Inefficiencies in the Cache Hierarchy: A Sensitivity Study of Cacheline Size with Mobile Workloads

Anouk Van Laer *
University College London
London, UK
anouk.vanlaer@ucl.ac.uk

William Wang
ARM Research
Cambridge, UK
william.wang@arm.com

Chris Emmons
ARM Research
Austin, USA
chris.emmons@arm.com

ABSTRACT

With the rising number of cores in mobile devices, the cache hierarchy in mobile application processors gets deeper, and the cache size gets bigger. However, the cacheline size remained relatively constant over the last decade in mobile application processors. In this work, we investigate whether the cacheline size in mobile application processors is due for a refresh, by looking at inefficiencies in the cache hierarchy which tend to be exacerbated when increasing the cacheline size: false sharing and cacheline utilization.

Firstly, we look at false sharing, which is more likely to arise at larger cacheline sizes and can severely impact performance. False sharing occurs when non-shared data structures, mapped onto the same cacheline, are being accessed by threads running on different cores, causing avoidable invalidations and subsequent misses. False sharing has been found in various places such as scientific workloads and real applications. We find that whilst increasing the cacheline size does increase false sharing, it still is negligible when compared to known cases of false sharing in scientific workloads, due to the limited level of thread-level parallelism in mobile workloads.

Secondly, we look at cacheline utilization which measures the number of bytes in a cacheline actually used by the processor. This effect has been investigated under various names for a multitude of server and desktop applications. As a low cacheline utilization implies that very little of the fetched cachelines was used by the processor, this causes waste in bandwidth and energy in moving data across the memory hierarchy. The energy cost associated with data movements is much higher compared to logic operations, increasing the need for cache efficiency, especially in the case of an energy-constrained platform like a mobile device. We find that the cacheline utilization of mobile workloads is low in general, decreasing when increasing the cacheline size. When increasing the cacheline size from 64 bytes to 128 bytes, the number of misses will be reduced by 10%-30%, depending on the workload. However, because of the low cacheline utilization, this more than doubles the amount of unused traffic to the L1 caches.

Using the cacheline utilization as a metric in this way, illustrates an important point. If a change in cacheline size would only be assessed on its local effects, we find that this change in cacheline size will only have advantages as the miss rate decreases. However, at system level, this change will increase the stress on the bus and increase the amount of wasted energy due to unused traffic. Using cacheline utilization as a metric underscores the need for system-level research when changing characteristics of the cache hierarchy.

CCS Concepts

•Computer systems organization → Multicore architectures; •Hardware → Communication hardware, interfaces and storage; *Power and energy; Interconnect power issues;*

Keywords

Mobile devices; Mobile workloads; False sharing; Cacheline utilization

1. INTRODUCTION

The cache hierarchy forms an important part of a shared memory multiprocessor and any inefficiencies will affect the overall performance and power consumption of the architecture.

In this work we take a closer look at the effect of an increase in the cacheline size by first looking at false sharing (Section 3). False sharing is a well-known problem in shared memory multiprocessors and has been looked into for various workloads and cases. With the advent of multicore in mobile devices such as smartphones and tablets, we investigate the occurrence of false sharing in mobile workloads. Understanding the severity of false sharing in mobile workloads can help us decide whether there is a need to eliminate false sharing via changes in the cache and interconnect designs, compilers, and operating systems. Mobile devices featuring quad-cores and even octa-cores are being more commonplace now. While we focus on a dual-core system in this work, the conclusions we draw holds for a higher core count as false sharing only occurs between collaborating threads sharing a common physical address space and research has shown that such thread-level parallelism is quite low in mobile applications [8].

The methodology used to detect false sharing (Section 2) can also be used to look at how the individual memory addresses in the cacheline are used. Multiple memory addresses are grouped into a cacheline to exploit spatial locality and reduce the communication overhead. The cacheline utilization captures how many bytes in the cacheline have been touched when that cacheline gets removed from the cache.

*This work was done whilst Anouk Van Laer was doing a HiPEAC industrial internship at ARM

A low cacheline utilization will have a double effect. Locally, the total cache capacity was not efficiently used with parts of the cacheline remaining untouched. Globally it indicates that as part of the data brought into the caches was not used, a proportion of the traffic on the bus was superfluous. Writebacks of unused data to storage class memories will reduce the endurance of those devices unnecessarily. With communication being quite power consuming, cache utilization is an important metric to consider. Whereas this has been explored for other workloads, we wish to investigate the cacheline utilization of mobile workloads (Section 4).

2. METHODOLOGY

2.1 Hardware based detection of false sharing

To detect false sharing we instrumented the full-system, cycle-approximate simulator gem5 [3]. This represents a scheme which could be implemented in hardware, however the hardware is not the focus of this paper.

Figure 1 gives a schematic reproduction of the complete detection mechanism. To detect the sharing patterns, information about the accesses to every cacheline will be kept at a word granularity. Firstly, an *access vector* gets attached to every cacheline. This access vector tracks the local accesses to the individual words in the cacheline. To compare the local and remote accesses, every message following a cache miss will be extended so it also contains which words in the cacheline have been requested exactly. This *remote access vector* can then be compared with the local access vector to detect false sharing. When any form of false sharing (Section 3.3) is detected during an invalidation, the cacheline address and the remote access vector causing the invalidation will be added to a list called the *false sharing map (FS Map)*. Upon a cache miss, the false sharing map will be checked to see whether the miss was caused by false sharing. If the address is present in the false sharing map, the miss can be classified according to Figure 2b. The cacheline address will then be removed from the false sharing map as every miss needs to be completed, upon which the cacheline is no longer falsely invalidated. Every cache also contains a second list, the *false sharing histogram (FS Histogram)* used to track statistics about past false invalidations and subsequent misses. It monitors the number of false sharing related events per cacheline address and the virtual addresses mapping onto this cacheline. This detection method resembles a hardware structure and as such will detect all false invalidations, without any false positives.

The structure as implemented in gem5 assumes every cacheline in the cache gets an access vector added as additional meta-data. Every D-L1 cache also gets extended with a false sharing probe, containing the FS Map and the FS Histogram. If this were to be implemented as a real hardware scheme, this would result in a rather high area footprint. As this work focuses on false sharing detection and not the actual implementation, we did not investigate schemes that have a reduced area and complexity.

2.2 Usability for cache utilization measurements

The hardware used to detect false sharing can also be used to detect cacheline utilization, as the access vectors already track the accesses to the cacheline on a word granularity. Cacheline utilization can then be measured by counting the number of words that are touched (i.e. read or written) upon the time of measurement, by a specialized *cacheline utilization probe*. Utilization can be measured upon cacheline eviction, cacheline invalidation or other specified moments.

