Increasing Cache Capacity via Critical-words-Only Cache

Cheng-Chieh Huang
Institute of Computer Systems Architecture
University of Edinburgh, Edinburgh, UK
cheng-chieh.huang@ed.ac.uk

Vijay Nagarajan
Institute of Computer Systems Architecture
University of Edinburgh, Edinburgh, UK
vijay.nagarajan@ed.ac.uk

***Abstract*—Current processors have multiple levels of caches. Choosing the right cache size for each level is critical for performance. The first-level cache (L1) is typically small, in order to match the speed of the processor. The lower level caches, on the other hand, are typically large, in order to reduce capacity misses. However, situations may arise in which the size of a lower level cache cannot be increased beyond a point – for example, recent Intel multi-core processors (including Nehalem and Sandy Bridge) have only 256 kB private L2 caches per core – which adversely affects the performance of benchmarks which have large working set sizes.**

In this paper, we propose a novel cache design known as the critical-words-only cache (co-cache) for increasing the effective cache capacity. Our approach involves rethinking the notion of cache blocks; instead of storing all the words that belong to a cache block, we only store the critical words, where the critical words are the words that are generally accessed before the others. Our experiments show that with our design a 256 kB L2 performs as well as a 512 kB conventional L2 cache on average.

I. Introduction

Current processors have multiple levels of caches. Choosing the right cache size for each level is critical for performance. The first-level cache (L1) is typically small, in order to match the speed of the processor. The lower level caches, on the other hand, are typically large, in order to reduce capacity misses.

How well a lower level cache is able to reduce capacity misses depends on how well the working set fits into the cache. Ideally, the cache should be large enough to fit the working set, but situations may arise in which the size of the lower level cache is significantly lesser than the working set. For instance, several modern multi-core processors (including Intel Nehalem and Sandy Bridge) choose to support L2 caches that are private to each core; although it would be desirable to have a large private L2, this might not be possible, as increasing the size of per core L2 has a multiplicative effect on the overall area. Indeed, the above processors support only 256 kB of L2 cache per core.

Is a 256 kB L2 large enough to reduce capacity misses? Figure 1 shows the miss-rates incurred by SPEC benchmarks as the size of the L2 cache is varied from 128 kB through 4 MB. As we can see, with a 256 kB L2 cache, the average miss-rate is greater than 50%. In fact, for programs such as *galgel* and *omnetpp* which have a miss-rate of over 90%, we found that bypassing the L2 cache and directly accessing the LLC proved to be more performance-efficient than accessing the L2 cache. This is because accessing the L2 in such situations ceases to provide any performance benefit, since the L2 miss rate is high; to make matters worse, it also adds the latency of a wasteful L2 access to the critical path. As the size of L2 is increased, however, we observe that the miss-rate significantly decreases in some benchmarks.

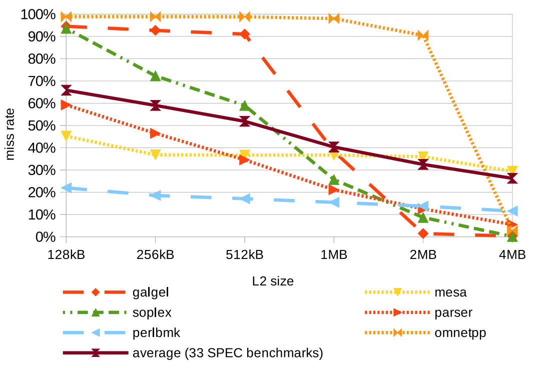
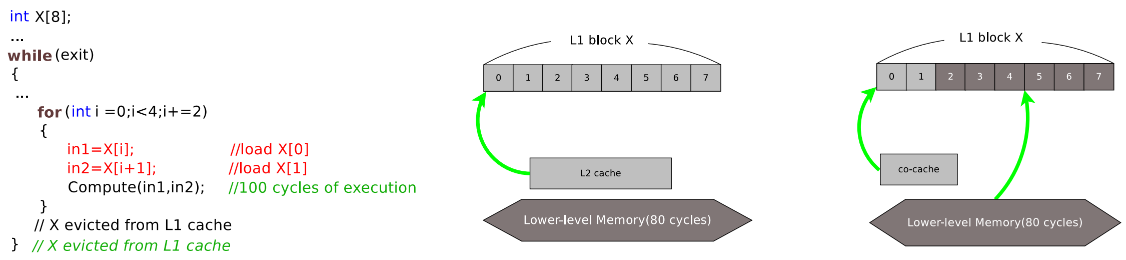


Fig. 1. Miss rate versus L2 cache size: as the size of the L2 cache (8-way) is increased from 128 kB through 4 MB, the local miss-rate significantly decreases. For this experiment 32 kB (4-way) L1 cache was used.

From the above example it is clear that a lower level cache that is too small compared to the working-set size could potentially hurt performance. In such situations, is there a way in which the cache can be used more effectively? In this paper, we propose a novel cache design for increasing the effective cache capacity. Although our design is applicable for any lower level cache, i.e. any cache other than L1, in this work we restrict ourselves to L2 caches.

