Evaluating the Reuse Cache for mobile processors

Urmish Thakker, Lokesh Jindal, Swapnil Haria
Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI, USA
{uthakker, lokeshjindal15, swapnilh}@cs.wisc.edu

***Abstract*—There has been a lot of research over the years on novel cache architectures to improve cache performance or reduce cache area. However, the focus of these efforts has been on desktop processors and compute workloads. Today, mobile processors represent a significant chunk of the market, and are becoming as complex as their desktop counterparts. Area optimizations are crucial to drive down costs and improve profits in this highly competitive market. Caches, which occupy 20-30% of the die area of mobile SOCs, are ideal candidates for downsizing. Hence, our work intends to extend the Reuse Cache architecture for use in a mobile processor as it promises reductions in cache size with minimal performance degradation.**

In this report, we confirm that the observations regarding high percentage of dead lines in the shared Last-Level Cache hold true for mobile workloads running on mobile processors. We propose new techniques to measure the utilization of cache lines to overcome the limitations of the original work. Our version of the reuse cache architecture was implemented in gem5, and the performance was evaluated for several memory-intensive Android applications from the AsimBench mobile benchmark suite. Compared to the baseline model with 4MB conventional L2 cache, our reuse cache design with 1MB data array and 4MB tag array is able to demonstrate 50% area reductions, 10% power savings at the cost of only 5% performance loss.

I. Introduction

Lately, the mobile SOC market has been growing at a breathless pace, and the competitive nature of the market is fueling immense innovation in mobile processors. Mobile processors like NVIDIA's Denver are trying to make their mark using techniques like dynamic code optimization. However, the market is still split between processors which sacrifice some power-efficiency for high performance (typically seen in tablets and high-end smartphones), and processors which deliver adequate performance under constrained power budgets (seen in most low-end smartphones). We believe that optimizing caches in these mobile processors for area and energy efficiency will help sustain aggressive processor techniques which deliver high performance.

Furthermore, given the low profit margins per chip in this volume-driven industry, there is a huge incentive for design innovations that help lower production costs. Reducing the die size is an obvious means of achieving this. Till recently, this was achieved primarily through technology scaling. However, the considerable issues associated with scaling beyond modern nanometer sizes have resulted in increasing costs and diminishing returns. Architectural innovations that can optimize the area of on-chip hardware are needed to drive costs down in this very competitive market. However, there seems to be a paucity of research work in this domain.

Shared-memory chip multiprocessors (CMPs) dominate the mobile processor market. A shared last-level cache (SLLC) is present in most CMPs to service miss requests from the private caches of all the cores, and also manages coherence. Previous studies have observed that the usefulness of the SLLC is mostly due to a few highly accessed cache lines, and that most of the cache lines are never accessed while cached in the SLLC. The spatial and temporal locality of memory accesses gets filtered out by the upper cache levels. This key observation is exploited by Alberico et al as the Reuse Cache [1] design. The reuse cache is based on the idea that the second reference to a particular line is indicative of future reuse, and only caches lines on their second reference. The current work aims at validating the observations regarding inefficient SLLCs for the mobile setup, and subsequently evaluating the merits of the reuse cache organization for mobile workloads.

Most mobile processors have smaller, simpler caches than their desktop counterparts. Also they have an L2 cache as the SLLC, unlike conventional CMPs which provide an L3 cache as the SLLC. Hence in this report, we first confirm that the cache access patterns being exploited by the reuse cache organisation hold true for mobile processors and workloads. An additional metric of evaluating SLLC efficiency is proposed to overcome the shortcomings of metrics used in the original paper. Then the reuse cache design is tweaked to fit into the mobile setup. The replacement policies are enhanced using ideas borrowed from the Shadow Directory proposal [2] to further improve cache performance.

Our implementation of the reuse cache is evaluated using a four-core CMP system running several memory-intensive applications from a mobile benchmarks suite, AsimBench [3]. We also simulate SPEC CPU2006 workloads to compare against the original paper. The reuse cache successfully identifies useful cache lines, while steering clear of dead lines. This results in about 50% cache area savings, and 10% energy reductions, with about 5% degradation in performance.

The report is structured in the following manner. Section II provides a brief background regarding the principles of the reuse cache and the shadow directory. Section III presents the evidence of the efficiency of the SLLC, in terms of ratio of live and dead lines present in the cache when running mobile workloads on our baseline system. Section IV describes our implementation of the reuse cache, giving details of the replacement policies as well as modifications needed to support the MOESI coherence protocol. Section V outlines the experimental methodology, the baseline model and the workloads simulated, and the experiments and results obtained are discussed in Section VI. Finally, section VII points out the

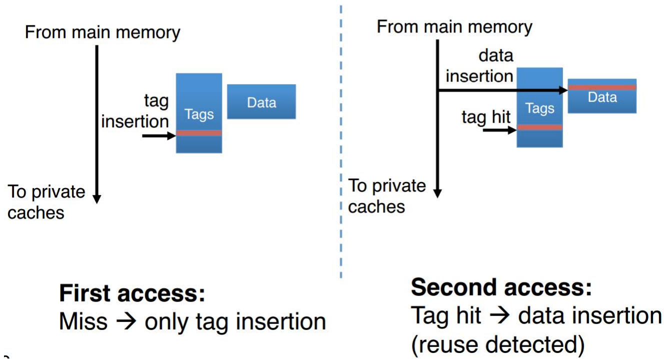

Fig. 1: Working principle of Reuse Caches, taken from[[1]]

key takeaways from our work, and identifies potential future directions to extend this work.