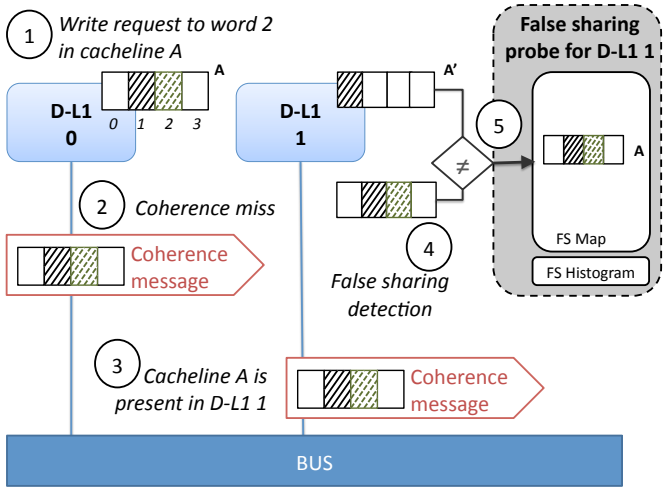


Figure 1: False sharing detection mechanism. The shaded blocks in the access vectors indicate locally used words (1) CPU 0 sends out a write request to a memory address in word 2 in cacheline A. (2) The write request misses as the cacheline is not in the exclusive state and a coherence message is broadcast, requesting the invalidation of other copies (3) The coherence message hits in D-L1 1 which invalidates A' (4) Before the invalidation, false sharing is detected by comparing the remote access vector (brought in via the coherence message) and the local access vector. (5) The access vector is stored in the FS Map to detect future misses caused by the false invalidations.

2.3 Workloads

To investigate the presence of false sharing in mobile applications, we used four different benchmarks, each representing a different type of mobile application and together encompassing a broad part of the mobile spectrum. BBench [9] is a browser benchmark. The AngryBirds benchmark [19] is based upon on the infamous game of the same name and represents gaming on the Android platform. AndEBench [21] is an open-source benchmark, aimed at Android mobile devices with a focus on integer and floating point operations. CaffeineMark [16] is a benchmark aimed at measuring Java performance. More information about these benchmarks can be found in Table II in [20]. We compared the values obtained for these mobile benchmarks with the streamcluster benchmark in the PARSEC suite [2], a benchmark with known false sharing.

2.4 System architecture

In this work we investigated a dual-core system (ARM v7) where every core has a private L1-D (2-way associative, 32 kB) and private L1-I (2-way associative, 32 kB). Both cores share an L2 (8-way associative, 1 MB). To investigate the presence of false sharing, a sweep over various cacheline size (from 16 bytes to 256 bytes) is performed while these parameters are kept constant. These simulations are done using the full system, cycle-approximate simulator gem5 [3]. We use the ARM architectural models in gem5 which are stable

and allows us to boot Android Kitkat. gem5 provides various types of architectural models: we use the atomic CPU which is a functional yet fast model. The coherence protocol used is a MOESI-based protocol. In mobile systems, screen rendering will done by a specialized GPU. However, as gem5 does not contain GPU models, software rendering is used. In this case, the screen is rendered by running software rendering code on the generic CPU. This is not realistic, as the software rendering instructions and data might affect the instructions and data of the actual threads, so some simulations are repeated without any screen rendering, removing the need for software rendering.

3. FALSE SHARING IN MOBILE WORKLOADS

In this section false sharing is defined. To allow a precise quantification of the false sharing problem, we will also classify the various types of false sharing and subsequent misses. This classification will help to determine an upper bound of the gains that can be made if a false sharing-optimized architecture were to be designed and used, such as [23].

3.1 Definition of false sharing

False sharing is a well-known problem in shared memory multiprocessors [22, 7]. Multiple memory addresses are grouped together into one cacheline, to increase cache performance as it allows to take advantage of spatial locality and reduce the communication overhead per memory address. However, in a chip multiprocessor where multiple cores are collaborating in the same global address space, grouping data at the cacheline granularity can have negative side-effects in the form of false sharing. When two or more cores are working on the same cacheline, but not the exact memory addresses in the cacheline, *false sharing* occurs.

False sharing can lead to *false invalidations*, causing cache lines to be needlessly invalidated. Imagine a thread wishing to write to a memory address which is currently not writeable in its local private cache (as it is not present or is not in the correct coherence state to allow writes). If the cacheline this memory address belongs to, is also privately cached by another remote core, this copy needs to be invalidated to keep the address space coherent and consistent. The invalidation of this cacheline is false as the remote core was not sharing the exact same memory address as the one currently causing the invalidation. This invalidation would not have occurred if the cacheline size were smaller and it contained only one memory address.

3.2 Effects of false sharing

False sharing in itself is harmless: if a cacheline is falsely invalidated in a cache, but never used again by that cache, false sharing does not affect performance. However, any subsequent accesses to the falsely invalidated cacheline will lead to a cache miss. Such cache misses lead to performance degradation and poor scalability for multithreaded applications.

False sharing also leads to unnecessary data movements which drain the battery of mobile devices. This becomes especially pronounced when both threads are trying to write to the same cacheline at the same time (ping pong-like behaviour): upon a write by one the threads, the cacheline currently residing in the other cache, gets invalidated and transferred completely to the other cache. Upon the next write by the other thread, the same process gets repeated and the whole cacheline is again (needlessly) transferred to the other cache.

The *linear_regression* benchmark in the Phoenix suite [18] is known to have false sharing [14, 13]. As an example, removing the false sharing present in *linear_regression* improves performance by more than 12× [13] and 5.4× [14] on Intel Xeon processors. We rerun the same *linear_regression* benchmark on the Arndale Octa board that features the ARM Cortex-A15 [1], the performance of the application improves by 73% and the energy consumption decreases by 64% after fixing false sharing.

3.3 Classification of false sharing and subsequent misses

Read/write requests from the CPU can span multiple words in one cacheline, which means there can be a grey zone where the locally requested memory addresses partially overlap with the remote memory addresses. To allow an exact quantification of the false sharing problem, we define the following types of invalidations (Figure 2a):

- **True invalidation**: the locally used memory addresses in the cacheline and the remotely access completely overlap
- **Grey invalidation**: there is some overlap between the locally used memory addresses in the cacheline and the remote write causing the invalidation.
- **False invalidation**: there is no overlap between the locally used memory addresses in the cacheline and the remote access causing the invalidation

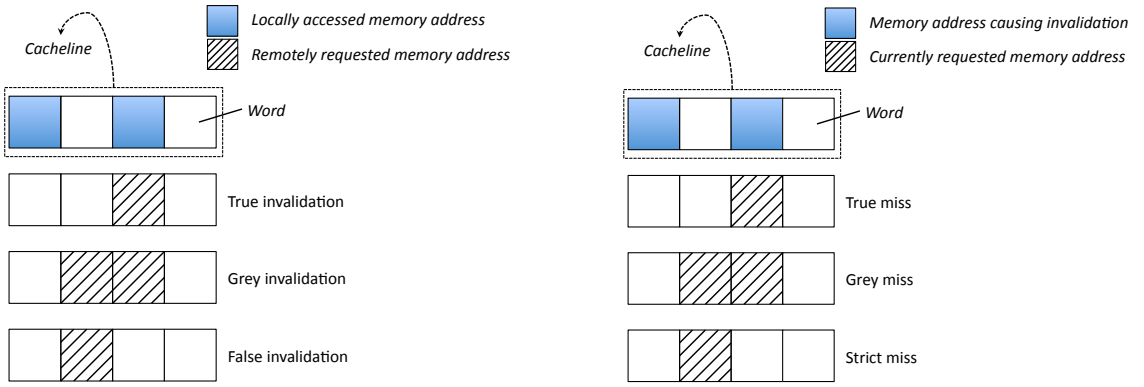
To determine how many misses following a false invalidation could be avoided by an optimisation, a similar classification can be made of all misses (Figure 2b).

