertains. Thus $F_m(x)$ will represent the probability that a reference is directed at an address that is referenced x times or less.

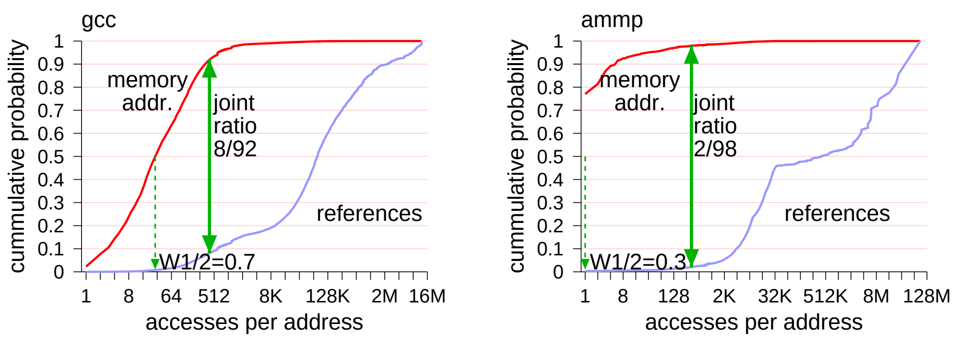


Figure 1: *Mass-count disparity plot for memory accesses in the gcc and ammp SPEC benchmarks.*

Mass-count disparity refers to the fact that the two graphs are quite distinct. An example is shown in Fig. 1, using the gcc and ammp SPEC 2000 benchmarks with the train input. The joint ratio is the unique point in the graphs where the sum of the two CDFs is 1. In the case of the gcc graph the joint ratio is 8/92. This means that 8% of the memory addresses, and more specifically those addresses that are highly referenced, get a full 92% of the references, whereas the remaining 92% of the addresses get only 8% of the references. Thus a typical address is only referenced a rather small number of times (up to about 500), whereas a typical reference is directed at a highly-accessed address (one that is accessed from 1000 to 10,000,000 times). More importantly for our work is the $W_{1/2}$ metric, which assesses the combined weight of the half of the addresses that receive less references. For gcc, these 50% of the addresses together get only 0.76% of the references. Thus these addresses are hardly used, and should not be allowed to pollute the cache. Rather, the caches should be used preferentially to store heavily used data items.

The results are even more extreme for ammp: the joint ratio is 2/98, meaning that only 2% of the addresses are the targets of a full 98% of the references, and vice versa. Moreover, a whopping 77% of the addresses are only referenced once, and together they account for only 0.3% of the total references. Looking at other SPEC benchmarks, we found joint ratios in the range from 1/99 to 8/92, and $W_{1/2}$ metrics from 0.03% to 3.42%. An especially extreme example is the apsi benchmark, where 99.5% of the addresses are accessed only once[1]!

3.2 Core Working Sets

In terms of classical locality theory, these findings imply that the notion of a working set is deficient in the sense that it does not distinguish between the heavily used items and the lightly used ones. The definition of a working set by Denning is the set of all distinct addresses that were accessed within a window of T instructions [5]. We will denote this set as $D_T(t)$, to mean "the Denning working set at time t using a window size of T". As an alternative, we define the *core working set* to be those addresses that appear in the working set and are reused a significant number of times.

[1]Of course, being accessed only once does not make sense in terms of storing and using data. It could be that these addresses are actually accessed again, beyond the 2 billion instruction window we used. However, this does not change the claim that they should not be cached.

This will be denoted $C_{T,P}(t)$, where the extra parameter P reflects a predicate used to identify core members.

The predicate P is meant to capture reuse of memory addresses. This can be expressed in a number of ways. Recall that caches are typically organized in blocks (or cache lines) that are bigger than a single word, in order to exploit spatial locality. We therefore need to decide whether to accept any type of repetitions within a block, or require repetitions in accessing the same word. To express this, we introduce the following notation. Let B represent a block of k words. Let w_i, $i = 1, \dots, k$ 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 define several types of predicates:

$n \times B$ — words in block B are referenced n times or more. Formally this is written

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

For example, the predicate $3 \times B$ identifies those blocks that were referenced a total of 3 times or more. This is meant to identify a combination of spatial and/or temporal locality, without requiring either type explicitly. n will typically be smaller than the block size k, so as to be able to capture strided accesses.

The space overhead for implementing this predicate is $\log n$ bits per cache block, in order to count the accesses. Note that this is only added to the filter, not to the cache proper.

$n \times W$ — some word w in block B was referenced n times or more. Formally this is written

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

For example, the predicate $2 \times W$ identifies those blocks that include some word that was referenced 2 times or more. This predicate is designed to identify blocks that display temporal locality.

The space overhead for implementing this predicate is $k \log n$ bits, because we need to count the accesses to each word individually.

$n \times AW$ — all words in block B were referenced, and at least one of them was referenced n times or more. Formally this is written

$$n \times AW \equiv (\forall w \in B : r(w) \geq 1) \wedge (\exists w \in B \quad \text{s.t.} \quad r(w) \geq n)$$

This predicate is designed to identify memory scans (each word accessed in sequence) that should nevertheless be cached because there is some reuse in addition to the scan. However, experience indicates that this is very rare, and even many scans do not access all the words in each block but rather use some stride. This type of predicate is therefore not used in the sequel.

$n \times ST$ — a non-uniform strided reference pattern with reuse was detected. 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

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

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.

The space overhead for implementing this predicate is $\log n$ bits for the repeat counter, and $2 \log k$ bits to identify the two previous addresses accessed; these then enable the strides to be calculated.

The above definitions are illustrated in Fig. 2. Using the SPEC gcc benchmark as an example, the top graphs simply show the access patterns to data and instructions. Below them we show the Denning working set $D_{1000}(t)$ (i.e. for a window of 1000 instructions) and the core working set $C_{1000,2\times W}(t)$. As we can easily see, the core working set is indeed much smaller, and in particular, it eliminates much of the sharp peaks that appear in the Denning working set and reflect memory scans. In the next section we will use this to design a filter cache that only caches the core, and filters out the rest. Additional examples are shown in Fig. 3. In some cases, such as the gzip benchmark, the reduction in working set size is very large. However, there were other benchmarks in which the difference was smaller, or even negligible. This is reflected in the performance evaluation results reported in Section 5.