We observe that not all words belonging to an L2 cache block are accessed around the same time – a subset of the words, in fact, are consistently accessed sooner than the others. We refer to the former as *critical words* and the latter as *non-critical words*. Our key idea stems from the realization that, if the time interval between the critical word accesses and the non-critical word accesses is at least as high as the latency of accessing the lower level cache (or memory), then an L2 that caches *only the critical words* of each block could potentially perform as well as a conventional L2 that caches the full block. This is because, whenever any of the critical words in L2 are

This work is supported by UK India Education and Research Iniative Thematic Partnership Award UKUTP201100256 and the Centre for Numerical Algorithms and Intelligent Software, funded by EPSRC grant EP/G036136/1 and the Scottish Funding Council to the University of Edinburgh.



```
int X[8];
...
while (exit)
{
 ...
    for (int i =0;i<4;i+=2)
    {
        in1=X[i];              //load X[0]
        in2=X[i+1];            //load X[1]
        Compute(in1,in2);      //100 cycles of execution
    }
    // X evicted from L1 cache
}  // X evicted from L1 cache
```

Fig. 2. Example to illustrate the idea of critical words

accessed, the L2 could additionally request the full cache block from the lower level and have it sent to the L1 – in time before any of the non-critical words are accessed by the processor. We call such a cache design which stores only the critical words for every block, the *Critical-words-Only cache* (co-cache), and we denote the number of critical words per block as the *depth* of the co-cache. In the ideal case, a co-cache of depth 1, will perform as well as a conventional cache with full-sized blocks.

An Example. We illustrate our idea with a simple example shown in Figure 2. For this example, let us assume a two-level cache organization with a block size of 8 words. Furthermore, let us assume that the cache block containing array X is cached in L2 throughout, but is only cached in the L1 when executing the inner *for* loop (as it is evicted from the L1 soon after). Consequently, every time $X[0]$ is loaded within the inner *for* loop, it causes an L1 miss, which in turn causes block X to be transferred from L2 to L1. Then, $X[0]$ and $X[1]$ are loaded and the loaded values are used to perform some computation which takes 100 cycles. Subsequently, $X[2]$ and $X[3]$ are loaded (and so on).

In this example, $X[0]$ and $X[1]$ constitute the *critical words*, as these are accessed sooner than the other words in the block. This example sheds light on the reason behind this observation of criticality: typically not all words within a block are accessed together; often one or two words are loaded and the loaded values are used to perform some computation before loading the subsequent values.

It is also important to note the above pattern repeats itself across each iteration of the outer *while* loop. The example also sheds light on the reason for this repeatability: often, like in this example, the same piece of code is responsible for causing misses to a given block in the L1 cache. Consequently, critical words for a particular block are often accessed in regular patterns.

Figure 2 (right) illustrates how an L2 that is organized as a co-cache of depth 2 achieves the same performance as that of the conventional L2 with 8 words per block. When $X[0]$ is accessed in the co-cache, it triggers a request to fetch full block X from memory. By the time the processor requests $X[3]$ after performing the computation (which takes 100 cycles), the block would have been fetched from the memory, as the memory latency is 80 cycles.

Contributions and Paper Organization. In this paper, we consider the problem of how best to use a lower-level cache whose size is much smaller than the working-set size. Our contributions are as follows:

- We observe that a subset of words from a cache block – the critical words – are consistently accessed sooner than others. We demonstrate this with the help of an empirical study (§II), in which we also demonstrate that the above critical words are predictable.
- We exploit this observation by proposing a novel cache design called co-cache (§III), which is a lower-level cache comprising only the critical words of every cache block, and thus uses the available cache size more effectively.
- We show how the L2 can be engineered as a co-cache (§III-B). For predicting critical words, we have a simple *critical words predictor* (§III-A), in which we add additional tag bits to the L1 cache (<500 bytes overhead) for remembering the order in which words are accessed; thus our design requires no complex prediction table.
- Whereas the co-cache is effective in situations in which the size of the L2 is significantly smaller than the working-set size, a conventional cache might perform better in situations where the working-set fits in the cache. For this reason, we propose the adaptive co-cache (aco-cache), a scheme for reconfiguring back to a conventional cache if the working set is determined to fit within L2.
- We describe our evaluation methodology in §IV. Among other things, we discuss how we model the co-cache access latency (§IV) and the space overhead of the co-cache (§III-D).
- Our experiments (§V) with SPEC2000 and SPEC2006 benchmarks show that a 256 kB L2 used as a co-cache (aco-cache) of depth 2 can achieve up to 36.8% (36.8%) speedup and on average 5.3% (6.1%) speedup compared to a conventional 256 kB L2 cache. As for multi-core workloads, the co-cache (aco-cache) in a 4-core system shows up to 29.3% (29.3%) speedup and on average 7.3% (8.1%) speedup across 32 randomly generated workload groups.

II. Motivation: Empirical Study

Clearly, the effectiveness of our idea hinges on whether or not each L2 cache block can be split into critical and non-critical components. In addition to this, we should be able to predict the critical words for each cache block – only then would we be able to know what to cache in the proposed co-cache.