- **True miss**: the addresses causing the invalidation and the currently requested addresses completely overlap.
- **Grey miss**: there is some overlap between the addresses causing the invalidation and the currently requested addresses
- **Strict miss**: there is no overlap between the addresses causing the invalidation and the currently requested addresses

Only strict misses following false invalidations could be avoided by any optimisation of the architecture. These strict misses tie in with [7] where false sharing misses are defined as non-essential misses: the invalidation leading to the miss could have been ignored and the program would still execute correctly.

3.4 Related work

False sharing has been investigated in various ways, for multiple types of workloads as it can have a detrimental effect on performance. In [14] false sharing is determined by tracking the number of invalidations per instruction via hardware counters, which might indicate false sharing. However, its hardware counter based first step does not distinguish between false and true sharing. When the number of



(a) Various types of invalidations

(b) Various types of misses

Figure 2: False sharing terminology

invalidations exceeds a preset threshold, the next steps of the process determine whether or not the contention is actually due to false sharing. They ascertain the presence of false sharing in the *linear_regression* benchmark of the Phoenix suite [18] and *streamcluster* in the PARSEC suite [2]. They also show the negative effect false sharing can have on performance by measuring the increase in speedup when adding more cores: by avoiding false sharing by remapping the addresses in linear_regression the speedup can increase by over 4×. PREDATOR [13] predicts false sharing based upon a single execution. It does this via compiler instrumentation that detects memory accesses. Once the number of writes and invalidations to a cacheline exceed certain thresholds, access information is tracked to allow for false sharing detection. PREDATOR detects false sharing in some benchmarks in the Phoenix and PARSEC suites. It also detects false sharing in real applications: MySQL and Boost. The Protozoa work [23] proposes a coherence protocol in which the coherence granularity and actual storage and communication granularity differ, to reduce unused data movement. They also detected false sharing in some scientific workloads, as a consequence of the coherence granularity being too large. False sharing has also been detected in various other cases such as the Java virtual machine [6] and the Linux kernel [4]

Unlike the software detection approaches, this work focuses on hardware based detection of false sharing in mobile applications. As we track the accesses to the individual words in the cacheline, we do not have any false positives in contrast to [14]. The hardware detection scheme also avoids any compiler instrumentation as used in [13]. The scheme used in this work only detects false sharing and does not involve any costly changes to the coherence protocol [23].

3.5 Results

In this section we will discuss whether false sharing is an issue for mobile workloads. This will be done by discussing three metrics and comparing the values obtained by the mobile workloads with the values obtained by streamcluster, a benchmark in the PARSEC benchmark suite with known false sharing issues [14][13]. First we will look at the number of writes involved in causing a false invalidation and subsequent miss, secondly we look at the effect of the false invalidation misses and lastly, we investigate the exact addresses involved in the false invalidations.

3.5.1 *False invalidations*

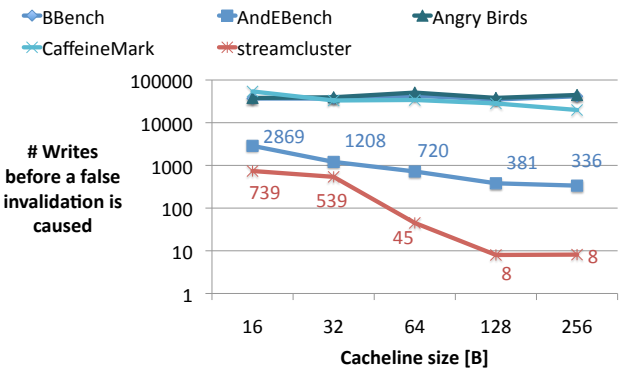
We define a new metric to measure the amount of false invalidations per benchmark: *number of writes per false invalidation*. This indicates the number of local write accesses before a false invalidation will be caused in a remote cache. This metric however does not indicate how the false invalidations will affect cache performance as not every false invalidation will cause a miss. To investigate the effect of false invalidations on performance we define *number of writes per miss caused by false invalidation*. Figure 3a shows the number of write accesses needed before a false invalidation will be caused in a remote cache. Increasing the cacheline size increases false sharing as the number of writes needed to cause a false invalidation decreases. The figure also indicates there are major differences between the various mobile benchmarks: when the cacheline size is set to 64 bytes it takes around 700 writes in AndEBench to cause a remote false invalidation. In BBench, however, this takes over 50,000 writes.

Of all mobile workloads, only AndEBench seems to have a tendency towards false sharing. However, if we compare this value to the value we obtained for streamcluster, it becomes clear the false sharing behaviour of AndEBench is limited: in the case of streamcluster 45 writes are enough to cause a false invalidation. This is a 16× decrease.

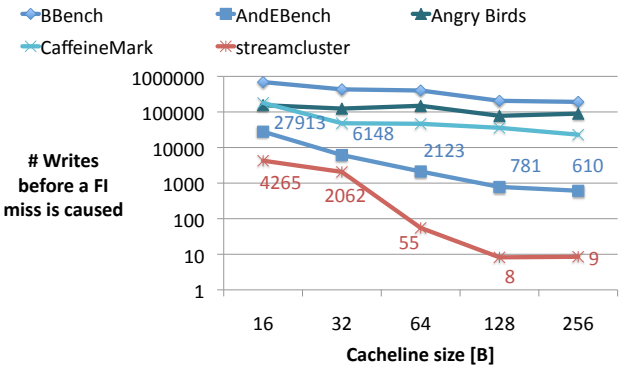
Looking at Figure 3b it becomes clear that not every false invalidation will result in a miss. Taking AndEBench as an example, while it only takes around 700 writes to cause a false invalidation, it takes over 2000 writes to cause a miss because of a false invalidation. Comparing this with streamcluster where it takes only 55 writes to cause a false invalidation miss, confirms the hypothesis that false sharing is not problematic in mobile workloads.

Another interesting point in Figure 3b is that while most mobile benchmarks show similar behaviour when it comes to the number of writes per false invalidation, they do differ in the number of writes it takes to actually cause a miss. This shows that it is important to take the cache accesses after a false invalidation into account as a false invalidation might not always be problematic if the invalidated cacheline is never referenced again.