3.3 Parameter Values

An important parameter of the filtering is the window size that needs to be considered. The bigger the window, the larger the filter has to be. We therefore need to base our design on a size that is as small as possible, but large enough to capture enough reuse so as to enable the core to be identified. Such data is shown in fig. 4. This shows the distribution of observed reuse of 8-word blocks and of single words as a function of the window size. According to this data, a window size of about 64 blocks suffices to identify about 75–90% of the total reuse of blocks for all the benchmarks but two. For word-level reuse, the corresponding range is around 50–70%. Note however that the plots become rather flat beyond this point, so a noticeable increase in the captured reuse would require a significantly larger window size. We therefore feel that a window size of 64 is a suitable design point, as it is implementable and somewhat after the knees in the curves.

An example of how filtering interacts with the access pattern is shown in Fig. 5. The top plot is a subset of the access patterns for gcc formerly shown in Fig. 2. Here, these scatter plots are color coded to show the parts that are filtered out using the 2xW filter. The bottom is from the gzip benchmark, and clearly shows how some memory scans are filtered out.

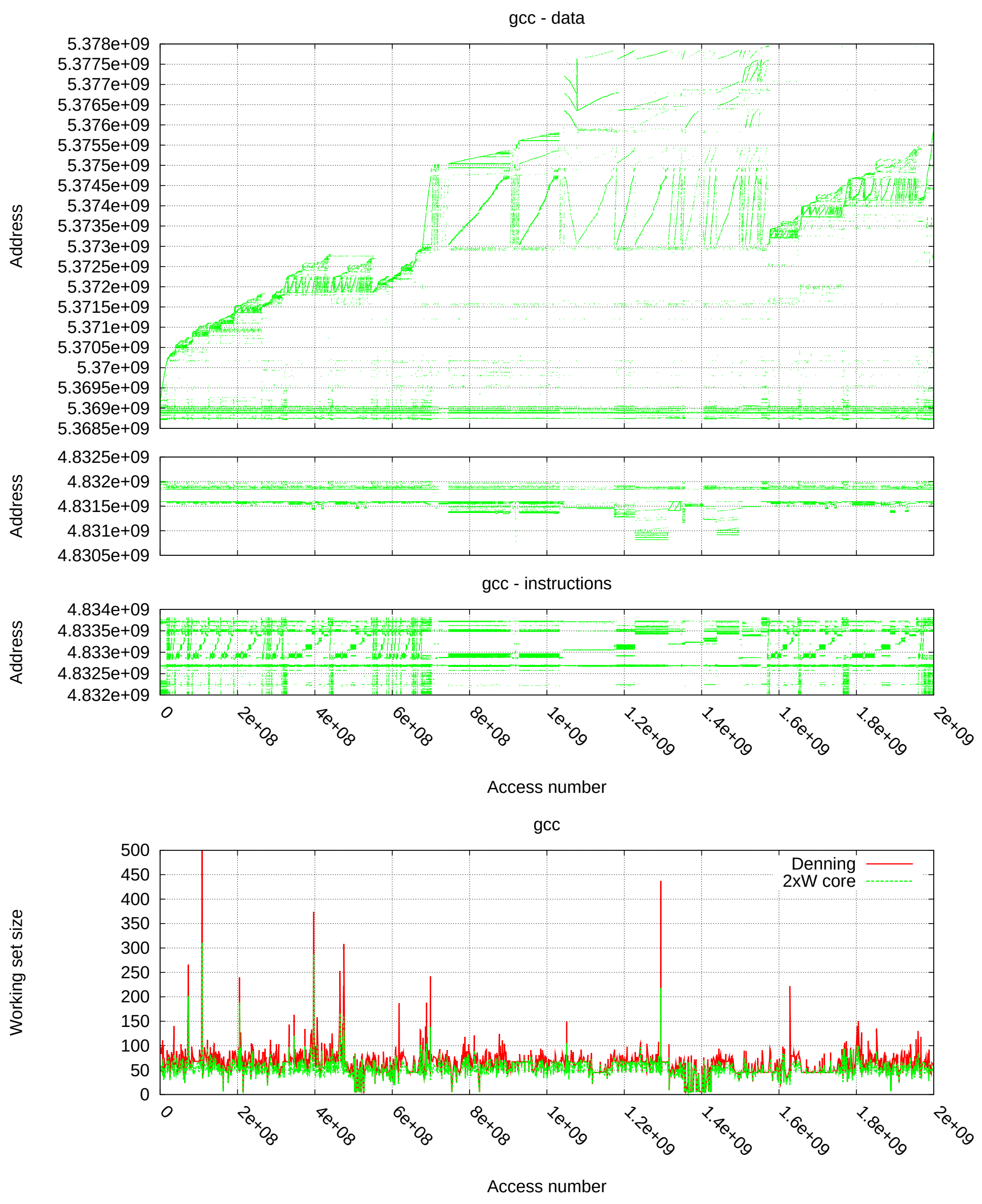


Figure 2: *Examples of memory access patterns and the resulting Denning and core working sets.*

4 Design of a Filter Cache

The working set cache management policy attempts to maintain the program's working set in the cache [6]. Our policy is to try and focus on only the *core* working set. This can be done by a filtering mechanism that implements the predicates outlined above.