A. *Critical Words vs Non-critical Words*

We conducted an experiment to ascertain whether, for each L2 cache block, a subset of words (the critical words) are consistently accessed before the others (as opposed to all words being accessed around the same time). To this end, we performed a cache simulation study in which we simulated a 4-way L1 of size 32 kB of 64 bytes block size across the SPEC benchmarks. Since the goal of the study is to better understand

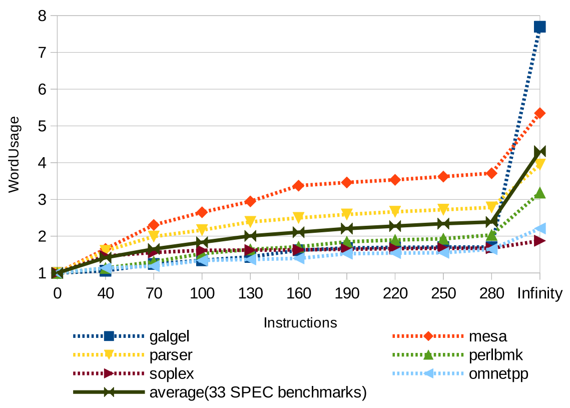


Fig. 3. The times (in terms of dynamic instruction count) at which the words within a block are first accessed; the time at which the first word is accessed is taken to be 0.

the properties of L2 accesses, simulating a finite L2 cache will only limit coverage; so we simulated an infinite L2 of 64 bytes block size. Whenever there is an L1 miss, the corresponding cache block is brought into the L1 from the L2, and the word that caused the miss is accessed. We start time at this instant, and measure the times at which the other words belonging to the same cache block are subsequently accessed. We use the dynamic instruction count as the measure of time.

As we can see from the result in Figure 3, on average, the second word is accessed after 100 dynamic instructions, whereas the third word is accessed after about 500 dynamic instructions (not shown in the figure). Assuming a CPI of 1, this would mean than the third word is accessed around 500 cycles after the first word. It is worth noting that this is likely greater than the memory latency, and definitely greater than the L3 latency if there is an L3 cache. Thus, we can infer from this study that generally the first 2 words are accessed much earlier than the rest of the words; in other words, there is very good evidence for the first 2 words being the critical words. The reason for this behavior can be explained by the fact that applications typically read a couple of words, after which they would likely use the above words that are read to perform some computation, before reading the next set of memory words, and so on.

B. *Critical Words Predictability*

To check if the critical words are predictable, we remember for every memory block, the word address that caused the L1 miss (the 1st critical word) and also the word subsequently accessed (the 2nd critical word). When the same memory block results in an L1 miss and is brought back into the L1 again, we compare the recorded addresses with the actual addresses of the 1st and 2nd critical words.

In our study, the average predictability of the 1st critical word is 65.4% across all benchmarks. For some benchmarks, such as *galgel* and *omnetpp* the critical words are extremely predictable with close to 100% predictability. On the other hand, for benchmarks such as *mesa* and *parser* the predictabilities are in the order of 60.0% and 49.4% respectively. It is important to note, however, that even for benchmarks with relatively poor predictabilities, they are still significantly better than the expected predictability of 12.5% if the critical words were uniformly distributed. We also measure the percentage of times *both* critical words are predicted correctly – as we can see this is about 62.5% across all benchmarks. From this we can infer that, in general, when the first critical word is predicted correctly, the prediction for the second critical word is also correct. The reason for this behavior can be explained by the fact that generally the same piece of code is responsible for causing L1 misses to a given memory block. Consequently, the critical words are accessed in a regular pattern and hence predictable.

C. *Useful Words vs Critical Words*

Despite the presence of spatial locality, not always do all the words belonging to a cache line end up being used; a number of techniques [1]–[5] leverage this observation to improve cache performance. For instance, Line Distillation [3] attempts to discard such unused words in a cache line to improve cache capacity. While our idea of exploiting critical words is closely related to the idea of exploiting useful words, there is one crucial difference. Techniques that exploit useful words benefit from cache lines in which not all the words in the cache line are used, i.e. the more unused words the better. In contrast, our idea exploits the factor of time; for example, even if all the words from a cache line are used, we can still benefit as long as some of the words in this cache line are accessed sooner than others. We conduct an experiment to find out, on average, how many words are accessed in a block before its eviction (i.e. word usage of the block). As shown in Figure 4, the average word usage is 4.3 words per block across all benchmarks, which amounts to 53.8% of the original block size. Intuitively, this ratio corresponds to the maximum benefit that is possible by exploiting useful words (cache that is about half the size of the original cache can perform as well as the original cache). On the other hand, the benefit that can be derived by exploiting critical words is dependent on the latency of accessing the lower level: lesser the latency, lesser the number of critical words (which is the word usage for critical words based techniques). Figure 4 shows the word usage for various latencies. As we can see, even with as high a latency as 160 cycles, the number of critical words (and thus the average word usage) is 2.1 words, which amounts to 26.25% of the original size (cache that is quarter the size of the original cache can perform as well as the original cache). From the above study, it is clear that exploiting critical words has potential for greater savings in comparison to useful words.

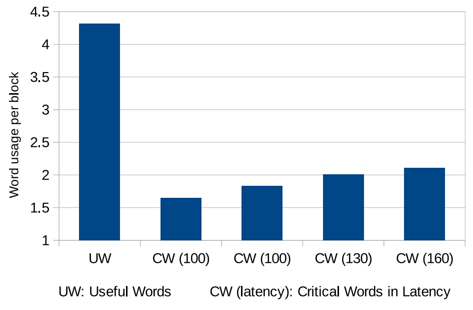


Fig. 4. Useful words filtering versus critical words filtering

III. Hardware

A critical component of our design is the *critical words predictor*, which we introduce first. Next, we show how we engineer the L2 cache as a *critical-words-only cache (co-cache)*. A conventional cache might perform better than the co-cache if the cache size is large enough to fit the working-set; for this reason we introduce a reconfiguration scheme called *adaptive co-cache* (aco-cache), for dynamically choosing between the co-cache or conventional cache based on the workload's miss-rate. Finally, we discuss the area overhead of our design.