3.5.2 *Effect of false sharing on miss rate*



(a) Number of local writes before causing a remote false invalidation, for various mobile benchmarks



(b) Number of local writes before causing a remote miss because of a false invalidation, for various mobile benchmarks

Figure 3: False sharing in mobile workloads

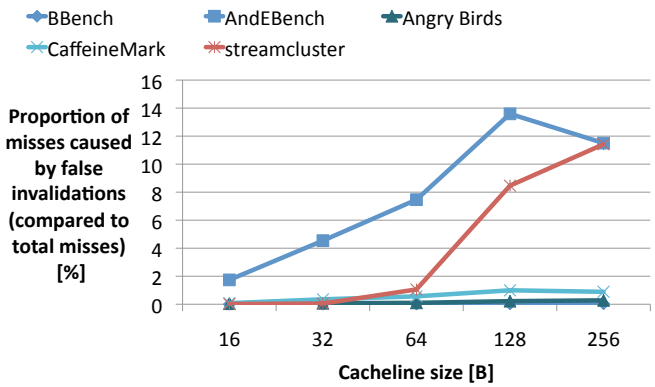


Figure 4: Proportion of all misses that can be attributed to false invalidations

The following section will examine the various types of false invalidation misses occurring and their effect on the overall miss rate. Figure 4 shows the proportion of all misses that are caused by false invalidations. For most benchmarks this proportion is relatively small: at most 1% of all misses are related to false invalidations. However, in the case of AndEBench, a substantial number of misses are caused by false invalidations as up to 14% of all misses originate from false invalidations. The proportion of false invalidation misses decreases when increasing the cacheline size past 128 bytes. This is due to the fact that, even though the absolute number of false invalidation related misses increases, the overall number of misses increases more dramatically. The proportion of misses caused by false invalidations is similar in the case of streamcluster and AndEBench. There are some differences though: in the case of streamcluster, less misses can be attributed to false invalidations which might be explained by its inherently high miss rate (Figure 5b).

To investigate the effect a hypothetical architecture optimization could have, we look into the miss rate of AndEBench. The full line in Figure 5a shows the normal miss rate of AndEBench. An architecture optimization focused on avoiding false invalidations and the subsequent misses would only be able to remove a subset of all false invalidation misses. Only strict false invalidation misses caused by a false invalidation as depicted in Figure 2 can be avoided, as in those cases there is absolutely no overlap between the access pattern causing the invalidation and the newly requested access pattern. Luckily, almost all false invalidation misses belong to this category: in the case of AndEBench with a cacheline size of 64 bytes, over 99% of all false invalidation misses are strict misses and thus avoidable.

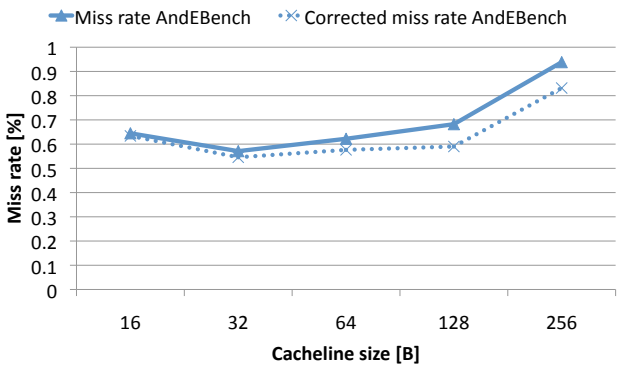
By removing all avoidable misses and recalculating the miss rate, we can get an estimation of the effect an optimization would have. This corrected miss rate for AndEBench is depicted in Figure 5a with a dotted line. The corrected and real miss rate both stay under 1% and are not too different. The costs of applying an optimization, whether this be in software by for example remapping the virtual addresses or in hardware by a change in the coherence protocol as proposed in [23], might outweigh this slight decrease in miss rate. Correcting for misses due to false invalidations in the case of streamcluster does not change the miss rate significantly (Figure 5b), which again might be due to the high miss rate.

Overall, AndEBench shows similar behaviour to streamcluster which might indicate false sharing could be an issue. However, correcting for misses due to false invalidations does not alter the miss rate significantly and might indicate false sharing is not an issue for AndEBench.

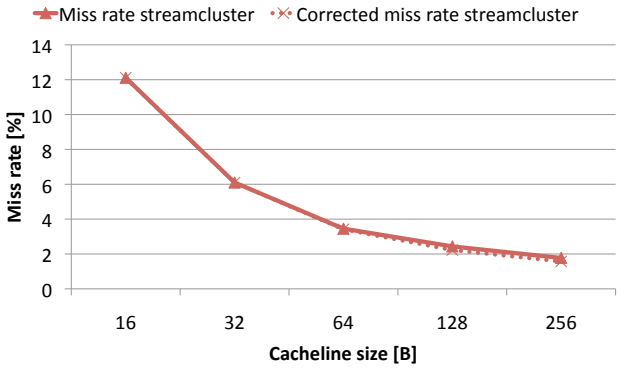
3.5.3 *Addresses involved in false sharing*

To confirm the hypothesis that false sharing in mobile workloads is harmless, we look into the exact addresses involved in false sharing. If there are only a few addresses involved in the false invalidations, this might indicate a contested data structure. To detect this, we kept a histogram of all physical and associated virtual addresses involved in false invalidations. The ten most invalidated physical addresses only account for 4% of all false invalidations which indicates there are no contested data structures. The distribution of the number of false invalidation across these ten most invalidated addresses confirms this as it is also quite uniform.

Comparing this to streamcluster gives some stark differences: the ten most falsely invalidated addresses account for over 99% of all false invalidations. The distribution of false invalidations among these ten addresses is not uniform either: four addresses account for almost all invalidations. This clearly indicates there are some contested data structures in streamcluster.



(a) D-L1 miss rate for AndEBench



(b) D-L1 miss rate for streamcluster

Figure 5: Miss rates. The dotted line indicates the case where all strict false invalidation misses are removed. This signifies an architecture where a hypothetical optimization has removed these avoidable misses.

3.5.4 *Effect of software rendering*

In all mobile simulations software rendering was used as gem5 currently does not simulate GPU's. This is an unrealistic scenario as normally the screen rendering gets offloaded to a specialized GPU. This could affect the false sharing results as both the data used by the software rendering threads and the 'actual' threads will reside in the same D-L1. The software rendering data could turn false invalidation misses into conflict or capacity misses. However, we argue that the effect of software rendering is minimal. Additional simulations without any screen rendering and hence no software rendering indicate software rendering does artificially increase the miss rate as it increases the working set. However, the software rendering data shows high spatial locality which makes that the first access might miss and push a cacheline belonging to the actual benchmark out of the D-L1 but subsequent accesses will hit the same cacheline and not cause further harm. The benchmarks with the most active software rendering threads (because of their continuously changing screens), BBench and AngryBirds, show such a negligible amount of false sharing that, even when accounting for the artificial reduction of false sharing, the amount of false sharing in a realistic simulation would still be inconsequential. We conclude that software rendering might artificially decrease the number of observed false sharing events but this does not change the overall conclusion that false sharing is limited in mobile benchmarks.