II. Background

A. Reuse Cache

Reuse Cache was proposed by Albericio et al for the L3 cache, which was the Shared Last Level Cache (SLLC) in their model. They observed that the reference stream seen by the SLLC for a Chip Multi-Processor (CMP) does not exhibit much temporal or spatial locality. As a result, most lines in the SLLC are dead and most hits come from a very small subset of cache lines. However, they found that the SLLC accesses do demonstrate a large amount of reuse locality- very few lines in SLLC get re-referenced and once a line receives a hit, there is a high probability that it will get referenced again. Thus, they proposed the reuse cache which improves the utilization of the SLLC by only fetching data for lines that exhibit reuse (Figure 1). This selective data allocation policy has improved the efficiency of the cache without negatively affecting performance. The performance of the reuse cache organization was demonstrated on an eight core CMP running multiprogrammed and multithreaded workloads. The results show that the reuse cache with a 4MB tag array and a 1MB data array could achieve the same performance as a conventional 8 MB L3 cache, reducing the area footprint of the SLLC by about 84%.

B. Shadow Directory

To further explore the ways to intelligently use the information stored in the tag array (with greater number of entries compared to the data array), we implemented and studied the performance of Shadow Directory [2]. Shadow Directory is a concept introduced in 1984 by Pomerene et al, which provides an improvement over the popular LRU algorithm. The shadow directory aims to distinguish between transient lines that pollute the cache, and lines that become active after long periods of inactivity. Thus it tries to protect data that has been dormant for a while, but which would be referenced again. A traditional LRU algorithm would evict such blocks and degrade performance.

The Shadow Directory maintains a copy of the actual cache, except that the copy only contains the directory but no data. The duplicate cache has the same organization as the original cache. It contains tags that have been evicted from the original cache. On a miss, a lookup is made to the shadow directory in order to evaluate whether this tag was originally present in the cache. If the tag is found in the shadow area, it is considered a shadow miss. Such tags are protected from eviction.

III. Motivation

As discussed in the previous section, SLLC exhibits a large proportion of dead lines. In this section, we analyze the observations made by the original paper regarding the percentage of dead lines in SLLC. Cache Access patterns were studied for the L2 SLLC after simulating benchmarks from SPEC CPU2006 as well as simulating mobile workloads from the Asimbench suite (Figure 2). A two pronged approach is proposed to compute the upper and lower bounds for estimating the proportion of dead lines.

The original paper categorized a cache line as live at a particular moment if it received at least one additional access during the rest of its stay in the cache; otherwise it is labelled as a dead line. In their work, the proportion of live lines in the SLLC was measured by simulating SPEC CPU2006 benchmarks on an 8 core system for 700M cycles, sampling the instantaneous number of live lines every 100K cycles. They reported that on an average, only 17.4% of the lines in the SLLC are live. To determine whether a line is live or dead, they looked at the state of the lines at sample time and categorized them appropriately based on their reference count. Their observations indicated that only 5% of all the lines loaded in SLLC are useful, receiving one or more hits. The remaining 95% of the loaded lines are useless as they will not receive any hit during their lifetime. This implied that a lot of space in SLLC was getting used up in caching data that did not serve any useful purpose. The whole idea of reuse cache was motivated to exploit this observation to reduce the SLLC data array size.

In this study, we compute the lower bound of the number of dead lines by analyzing them at the time of eviction. There might be lines sitting in the SLLC that might never get reused in the future. These are not accounted for as they never get evicted due to the applications working set size being smaller than the SLLC. In order to arrive at a better statistic, we propose another technique which is the upper bound of the number of dead lines. For this second method, we count the number of new lines being fetched into the SLLC. Further, whenever any cache line in the SLLC sees a second reference, we categorize it as a live line. We consider all the other lines as dead. This is the optimistic count of dead lines as the lines which are not live currently may be reused in the future. The above two ways of computing dead lines can be represented mathematically in the following way:

$$\%Dead\ Lines(on\ Fetch) = 1 - \frac{\#\ Lines\ Reused}{\#\ Lines\ Fetched}$$

$$\%Dead\ Lines(on\ Eviction) = \frac{\#\ Not\ Reused\ Evicted\ Lines}{\#\ Line\ Evicted}$$