The flow of data between the different levels of the memory hierarchy is as shown in Fig. 6. In a conventional caching system, requests that cannot be satisfied at one level are forwarded to the next level, and when the data finally arrives it is cached. In our scheme, in contradistinction,

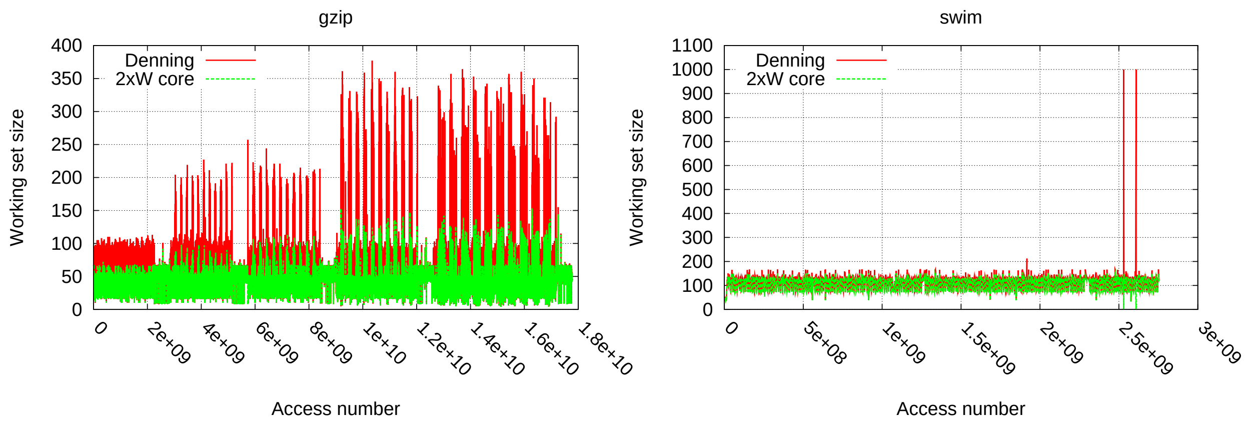


Figure 3: *Additional examples of Denning and core working sets.*

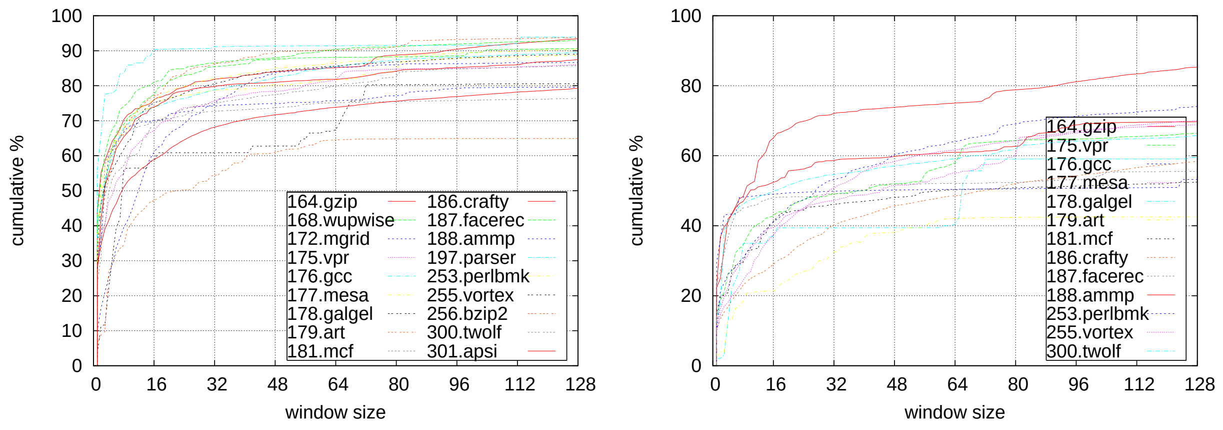
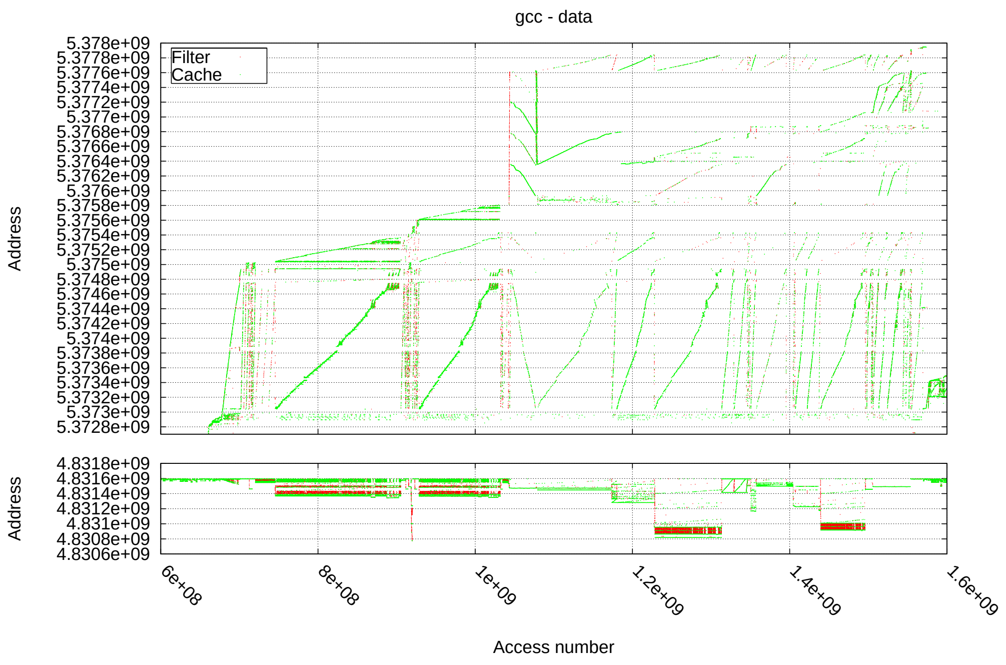


Figure 4: *Distribution of observed repeated accesses to memory, at the block level (left) and word level (right). It is important that the filter size used be beyond the knee in these graphs.*

newly arrived data is not immediately admitted into the cache. Instead, it is placed in a filter that implements the desired predicate.

