Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines

Moinuddin K. Qureshi M. Aater Suleman Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{*moin, suleman, patt*}@*hps.utexas.edu*

Abstract

Caches are organized at a line-size *granularity to exploit spatial locality. However, when spatial locality is low, many words in the cache line are not used. Unused words occupy cache space but do not contribute to cache hits. Filtering these words can allow the cache to store more cache lines. We show that unused words in a cache line are unlikely to be accessed in the less recent part of the LRU stack. We propose* Line Distillation (LDIS)*, a technique that retains only the used words and evicts the unused words in a cache line. We also propose* Distill Cache*, a cache organization to utilize the capacity created by LDIS. Our experiments with 16 memory-intensive benchmarks show that LDIS reduces the average misses for a 1MB 8-way L2 cache by 30% and improves the average IPC by 12%.*

1. Introduction

Caches are organized at a *line-size* granularity to exploit spatial locality. Using large line-size provides a performance improvement proportional to the amount of spatial locality in the memory reference stream. However, spatial locality varies across applications and between different accesses of the same application. When spatial locality is low, majority of words in a line are never used. Cache performance can be improved by discarding such words and using the cache space to store useful data. Removing unused words from cache lines is called *spatial filtering*.

For set-associative caches, the information about which words are used and which words remain unused stabilizes as the line traverses the LRU stack. We show that unused words in a cache line are less likely to be accessed in the less recent part of the LRU stack. We propose, *Line Distillation (LDIS)*, a technique to discard the unused words. LDIS tracks the word usage information of a cache line until it reaches a predefined recency position in the LRU stack and then evicts the words that have not been used. Unlike previous proposals [7][9][3][11] for spatial filtering, LDIS does not require a separate prediction structure for tracking spatial locality.

LDIS requires a cache organization that can utilize the extra capacity created by filtering the unused words. Organizing the cache at word-size granularity as done in [7] requires a tag overhead as much as half of the cache size (approximately 4B of tag overhead for 8B data). Kumar et al. [9] use a decoupled sectored cache [14] to reduce the tag overhead. However, a decoupled sectored cache constrains the position where a word can be installed, which reduces the benefit of spatial filtering [9]. We propose a novel cache organization, *Distill Cache*, which consists of two structures: *Line-Organized Cache (LOC)* and *Word-Organized Cache (WOC)*. Cache lines are initially placed in the LOC. When the line is evicted from the LOC, the used words of the line are transferred to the WOC and the unused words are discarded. The distill-cache provides a good trade-off between tag-overhead and flexible placement of the words that remain after distillation. We describe the design and operation of the distill-cache in Section 5.

Installing lines that have a large number of words used can evict several useful lines from the WOC. The number of useful lines in the WOC can be increased by not installing lines for which the number of used words exceed a certain threshold. We propose *median-threshold filtering* that dynamically tracks the median number of words used in the cache line for a given application. If the cache line evicted from LOC has more than the median number of words used, then none of its words are installed in the WOC.

LDIS assumes that unused words are unlikely to be used in the less recent part of the LRU stack. This assumption, although true for most workloads, is violated for some benchmarks. To make LDIS applicable to a wide variety of workloads, we propose a low-overhead *reverter circuit*. The reverter circuit disables LDIS when it is likely to hurt cache performance.

Most of the previous spatial filtering studies [9][3][11] were evaluated for first-level data (L1D) caches. The design of L1D cache is heavily constrained by cycle time which makes it less amenable to performance optimizations. Moreover, with out-of-order execution, the processor can tolerate some of the L1D misses [8]. Our study is focused on improving the performance of second-level (L2) caches. In Section 4, we describe the framework to enable LDIS for L2 cache. Our evaluations with 16 memory-intensive benchmarks show that LDIS reduces the misses for a 1MB L2 cache by 30% and improves IPC by 12% (other parameters of the study are described in Section 6).

An orthogonal approach for increasing cache capacity is cache compression. Although the goal of compression and LDIS is identical, they exploit fundamentally different inefficiencies in cache design. Compression exploits the redundancy in information stored in a cache line, whereas, LDIS exploits the useless words in a cache line. In Section 8, we study the interaction between LDIS and compression. To our knowledge, this is the first study to investigate the interaction between spatial filtering and compression. We show that they interact positively and can be combined for greater capacity benefits than either scheme by itself. The proposed combination, *footprint-aware compression*, reduces the average misses for the 1MB L2 cache by 50%.

2. Problem

The line-size of a traditional cache is fixed at design time. When the spatial locality is low, a large amount of cache space is allocated to data that is never accessed. To quantify the amount of useful data in a cache line, we divide the cache line into equally sized units called *words*. In our studies, the size of each word is equal to 8B.[1] Figure 1 shows the histogram of the number of words used in the cache line of the baseline L2 cache with 64B line-size.

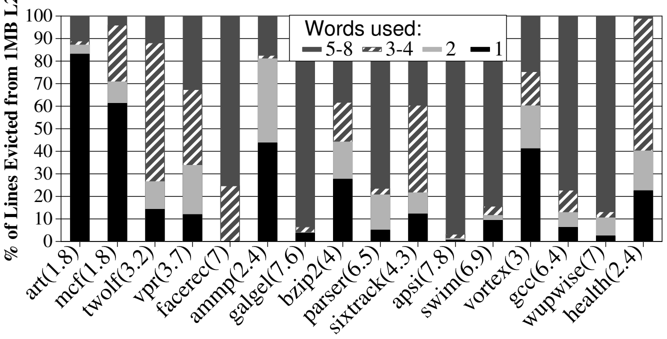

Figure 1. Distribution of the words used in a cache line. Average number of words used is also shown for each benchmark.

A line-size of 64B allows the cache to exploit spatial locality for facerec, galgel, apsi, swim, and wupwise. However, for art and mcf, on average less than two out of the eight words are used, indicating that more than 75% of cache space is used for storing words that are never accessed. For 8 out of the 16 benchmarks, on average four or fewer words are used indicating that more than half the words in the cache line remain unused. The unused words consume cache space without contributing to cache hits. Cache performance can be improved if unused words are filtered out of the cache and the reclaimed cache space is used for storing useful lines. An obvious way to reduce the number of unused words is to reduce the line-size. However, spatial locality varies across applications and within different memory regions of an application. Therefore, reducing cache line-size from 64B to 32B increases the cache misses for most of the benchmarks.[2] Our objective is to reduce cache misses by means of filtering the unused words.