We ran SPEC CPU2006 and Asimbench [3] workloads

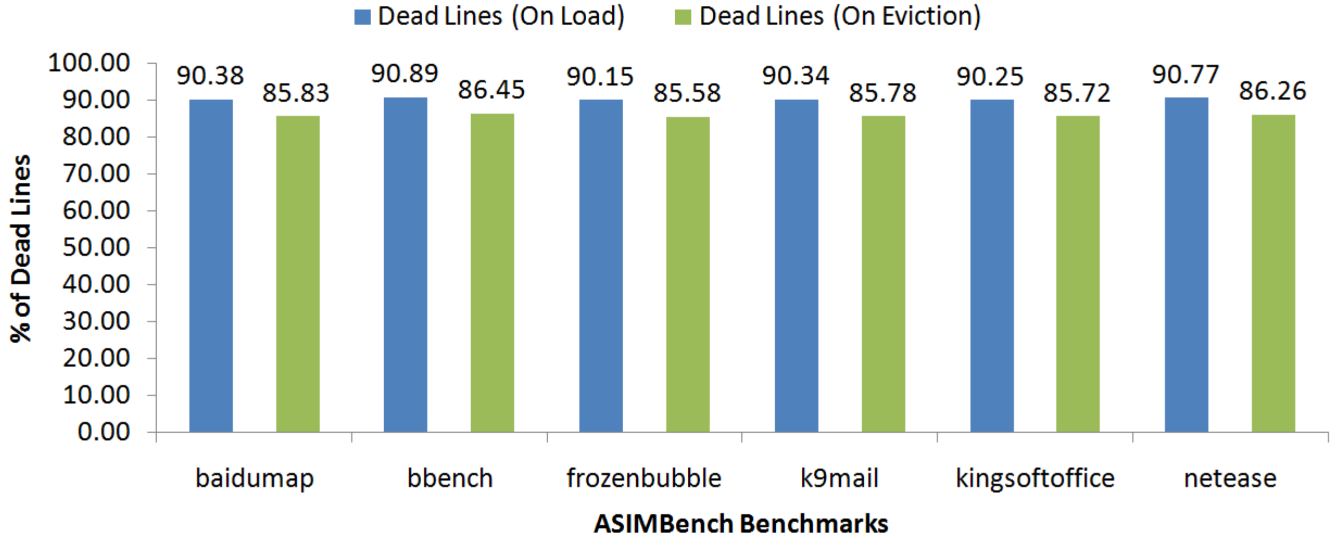

Fig. 2: Liveness of lines in the L2 SLLC for mobile workloads

for 10 Billion instructions after warmup and used the above metrics to observe the liveness of cache lines. As shown in Figure 2, for each benchmark run, we obtained a range for the number of dead lines in the SLLC. Thus, the observation in original paper was found to be true - a minimum of 85% of lines in the SLLC are dead, which could be in excess of 90% in certain scenarios. Thus, we see that the reference stream forwarded to SLLC does not exhibit significant spatial and temporal locality. This creates an opportunity to conserve space by shrinking SLLCs data array and modifying the data block allocation policy to store only the lines that will get reused. With the reuse cache implementation, the percentage of live lines in SLLC was observed to increase to about 50% for the performance-optimal design. These results are discussed in detail in Section VI.

IV. Implementation

A. Organization

Traditional Caches maintain a tight coupling between the tag and the data array. This results in a one to one mapping between a tag set and a data set. However, the reuse cache attacks this implicit notion. Here, a tag does not always have a data block associated with it. A particular coherence state helps identify this scenario. Since the tag and data are decoupled, the organization of the two arrays needs to be different. In order to assist in replacement and lookup, forward and backward pointers are maintained. The tag metadata is additionally increased in order to accommodate bits to detect shadow hits. Figure 3 shows an overview of the reuse cache organization, and Table I shows the extra metadata stored in the tag array. The forward pointer ensures that no additional

Tag	Forward Pointer	Coherence State	Recently Evicted	Shadow Hit

TABLE I: Tag Metadata

lookup is required in the data array. This enables the use of a fully associative data array. An increased associativity helps the replacement policy by providing more blocks to ch ose from. However, increasing associativity increases the size of the pointers stored by the arrays. Our results show that a 4:1 mapping is optimum, i.e., 4 sets of tag map to 1 set of data. Due of this fixed mapping, a backward pointer from the data block to a tag block is only 2 bits long.

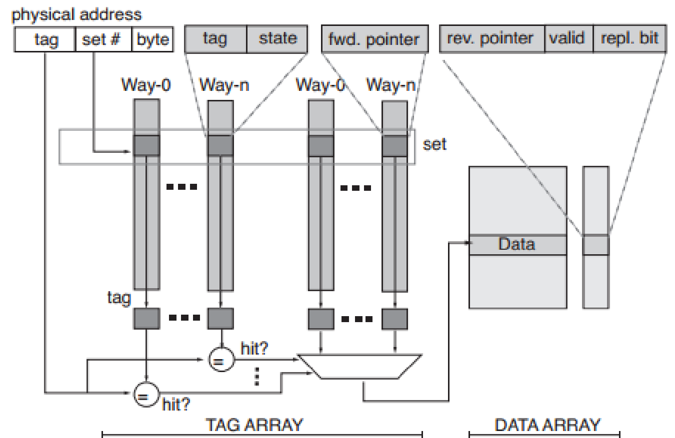

Fig. 3: Reuse Cache Overview, taken from [1]

B. Replacement

Different tags and data array imply that replacement policy can be different. In this project we use a Not Recently Reused policy [4] for the tag array and a modified Least Recently Used policy for the data array. The LRU policy is enhanced by borrowing concepts from shadow directory. This modified replacement policy differentiates between data that was a shadow hit and data that was a reuse hit. The shadow hit data is given the highest priority to protect it from eviction during dormant phases. During data eviction, the state of the corresponding tag is changed to tag only and its state is modified to indicate the recent eviction.

Figure 4 depicts the scenario prevented by the shadow directory. When the data is initially accessed, the cache line is in the tag only state. The second access changes its state to tag and data. However, the data is not referenced for a while and is thus evicted from the cache. When the data is referenced again, it takes 2 accesses to fetch the data back into the cache. A traditional LRU could not detect such patterns. Shadow Directory is specifically designed to identify such patterns and protect data in state 4 (of the figure).

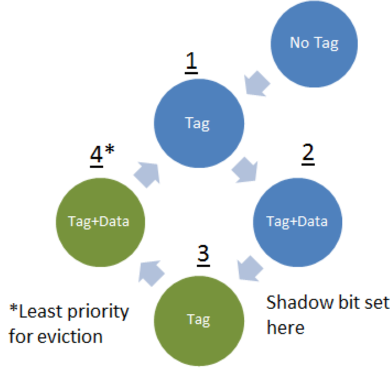

Fig. 4: Incorporation of Shadow Directory in replacement policy

In order to implement a Shadow Directory, extra metadata is stored in the tag along with the usual coherence and valid bits. The metadata consists of two bits, recently evicted and shadow hit. If a data of a tag is eliminated, the recently evicted bit is set. If this tag is referenced again, then the access is a shadow hit. Thus the corresponding bit is set while the recently evicted bit is cleared.