The filter is fully associative and managed in an LRU manner. It operates according to the following rules. When an address is referenced, the CPU searches for it in the cache and filter in parallel. If it is in the cache it is retrieved as in conventional systems. If it is found in the filter, this new reference is tabulated, and the filter predicate is applied. If the cache line containing the referenced address now satisfies the predicate, it is promoted into the cache proper. If not, it is served from the filter and moved to the MRU position. If the referenced address is not found in neither the cache not the filter, it is retrieved from the next higher level, and inserted into the MRU position of the filter. To make space for it, the cache line in the LRU position needs to be evicted. If this cache line has been modified while in the filter, it is written back to the next higher level in the hierarchy. If not, it is simply discarded.

The main algorithmic aspect of the workings of a filter cache is the promotion algorithm. This



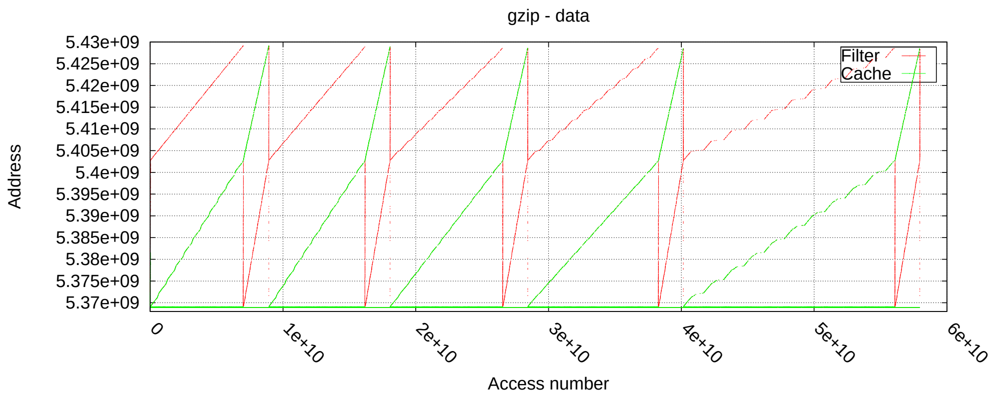


Figure 5: *Examples of memory access filtering. Top: gcc benchmark (a subset of the data shown in Fig. 2). Bottom: gzip benchmark.*

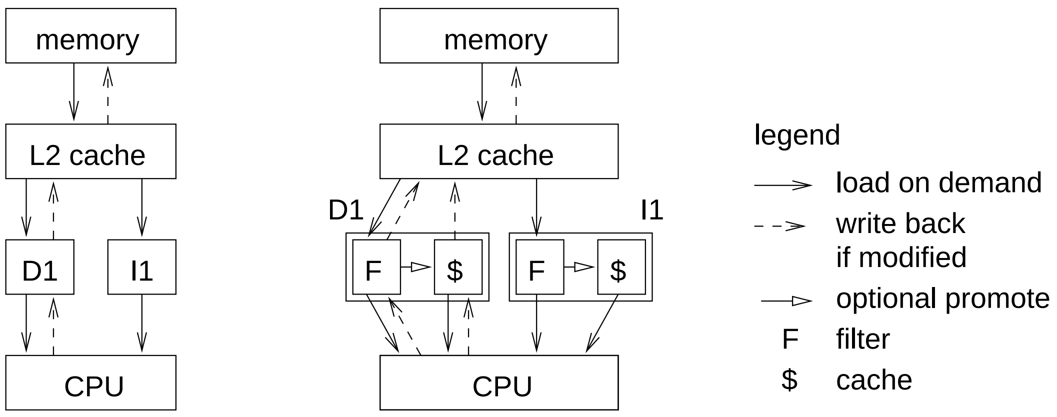


Figure 6: *Basic design and data flows of a system with filtered D1 and I1 caches (right) compared with the conventional design (left).*

implements the chosen predicate, that is supposed to identify members of the core working set. The specific predicates that we evaluated are listed in Section 5. Note that our cache lines contain 8 words by default, which implies that we may consider them to be members of the core even if most of the words were not referenced. This reflects experience with trying out various alternatives, which showed that strided accesses often cause many words to be skipped altogether, and that this should not be taken to imply that other words in the cache line will not benefit from caching.

The default parameters for our filters are:

- 64 entries
- 8 words per entry
- fully associative

These parameters were selected as a compromise between the expected performance gains and the implementation cost. A larger fully associative filter could be too costly in three ways. First, the latency of the search logic depends on the number of entries that need to be searched. We want to keep the filter latency similar to that of the L1 cache, so we must limit its size. Second, because a fully associative search is done in parallel on all entries, the energy consumption is also proportional to the size. Finally, a larger filter takes up more die area. In fact, our selected parameter values are quite modest, as witnessed by the HP assist cache implementation from 1997. We study their effect and the various cost-performance tradeoffs in the next section.

5 Evaluation of Benefits

To evaluate the implementation of cache insertion policies based on a filter cache, we ran extensive simulations using SimpleScalar [1] and the SPEC 2000 benchmarks [18]. All the SPEC benchmarks were used except for six: eon and equake, in which the miss rate was extremely low to begin with (under 0.1%), gap, sixtrack, and fma3d, who failed to run in our environment, and

cache		micro-architecture	
L1 design	split	fetch/issue/decode width	8
D1/I1 size	16 KB	functional units	8
D1/I1 line size	64 B	window size	128
D1/I1 set assoc.	4	branch prediction	perfect
D1/I1 sets	64	memory	
D1/I1 latency	2 cy.	memory latency	100 cy.
L2 design	unified	filter	
L2 size	256 KB	filter size	4 KB
L2 line size	128 B	filter line size	64 B
L2 set assoc.	8	filter entries	64
L2 sets	256	filter assoc.	full
L2 latency	10 cy.	filter latency	3 cy.