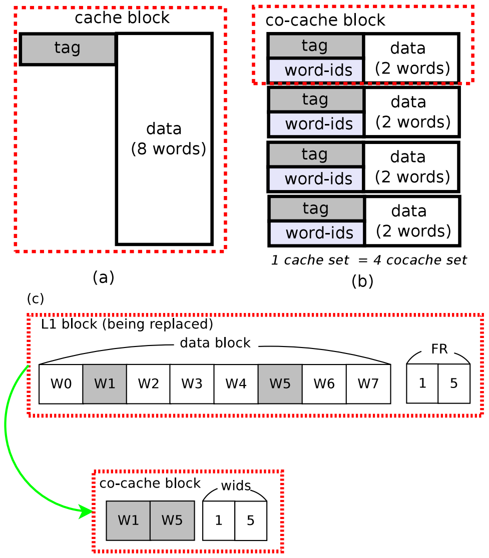


Fig. 5. (a) cache block in a conventional cache (b) 4 blocks in a co-cache (c)Block placement – FR: Footprint registers (§ III-A)

A. Critical Words Predictor

We use a simple predictor, in which we simply remember the order in which the words of a cache block are accessed in the L1 and use it as our prediction – our approach is similar to the predictor used in [3]. In order to remember the identities of the critical words, we add additional tag bits – the *footprint registers* – to each L1 cache block. For example, for a 32 kB cache size, with a 64B block size, and a depth of 2 (which means 2 critical words per block), we require just two 3-bit footprint registers per cache block to remember the critical words – which amounts to only 384 bytes overhead. Thus, our predictor design does not require any prediction table, and its area overhead is relatively minimal.

B. Co-cache Structure and Implementation

The co-cache is a cache organization for caching only the critical words for every cache block. Its organization is similar to a traditional cache, except in the following respects. For the purpose of this discussion let us assume all caches use write-back policy:

- **Checking for hit/miss**. Since the co-cache caches only the critical words from each cache block, we need to be able to identify what words are currently cached. For this purpose, each block in the co-cache is associated with additional *word-ids* which stores the identities of the words that are currently cached, as shown in Figure 5 (b). Note that the word-ids would also need to be compared as part of tag checking. In other words, adding the word-ids has the same effect of increasing the tag size of each cache block. We model the delay of this increased tag size in our experiments (§IV) and find that this is negligible.
- **Accessing the co-cache**. It is worth noting that even if the requested word is found in the co-cache, the request has to be forwarded to the lower level in order to bring the remaining (non-critical) words to L1. Consequently, upon an L1 miss, both the L2 (functioning as a co-cache) and the lower-level are accessed. If the requested word is found in the co-cache, the requested word and the other critical words are sent to the L1; later, the rest of the words would arrive from the lower-level. On the other hand, if the requested word in not found in the L2 co-cache, the processor will have to wait until the whole block arrives from the lower-level. Thus, accessing the co-cache proceeds as detailed in Table I.
- **Updating the co-cache (block placement)**. When *a cache block is replaced from the L1*, we need to decide what words from the block need to be cached in the L2 co-cache. We use the output of our critical words predictor (*footprint registers*) and only cache those words that are marked as critical (as shown in Figure 5(c)). It is worth noting that all blocks in the L1 should have at least one access that caused the block to be fetched, so footprint registers cannot be empty.
- **Mis-prediction handling**. It is worth noting that since the processor forwards the request to lower level memory for the full block data, this would automatically handle the case in which the critical words are mis-predicted.
- **Handling write backs**. When the L1 writes back dirty data to the L2 co-cache, since the co-cache cannot hold the full block, the dirty block will also have to be written to the L3.
- **Cache coherence**. In contrast to conventional cache coherence (with local L1/L2 and shared L3), ensuring cache coherence with a L2 co-cache involves one minor change. When a block with read-write permissions is evicted from the L1, since the co-cache cannot hold the full block, the co-cache will in turn have to write the block back to the L3. However, in doing so, the block in L3 will have to be maintained in read-only state as the co-cache continues to hold part of the block.

TABLE I. Accessing the co-cache

1	If there is a miss in L1, the fetching request is sent to both co-cache (for partial block containing the critical words) and the L3 (for the whole block).
2	If the co-cache access is a hit, the partial block in the co-cache is brought back to L1 and satisfies processor's request. If the co-cache access is a miss, the processor waits for the whole block to be brought from L3.
3	When the requested (whole) block returns from L3, it will be placed in L1, but not in the co-cache [1]. If the co-cache access had resulted in a hit (in step 2), the whole block will replace the previously present partial block in L1.

C. Adaptive Co-cache

A co-cache targets situations in which the size of the L2 is significantly smaller than the working set size. However, when the working-set size does fit in the cache, a conventional cache might perform better. Thus, it would be beneficial to have an scheme for switching back to a conventional cache

[1]The co-cache is *exclusive* with respect to the L1

if the working-set is small enough to fit the L2. We call this *adaptive co-cache (aco-cache).*

Obtaining miss-rates. This design requires that we obtain the miss-rate of a conventional cache while we are operating in co-cache mode. In order to achieve this, we make use of shadow tags [6], [7] to monitor the cache miss-rate. In contrast to a conventional cache, the shadow tags only perform tag operations – i.e they only keep the information of blocks but not their data.

Scheme. In our proposed aco-cache scheme, we start in co-cache mode, but keep monitoring the miss-rate of the conventional cache with the shadow tags. At relevant checkpoints, we compare the miss-rate with a pre-defined threshold; if the measured miss-rate is lower than a threshold, we reconfigure the cache back to conventional cache mode. With regard to the checkpoints, these can be placed either by the software (operating system or compiler) or the hardware. In this paper, we assume the latter – specifically, we assume *a hardware monitoring unit* [7] which simply checks the miss rate every fixed number of instructions.