C. Coherence

Conventional cache coherence protocols assume that each line in a cache has an entry in both the tag and the data array. However, this assumption is not valid for the reuse cache. As a result an additional state of tag only needs to be maintained for coherence. We have modified the MOESI based coherence protocol to include such a state. Table II below describes the different states in our coherence protocol.

Name	Cache	Memory	Data
M	Modified, not shared	Stale	Yes
O	Shared, modified	Stale	Yes
S	Shared	Up-to-date	Yes
E	Exclusive, unmodified	Up-to-Date	Yes
I	Invalid or not present	-	No
TO	Only Tag, No Data	Up-to-Date	No

TABLE II: States of the TO-MOESI example coherence protocol

D. Hardware Cost

In this section, the actual reduction in size offered by a reuse cache is compared against a conventional cache, by considering the reduction due to reduced data array entries and the increase brought about mainly by forward and backward pointers. We consider the case of a 4MB, 8-way conventional cache, and a reuse cache with a 4 MB, 8-way tag array and a 1 MB, 8-way data array. The cache line size for both caches is 64 bytes, and we assume 40 bits of physical address in a 64-bit address space.

A tag entry in a reuse cache is similar to a conventional tag entry, with the addition of a forward pointer, a shadow bit and an extra bit for the Tag-Only coherent state. The data entry in a reuse cache requires additional bits for the backward pointer, a valid bit, a bit for the LRU replacement policy along with the 512 bits for data. We optimize the number of additional bits required for the forward and backward pointers by maintaining the same level of associativity between the tag and the data arrays. In such an arrangement, since there is a 4:1 ratio between the number of tag and data entries, 4 tag sets can be mapped to one data set. Thus, the forward pointer needs to be only 3 bits to identify the correct way in the appropriate data set. Conversely, the backward pointer needs to have 5 bits, 2 to select between the four potential tag sets, and 3 to point at the correct way in the set. The exact distribution of bits in the tag and data arrays in the two cache designs is compiled in Table III. Thus, the reuse cache with 4MB of tag entries and 1MB of data entries needs 10736 Kbits, which is about 30.72% of the storage capacity of the conventional 4MB cache.

Component	Conventional 4MB	Reuse 4MB + 1MB
Tag *	40-(13+6)= 21	21
Coherence	4	5
Full map vector	8	8
Replacement	1	1
Fwd pointer	-	3 (8-way data)
Total (bits)	34	38
Component		
Data	512	512
Valid	-	1
Replacement	-	1
Rev Pointer	-	2+3 = 5
Total (bits)	512	519
Tag Array (K entries	64	64
Data Array (K entries	64	16
Total Size (Kbits)	34944	10736

TABLE III: Hardware Cost of Reuse Cache

V. Methodology

Figure 5 gives an overview of the methodology used to evaluate the reuse cache implementation in this study. The Reuse Cache architecture described in section IV was implemented using the Classic Memory model in gem5. The ruby memory model of gem5 provides greater flexibility, but lacked support for ARM architecture at the time of writing. gem5 is the most widely used simulator in CPU architecture research today with support for ARM ISA (our target ISA for the study). It is being actively developed by several corporate organizations including ARM Holdings and is capable of booting a full fledged Linux Kernel as well as Googles Android operating system.

The performance statistics obtained from gem5 were used to analyze the IPC of the baseline conventional cache as well as the reuse cache. A system based on the A15 ARM

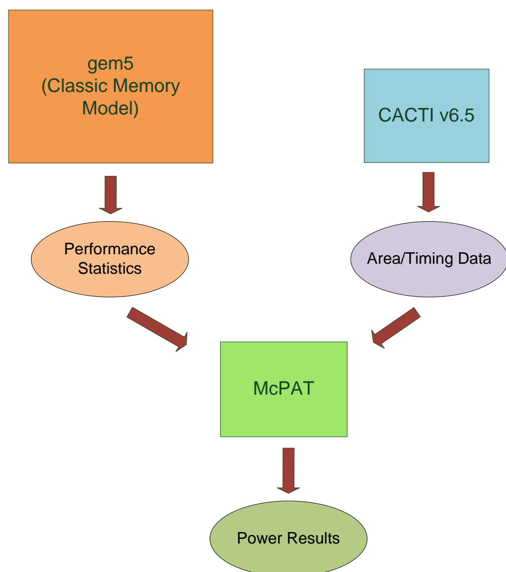

Fig. 5: Block Diagram of the methodology used

CPU was modelled in McPAT, an integrated power, area and timing modelling framework. The gem5 statistics from running different benchmarks were used with McPAT [5] to generate the power numbers. The original paper did not analyze the power consumption of reuse caches, which is an aspect covered in the current project. Cacti v6.5 [6] was used to calculate the area of the reuse and conventional cache models.

For our baseline system, we did a survey of the common mobile SoCs in market from leading companies like Qualcomm, Nvidia and Samsung. Finally an ARM A15-like processor model with the following parameters was chosen as the baseline:

- 32 KB/4 way L1 I-Cache
- 32 KB/4 way L1 D-Cache
- 4 MB/8 way L2 Cache (Shared Last Level Cache)
- 8 wide Out-Of-Order pipeline
- 1.4 GHz operating frequency
- 32 nm technology node

Precautions were taken to avoid sources of error in a full system simulations by following guidelines reported in [7].