Table 1: *Basic configuration used in the simulations.*

	DL1			IL1			UL2		
benchmark	avg lat	missrate	misses	avg lat	missrate	misses	avg lat	accesses	missrate
mgrid	4.5586	0.0344	25e6	2.0006	0	30226	47.8032	32e6	0.3793
facerec	9.6268	0.0335	24e6	2.0002	0	5647	59.4110	36e6	0.5089
lucas	17.9727	0.0789	33e6	2	0	808	57.2598	51e6	0.4365
vpr	7.7267	0.0532	37e6	2	0	360	48.2863	50e6	0.3889
mesa	2.5473	0.0051	3e6	2.0123	0.0014	3e6	20.1625	9e6	0.0916
parser	4.2167	0.0322	22e6	2.0017	0.0001	212578	29.8848	28e6	0.1833
crafty	2.4003	0.0246	18e6	2.1505	0.0185	49e6	11.6491	69e6	0.0169
mcf	27.5424	0.2078	186e6	2.0007	0.0001	292070	58.5592	265e6	0.4362
ammp	27.9668	0.1823	128e6	2.0001	0	7567	97.1783	133e6	0.8227
gzip	2.4609	0.0221	14e6	2	0	190	12.4497	20e6	0.0229
perlbmk	2.3888	0.0203	18e6	2.1144	0.0138	39e6	11.9269	61e6	0.0197
swim	7.9702	0.0868	57e6	2.0003	0	36133	38.7305	77e6	0.2959
galgel	6.3150	0.1813	161e6	2	0	384	15.5151	162e6	0.0568
art	34.1540	0.3421	254e6	2	0	125	79.4809	308e6	0.5670
twolf	8.8613	0.0647	42e6	2.0234	0.0026	6e6	43.2337	64e6	0.3425
wupwise	3.7475	0.0128	7e6	2	0	291	50.8031	9e6	0.3701
apsi	4.5482	0.0389	29e6	2.0945	0.0117	25e6	28.0604	66e6	0.1839
bzip2	3.3189	0.0267	19e6	2	0	758	36.4921	24e6	0.2659
vortex	2.7585	0.0178	15e6	2.1461	0.0156	43e6	14.4925	63e6	0.0438
gcc	21.3845	0.0546	76e6	2.0847	0.0049	11e6	49.2831	131e6	0.3015

Table 2: *results for SPEC 2000 benchmarks using the base configuration, with no filtering.*

DL1 filter	IL1 filter	description
$2 \times W$	none	cache data blocks that include any single word that is referenced for the second time, and all instruction blocks
$3 \times B$	none	cache data blocks to which three references are made, and all instruction blocks
$3 \times ST$	none	cache data blocks that exhibit non-uniform strides or more than 3 consecutive accesses to the same word, and all instruction blocks
$2 \times W$	$2 \times W$	cache data or instruction blocks that include any single word that is referenced for the second time
$3 \times B$	$2 \times W$	cache data blocks to which three references are made, and instruction blocks with a word referenced for the second time
$3 \times ST$	$2 \times W$	cache data blocks that exhibit non-uniform strides or more than 3 consecutive accesses to the same word, and instruction blocks with a word referenced for the second time

Table 3: *Filter configurations compared in the simulations.*

applu, which had a truncated binary[2]. We simulated the benchmarks as compiled by the DEC compiler for the Alpha architecture [16]. We also checked compilation with gcc, for both the Alpha and x86 architectures; in most cases checked the results were essentially the same, and they are not reported here. When multiple input datasets exist for a benchmark, input 0 was used.

The simulation was carried out with conventional default architectural parameters, as outlined in Table 1. The latency of accessing the filter is conservatively assumed to be 3 cycles, as opposed to only 2 cycles for the cache, based on a timing analysis using the CACTI 3.2 model [15]. When the train workloads were used, the full execution was simulated. For the reference workloads, we first skipped one billion instructions, and then tabulated the performance of the next two billion. Most of the results reported here are for the reference workload. The results for the base configuration, with no filtering, are listed in Table 2. The effect of different types of filtering are shown in subsequent figures as an *improvement* relative to these results (so positive values are better).

The main goal of the simulations is to evaluate the effectiveness of cache insertion policies, as embodied in different filtering configurations. Most simulations compared the base configuration described above with six optional filters as described in Table 3. For the IL1 cache we define the word size to be 4 bytes, so as to match the instruction size in the Alpha architecture [16] (in the DL1 cache a word is defined to be 8 bytes).

Results pertaining to the DL1 cache are shown in Fig. 7. As the number of accesses is constant, there is a strong correlation between the miss rate and the number of misses, so we show only one of them. Most of the benchmarks show an improvement in the average latency and/or the cache

[2]Dec OSF-compiled binaries were downloaded from the SimpleScalar web site.