Based upon the small impact the removal of false invalidation misses has on the miss rate, the uniform distribution of false invalidations and the high number of writes needed to cause a false invalidation related miss, we can conclude that any false sharing occurring in mobile workloads is accidental and not related to contested data structures. This ties in with the work in [8] which states there is limited thread level parallelism in mobile workloads. Overall, we come to the conclusion that false sharing is not an issue for mobile workloads.

We analysed these mobile workloads on a dual core system. However, as most mobile applications use less than two cores on average [8], we conclude that false sharing will not form an issue either at higher core counts.

3.6 Conclusions

By implementing a hardware based method to detect false sharing, we show that false sharing is not an issue for most mobile workloads. The workloads that do exhibit false sharing do so to a very limited extent, when compared to scientific workloads known for false sharing. The absence of false sharing in mobile workloads ties in with the low level of thread-level parallelism in these workloads. We set out to find whether an increase in the cacheline size will affect certain inefficiencies in the cache hierarchy: in the case of false sharing, we conclude that while the increase in cacheline size will result in a higher amount of false sharing, false sharing remains negligible in mobile workloads and should not be taken into account when deciding upon a change in the cacheline size.

4. CACHELINE UTILIZATION IN MOBILE WORKLOADS

The infrastructure used to detect false invalidations and subsequent misses can also be used to track the utilization of cachelines. To measure utilization, the number of individual memory addresses in a cacheline that have actually been used, will be counted, when the cacheline reaches the end of its life in the cache. This can be due to two reasons: in the most common case, the cacheline gets evicted to make space for another cacheline. In a limited number of cases however, cachelines gets invalidated: because of a coherence transaction initiated by a remote cache for example, the local cache is not allowed to hold on to its copy of the cacheline.

In the previous section on false sharing we only considered the D-L1 as false sharing cannot occur in the I-L1. In this section we will consider the I-L1 as well though. To make the figures more clear we have averaged the utilization of three mobile workloads (AndEBench, BBench and CaffeineMark), with error bars to denote the maximal and minimal utilization over all workloads.

4.1 Definition and related work

Cacheline utilization can be an important metric when assessing the efficiency of the L1 caches. We define cacheline utilization as the number of bytes in the cacheline that have actually been used (read or write) by the CPU upon time of measurement. When the CPU requests an address, the complete cacheline will be brought into to the cache, to exploit spatial locality. However, those bytes in the cacheline

that were never referenced will not only have a negative effect on the cache itself but also on the communication fabric. A low cacheline utilization means that the storage capacity available in the cache was not optimally exploited. The second effect, on the communication fabric, might be even more severe because of the increased power consumption of communication [15]: all unused bytes have been brought into the cache, thereby consuming power and wasting communication bandwidth. When writing back evicted cachelines, those unused bytes are transmitted a second time, thereby consuming power and bandwidth a second time.

We record the cacheline utilization using the same structure used for the detection of false sharing. The access vector tracks the references to the cacheline on a word granularity. In the ARM v7 architecture however, requests can be shorter than one word: they can span a byte or a half word (2 bytes). Because of this, the recorded cacheline utilization will be higher than the effective cacheline utilization as a half word access for example will only use 2 bytes but will be recorded as a full word, hence 4 bytes. This makes the cacheline utilization as measured here an upper limit.

Investigating cacheline utilization allows for more precise trade-offs to be made, with regards to cacheline size for example (Section 4.4). Cacheline utilization, as defined above, has been investigated in multiple works, albeit with slightly different terminology sometimes. Some authors refer to cacheline usage [5] or cache noise [17]. Though the exact terminology might differ, most of the works below look at cacheline utilization to reduce power consumption. In [12] cacheline utilization is investigated to argue for a variable cacheline size. By looking at various benchmarks from the SPEC2006 and PARSEC suites, Java and commercial workloads, it becomes clear cacheline utilization varies both across workloads but also over the runtime of workloads. The average utilization is quite low: only 50% of all words in the cacheline are used when the cacheline is evicted, when assuming a cacheline size of 64 bytes. [5] looks at the SPEC2006 benchmarks in more details in the context of spatial pattern prediction. They compare the utilization across cacheline size and increasing the cacheline size from 64 bytes to 256 bytes decreases the unreferenced data in the cacheline from 45% to 66%. In [17] cacheline utilization is predicted so only words that are predicted to be used will be brought into the cache based upon utilization history. They measure the cacheline utilization in the SPEC 2K benchmarks, with a cacheline size of 32 bytes. On average, the integers benchmarks exhibit a lower utilization (42%) than the floating point benchmarks (72%) as the latter show more sequential accesses and use more doubles which occupy consecutive words in the cacheline. On average, the cacheline utilization is quite poor (57%), even for a relatively small cacheline size. The work in [11] investigates cacheline utilization as a way to reduce dynamic power consumption in the network on chip for chip multiprocessors. They find that in the PARSEC benchmarks unused words are, on average, responsible for 35% of the dynamic power consumption. In the case of the fluidanimate benchmark, this increases up to 55%.

However, none of these works investigated cacheline utilization in a mobile environment where power consumption is an even more stringent constraint. We aim to characterize cacheline utilization of mobile workloads and see whether the previous trends exist as well in the mobile setting.

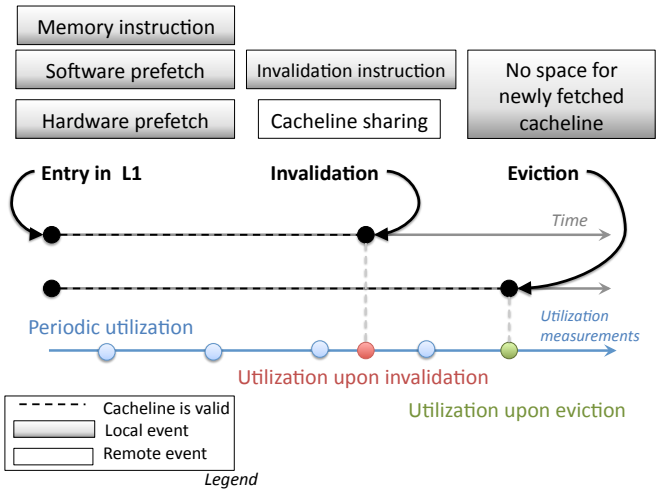


Figure 6: Life of a cacheline in the L1 caches

4.2 Utilization over cacheline lifetime

Figure 6 shows the life of a cacheline in an L1 and how the various possible end-of-life causes are related to the cacheline utilization measurements.

4.2.1 *Utilization upon eviction*

As shown in Figure 6 an eviction occurs when a new cacheline gets fetched for which there is currently no space in this cache. The life of the cacheline is determined by a local event. Measuring the cacheline utilization at the time of eviction gives the *cacheline utilization upon eviction*. Figure 7 shows the utilization upon eviction, averaged over multiple mobile workloads, for both the D-L1 and I-L1, where the error bars denote the maximum and minimum value obtained by individual benchmarks. In general the utilization of instruction cachelines is higher and shows less variation across workloads, indicating good spatial locality. However, the utilization is still relatively low. Around 60% of an evicted data cacheline is never used when the cacheline gets evicted, in the case of an evicted instruction cacheline this decreases to around 45%. This indicates there is a lot of unused traffic: bytes are transmitted across the bus and brought into the cache, but are never used.