To evaluate and compare the reuse cache against the baseline, benchmarks from 2 suites were simulated - SPEC CPU2006 and Asimbench (recently renamed to Moby) [3]. SPEC CPU2006 is the popular choice for evaluating processor optimizations and has been chosen to establish a direct line of comparison against the results from the original paper. AsimBench is a benchmark suite composed of mobile applications. The suite focuses on popular mobile applications and is designed to study the behavior of mobile applications on simulators like gem5. All the applications contained in AsimBench are popular mobile applications with high downloads on Google play store and span 10 categories, including web browser, social network, online shopping, email, audio, video, document, map and game. These results are of primary interest for our study as they represent realistic workloads running on mobile systems. For realistic performance estimation, AsimBench applications were simulated in Full System mode for 10 billion instructions, after a checkpointed boot to simulate about 6 seconds of real time. A snapshot of the gaming application, Frozen Bubble, being simulated in gem5 with the reuse cache implementation can be seen in figure 6. The results obtained are discussed at length in Section VI.

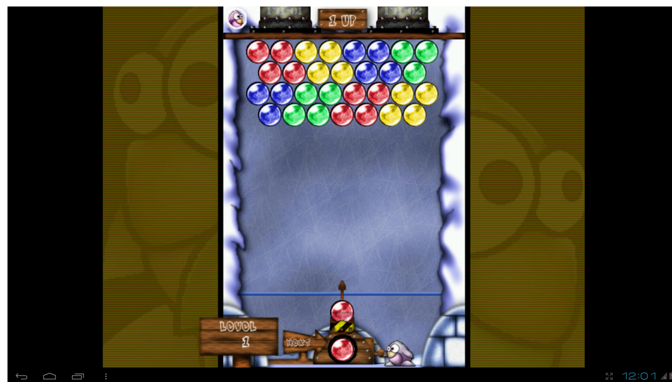

Fig. 6: Snapshot from simulation of Frozen Bubble

During the course of implementing the reuse cache in gem5, microbenchmarks were heavily used to validate the implementation. These were self written tests in C which made sequential memory accesses directed at generating a deterministic memory trace. The statistics regarding the number of SLLC hits and misses for tag and data were obtained by simulating the microbenchmark. These simulation results were compared for correctness against theoretically calculated values. Microbenchmarks proved to be a highly effective tool to flush implementation issues.

VI. Results

Firstly, we identify the optimal tag and data array size, reporting the results in Section VI A. Then, we look at the optimal size ratio between the tag and data arrays (Section VI B). We also compare the performance of a pseudo LRU replacement policy for data with the modified LRU+Shadow Directory guided replacement policy. Subsequently in sections VI D and VI E, performance of the optimal reuse cache organization is compared with our baseline model, described in section V. In section VI F, we demonstrate that the reuse cache architecture has significantly improved the liveliness of the cache. Finally in section VI G, we discuss our results in relation to the results reported in the original reuse cache paper.

A. Tag Array Size

We ran simulations for various tag array sizes in order to evaluate the performance of reuse cache. Figure 7 compares the performance of 3 different tag arrays for different SPEC2006 runs. We observe that the 4MB tag array provides the best performance. The reuse cache aims to provide reduced cache area without significant reduction in performance. Thus, we believe that the ideal tag array size for the reuse cache in the current scenario should be 4MB.