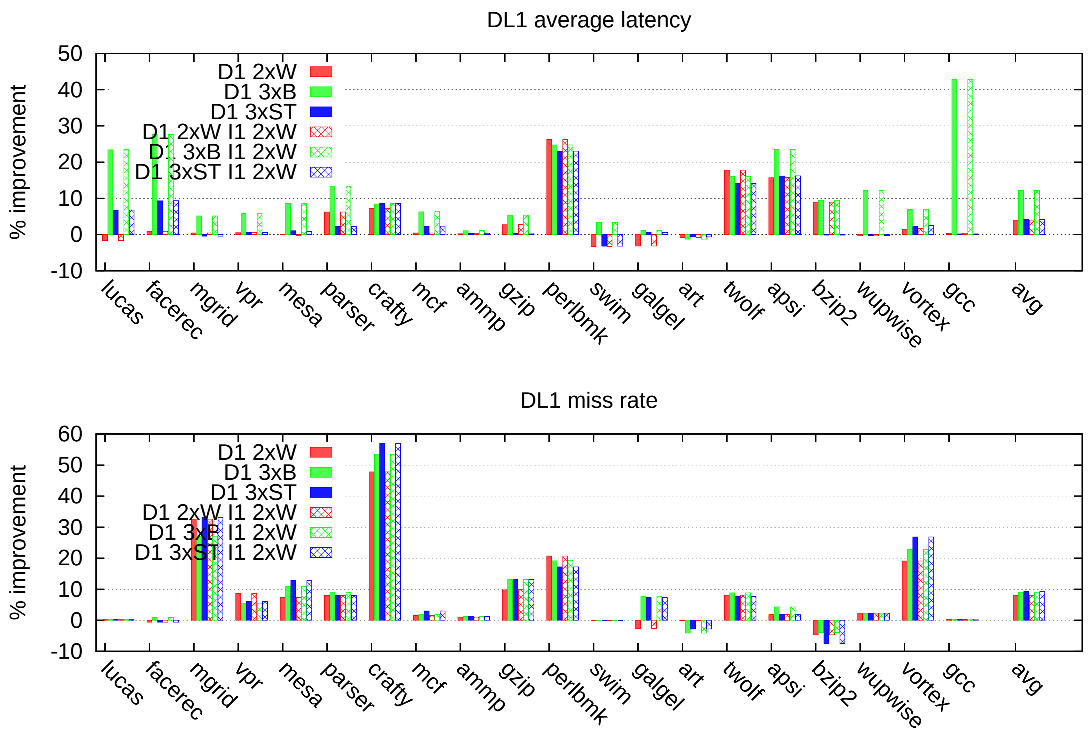


Figure 7: *Effect of filtering on the performance of the DL1 cache.*

misses; a few show minor degradation in performance for some filters. The 3xB filter is especially effective in reducing the average latency (relative to the base case), and leads to a reduction of 12.2% on average. The 2xW and 3xST filters reduce the average latency by about 4% and 4.2% on average. For the miss rates and number of misses there is little difference between the performance of the different filters. The 2xW filter reduces the miss rate by about 8% on average, while the other two reduce it by about 9% and 9.3%. In general, adding filtering to the IL1 cache does not affect