Reconfiguration: hardware. In principle, performing this reconfiguration is as simple as coalescing the (partial) blocks of the co-cache and turning off [8] the additional tags and word-ids; for instance, 4 blocks of the co-cache can be coalesced into 1 conventional block in the example shown in Figure 5 (b). There are two ways to do this: set-based reconfiguration [8], [9] or way-based reconfiguration [10]. We use the set-based reconfiguration (that configures 4 co-cache sets to 1 conventional cache set), as this requires *no change in associativity* (we wanted to avoid a design in which the co-cache's associativity is higher than the conventional cache).

Reconfiguration: single vs multiple transitions. Our reconfiguration scheme allows only a single transition: from the co-cache to the conventional cache (if the working set fits L2). Another alternative is to allow multiple back-and-forth transitions between the co-cache and the conventional cache depending on the program phase behavior. However, we chose the former, primarily because of the cost associated with a single reconfiguration transition. Reconfiguration involves the following costs. First, the cost of flushing all dirty L2 blocks [2]. Second, invalidating the L2 blocks that were flushed. Third, updating the L3 coherence directories. Fourth, performing the circuit changes associated with the reconfiguration [9]. It is worth noting that since the L2 cache is larger than L1, the reconfiguration cost of the above steps could be more expensive than prior work which had considered a similar scheme for L1 caches [9]. In addition to the cost issue, another reason why we considered a simple scheme is as follows. Our motivation for adaptive co-cache is to ensure that workloads whose working-set sizes already fit into the conventional L2 do not incur any slowdown due to our co-cache; in other words, this is simply a *fail-safe* option for benchmarks that already perform well with a conventional cache. A more sophisticated fine-grained reconfiguration scheme is therefore beyond the scope of this paper.

[2]Since all blocks are read-only in co-cache (§III-B), flushing operation only applies when transitioning from conventional cache to co-cache

TABLE II. ARCHITECTURAL PARAMETERS

Processor	3GHz, 4-core, in-order superscalar (4-issue)
L1 I/D caches	each 32 KB/4way, 2-cycle
L2 cache	256 KB/8 way, 12-cycle
Co-cache	256 KB/8 way, 12-cycle (§IV), depth 2
Reconfg. threshold	40%
ToL3 bus	12 GB/s, single core; 48 GB/s, 4-cores
L3 cache	16MB (4-core)/4MB(single core), 16way, 40-cycle
Memory	DDR3-2000, 9-9-9
Memory Bus	1GHz, 8-byte, single core 2GHz, 8-byte, 4 cores

TABLE III. L2 MISS-RATES

A (above than 70%)	B (40 - 70%)	C (less than 40%)
2.applu 3.art 7.facerec 8.galgel 9.gcc2k 11.mcf 13.mgrid 16.sixtrack 17.swim 21.wupwise 29.Gems 30.milc 31.soplex 33.omnetpp	1.apsi 4.bzip22k 14.parser 18.twolf 20.vpr 22.bzip2 23.gcc 27.leslie3d 32.hmmer	5.crafty 6.eon 10.gzip 12.mesa 15.perlbmk 19.vortex 24.gromacs 26.sjeng 25.gobmk 28.calculix

D. Space Overhead

Each cache block in the co-cache requires additional space for the word-ids. Also, with the same data array size, a co-cache has more sets compared to a conventional cache – consequently, more tag and status registers are required for those extra sets. On the L1 side, word-ids are again required to identify the words brought from the co-cache, in the transition period before the full block is back from the lower-level. Besides, footprint registers are also required to remember the critical words for prediction. Thus, for a co-cache with a depth of 2, every block in L1 cache needs 2 word-id registers and 2 footprint registers. If we consider a 32 kB L1 (4-way, 64B block), a 256 kB L2 (8-way, 64B block), and 5-bit status registers, the additional overhead of using the L2 as a co-cache amounts to 36.75kB (predictor: 0.75KB, additional tags: 36kB). An aco-cache additionally requires shadow tags to monitor the cache miss-rate, which requires around 8 kB space, bringing the overall overhead to 44.75kB overall. The co-cache (aco-cache) only adds 2.2% (2.2%) overhead to L1 and 13.4% (16.4%) overhead to L2. Thus, to consider 179KB in a 4-core system, it is less than 1% in overall cache hierarchy (L1s+L2s+L3).

IV. EVALUATION METHODOLOGY

We implemented our technique in the *gem5* cycle accurate simulator [11] and the default architectural parameters are shown in Table II. We used a depth of 2 for the co-cache experiments as we found that this was the depth that gave the best performance results.

Workloads We evaluated our technique across the SPEC benchmark suite (20 from SPEC2000 and 12 from SPEC2006) with ref input. We used all programs except those we could not get to compile and/or run correctly in our infrastructure. Recall that the co-cache is most effective for programs without sufficient L2 cache capacity to accommodate their working sets (i.e L2 miss-rate is high). Accordingly, we show L2 miss-rates for these benchmarks in Table III. As we can see, with a 256 kB L2 cache, 14 of the 33 programs have an L2 miss-rate of greater than 70%.

Co-cache's access latency. The co-cache does not involve tag searching operations, since the co-cache increases the number of sets but *not* the associativity. However, the co-cache requires additional tag bits: 6 additional bits for identifying the critical words, and also the additional tags bits due to increase in the number of sets. We use CACTI to model the effect of the increased tag size on the access latency. Our results shows that the increased tag access latency is negligible (less than 0.1 ns), and can be overlapped with data access like in the conventional cache.