It needs to be noted that only taking cacheline utilization upon eviction into account might skew conclusions, for example, the utilization of data cachelines in the case of AndEBench is quite low compared to the other benchmarks. This is due to hot lines: these lines do not get evicted regularly and their high utilization will not appear in the cacheline utilization upon eviction, giving the skewed impression that cacheline utilization is poor. The presence of hot cachelines can be ascertained by looking at the utilization of all valid cachelines on a regular basis. This is marked in Figure 6 as *periodic utilization*. If this periodic utilization is higher than the utilization upon eviction, hot cachelines are present in the cache.

The utilization of a data cacheline can affect the traffic twice: upon a fetch, unused data is brought into the cache. When a dirty cacheline is evicted, the complete cacheline will be written back to a lower level cache, transmitting those unused memory addresses again. Figure 8 shows the

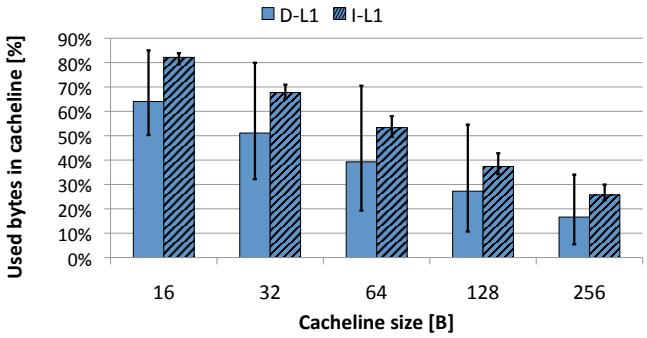


Figure 7: Cacheline utilization upon eviction in D-L1 and I-L1, averaged over various mobile workloads.

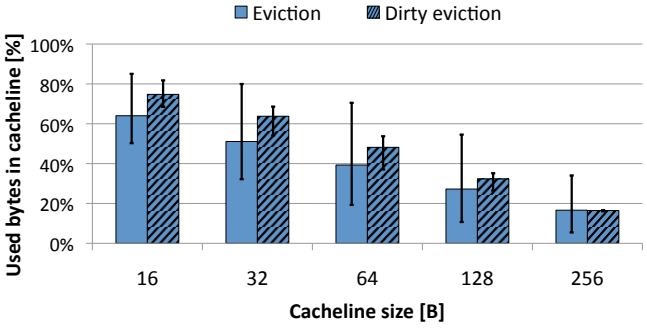


Figure 8: Cacheline utilization upon eviction and dirty eviction, averaged over various mobile workloads. The dirty evictions are a subset of the overall evictions.

utilization of dirty cachelines. Dirty cachelines have a higher utilization, as most writes follow a series of reads. However, between 30% and 60% of the writeback data was never used by the D-L1.

4.2.2 *Utilization upon invalidation*

Eviction of a cacheline is in a sense the natural end of life of a cacheline: the cacheline gets removed because of events in the local cache. Figure 6 shows the possible causes of an invalidation. Some invalidations are caused by local events like cache flushing instructions. Most invalidations however are caused by events in a remote cache, e.g. a false invalidation. It is to be expected that the cacheline utilization of invalidated lines will be lower than that of an evicted line as cachelines cannot reach their 'natural' end of life and have not built up utilization. This is confirmed in Figure 9: on average the utilization upon invalidation is 10% lower than the utilization upon eviction, at 64 bytes. This difference can increase up to 44% for BBench. Luckily, invalidations are very rare in comparison to evictions: at 256 bytes, which has the highest chance of invalidations, less than 1% of all cacheline removals is because of an invalidation.

4.3 Effect of software rendering on cacheline utilization

To look at the effect of software rendering on cacheline utilization, we look at BBench and AndEBench, running with and without software rendering, as they capture two ends of the spectrum. BBench, being a browser benchmark requires much more software rendering than AndEBench which has a fixed screen on which only a couple of variables are updated.

Figure 10a shows software rendering artificially decreases the I-L1 miss rate as the rendering code is quite repetitive. However, as Figure 11a the utilization of the instruction cachelines is slightly higher in the more realistic case. This could be due to the fact that the software rendering instructions are competing with the workload instructions which prevents them from building up utilization before eviction.

The D-L1 miss rates as depicted in Figure 10b show that software rendering increases the miss rate as it increases the working set: both the data needed by the software rendering threads and the workload threads now needs to fit in the D-L1. Figure 11b shows AndEBench and BBench exhibit different behaviour when it comes to cacheline utilization. In the case of BBench, where software rendering is more prominent, the more realistic simulation shows a lower utilization. This could be explained by the repetitive data usage in software rendering, which artificially increased the utilization. In the case of AndEBench, the cacheline utilization is higher when disabling software rendering. This ties in with the change in miss rate: as the miss rate is lower without software rendering, the cachelines remain in the D-L1 longer and can build up cacheline utilization.

In general though, while the quantitative results might differ, the qualitative conclusions hold: overall, the cacheline utilization of mobile workloads is quite low, indicating most of the data brought into the caches remains unused.

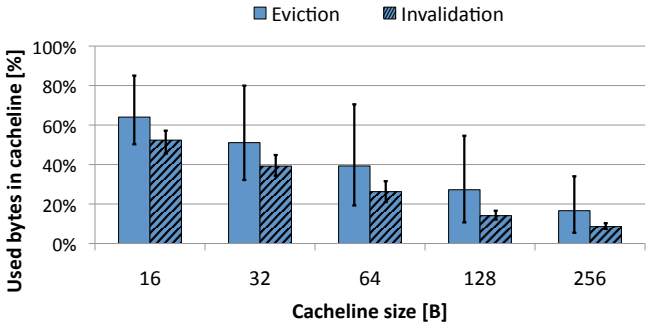
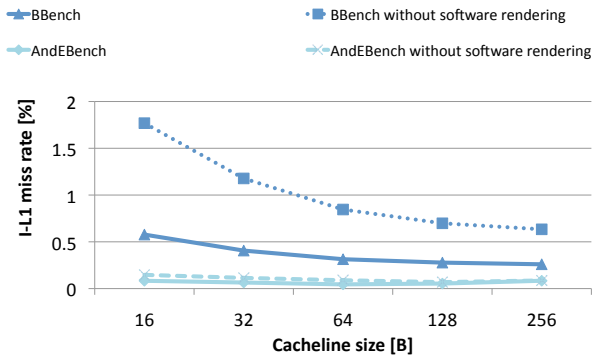


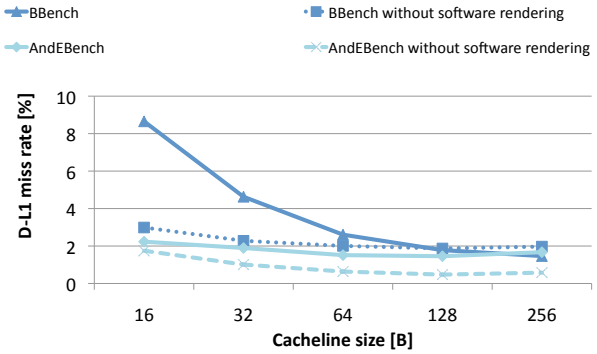
Figure 9: Cacheline utilization upon invalidation in D-L1, averaged over various mobile workloads.