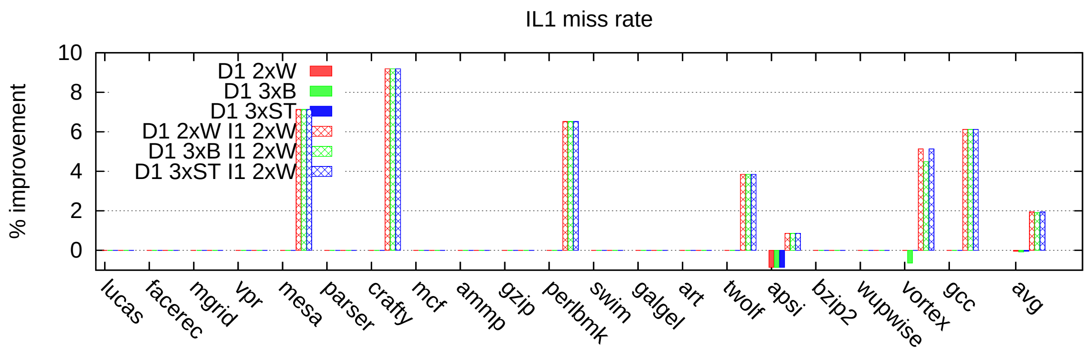


Figure 8: *Effect of filtering on the performance of the IL1 cache.*

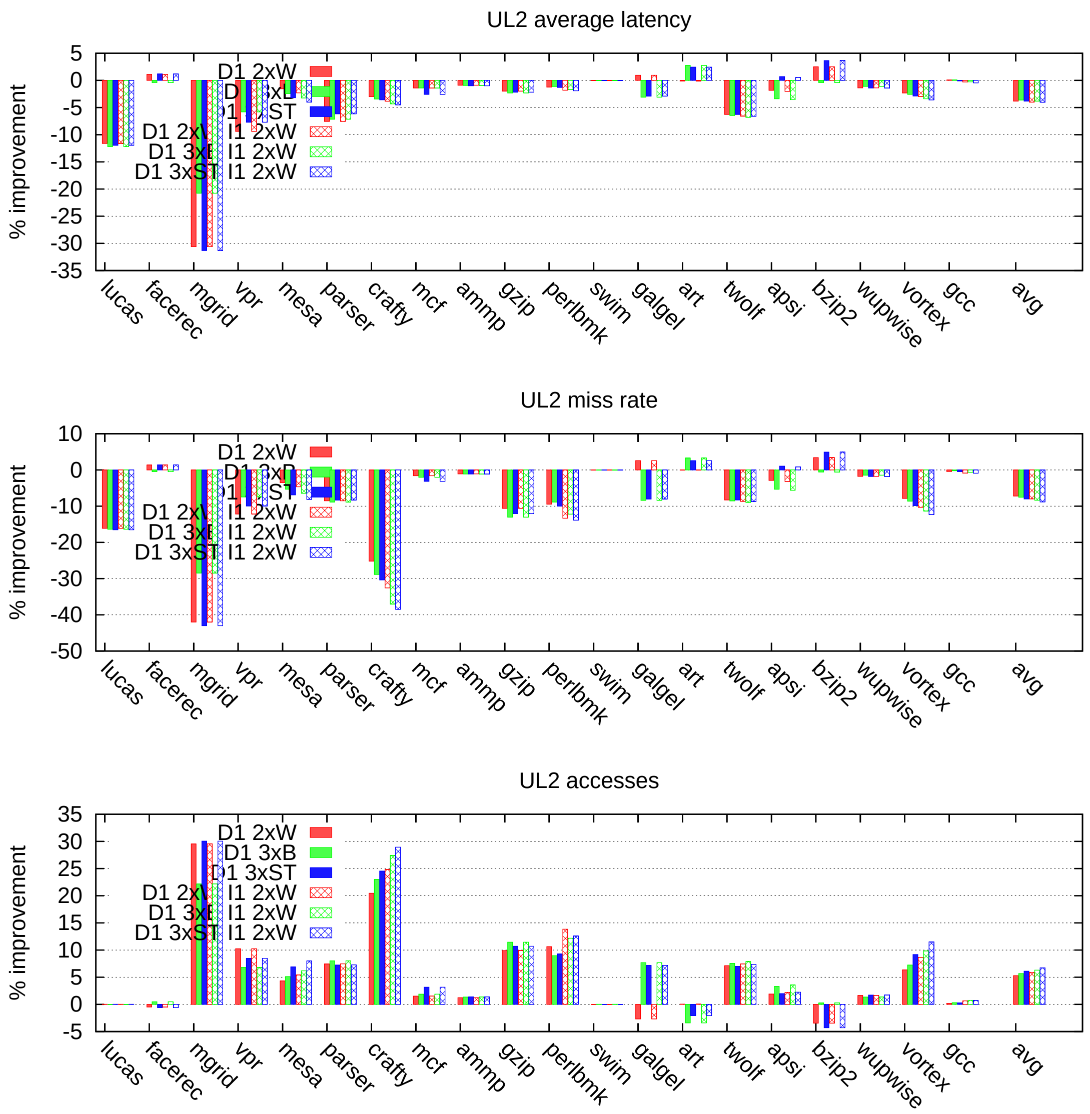


Figure 9: *Effect of filtering on the performance of the UL2 cache.*

the performance seen for the DL1 cache.

The effects of filtering on the IL1 cache were minimal, as shown in Fig. 8. Some benchmarks show a marginal degradation in access latency, but this is limited to less than 0.5% (not shown). In most benchmarks the miss rate is essentially zero, because the cache is large enough to include all working sets. In those that do exhibit misses, filtering reduces the miss rate by up to 9.2%, but the average is only 1.9% due to the many benchmarks with negligible misses. Naturally, these effects are largely confined to filter configurations that include filtering of the IL1 cache.

Simulation results for the UL2 cache are shown in Fig. 9. for most benchmarks, the average latency to the L2 cache is degraded. The maximal degradation is just over 31%, but the average is around 3.6–4%. This is true for all filter configurations, with only minor variations. The results

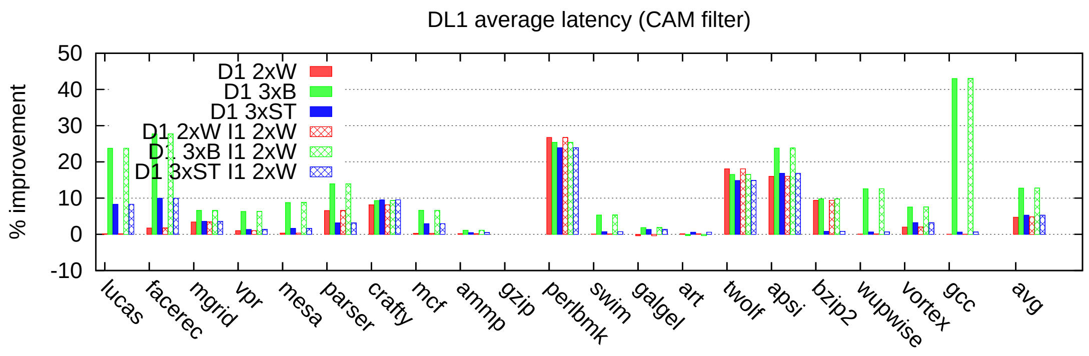


Figure 10: *DL1 performance with an improved filter implementation.*

	base config			with filters		
	8KB	16KB	32KB	8+4KB	16+4KB	32+8KB
avg latency [cy.]	10.48	10.28	10.11	10.20	10.12	9.96
miss rate [%]	9.47	7.92	6.53	8.21	7.60	6.37

Table 4: *Comparison of adding a filter to the DL1 cache with enlarging the DL1 cache.*

are similar for the miss rate, with the average degradation in the range 7.2–8.8%. However, these results are actually misleading. The miss rate is the fraction of misses out of the total accesses (misses + hits). An increased miss rate can therefore reflect more misses, or less hits. In our case it is mainly the latter, as witnessed by the chart showing the number of accesses to the UL2 cache, which is reduced by 5.3–6.7% on average (in all the charts, including this one, a positive “improvement” means reduction). Note also the strong correlation between reduced accesses and degraded miss rate. Thus it is the improved performance of the L1 caches, and specifically the reduction in L1 misses, that make the L2 cache look slightly worse.

The reduced accesses to the L2 cache are actually an important benefit. In forthcoming multi-core microprocessors the L2 cache is shared by the different cores. Thus contention for the L2 cache and its ports may become a serious performance issue. Filters on the L1 caches may help alleviate this concern by reducing the number of requests that are forwarded to the L2 cache.

All the simulations reported so far were conservative in the sense that they assumed that access to the filter costs 3 cycles, whereas access to the DL1 and IL1 caches only costs 2 cycles. It is possible, however, to reduce the access time to the filter by implementing it using a content-addressable memory (CAM) [3, 21]. This is a realistic option, as witnessed by the use of such memories in the core of modern microprocessor architectures, for example in the implementation of instruction issue lookup [7].

Re-running the simulations with filters that are assumed to require only 2 cycles per access leads to the results shown in Fig. 10. These are only slightly better than the results shown above in Fig. 7 for filters that require 3 cycles. The average improvement in average latency is now about 4.7% and 5.3% for the 2xW and 3xST filter configurations, and 12.8% for the 3xB filters.

An important question when adding an auxiliary structure like a filter is whether this is good

technology	4-way	8-way	4-way+filter
90nm	0.38nJ	0.68nJ	0.51nJ
65nm	0.26nJ	0.48nJ	0.34nJ

Table 5: *Results of power estimates using the CACTI 3.2 model, for a 16KB cache and a 4KB (64-entry) filter.*

use of the chip real-estate. By adding a filter, we are effectively enlarging the cache; maybe the above results would also be obtained by a simple corresponding increase in cache size, without the filtering mechanism? To check this we compared our base configuration of a 16KB DL1 cache with 8KB and 32KB caches, both with and without an added filter. For the 16KB cache we used a 64-entry filter, as above. The same filter size was used with the 8KB cache, because a reduced 32-entry filter is not large enough to capture reuse effectively. For the 32KB cache, we enlarged the filter to maintain the same ratio as for 16KB, i.e. have a filter that adds 25% to the cache.