3. Motivation

The unused words in the cache line can be evicted if such words can be identified. To track unused words, we associate a bit-vector, *footprint*, with each cache line. The footprint contains eight bits - one bit for each word in the line. When a line is placed into the cache, its footprint is reset to all zeros. When a word is accessed, its bit in the footprint is set to 1. To track the distribution of footprint changes with respect to the position of the line in the LRU stack, we associate each recency position with a numerical value. We term MRU as *position 0*, the recency position next to MRU as *position 1*, and so on. LRU corresponds to *position 7* for the baseline 8-way cache. For each cache line we record the maximum position attained by the cache line before its footprint gets changed. Consider a cache line A that is in position 0 when its first footprint-change occurs. Later, line A moves to position 5. An access to a different word causes another footprint-change and the line moves back to position 0. Line A is never accessed again and is eventually evicted from the cache. In this scenario, we say that the maximum recency position before footprint-change is position 5. Figure 2 shows the distribution of maximum recency position before footprint-change for all the lines when they are evicted from the cache.

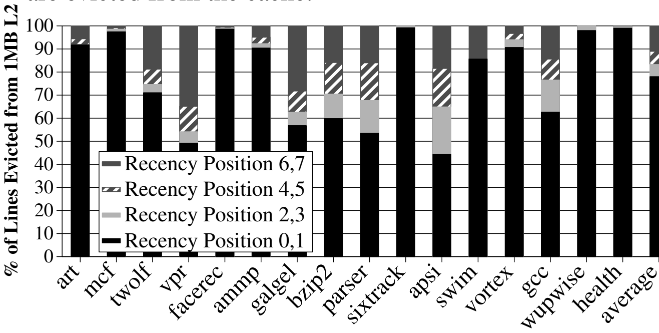

Figure 2. Distribution of maximum recency position before footprint-change. For an 8-way cache, recency position 0 denotes MRU and recency position 7 denotes LRU.

On average, 83% of footprint-changes occur when the cache line is between position 0 and 3. Less than 12% of the footprint changes occur after the line reaches position 6 or 7. Thus, the footprint stabilizes as the line reaches the bottom quarter (position 6 and 7) of the recency stack. By tracking footprints during run-time, the unused words of a cache line can be identified and discarded. This technique of retaining used words and discarding unused words is called *Line Distillation (LDIS)*. The next section describes the framework to support LDIS.

[1]We use the Alpha ISA. The maximum size of a memory access generated by an Alpha instruction is 8B. Therefore, we use 8B for word size.

[2]We use a line-size of 64B, similar to commercial processors. The problem of unused words is worse for larger line-sizes.

4. Framework for Line Distillation

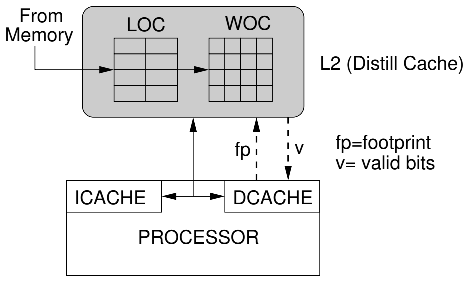

Figure 3. Framework for Line Distillation (Figure not to scale)

Figure 3 shows the framework for implementing LDIS in L2 cache. As instruction cache lines have high spatial locality, we perform LDIS only for the data lines. The cache organization that supports LDIS is called *Distill Cache*. It consists of two structures: *Line-Organized Cache (LOC)* and *Word-Organized Cache (WOC)*. Cache lines are initially placed in the LOC. When the line is evicted from the LOC, the used words of the line are transferred to the WOC and the unused words are discarded. We discuss the design of the distill-cache in detail in Section 5.

4.1. Tracking Footprint for Lines in LOC

The access stream generated by the processor is visible only to the first level (L1) caches. To track which words in the cache line are used, each tag-entry in the LOC contains an 8-bit footprint field. When the line is installed in LOC, the footprint associated with the line is reset to all zeros and the line is transferred to the L1. Each line in the L1D also has a footprint field associated with it. The footprint in L1D is updated as the processor accesses the word in the cache line. When the line is evicted from the L1D, the footprint associated with it is sent to the LOC. If the line evicted from the L1D is present in the LOC, the footprint of the line is OR-ed with the footprint already present in the LOC. Thus, the LOC is able to obtain word usage information even though the processor accesses are not directly visible to it.

4.2. Variable Number of Valid Words in L1D

On a WOC hit, all words for the requested line in the WOC are sent to the L1D. The WOC also sends a bit vector, *valid bits*, containing one bit of valid/invalid information for each word in the cache line. To accommodate variable number of valid words in the L1D line, we use a sectored cache for the L1D. If an invalid word in the line is accessed by the processor, a request for the line (along with the sector id) is sent to the distill-cache. For L1D misses that are satisfied by the LOC or memory, all words in the L1D line are marked as valid.

5. Distill Cache

LDIS requires a cache organization that can utilize the extra capacity created by filtering the unused words. Organizing the entire cache at word-size granularity as done in [7] requires a tag overhead as much as half the cache size. The proposed *Distill Cache* organization allows flexible placement of words that remain after LDIS while incurring the tag overhead for only a subset of the cache.

5.1. Organization

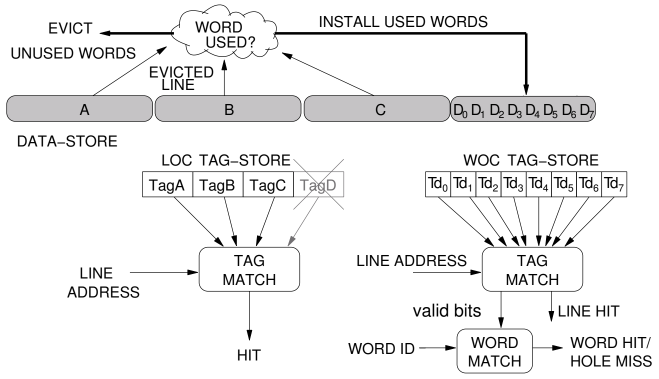

Figure 4. Organization of one set of a 4-way distill-cache. LOC consists of ways A, B, and C and WOC consists of way D.