4.4 Miss rate versus traffic trade-off based upon cacheline utilization

Knowing the cacheline utilization allows to make more detailed trade-offs at design time as it gives an idea of the amount of unused data brought into the L1 caches. As the power consumption associated with communication is quite significant, the cacheline utilization will also affect the overall power consumption. [15] looks at the cost of data movement in a mobile environment by measuring the power consumption of a Samsung Galaxy S3 smartphone, when running micro-benchmarks with known cache behaviour. They find that the energy cost of a LD operation which hits in the L1 is equal to 1.83 ADD operations. If however, the same LD request misses in the L1 and needs to be fetched from the L2, the LD operation will now be equivalent to 7.65 ADD operations. This shows the need for efficient data movements.

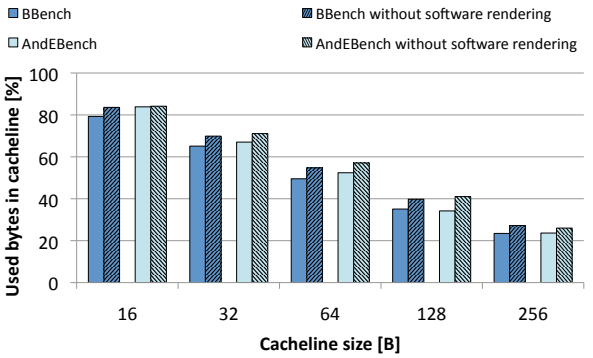


(a) Effect of software rendering on I-L1 miss rate

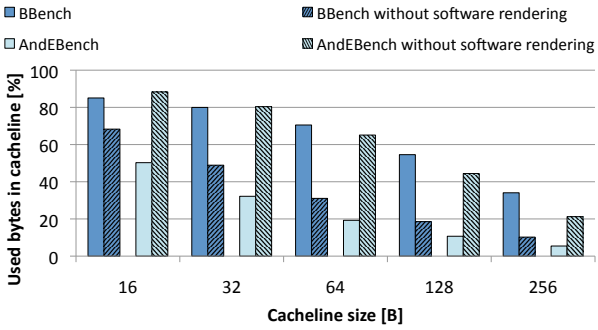


(b) Effect of software rendering on D-L1 miss rate

Figure 10: Effect of software rendering on L1 miss rates



(a) Effect of software rendering on cacheline utilization in I-L1 upon eviction.



(b) Effect of software rendering on cacheline utilization in D-L1 upon eviction

Figure 11: Effect of software rendering on cacheline utilization

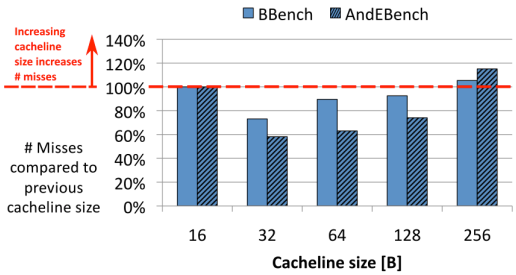
Section 4.2 showed that cacheline utilization decreases when increasing the cacheline size. However, in general, increasing the cacheline size will decrease the L1 miss rate because of spatial locality. This will continue until the ever larger cachelines will contend over space in the cache and the L1 miss rate will start increasing again [10]. In this section we will look at the effect of increasing the cacheline size on traffic. Increasing the cacheline size will (initially) decrease the miss rate and reduce the number of cachelines that need to be fetched whilst, at the same time, the cachelines become larger and more parts remain unused, which will affect the amount of unused traffic on the bus. As traffic can be see as a stand-in for energy consumption, this allows us to make an (albeit simple) trade-off between a better missrate and a higher energy consumption when choosing cacheline size. We do this analysis for both BBench and AndEBench, without software rendering.

Figure 12a shows the effect of increasing the cacheline size on the number of cache misses, compared to the previous cacheline size. This shows that for both benchmarks, increasing the cacheline size up to 128 bytes will have a positive effect on the number of cache misses as the number of misses is lower than at a smaller cacheline size. However, increasing the cacheline size beyond 128 bytes will increase the number of misses, due to conflict and/or capacity misses. Figure 12b looks at how this affects the amount of unused traffic on the bus. We define unused traffic as the amount data brought into the L1 caches which will remain unused over the lifetime of the cacheline. To calculate this, we assume every miss in the L1 is due to the cacheline not being present in the cache and will require the complete cacheline to be brought in. The majority of these cachelines will be evicted, a minority will be either be invalidated or remain in the cache. This allows us to calculate the amount of unused traffic by using the following formula:

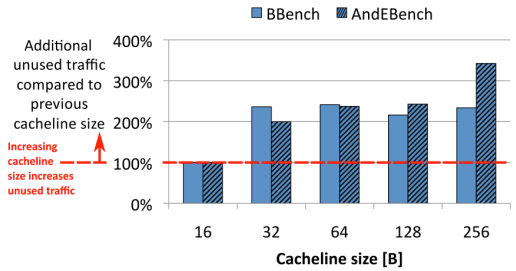
$$\begin{aligned} \text{Unused traffic} = {} & \text{L1 misses} \times \text{cacheline size} \\ & -[\#\ \text{Evictions} \times \text{Util. upon eviction} \\ & +\ \#\ \text{Invalidations} \times \text{Util. upon invalidation} \\ & +\ \#\ \text{Hot cachelines} \times \text{Util. of hot lines}] \end{aligned} \tag{1}$$

In reality though this formula is imperfect: some of the misses in the D-L1 will be coherence misses which will not require a data message but a shorter control message e.g. a write to an already present cacheline, currently in the shared state, will require other sharers to acknowledge the invalidation of their local copy, via a control message (upgrade). However, for AndEBench the proportion of coherence misses compared to the overall number of misses is below 18% for all cacheline sizes. For BBench this number is even lower: less than 1% of all misses in the D-L1 are coherence misses. A second imperfection lies in the fact that in the case of an out-of-order core with non-blocking caches multiple subsequent read misses can be aggregated into one fetch, if they hit in the Miss Status Handling Register. In this work however, we assume an in-order core.