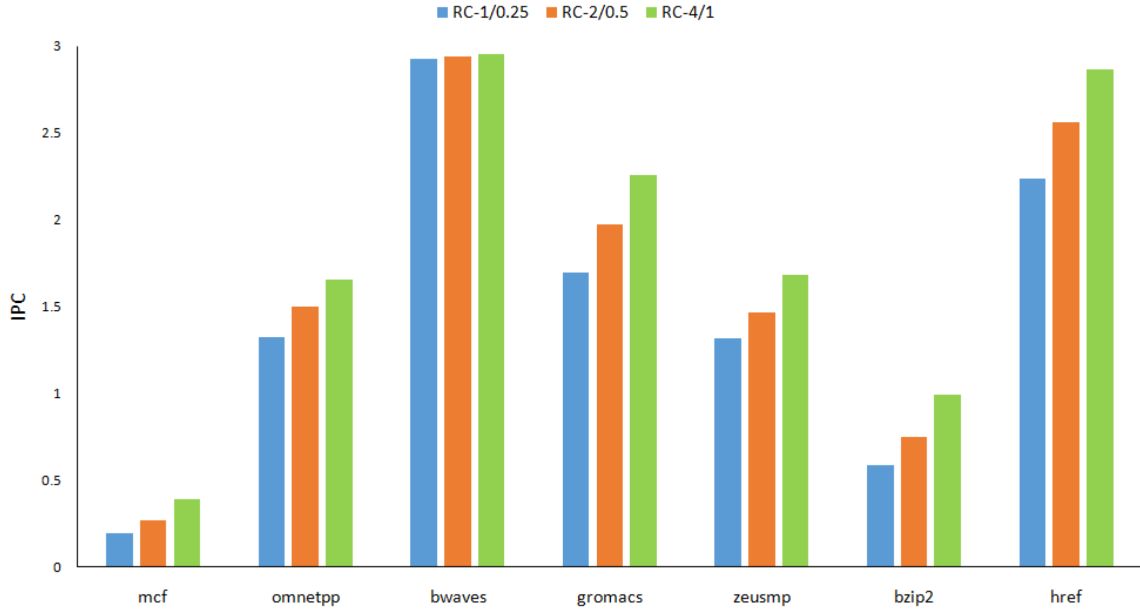

Fig. 7: Performance comparison for different tag array sizes

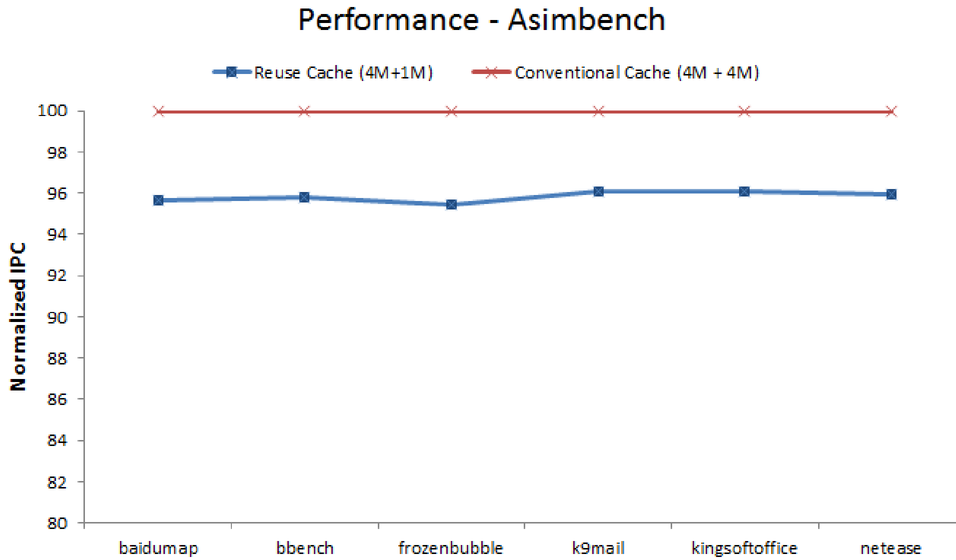

Fig. 8: Performance comparison for AsimBench applications

B. Tag Array to Data Array Ratio

In this section, we explore which data array size achieves the best performance for a 4MB tag array size. Figure 10 compares the performance of a reuse cache with 2 MB data array with a reuse cache with 1 MB data array. Apart from libquantum, there is no other benchmark that achieves significant speedup. Also, the average power consumption of the system increases by 3% when we move to 2MB from 1MB. We also ran similar tests for a data array size of 512kB and found that the performance degradation was significant. Going by these results, the reuse cache organization with a 4MB Tag Array and a 1MB Data array is the most optimal design considered for the rest of the study.

C. Data Replacement Policy

We now evaluate the impact of incorporating Shadow Directory into our replacement policy. From Figure 11, we observe that the performance improves by a factor of 2-3%. Although this improvement is very small, we believe that the performance improvement would be significantly better for an application running for a longer period of time. The reason for this is that the Shadow Directory is incorporated to protect useful lines which are dormant for long intervals. Additionally, this feature could be incorporated without any

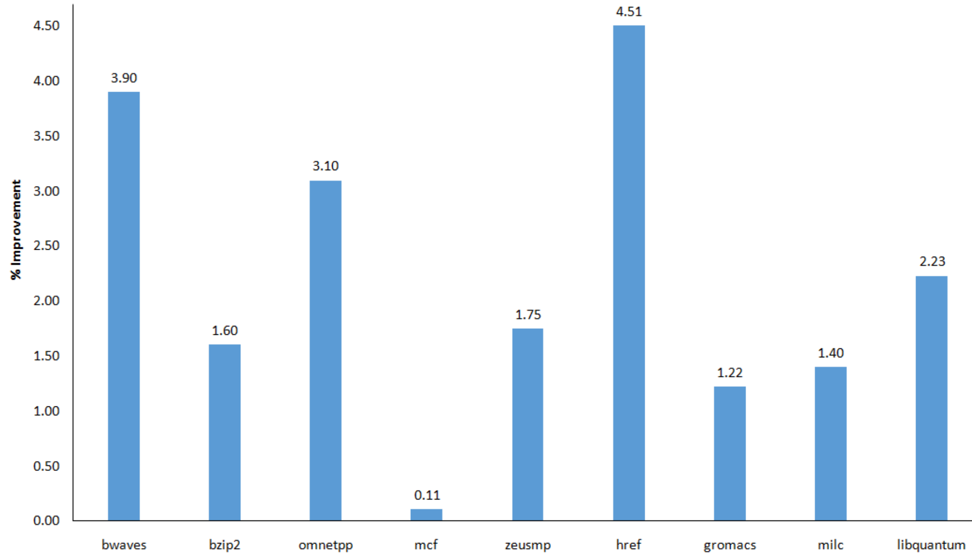

Fig. 9: Power Consumption

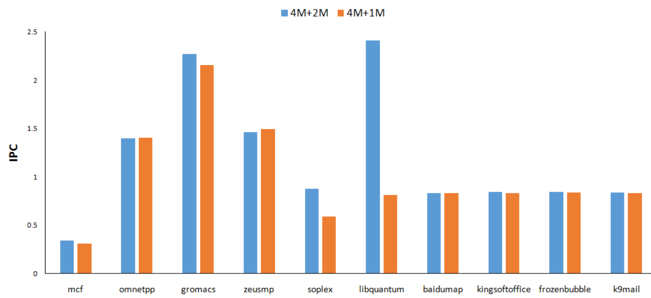

Fig. 10: Performance comparison for different tag array to data array ratios

additional hardware cost. If we view a reuse cache as a combination of data free tags and coupled tag-data blocks, it is clear that this organization inherently supports the structures required for the implementation of a shadow directory. By adding a single bit to metadata of the tag, we could implement this feature easily. Thus, all the results from this point onwards would assume an replacement policy which is a combination of the Shadow Directory and LRU algorithm.