Figure 4 shows how a distill-cache can be implemented with a four-way traditional cache. For simplicity, we show only one set of the cache and assume that tag access and data access are done sequentially.[3] The distill-cache shown in Figure 4 devotes three ways (ways A, B, C) to LOC and the remaining one way (way D) to WOC. The tag-store of WOC contains one tag-entry for each word in way D. For example, tag-entry Td_0 corresponds to word D_0 and tag-entry Td_1 corresponds to word D_1. Each tag-entry in WOC also contains *word-id* bits to identify the address of the stored word. The data-store for WOC remains unchanged as the data-entries in WOC are only *logically* partitioned into words. The incoming line from memory is always placed in the LOC. Each tag-entry in LOC contains a footprint field to track which words are used while the line is in the LOC. When the line is evicted from LOC, the used words in the line are installed in WOC and the unused words are evicted.

Multiple words of the same cache line are always stored in consecutive positions in the WOC. For example, for a cache line X evicted from the LOC, only the first word (X_0) and the last word (X_7) are used. Then, if X_0 is stored in Td_2 then X_7 must be stored in Td_3. We allow only power-of-two (1, 2, 4, or 8) words for each cache line installed in the WOC. Furthermore, multi-word lines must be aligned.

[3]Existing processors serialize tag comparison and data lookup to reduce the power dissipation of large cache arrays [4][17]

For example, a line with four words used can only start at Td_0 or Td_4 and a line with two words used can only start at Td_0, Td_2, Td_4 or Td_6. The restriction of aligned placement helps in getting all the words of a cache line from a single way in the data-store. The limitation of aligned placement also simplifies replacement decisions because the candidates for replacement must also be aligned.

5.2. Operation

An access to the distill-cache is sent to both the LOC tag-store and the WOC tag-store. The access to the LOC tag-store is similar to a traditional cache and returns either a hit or a miss. The access to the WOC tag-store is checked first for a line hit. If any word of the requested line is present in the WOC, the tag-match in the WOC signals a line hit. The tag-match also gives an 8-bit field, *valid-bits*, corresponding to which words of the requested line are present in the WOC. If the requested word is present in the WOC, the word-match logic signals a word-hit. Thus, an access to distill-cache can result in one of four cases. First, a hit in the LOC *(LOC-Hit)*. Second, a line hit and word hit in the WOC *(WOC-Hit)*. Third, a line hit and word miss in the WOC *(Hole-Miss)*. Fourth, a line miss in both WOC and LOC *(Line-Miss)*. We discuss the operation of distill-cache for each of the four cases.

1. *LOC-Hit*. A hit in the LOC is serviced similar to a traditional cache. The matching data way is accessed and the line is sent to the L1 cache. The replacement and footprint information for the line is updated in LOC.

2. *WOC-Hit*. The way in the data store corresponding to the WOC hit is accessed and the data-line (64B) is obtained. The words of the matching line are rearranged to their position. For example, for a line X, if the first word (X_0) is stored in Td_2 and the last word (X_7) is stored in Td_3, then the words in the data-line are rearranged such that X_0 becomes the first word and X_7 becomes the last word in the line. The WOC sends the cache line to L1 along with a bit vector *(valid-bits)* that identifies the valid words.

3. *Hole-Miss*. All words for the requested line in WOC are invalidated. If any word of the requested line is dirty, it is read out before invalidation. A request for the line is sent to memory. The incoming line from memory is installed in LOC. The cache line is updated with dirty words (if any) and sent to the L1 cache.

4. *Line-Miss*. If the requested line is neither in the LOC nor in the WOC, a request for that line is sent to memory. A victim line is evicted from the LOC and the used words from this line are transferred to the WOC. Note that the transfer from LOC to WOC happens in parallel with the memory access.

5.3. Replacement

The LOC uses the LRU policy for replacement decisions. A line evicted from the LOC can require space for storing 1, 2, 4, or 8 words in the WOC. Therefore, the replacement policy in the WOC needs to support variable size replacements. Replacement in WOC can occur only at alignment boundary. Thus, for installing a line that has two words used, the only candidates for replacement start at Td_0, Td_2, Td_4 and Td_6. Furthermore, to reduce hole-misses, we evict all the words of a line if any of its word words are evicted from the WOC. To support this, we add a *head-bit* with each tag-entry in the WOC. The head-bit is set only for the first word of a line stored in the WOC. Thus, multiple words of a line start with a head bit set and end when another entry with the head-bit set or an invalid entry is encountered. Only WOC entries which are invalid or for which the head-bit is set are eligible for replacement. The replacement engine randomly[4] picks from all candidates eligible for replacement.

5.4. Threshold-Based Distillation

LDIS increases cache capacity by retaining only used words of the cache lines. If a cache line evicted from the LOC has a large number of used words, then installing the line in WOC can evict several useful lines from the WOC. For example, if a line has eight words used then installing it in WOC can reduce one cache miss. However, it may evict eight lines (each with only one word used) which could have saved eight cache misses. The number of unique lines in the WOC can be increased by not installing lines for which the number of used words exceed a certain threshold *K*. We call this threshold the *distillation threshold*.

The best distillation threshold depends on word usage of the application. A low value of K means that almost no line gets installed in the WOC, rendering the cache space devoted to WOC unusable. With a high value of K, threshold-based distillation would provide no benefit over normal LDIS. If the distillation threshold is set to the median number of words used by the application, then approximately half of the lines evicted from the LOC are installed in the WOC. We call this mechanism *median-threshold (MT) filtering*.

To compute the median number of words used in a line we use eight counters. The first counts the LOC evictions that had one word used, the second counts the LOC evictions that had two words used and so on. A separate counter, *eviction-sum*, counts the total lines evicted from the LOC. The median is calculated by adding the counts starting from the first counter to the last counter until one-half of the value of the eviction-sum is reached. We compute the value of median once every 4k LOC evictions.

[4]LRU policy for variable-sized replacement requires multiple LRU lists. Random selection is simpler than LRU and has similar performance.

5.5. Reverter Circuit

The implicit assumption of LDIS is that unused words are unlikely to be used in the less recent part of the LRU stack. This assumption is violated in some benchmarks which access a large number of unused words in the less recent part of the LRU stack. Furthermore, with the capacity boost provided by LDIS, cache lines remain in the cache longer, which increases the likelihood that unused words become used. An access to a line present in the WOC for which the requested word is not present results in a hole-miss. If most of the lines stored in the WOC cause a hole-miss without giving any cache hits, then the space devoted to the WOC is useless. In such cases, LDIS can hurt performance. To make LDIS applicable to a wide variety of workloads, we propose a low-overhead *reverter circuit*. The reverter circuit uses *dynamic set sampling* [12] to enable or disable LDIS depending on whether LDIS has fewer misses compared to a traditional cache.