As Figure 12b shows increasing the cacheline size will always increase the amount of unused traffic on the bus. However, by combining both Figure 12a and 12b it is possible



(a) Effect of cacheline size on the number of L1 misses



(b) Effect of cacheline size on the amount of unused traffic

Figure 12: These figures show the effect of increasing the cacheline size on both the number of misses and the traffic on the bus. The bars indicate the change in respectively cache misses or unused traffic, compared to the previous value of the cacheline size. A value above 100% indicates an increase compared to the previous value of the cacheline size (a) In the case of BBench, changing the cacheline size from 64 bytes to 128 bytes for example will make that the number of misses in the L1 at 128 bytes is equal to 90% of the misses at 64 byte (b) However, this will cause the unused traffic at 128 bytes to be 216% of the unused traffic at 64 bytes.

to make a more precise trade-off. In the case of BBench, increasing the cacheline size from 64 bytes to 128 bytes will result in only 90% of the misses occurring. However, there will 216% more unused traffic on the bus. In the case of AndEBench, the same increase in cacheline size will result in only 74% of the misses at the cost of 243% of unused traffic on the bus.

This example makes clear that while a large cacheline size will reduce the number of misses and increase cache performance, the price to pay for this is steep: the amount of unused traffic and hence energy wasted in superfluous data movements will more than double.

4.5 Conclusion

We find that mobile workloads show low cacheline utilization in general. Instruction cachelines are utilized more than their data counterparts and show more uniform behaviour across benchmarks. If the life of a cacheline gets cut short by a false invalidation for example, the cacheline utilization will lower significantly. By comparing the cacheline utilization of simulations with and without software rendering, we find that whilst software rendering does change the quantitative results, the qualitative results remain: the cacheline utilization is low, indicating the majority of the data brought into the L1 caches remains unused.

It also needs to be noted that the cacheline utilization as measured here is an upper limit as the utilization was tracked on a word granularity, even though accesses could be made to single bytes or half words. This makes the situation as presented here is in reality even more dire: when increasing the cacheline size, the miss rate will decrease (up to a certain point) but as the cacheline utilization lowers, the efficiency of the cache hierarchy decreases. This was exemplified by looking at a miss rate versus unused traffic trade-off, showing that cacheline utilization, in contrast to false sharing, is an important metric when looking at cacheline size optimizations in a mobile environment.

5. OVERALL CONCLUSIONS

The number of cores in mobile chip multiprocessors has steadily increased over the last decade. Meanwhile, the cacheline size however remained quite constant. We have investigated whether the cacheline size might be ready for a refresh by looking at sources of inefficiency in the cache hierarchy, affected by cacheline size.

A first form of inefficiency is false sharing, which has been shown to have a severe impact on performance in the case of scientific and other workloads. While an increase in cacheline size would increase the amount of false sharing in mobile workloads, the amount of false sharing is still negligible when compared with known cases of false sharing. This ties in with the low amount of thread level parallelism present in mobile workloads. Overall, we can exclude false sharing as a source of concern when changing the cacheline size in a mobile environment.

Secondly, we look at cacheline utilization. Similar to scientific and other workloads, the cacheline utilization of mobile workloads is quite low. Whilst this has implications for the caches themselves (inefficient use of cache capacity for example), the effect on the system is far graver. When changing the cacheline size from 64 bytes to 128 bytes, the number of misses will reduce by 10%-30%. However, the amount of unused traffic on the bus will be more than double. This shows a change in cacheline size will have system-wide ramifications: apart from the redesign of the caches themselves, other system characteristics will need to adapt to this change as well.

If a change in cacheline size would only be assessed by looking at the caches themselves, the direct consequences become clear: while initially the spatial locality favours an increase in cacheline size, eventually the conflict and/or capacity misses start playing up. However, by looking at a metric like cacheline utilization, it becomes clear that there are many indirect consequences: the bus, for example, whilst it might need to be redesigned from a purely hardware-based perspective to adapt to a change in the cacheline size, the amount of traffic it will handle will increase significantly as well, which also needs to be taken into account.

In this work we have only looked at the cacheline size as the initial parameter to change and assessed its effect on inefficiencies such as false sharing and cacheline utilization. However, the choice for a cacheline size will in its turn, be affected by other system characteristics such as the size of the memory bursts etcetera.

The parameter sweep, as presented here, across various cacheline sizes where the focus lies solely on cacheline utilization, shows how cacheline size affects and in turn, is affected by other system characteristics, emphasizing the need

for system-level research.

6. ACKNOWLEDGMENTS

Anouk Van Laer would like to thank the HiPEAC Industrial PhD Internship program.

7. REFERENCES

[1] ArndaleBoard.org. Arndale Octa Board.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University, Jan. 2008.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[4] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, and Kaa. An analysis of Linux scalability to many cores. *OSDI*, pages 1–8, Oct. 2010.

[5] C. Chen, B. Falsafi, and A. Moshovos. Accurate and Complexity-Effective Spatial Pattern Prediction. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 276–276. IEEE, Feb. 2004.

[6] D. Dice. False sharing induced by card table marking, 2011.

[7] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. *ACM SIGARCH Computer Architecture News*, 21(2):88–97, May 1993.

[8] C. Gao, A. Gutierrez, R. G. Dreslinski, T. Mudge, K. Flautner, and G. Blake. A study of Thread Level Parallelism on mobile devices. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 126–127. IEEE, Mar. 2014.

[9] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-system analysis and characterization of interactive smartphone applications. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 81–90. IEEE, Nov. 2011.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[11] H. Kim and P. V. Gratz. Leveraging Unused Cache Block Words to Reduce Power in CMP Interconnect. *Computer Architecture Letters*, 9(1):33–36, Jan. 2010.

[12] 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 *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 376–388. IEEE, Dec. 2012.

[13] T. Liu, C. Tian, Z. Hu, and E. D. Berger. PREDATOR: predictive false sharing detection. *19th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming,*, 46(10):3, Oct. 2014.

[14] M. Nanavati, M. Spear, N. Taylor, S. Rajagopalan, D. T. Meyer, W. Aiello, and A. Warfield. Whose cache line is it anyway? In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*, page 141, New York, New York, USA, Apr. 2013. ACM Press.

[15] D. Pandiyan and C.-J. Wu. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *2014 {IEEE} International Symposium on Workload Characterization, {IISWC} 2014, Raleigh, NC, USA, October 26-28, 2014*, pages 171–180, 2014.

[16] Pendragron Software Organization. CaffeineMark 3.0.

[17] P. Pujara and A. Aggarwal. Cache Noise Prediction. *Computers, IEEE Transactions on*, 57(10):1372–1386, Oct. 2008.

[18] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE, 2007.

[19] Rovio Entertainment Ltd. Angry Birds.

[20] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver. A structured approach to the simulation, analysis and characterization of smartphone applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 113–122. IEEE, Sept. 2013.

[21] The Embedded Microprocessor Benchmark Consortium. AndEBench, 2015.

[22] J. Torrellas, H. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[23] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas. Protozoa: Adaptive Granularity Cache Coherence. *ACM SIGARCH Computer Architecture News*, 41(3):547, July 2013.