V. Results

A. Performance Improvement

The primary goal of our evaluation is to to compare the performance of the L2 used as a co-cache (aco-cache) with the conventional cache. We evaluate the performance of the following configurations:

- *baseline*. Conventional cache, 256 kB/8 way/64B.
- L2-320kB / L2-512kB. Conventional cache, 320 kB/10 way/64B (resembles the aco-cache area) and 512 kB/8 way/64B.
- *bypassing / auto-bypassing*. In bypassing scheme, the L2 cache is bypassed to reduce the access latency to the lower level memory (i.e. No L2 cache). Auto-bypassing automatically selects the better-performing alternative between accessing L2 and bypassing L2 for that benchmark.
- *co-cache / aco-cache*. The proposed design, 256 kB/8 way/16B.

The speedups (normalized to the baseline) are shown in Figure 6 . From these result, we can see that *bypassing* (No L2) provides speedup for some benchmarks with high miss-rate (as shown in Table III) such as soplex (speedup of 2.1%). In comparison, we observe that the co-cache provides much better speedups – for example speedups of 36.8% for soplex. On average, co-cache provides a speedup of 5.3% whereas bypassing incurs a slight slowdown (about 1% slower).

However, in some benchmarks the co-cache is slower than the baseline – for example, gromacs is 11.8% slower. This is because the 256 kB L2 in gromacs incurs low miss-rate and hence already provides good performance. This is one of the reasons we proposed the aco-cache; recall that the aco-cache is able to reconfigure the co-cache back to a conventional cache depending on the miss-rate. With the aco-cache, the average speedup is 6.1% which is better than the speedup provided by doubling the conventional cache size (5.0%). Most of the benchmarks which incur a performance loss with the co-cache have now improved significantly with the aco-cache – for example, gromacs which was 11.8% slower now performs as well as the conventional cache. Also note that the speedup incurred with *auto-bypassing* is only 2.7% and is significantly lesser than the co-cache/aco-cache speedups.

B. Sensitivity to L2 Size

Figure 7 shows the performance as the size of L2 is varied. As we can see, for the lower sizes (128 kB, 256 kB, 512 kB) the co-cache performs better than a conventional cache. However, for the larger cache sizes (1 MB, 2 MB) the conventional cache performs better. As we mentioned in previous section, this is because, as the size of the cache is increased, the conventional cache starts becoming more and more effective, and its miss-rate drops. On the other hand, the co-cache performance will be dependent on whether or not the particular benchmark displays criticality. Motivated by this observation, we show the performance results of aco-cache which could be configured into a conventional L2 or a co-cache depending on benchmark's miss-rate. As we can see, this performs significantly better than the other two, especially when the size is neither too small nor large.

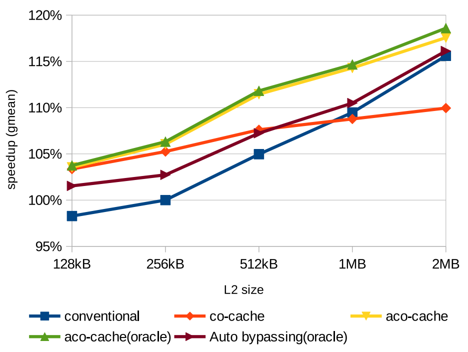


Fig. 7. Sensitivity to cache size

C. Sensitivity to Lower Level Latency

In this experiment, we want to examine our technique's sensitivity to lower level latency. Therefore, instead of a 3-level cache hierarchy, we *remove the L3 cache* and use a SimpleMemory (i.e. fixed access latency) module in gem5 and vary its latency. In our 3-level cache hierarchy, the average latency of *L3 + memory* (i.e. L2 miss latency) is about 100 cycles. Therefore, we vary the latency from 30 cycles through 240 cycles.

In this experiment, all results are normalized to the baseline (256 kB conventional cache). In addition to overall speedup results, we also show co-cache speedups under different L2 miss-rate groups that are shown in Table III. As we can see from Figure 8, the speedup of group A benchmarks (high miss-rate) increase with increase in latency. Since these benchmarks have high miss-rate, a co-cache hit becomes all the more important with increasing latency. However, for benchmarks from group C (low miss-rate), co-cache would lose more performance because most of accesses are already hits in the baseline configuration.

Overall, the speedup of the co-cache first increases from 5.2% to 5.9% and then decreases to 1.1% when the latency is varied from 30 cycles to 240 cycles. Since the aco-cache enables L2 to choose between co-cache and conventional cache based on L2 miss-rate, it provides a relatively consistent speedup (5.5% to 7.4%). This shows that aco-cache can continue to work even if the access latency to lower level memory is high such as a no-L3 system.