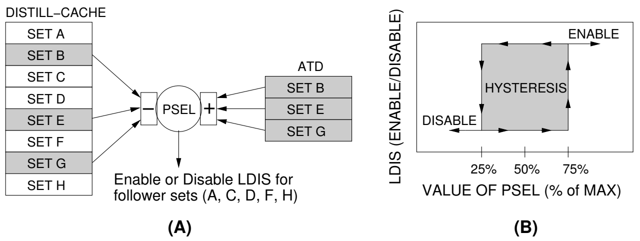

Figure 5. (A) Reverter circuit for a distill-cache with eight sets. (B) Hysteresis curve to enable/disable LDIS.

Figure 5 demonstrates the working of the reverter circuit for a distill-cache containing eight sets. Sets B, E, G are called *leader sets* because they decide if LDIS is enabled or disabled for the remaining sets. The remaining sets (A, C, D, F, and H) are called *follower sets*. LDIS is always enabled for the leader sets. The state of the traditional cache is tracked only for the leader sets using a separate structure called Auxiliary Tag Directory (ATD). The policy selector (PSEL) is an 8-bit saturating counter that tracks which of the two configurations—LDIS or traditional cache—has fewer misses. A miss in the leader sets of the distill-cache decrements PSEL, whereas, a miss in the ATD increments PSEL. The value of PSEL thus represents which of the two configurations—LDIS or traditional cache—incurs fewer misses. To avoid frequent enabling and disabling of LDIS, we incorporate hysteresis in decisions based on the value of PSEL. LDIS is disabled if the value of PSEL is less than 64 and enabled if the value of PSEL is more than 192. If the value of PSEL is between 64 and 192, the previous decision is retained. We use 32 leader sets for the baseline cache with 2048 sets. The reverter circuit enables/disables LDIS for the remaining 2016 $(2048 - 32 = 2016)$ sets. The reverter circuit requires a storage overhead of 1kB $(32\ sets \cdot 8\ ways/set \cdot 4B/ATD\text{-}entry)$.

6. Experimental Methodology

6.1. Configuration

We use a trace-driven cache simulator for all the experiments in the paper, except for the IPC results shown in Section 7.4. For the IPC experiments, we use an in-house execution-driven simulator that models the Alpha ISA. The parameters of our baseline configuration are shown in Table 1. The baseline L2 cache is 1MB 8-way set associative. The distill-cache has the same size as the baseline L2 cache. Unless stated otherwise, the distill-cache devotes six out of the eight ways to the LOC and the remaining two ways to the WOC. We do not enforce inclusion in our memory model.

Table 1. Baseline processor configuration

Inst. Cache	16kB, 64B line-size, 2-way with LRU replacement;
Branch Predictor	64k-entry gshare/64k-entry PAs hybrid minimum branch misprediction penalty is 15 cycles.
Exec. Engine	8-wide; reservation station contains 128 entries
Data Cache	16kB, 64B line-size, 2-way with LRU replacement,
Unified L2 Cache	1MB, 64B line-size, 8-way with LRU replacement, 15-cycle hit latency, 32-entry MSHR.
Memory	32 DRAM banks; 400-cycle access latency; bank conflicts modeled; maximum 32 outstanding requests;
Bus	16B-wide split-transaction bus at 4:1 frequency ratio.

6.2. Benchmarks

The SPEC CPU2000 benchmarks used in our study were compiled for the Alpha ISA. A representative sample of 250M instructions was obtained for each benchmark using a tool similar to Simpoint [10]. Benchmarks eon, perlbmk, and crafty were excluded from our study because of extremely low miss rate ($<$ 0.1 MPKI). We also excluded benchmarks for which the MPKI reduced by less than 10% when the cache size was quadrupled from 1MB to 4MB.[5] In addition to the SPEC benchmarks, we also used the health benchmark from the olden suite to show the effect of LDIS on pointer chasing workloads. We ran the health benchmark to completion. Table 2 shows the number of L2 misses per 1000 instructions (MPKI) and the percentage of misses that are compulsory misses for each benchmark.

Table 2. Benchmark summary (B = Billion)

Name	MPKI	Compulsory Misses	Name	MPKI	Compulsory Misses
art	38.3	0.5%	parser	1.6	20.3%
mcf	136	2.2%	sixtrack	0.4	20.6%
twolf	3.6	2.9%	apsi	0.3	22.8%
vpr	2.2	4.3%	swim	26.6	50.4%
ammp	2.8	5.1%	vortex	0.7	53.4%
galgel	4.7	5.9%	gcc	0.4	77.4%
bzip2	2.4	15.5%	wupwise	2.3	83.0%
facerec	4.8	18.0%	health	62	0.73%

[5]Key results for the 11 SPEC benchmarks excluded from our study are shown in Appendix A.

7. Results and Analysis

7.1. Change in MPKI with Line Distillation

Figure 6 shows the percentage reduction in MPKI over the baseline cache for three LDIS configurations. The first configuration, *LDIS-Base*, always transfers all used words from the evicted line of LOC to WOC. The second configuration, *LDIS-MT*, employs median-threshold filtering. Finally, *LDIS-MT-RC* employs both median-threshold filtering and reverter circuit. The bar labeled *avg* represents the reduction in the arithmetic mean MPKI over all the 16 benchmarks. As mcf has a high value for MPKI, the average MPKI reduction excluding mcf *(avgNomcf)* is also shown.

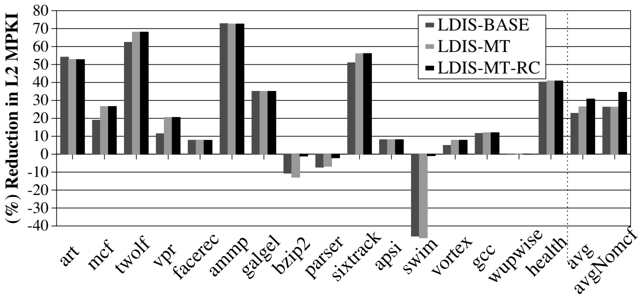

Figure 6. Reduction in MPKI with three LDIS configurations.

LDIS-Base reduces MPKI by more than 40% for art, twolf, ammp, sixtrack, and health. LDIS-Base reduces the average MPKI by 22.8%. LDIS-MT allows the LDIS mechanism to store more useful lines in the WOC, which further reduces the MPKI for mcf, twolf, vpr, and sixtrack.