The results in terms of average latency and miss rate are shown in Table 4. They indicate that in terms of average latency the filter is very competitive: an 8KB cache with filter achieves a lower average latency than a 16KB cache, and a 16KB cache with filter achieves a latency similar to that of a 32KB cache, in both cases using less space and less energy (energy calculations are given below in Section 6). In terms of miss rate the filter is equivalent to increasing the cache size. For example, the miss rate of a 16KB cache with filter is 7.60%. The effective size of this combination is 20KB. And indeed, by interpolating between the 16KB and 32KB miss rates for caches without a filter, we find that a 20KB cache would be expected to suffer a miss rate of 7.57%.

6 Energy Considerations

An obvious concern with our filter design is energy consumption. It is well-known that the power expenditure of a cache structure depends on its size and associativity. We used the CACTI 3.2 model to perform an initial evaluation of our design [15]. The estimated power consumption is calculated simply as the sum of the power consumption of the filter and the cache itself, using each one's specific configuration.

The results of such evaluations are shown in Table 5. We compare 16KB caches with different levels of set associativity, and with an added 64-entry fully associative filter. A notable result is that the estimated power consumption of a 4-way associative cache with an added filter is *lower* than that of an 8-way associative cache without a filter.

Considering the optional use of CAM to implement the filter in order to reduce its access time, we note that a CAM structure of 64 lines is much more energy efficient compared to even smaller RAM structures [21].

	L1 avg latency	L2 avg latency	IPC
base case	10.68	40.68	0.712
with filter	8.86 (-17.07%)	43.36 (+6.58%)	0.718 (+ 0.83%)
with 512KB L2	8.24 (-22.89%)	30.42 (-25.22%)	0.860 (+20.80%)

Table 6: *Measured IPC is sensitive to the L2 average latency, but not to the L1 average latency, motivating the study of attaching a filter mechanism to the L2 cache. Data for the twolf benchmark.*

7 Conclusions and Future Work

Computer programs use different parts of their address space in different ways. In particular, some memory words are accessed very many times, while others are not. This may have a big impact on caching policies: caching memory words that will not be reused pollutes the cache and takes the place of other words that could have been cached to benefit. We therefore propose that caches be managed using an insertion policy, that tries to filter out a large part of the references. The filter implements a predicate that tries to identify those parts of the address space that constitute the core working set, and that really deserve to be cached.

We have presented an initial study of the parameters of the filter, and how they affect performance and cost. But there is still much to be done. One possible improvement is to replace the fully associative filter by a highly associative filter. This will enable the use of a larger filter, while limiting the access latency and power consumption. Another is the possible divergence of the filter block size and the cache block size. In our simulations they were always the same, but it is possible that some benefits can be obtained by using different sizes.

As we have shown, the filter cache leads to a significant reduction in the average memory access latency, and also to a significant reduction in the bus traffic between the L1 and L2 caches. However, this only led to a marginal improvement in the IPC. This implies that L1 access is not the bottleneck in our combination of processor configuration and benchmarks. However, when we experimented with an artificially improved L2 cache we did achieve an improvement in the IPC (Table 6). This means that the IPC is actually sensitive to the *long* latencies which cause the whole pipeline to empty out. Thus an important and promising area for future research is to include a filter mechanism in the L2 cache, so as to reduce the average latencies involved in L2 access. This is complicated by the fact that the L2 cache does not see all memory accesses, as the lion's share are absorbed by the L1 caches. Thus we need to devise a mechanism to keep the L2 filter informed of the locality properties of the reference stream, so that it can make and informed decision regarding what to insert into the cache.

Another direction for future research is to try and incorporate criticality information with the reuse information. The schemes proposed above are oblivious of request criticality. However, in real applications, some requests are more critical then others, in terms of their effect on the IPC [19]. Adding such considerations to the filter can further improve its fidelity, and allow even higher focus on caching only those addresses that lead to improved performance.

References

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

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

[4] P. J. Denning, “*The locality principle*”. *Comm. ACM* **48(7)**, pp. 19–24, Jul 2005.

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

[6] P. J. Denning, “*Working sets past and present*”. *IEEE Trans. Softw. Eng.* **SE-6(1)**, pp. 64–84, Jan 1980.

[7] D. Ernst and T. Austin, “*Efficient dynamic scheduling through tag elimination*”. In *Intl. Symp. on Computer Architecture*, pp. 37–46, May 2002.

[8] A. González, C. Aliagas, and M. Valero, “*A data cache with multiple caching strategies tuned to different types of locality*”. In *Intl. Conf. on Supercomputing*, pp. 338–347, ACM Press, New York, NY, USA, 1995.

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

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

[11] N. Megiddo and D. S. Modha, “*ARC: a self-tuning, low overhead replacement cache*”. In 2nd *USENIX Conf. File & Storage Tech.*, pp. 115–130, Mar 2003.

[12] 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*, pp. 108–116, ACM Press, New York, NY, USA, 2002.

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

[14] J. Sahuquillo and A. Pont, “*The filter cache: a run-time cache management approach*”. In *EUROMICRO*, pp. 1424–1431, IEEE Computer Society, 1999.

[15] P. Shivakumar and N. P. Jouppi, *CACTI 3.0: An Integrated Cache Timing, Power, and Area Model*. Technical Report WRL Technical Report 2001/2, Compaq Western Research Laboratory, Aug 2001.

[16] R. L. Sites, "*Alpha AXP architecture*". *Communications of the ACM* **36(2)**, pp. 33–44, Feb 1993.

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

[18] *Standard performance evaluation corporation*. URL http://www.spec.org.

[19] S. T. Srinivasan and A. R. Lebeck, "*Load latency tolerance in dynamically scheduled processors*". In *31st Intl. Symp. on Microarchitecture*, pp. 148–159, IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

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

[21] M. Zhang and K. Asanovic, "*Highly-associative caches for low-power processors*". In *Kool Chips Workshop, 33rd Intl. Symp. on Microarchitecture*, IEEE Computer Society, 2000.