D. Prefetcher Effects

Hardware data prefetching is a technique to predict miss stream and fetch blocks before they are accessed. If the prefetcher brings data into the L3, our technique is orthogonal to it. However, if the prefetcher brings data into the L2, this cannot directly apply to our L2 (co-cache) as the co-cache only holds partial blocks (critical words). As an alternative, we can have the prefetcher bring data into the L1 instead, with the potential downside of prefetcher induced L1 cache

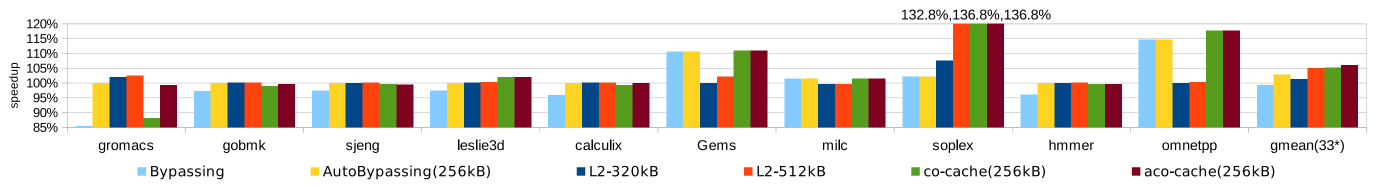


Fig. 6. Performance Improvement — due to space constraints, only SPEC2006 benchmarks are shown, but the *gmean* is across all 33 benchmarks.

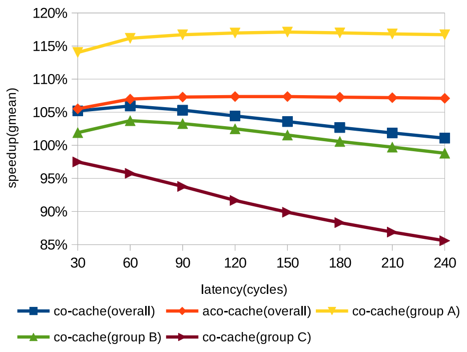


Fig. 8. Performance improvement with different latencies – Group A, B, C are defined in Table III.

pollution. In this experiment, we apply a stride prefetcher to L2 in our baseline system to see if our alternative (co-cache + L1 prefetcher) can still improve over the baseline (prefetcher in L2). With the prefetcher included, some of the benchmarks no longer experience significant capacity misses. In other words, for such benchmarks there is no problem to solve. Nonetheless, despite using a prefetcher, a significant number of benchmarks (19 out of the 33) continue to suffer from capacity misses. We are able to get 6.3% (up to 36%) speedup if we consider these 19 benchmarks (3.4% speedup if we consider all 33 benchmarks). From this result, we believe that the co-cache/aco-cache can continue to boost performance even under the presence of a prefetcher.

E. Energy Impact

In this section, we discuss the energy impact of our technique. We use CACTI [12] to model co-cache's power consumption. In CACTI, we use *itrs-hp* cell and 32 nm technology. The access mode is set to *fast* in L1 and *normal* for L2 and L3. The co-cache potentially has both static and dynamic energy overheads compared to conventional cache.

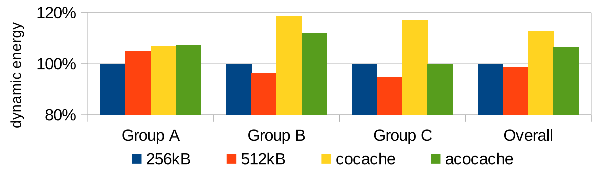


Fig. 9. Dynamic energy comparison (the lower the better)

With regard to static energy, the aco-cache induces additional space overhead of 44.75kB as described in §III-D. On a 4-core system, this amounts to 179KB overhead, which is a mere 1% space overhead if we consider the overall cache hierarchy (L1s+L2s+L3). Therefore, the increase in static power is about 1%. However, since co-cache (aco-cache) improves the performance by 5.3% (6.1%), there is an overall *reduction* in static energy of 4.2% (5.3%) with the co-cache (aco-cache).

With regard to dynamic energy, the co-cache requires simultaneously accessing the L3 even for L2 co-cache hits and the aco-cache requires additional accesses to shadow tags for obtaining miss-rate (as in §III-C). As we can see from Figure 9, because most of accesses in group A (high miss-rate benchmarks) miss in the L2 and go to the L3, the co-cache (aco-cache) only causes a moderate increase of 6.8% (7.5%) in dynamic energy. In group B and group C, the dynamic energy overhead increases to 18.5% and 16.9% respectively. However, the aco-cache improves the energy overhead (compared to co-cache) in group C by reconfiguring back to conventional mode and shows almost no energy overhead. It is worth noting that the monitoring will be turned off [8] after the decision to reconfigure it as a conventional cache. Overall, the dynamic energy overhead of aco-cache is 6.8%. If we consider the total energy impact (static + dynamic), there is a 3.8% (5.0%) improvement with the co-cache (aco-cache), as the static energy dominates dynamic energy.

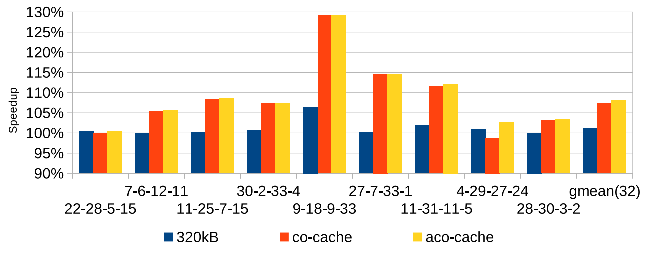


Fig. 10. Multi-core workloads — Due to space constraints, only 9 workload groups are shown, but the *gmean* is over the 32 workload groups

F. Multi-core Workloads

In this section, we study how our co-cache design performs under multiprogrammed workloads. We randomly generate 32 workload groups, with each group consisting of 4 workloads (the numbers identify the benchmark programs as shown in Table III). The 4-core system configuration is shown in Table II. The speedup results are shown in Figure 10. We use the sum of each workload's IPC (throughput metric) and normalize them to 256 kB baseline. As we can see, the co-cache is up to 29.3% faster than the baseline and on average 7.3% (8.1% for aco-cache) faster than the baseline, whereas the 320kB size cache (that resembles aco-cache in area) only provides 1.1% speedup. Since this is comparable to the speedups we obtained with our uniprocessor workloads, this shows that the co-cache continues to work well under multi-programmed workloads. One interesting observation here is that aco-cache performs consistently better than the co-cache with multiprogrammed workloads for most cases. We believe this is because aco-cache benefits here from two reasons. First, like before, aco-cache avoids performance loss for conventional-preferring benchmarks. In addition to this, since conventional caches cause lesser traffic on the L3 bus, this frees up bandwidth which can be potentially exploited by co-cache-preferring benchmarks.