Both LDIS-Base and LDIS-MT significantly increase MPKI for swim. For swim, 45% of the lines evicted from a 0.75MB cache (the size of LOC) have only one word used and the remaining lines have all the words used.[6] LDIS tries to increase the number of useful lines by retaining in WOC only the used words of a cache line. However, when the cache size is increased to 1.25MB, swim uses all the words in the cache line for 99% of the cache lines. Thus, retaining in WOC only the used words of a line evicted from LOC is futile because the unused words are referenced soon causing hole-misses in the WOC. As most of the lines stored in the WOC result in hole-misses without contributing to cache hits, the distill-cache performs worse than the baseline. In such cases, the reverter circuit in LDIS-MT-RC disables LDIS and allows the distill-cache to perform similar to the baseline cache. The reverter circuit limits the increase in misses for benchmarks bzip2, parser, and swim. LDIS-MT-RC reduces the average MPKI by 30.7% compared to the baseline, while never increasing misses by more than 2%. As LDIS-MT-RC is robust across all the benchmarks, we use LDIS-MT-RC as the default LDIS configuration in the rest of the paper.

[6]Appendix B shows the average number of words used in a cache line for each benchmark as the cache size is varied.

7.2. Analysis of Hit-Miss Distribution

An access to a traditional cache results in either a hit or a miss. An access to a distill-cache can result in one of the four cases: LOC-hit, WOC-hit, Hole-miss, or Line-miss. Figure 7 shows the breakdown of cache accesses in terms of hit-miss for both the baseline cache and the distill-cache.[7]

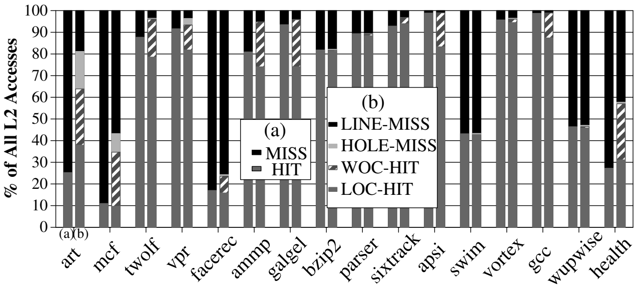

Figure 7. Breakdown of cache accesses in terms of hit-miss for the (a) baseline cache (b) distill-cache.

For mcf, 12% of the accesses hit in the baseline cache. The LOC has only 6 ways compared to the 8-ways in the baseline, which reduces the fraction of hits from 12% to 10%. However, the 2-ways devoted to WOC provide an additional 25% hits. Thus, for mcf, the distill-cache has three times the hits in the baseline cache. For twolf and ammp, the large fraction of WOC-hits helps the distill-cache to outperform the baseline cache.

For art and health, the number of LOC-hits is greater than the number of hits in the baseline cache. This happens because these two benchmarks have a dataset bigger than the cache size which causes thrashing with the LRU policy employed in the baseline cache. The distill-cache accommodates some part of the dataset in WOC which reduces the thrashing in LOC and hence the increased hits.

Although LDIS increases the hit rate of art from 25% to 63%, half of the misses for art are hole-misses. With LDIS a significant fraction of the dataset of art fits in the cache, which means that cache lines stay in the distill-cache significantly longer than in the baseline cache. This increases the likelihood of unused words being used which results in hole-misses. The problem of hole-misses is inherent when spatial filtering decisions are based on word usage patterns for a given cache size. Spatial filtering tries to increase the cache size which changes the word usage patterns. We also analyzed other spatial filtering scheme that uses a separate predictor [9] and found that art incurs a significant number of hole-misses even when a separate predictor is used.

[7]A distill-cache can incur extra cache accesses due to sector miss in the first-level cache. For all benchmarks, except art, distill-cache incurs less than 1% additional accesses compared to the baseline cache. For art, the distill-cache has 1.5% additional accesses than the baseline cache. As the number of L2 accesses for the baseline cache and distill-cache are similar, their hit-miss distributions are comparable.

7.3. Capacity Analysis

Figure 8 shows the reduction in MPKI with the distill-cache, a 1.5MB cache, and a 2MB cache over the baseline 1MB cache. For facerec, ammp, and sixtrack, distill-cache is comparable to increasing the cache size by 50%. For mcf and health, distill-cache provides more benefit than doubling the cache size.

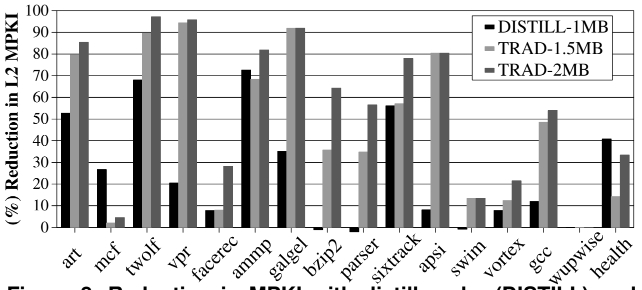

Figure 8. Reduction in MPKI with distill-cache (DISTILL) and traditional (TRAD) cache of bigger size.

7.4. Impact on System Performance

To measure the IPC improvements provided by LDIS, we use an execution-driven simulator. As the distill-cache contains more tag-store entries than the baseline cache, we add an extra cycle latency for the tag access of the distill-cache. The access latency for the data-store of the distill-cache remains the same as the baseline cache. For accesses that hit in the WOC, we add a latency of two cycles for rearranging the words in the cache line before it is sent to the first-level cache. Figure 9 shows the performance improvement measured in instructions per cycle (IPC) between the baseline processor and the same processor with the distill-cache. The bar labeled *gmean* is the geometric mean of the individual IPC improvements of each benchmark.

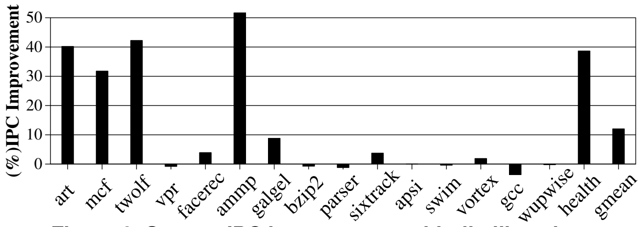

Figure 9. System IPC improvement with distill-cache.

The processor with the distill-cache outperforms the baseline by an average of 12%. The IPC of art, mcf, twolf, ammp, and health increases by more than 30%.

Instructions on the wrong path can cause the footprint to show a higher number of words used which reduces the benefit of LDIS.[8] Therefore, the reduction in misses for vpr and gcc does not translate into IPC improvements. As gcc is instruction-cache intensive, the extra cycle in cache access causes a minor IPC reduction.