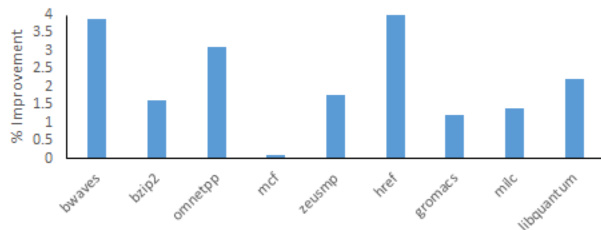

Fig. 11: Performance Evaluation of Shadow Directory

D. Comparison with baseline - IPC

Figures 8, 12 and 13 show the performance of reuse cache with respect to our baseline model. We observe that the average performance degradation for SPEC benchmarks is about 10%. For a multi core scenario, the average performance loss drops to around 15%. However, for mobile workloads, the average performance degradation is only 4%. This marginal performance degradation is acceptable considering the significant gains in area and power (discussed in the next section).

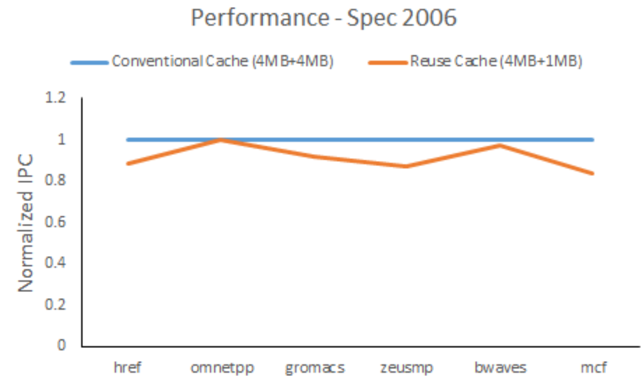

Fig. 12: Performance comparison for SPEC2006 benchmarks

E. Comparison with Baseline - Power and Area

As expected, the reuse cache leads to significant reduction in cache area. The results presented in Table IV have been obtained using Cacti 6.5. The reduction of area is observed to be about 50%. This reduction in area is not proportional to the scaling factor of the data array. The decrease in the size of the data array is slightly offset by the increase in the size of the tag array due to the additional metadata. The backward pointers and the valid bit add to the size of each individual data block as

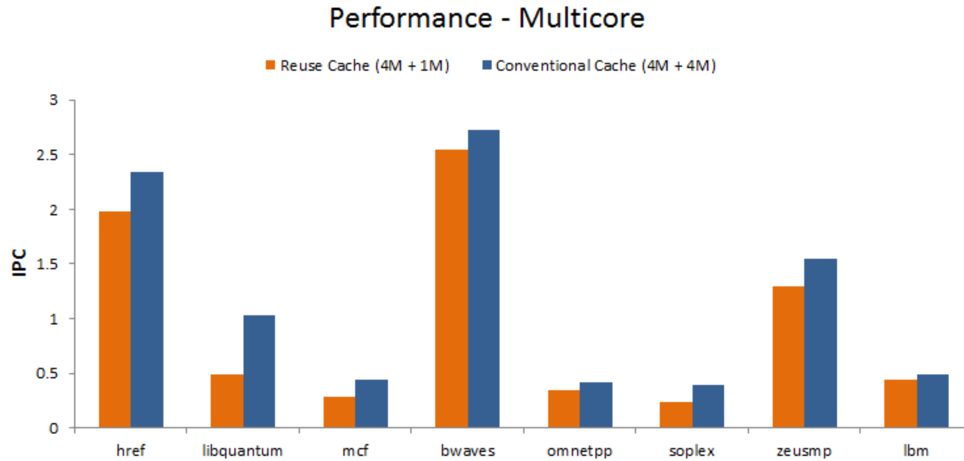

Fig. 13: Performance results for multi-core scenarios

well. Finally, the hardware involved in the lookup such as the multiplexer and comparator logic is not directly proportional to the size of the cache.

The reduced data array also greatly decreases the leakage power of the cache. However, this reduction in leakage power is offset by the increase in DRAM access energy due to two DRAM accesses for every reused cache line. Figure 9 shows the relative gain in power consumption for our reuse cache. We see an average power reduction of 10% in SPEC benchmarks. The power results for the reuse cache should improve with simulation time as the cost of the two DRAM accesses will be amortized across the number of hits made to these reuse lines.

	Conventional (4M)	Reuse (4M+1M)
Area (in mm2)	2.37	1.5
Access Latency (in ns)	2.97	3.28

TABLE IV: Cache area and latency comparisons

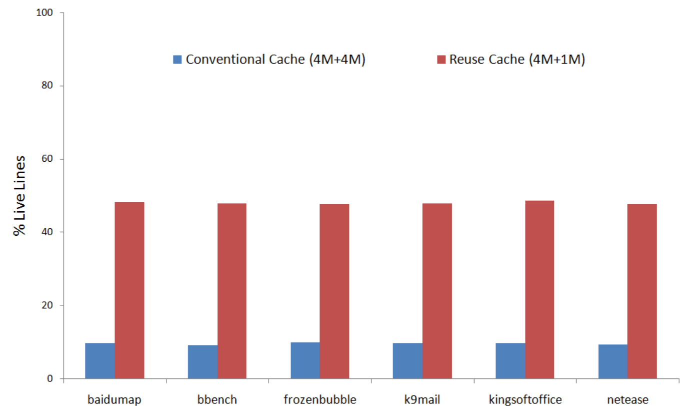

Fig. 14: Live lines

F. Improvement in Cache Utilization

Reuse Cache aims to improve the utilization of the cache by fetching only relevant lines from DRAM. This improvement is clearly visible in figure 14. The number of live lines is taken as the average of the two bounds proposed in Section II. We observe that the average number of live lines improves from 9.54% in the conventional cache to 47.95% in the reuse cache implementation.