VI. Related Work

Our idea is inspired by the classic *critical word first* [13] technique for reducing miss penalty, in which, instead of providing the full cache block, the word causing the miss (the critical word) is provided first. Recent works [14], [15] adapted this idea to reduce the access latency by proposing a memory system that is able to service the critical words faster. In contrast to the above works which seek to reduce the access latency, we seek to increase the cache capacity in the lower level caches.

Useful words fetching. A number of works exploit the fact that not all words belonging to a cache line end up being used; only a subset of the words – the *useful words* – end up being used. Several prior works [1], [2], [4], [5] focus on predicting these useful words and fetching only these. Kumar et al. [1] first observed that not all words belonging to a cache block end up being used, i.e spatial locality is not always high in all cache blocks. They exploit this observation to improve the miss-rate of a *sectored cache* [16] – which is a design to optimize bandwidth by fetching only sub-blocks. However, the performance with this technique is still worse than the conventional cache because of the low useful words predictability. To improve that, Chen et al. [4] and Pujara et al. [5] proposed more accurate PC-based predictors to provide better predictability. While the above techniques focus on primarily reducing bandwidth, the recently proposed *amoeba cache* [2] also focus on the performance. They do this by allocating a suitable number of entries for each cache block depending on the number of consecutive useful words in that block. We consider these techniques to be orthogonal to our work. While our co-cache works by retaining the critical words when a block is replaced from the higher level, the above techniques work by only fetching useful words. In other words, it would be possible to integrate our technique with any of the above, which we leave for future work.

Useful words filtering. Qureshi et al. [3] proposed *Line Distillation* to achieve better performance. They use a separate *word-organized cache* (WOC) as a victim cache which caches only the useful words. When a block is replaced from the higher level, they filter the useful words and put it in the WOC. In doing so, the cache is utilized more effectively which in turn leads to better performance. The downside to this approach is the complexity of supporting WOC. WOC is a highly associative cache – compared to a cache with 64 byte blocks, a WOC with 8-byte words increases associativity by 8 times.

Our approach, like Line Distillation, is a filtering approach; however, we exploit critical words rather than useful words. Even if all words in a cache block end up being used (i.e spatial locality is high), we can still benefit as long as the critical words are accessed before the non-critical words, as we illustrated in the limit study in §II-C. Indeed, we found that for most benchmarks the number of critical words is only 2 – which is why caching only these is sufficient. In other words, we do not need the complexity of having to deal with blocks of various degrees of spatial locality. Thus, unlike WOC used in Line Distillation, our approach does not increase associativity.

VII. Conclusion

In this paper, we proposed a cache design called *critical-words-only cache* for increasing cache capacity of an L2 cache. Our design is based on the observation that for every L2 cache block, a subset of words (the critical words) are accessed sooner than the others. In contrast to a conventional L2, a co-cache only caches the critical words for each cache block. Our experimental results provide evidence to support the hypothesis that a co-cache is able to utilize the cache space more effectively, especially in situations in which the cache size is significantly less than the working-set size. However, in situations in which the cache size is larger than the working-set size, a conventional cache could perform better. For this reason, we also proposed adaptive co-cache (aco-cache) that can dynamically choose to behave like a co-cache or a conventional cache. In our experiments, a 256 kB L2 organized as an aco-cache performed as well as a 512 kB conventional L2 cache on average.

References

[1] S. Kumar and C. B. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," in *ISCA*, 1998, pp. 357–368.

[2] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy," in *MICRO*, 2012, pp. 376–388.

[3] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *HPCA*, 2007, pp. 250–259.

[4] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos, "Accurate and complexity-effective spatial pattern prediction," in *HPCA*, 2004, pp. 276–287.

[5] P. Pujara and A. Aggarwal, "Increasing the cache efficiency by eliminating noise," in *HPCA*, 2006, pp. 145–154.

[6] R. K, T. Warrier, and M. Mutyam, "Skipcache: miss-rate aware cache management," in *PACT*, 2012, pp. 481–482.

[7] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006, pp. 423–432.

[8] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-v dd : A circuit technique to reduce leakage in deep-submicron cache memories," in *ISPLED*, 2000, pp. 90–95.

[9] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar, "An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches," in *HPCA*, 2001, pp. 147–157.

[10] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *MICRO*, 1999, pp. 248–.

[11] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[12] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to model large caches," 2009.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[14] E. J. Gieske, "Critical words cache memory: exploiting criticality within primary cache miss streams," Ph.D. dissertation, Cincinnati, USA, 2008.

[15] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Fang, R. Illikkal, and R. Iyer, "Leveraging heterogeneity in dram main memories to accelerate critical word access," in *MICRO-45*, 2012, pp. 13–24.

[16] A. Seznec, "Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio," in *ISCA*, 1994, pp. 384–393.