[8]The effect of footprint update by wrong path instructions can be mitigated by delaying the footprint update until the instruction is confirmed to be on the correct path. We do not explore such optimizations in this paper.

7.5. Overheads of Distillation

In this section, we evaluate the storage, latency, and energy costs associated with the distill-cache. Storage is measured in terms of register bit equivalents. To model cache access latency and energy we used Cacti v3.2[18].

7.5.1. Storage The extra hardware for the distill-cache consists of the following: (1) Extra tags for the WOC, (2) Footprint bits in each tag-store entry in the LOC, (3) Footprint bits in each tag-store entry of the first-level data cache, (4) Counters for median threshold distillation, and (5) Extra tags in the ATD of the reverter circuit. The storage requirement for the distill-cache is calculated in Table 3. We assume a physical address space of 40 bits.

Table 3. Storage overhead of Line Distillation.

Size of each tag-entry in WOC (valid + dirty + head-bit + 23-bit tag + 3-bit word-id)	29 bits
Total number of tag-entries in WOC (2k sets * 2ways/set * 8entries/way)	32k
Overhead of tag-entries in WOC (29 bits/entry * 32k entries)	116kB
Total number of tag-entries in LOC (1MB/64B)	16k
Overhead of footprint bits in LOC (8bits/line * 16k lines)	16kB
Total number of lines in L1D Cache (16kB/64B)	256
Overhead of footprint bits in L1D Cache (256 lines * 8bits)	256B
Overhead for median threshold distillation (9 * 2B-counters)	18B
Size of each ATD entry	4B
Number of ATD entries (8 ATD-entries/set * 32 sets)	256
Overhead of reverter circuit (4B/ATD-entry * 256 entries)	1kB
Total storage overhead of distill-cache (116kB+16kB+256B+18B+1kB)	133 kB
Area of baseline L2 cache (64kB tags + 1MB data)	1088 kB
% increase in L2 area with distill-cache (133kB/1088kB)	12.2%

The distill-cache incurs a total storage overhead of 12.2% of the area of the baseline cache. However, this storage overhead is dependent on the line-size of the cache. For a line-size of 128B, the storage overhead reduces to 7% and for a line-size of 256B, the storage overhead reduces to 4%.

7.5.2. Latency The distill-cache incurs a latency penalty due to the additional tag-store entries in the WOC. The access latency is also extended by a mux delay to select the information about the matching way between the LOC tag-store and the WOC tag-store. For 65nm technology, Cacti estimates the additional delay to be 0.14ns. In our IPC experiments, we assume this additional delay increases the tag access of the distill-cache by one cycle.

7.5.3. Energy We use Cacti to measure the energy consumed in the extra tags of the WOC tag-store. For each access in the distill-cache, the extra tags of the WOC consume 3.76 nJ per access in addition to the 3.06 nJ consumed in the tag-store of the LOC. The access to the data-store in a distill-cache is similar to a traditional cache. Therefore, for accesses that hit in the distill-cache, the energy consumed in the data portion of the distill-cache remains similar to the baseline cache.

8. LDIS and Compression: Differences, Interactions, and Optimizations

This paper proposes LDIS for increasing cache capacity. An orthogonal approach for increasing cache capacity is cache compression [19][1][6]. Although the goal of compression and LDIS is identical, they both try to exploit fundamentally different inefficiencies in cache design. Compression exploits the redundancy in the information stored in a cache line, whereas, LDIS exploits the words that remain unused in a cache line. In this section, we analyze the interaction between compression and LDIS, and propose a mechanism that combines both schemes.

8.1. Compressibility of Cache Lines

To analyze the compressibility of cache lines, we use the following[9] encoding that operates on a 32-bit granularity:

Table 4. Encoding scheme for 32-bit data.

Code	Value of the 32 bit data
00	0
01	1
10	bits[31:16] are 0, only bits[15:0] stored.
11	Incompressible, all bits[31:0] stored.

We sample the contents of the baseline cache once every 10M instructions and invoke the compression scheme for all the valid lines in the cache. Based on the size of the compressed line, each cache line is classified into four categories. The first three categories contain cache lines that can be stored in at least *one-eighth*, *one-fourth*, and *one-half* of their original size, respectively. The fourth category, *full* size, contains all the remaining lines. Figure 10(a) shows the distribution of cache lines for the baseline.

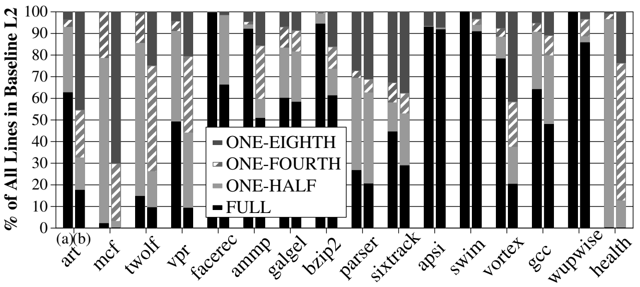

Figure 10. Compressibility of lines when (a) all words are considered for compression (b) only used words are compressed

All lines that do not belong to the *full* category are called *compressible* lines. For 10 out of the 16 benchmarks, less than half the lines are compressible. Even among compressible lines, most of the lines are in the *one-half* category. Only mcf, parser, and sixtrack have more than 20% lines in either the *one-fourth* category or the *one-eighth* category. Due to limited compressibility, some of the proposals for cache compression try to compress a line to only half its original size [19], which severely restricts the potential gains from compression. However, not all words in a cache line are useful. If only used words are compressed then the compression engine might be able to reduce the lines to a much smaller size. Figure 10(b) shows the breakdown of the cache lines into the four categories when only used words are considered for compression. For most benchmarks, a majority of the lines are compressible. For art, mcf, twolf, vpr, vortex, and health, more than half the lines are in either *one-fourth* category or *one-eighth* category indicating significant capacity benefits from such a compression scheme.

[9]We also studied more complex compression schemes [2] but the compression ratio and the reduction in MPKI were similar.

8.2. Footprint-Aware Compression

The information about which words are used is available in the footprint field of the LOC. A compression scheme that uses the footprint information to compress only used words is called *Footprint-Aware Compression (FAC)*. FAC can easily be implemented in conjunction with the distill-cache. When the cache line is evicted from LOC, the used words are compressed and stored in the WOC. The tag-entries in WOC are modified to support both compressed and uncompressed lines. Figure 11 shows the percentage reduction in MPKI for four cache configurations. First, LDIS-3xTags is a distill-cache in which two ways out of the eight ways are devoted to WOC. Second, LDIS-4xTags is a distill-cache in which three out of the eight ways are devoted to WOC. Third, CMPR-4xTags is a traditional cache that implements compression and has 4 times as many tag-entries as cache lines. Finally, FAC-4xTags is a distill-cache which implements FAC and devotes three out of the eight ways to WOC.