G. Comparison with the Original Paper

The original paper chose a baseline model with a conventional 8MB, 16-way set associative L3 cache. A major difference between our work and the original study is that we focus on mobile processors fow which L2 cache is the SLLC. The performance evaluation in the original study was done by simulating SPEC benchmarks for 700 million instructions, following a warmup period of 300 million instructions. This is a very short amount of time to evaluate the performance of reuse cache, as the benefits of this design come from repeated accesses to the reused lines. Also, there will not be sufficient accesses to the L3 cache in their simulations, as the L1 and L2 caches will serve most of the memory accesses. With this approach, they demonstrate that the 4MB + 1MB reuse cache shows performance comparable to an 8MB conventional cache. In the current project, we simulated our workloads for 10 billion instructions with a warmup period of 300 million instructions. This setup is far more suited to accurately evaluating the performance of our design, as the number of L2 misses seen is a couple of orders of magnitude greater than the number of cache blocks in the L2 cache. Our results showing a performance degradation of about 5-10% are less impressive than the results reported by Albericio et al, but we believe our results are more indicative of the actual performance of the reuse cache model.

The original paper also claimed 84% reduction in the cache area, but this was based on a back of the envelope calculation by considering only the number of bits saved in the reuse cache design. Our results of 50% area savings, generated by modelling our cache design in Cacti v6.5, are significantly more representative of the actual area benefits offered by the reuse cache. This current work also focuses on the power consumption of the reuse cache, which was an aspect not

considered in the original study.

VII. Related Work

In addition to the original paper on reuse caches, there are two other studies related to our project [8] [9]. These exploit the decoupling of tag and data arrays to retain the inclusion property of tags while modifying the data allocation policy on a cache miss. Multiple tag/data decoupling designs are proposed in the non-inclusive cache, inclusive directory [NCID] architecture [9]. One such proposal does reduce cache size by allocating tag as well as data for a random X% of the cache lines, while maintaining the tag information for the other lines. Alberico et al found the performance of such a design to be comparable to their reuse cache design for the case of X=5. A selective data allocation policy was employed by Lodde et al [8]. They categorized cache lines as shared or private based on the coherence state, and used this information to selectively allocate data for shared lines. However, this approach doesn't consider reuse locality, and allows transient cache lines to pollute the cache contents.

VIII. Conclusion and Future Work

From this work, we believe that the reuse cache is an idea that demonstrates great potential for optimizing the L2 cache of a mobile processor. The observations regarding the efficiency of the SLLC made in the original reuse cache paper have been validated for the mobile environment, and about 85-90% of L2 cache lines are found to be dead. Two techniques have been proposed to study the live-ness of cache lines with better accuracy. Extra tag bits available in the reuse cache have been exploited to incorporate the Shadow Directory for improved replacement. The decoupled tag and data organization could be explored further to look into various replacement policies and their impact on performance. Better ideas need to be found to further exploit the decoupling of the tag and data arrays, as well as use the extra tag entries.

Realistic performance estimation has been performed here, by simulating 10 billion instructions after boot in gem5's Full System mode. For the purpose of this study, the simulation infrastructure was created from scratch. Enhancing the infrastructure to run simulations more efficiently using simpoints and to evaluate reuse cache on multi-core systems would lead to a more practical study of the proposed design.

Cache area is computed using Cacti, which results in highly accurate area estimates. A crucial area that needs to be explored further is the energy analysis of the reuse cache design. The original work by Albericio et al did not discuss the power/energy impact of reuse cache. The reuse cache leads to increased DRAM accesses, which are very expensive in terms of energy. For a constrained energy and power environment, like a mobile SoC, this could be a cause for concern. The power numbers presented in the paper, have been generated by incorporating a very large error margin. This results in a modest power improvement of 10%. However, an accurate study for such a design calls for detailed modeling of the reuse cache design in McPAT. Given the limited time frame of the project, this study has been marked for future work.

Thus, we see that for mobile workloads, the reuse cache promises significant cache area reduction of about 50% and decent power savings of 10% for the marginal performance loss of about 6%.

Acknowledgment

The authors would like to thank Professor David Wood, for taking us on a grand tour of computer architecture through the CS752 course at UW-Madison, and for guiding us as we worked our way through this project.

References

[1] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería, "The reuse cache: Downsizing the shared last-level cache," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 310–321. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540735

[2] J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio, "Prefetching mechanism for a high speed buffer store," Oct. 9 1985, eP Patent App. EP19,850,102,204. [Online]. Available: http://www.google.im/patents/EP0157175A2?cl=ru

[3] Y. Huang, Z. Zha, M. Chen, and L. Zhang, "Moby: A mobile benchmark suite for architectural simulators," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, March 2014, pp. 45–54.

[4] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería, "Exploiting reuse locality on inclusive shared last-level caches," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 38:1–38:19, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400697

[5] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 469–480.

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

[7] A. Gutierrez, J. Pusdesris, R. Dreslinski, T. Mudge, C. Sudanthi, C. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, March 2014, pp. 13–22.

[8] M. Lodde, J. Flich, and M. E. Acacio, "Dynamic last-level cache allocation to reduce area and power overhead in directory coherence protocols," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 206–218.

[9] L. Zhao, R. Iyer, S. Makineni, D. Newell, and L. Cheng, "Ncid: A non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 121–130.