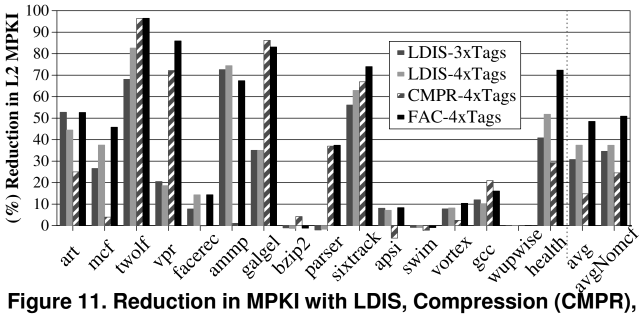

Figure 11. Reduction in MPKI with LDIS, Compression (CMPR), and Footprint-Aware Compression (FAC)

For some benchmarks LDIS reduces more misses than CMPR and for some CMPR reduces more misses than LDIS. FAC reduces more misses than either scheme for benchmarks mcf, vpr, sixtrack, and health. CMPR reduces more misses than FAC on galgel and gcc because we use perfect LRU replacement for CMPR and the practical size-based random replacement (discussed in Section 5.3) for FAC. On average, FAC reduces the average MPKI by 50% indicating that LDIS and CMPR interact positively.

9. Related Work

Sectored caches reduce the overhead of tag directories. Instead of a cache line, only a portion of a cache line, i.e. a sector, is fetched on a miss. Although a sectored cache utilizes bandwidth efficiently, it utilizes cache space inefficiently as some sectors remain invalid. A sectored cache typically has a higher miss rate than a traditional cache [14].

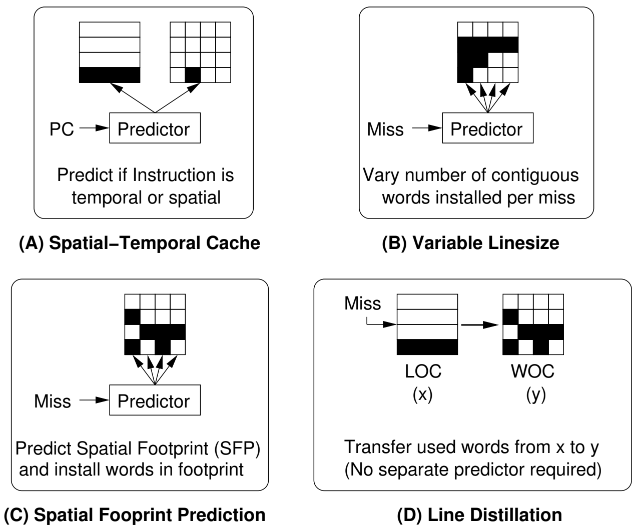
PC
Predictor
Predict if Instruction is temporal or spatial
(A) Spatial–Temporal Cache
Miss
Predictor
Vary number of contiguous words installed per miss
(B) Variable Linesize
Miss
Predictor
Predict Spatial Footprint (SFP) and install words in footprint
(C) Spatial Fooprint Prediction
Miss
LOC (x)
WOC (y)
Transfer used words from x to y (No separate predictor required)
(D) Line Distillation

Figure 12. Organizations for Spatial Filtering

Several studies have looked at improving cache performance by using a predictor to detect spatial locality. Figure 12 classifies some of the proposals for spatial filtering. The design proposed by Gonzalez et al. [5] consists of a spatial cache, a temporal cache, and a predictor which predicts if the miss-causing instruction is spatial or temporal. Depending on the prediction, the fetched line is placed in either the spatial cache or the temporal cache. The proposed scheme works well for numerical codes but it does not scale well to integer codes. Furthermore, the spatial-temporal cache physically partitions the cache such that a line can only be allocated to one of the two caches. Whereas, with the distill-cache all lines can be placed in either of the two structures which provides more efficient use of cache space.

Johnson [7] propose a mechanism to vary the line-size by predicting spatial locality of access over a region of memory. Similarly, Viedenbaum et al. [16] propose an algorithm to gradually change the line-size based on reuse information of adjacent cache lines. These techniques exploit spatial locality at a coarser granularity. For example, if only the first and the last word of a cache line are used, then these techniques will also fetch all the words between the first and the last word. Furthermore, both techniques use a cache organized at a word granularity which results in a tag overhead of as much as half the cache size.

Only useful words of a cache line can be installed in the cache, if the information about which words in the line are useful is available. Kumar et al. [9] use a spatial footprint predictor (SFP) to predict the useful words in a cache line. The prediction of the SFP determines which words in the cache line are installed in the cache. They used a decoupled sectored cache [14] to limit the tag overhead. Figure 13 compares the reduction in MPKI provided by SFP and LDIS. We simulated the same number of tag-entries in the decoupled sector cache (used in SFP) as there are in the distill-cache (used in LDIS). However, SFP also incurs additional overhead of the footprint predictor. We show SFP results for a predictor that has 16k entries (64kB) and 64k entries (256kB). We also added the reverter circuit to the SFP to limit the increase in MPKI for some benchmarks.

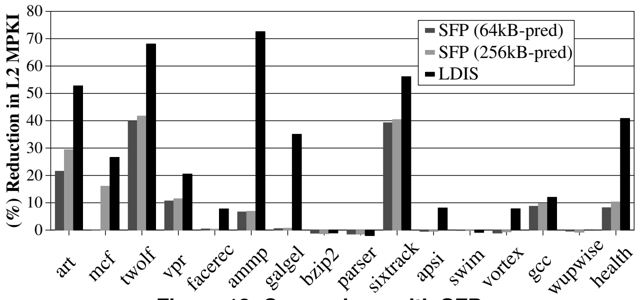

Figure 13. Comparison with SFP

Figure 13 shows that SFP reduces the misses compared to the baseline, however, this reduction is significantly lower than the reduction from LDIS. SFP makes prediction at install time which means that an incorrect prediction results in a miss which would be a hit in the traditional cache. On the other hand, LDIS performs filtering only at eviction time, which means that an access to a word that is not present will result in a miss which may also have been a miss in the traditional cache. Also, the SFP stores spatial footprint for a particular sized cache. An attempt to store more lines can change the footprint information for a cache line, causing mispredictions in SFP. Furthermore, the decoupled sectored cache restricts the placement of words. Thus, if two lines require only the first word in the line then they cannot reside together in the same data line of a decoupled sector cache. Whereas, these words can easily be stored in any two entries of WOC in a distill-cache.

SFP also requires huge overhead of the prediction table. This overhead can be reduced by learning the spatial patterns rather than memorizing the spatial footprint of each line. Chen et al. [3] and Somogyi et al. [15] describe mechanisms to learn the spatial patterns and use this information for prefetching. These schemes do prefetching at a cache line granularity so LDIS can be used with these schemes for removing unused words in both demand and prefetched lines. Pujara et al. [11] use spatial footprint prediction for reducing cache leakage power, however, they do not describe any mechanism to improve cache performance.

10. Concluding Remarks

A significant number of words in the cache line remain unused because of the variation in spatial locality. Unused words occupy cache space without contributing to cache hits. The usage of words in a cache line stabilizes as the line traverses through the LRU stack. This information can be used to filter the unused words which can allow the cache to store more cache lines. Based on this observation, this paper makes the following contributions:

1. We propose *Line Distillation (LDIS)* to filter unused words in the cache line. Unlike previous proposals, LDIS uses only in-cache information for spatial filtering and requires no separate prediction structure.

2. We propose the *Distill Cache* organization to increase cache capacity by leveraging LDIS. The distill-cache has extra tags for a subset of the cache to provide flexible placement for words that remain after distillation.

3. We propose *median-threshold filtering* and the *reverter circuit* to improve the performance of LDIS and to make it applicable to a wide variety of workloads. Our evaluation shows that LDIS reduces average misses for a 1MB 8-way L2 cache by 30% and improves average IPC by 12%.

4. We show that compression and LDIS interact positively and can be combined for a greater capacity benefit than either scheme standalone. The proposed combination, *footprint-aware compression*, reduces average misses for the 1MB 8-way cache by 50%.

Acknowledgments

Initial part of this work was done while the first author was an intern at the IBM T.J. Watson research center during Summer 2004. Thomas Puzak and David Thompson contributed substantially to that work [13] .

Thanks to Aamer Jaleel, Lee Baugh, Jose Joao, Veynu Narasimhan, and Chris Wilkerson for their comments and feedback. This work was supported by gifts from IBM, Intel, and the Cockrell Foundation. Moinuddin Qureshi was supported by an IBM PhD fellowship during this work.

References

[1] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *ISCA-31*, page 212, 2004.

[2] A. R. Alameldeen and D. A. Wood. Frequent pattern compression: A significance-based compression scheme for L2 caches. Technical Report 1500, CS Dept., University of Wisconsin - Madison, 2004.

[3] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *HPCA-10*, 2004.

[4] Digital Equipment Corporation, Hudson, MA. *Digital Semiconductor 21164 Alpha Microprocessor Product Brief*, Mar. 1997. Technical Document EC-QP97D-TE.

[5] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *ICS-9*, 1995.

[6] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *HPCA-11*, 2005.

[7] T. L. Johnson. *Run-time adaptive cache management*. PhD thesis, University of Illinois, Urbana, IL, May 1998.

[8] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Second Annual Workshop on Memory Performance Issues*, 2002.

[9] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ISCA-25*, pages 357–368, 1998.

[10] E. Perelman et al. Using SimPoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[11] P. Pujara and A. Aggarwal. Increasing the cache efficiency by eliminating noise. In *HPCA-12*, 2006.

[12] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for MLP-aware cache replacement. In *ISCA-33*, 2006.

[13] M. K. Qureshi, D. Thompson, T. R. Puzak, and Y. N. Patt. Line distillation: A mechanism to improve cache utilization. Technical report, TR-HPS-2006-002. University of Texas, Austin, Feb. 2006.

[14] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *ISCA-21*, pages 384–393, 1994.

[15] S. Somogyi et al. Spatial memory streaming. In *ISCA-33*, 2006.

[16] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *ICS-13*, 1999.

[17] D. Weiss, J. J. Wuu, and V. Chin. The on-chip 3-mb subarray-based third-level cache on an itanium microprocessor. In *IEEE journal of solid state circuits*, pages 1523–1529, Nov. 2002.

[18] S. J. E. Wilton et al. Cacti: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31:677–688, May 1996.

[19] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *MICRO-33*, pages 258–265, 2000.

Appendix A: Cache-Insensitive Benchmarks

If increasing the cache size does not reduce misses, then LDIS cannot reduce misses either. The MPKI of gzip (1.45), fma3d (4.61), perlbmk (0.04), and eon (0.01) remain unchanged for the four configurations shown in Table 5.

Table 5. MPKI for cache configurations (Trad=Traditional)

Config.	equake	lucas	mgrid	applu	mesa	crafty	gap
Trad 1MB	18.42	16.17	7.73	13.75	0.62	0.09	1.65
LDIS 1MB	18.40	16.16	7.74	13.75	0.62	0.10	1.66
Trad 2MB	18.02	16.16	7.57	13.50	0.61	0.08	1.65
Trad 4MB	16.88	16.16	7.13	12.89	0.59	0.08	1.65

Appendix B: Cache Size vs. Words Used

Table 6. Average number of words used in the cache line

Size	art	mcf	twolf	vpr	fac.	ammp	galgel	bzip2
0.75MB	1.80	1.82	3.20	3.10	6.82	2.58	7.75	3.76
1.00MB	1.81	1.83	3.24	3.71	7.01	2.40	7.60	4.13
1.25MB	2.39	1.85	3.26	4.59	7.08	2.62	7.39	4.46
1.50MB	3.27	1.86	3.33	5.77	7.14	2.95	7.73	4.85
2.00MB	3.63	1.91	3.83	6.09	7.35	3.16	7.73	6.13

Size	parser	sixtrk	apsi	swim	vrtx	gcc	wup.	health
0.75MB	6.01	4.34	7.88	4.71	3.03	6.13	7.01	2.44
1.00MB	6.42	4.34	7.80	6.91	3.04	6.38	7.01	2.44
1.25MB	6.84	4.37	7.85	7.98	3.09	6.69	7.01	2.44
1.50MB	7.27	4.39	7.94	7.98	3.15	6.88	7.01	2.44
2.00MB	7.59	4.38	7.94	7.98	3.25	7.04	7.01	2.44