Note that the two boolean operations ($(a < b)$ and $(a >= b)$) are mutually exclusive, being interpreted internally by the compiler as 0 (false) or 1 (true). So, assuming that ***a*** is smaller than ***b***, the result of the algebraic operation is ***(1) * a + (0) * b*** which, ultimately, will return only the value of ***a***.

The same strategy can be applied to lines 19 to 23 of Listing 1.1, that represent the third step of the first stage. The new code is shown in Listing 1.4, where $iLocalSize$ stores the number of active local work-items and iLI represents the *work-item's* local identifier.

Here, in each iteration of the loop, $iPos$ is divided by 2 ($iPos >>= 1$) and $bFlag$ is expanded to either 1 or 0, thus reducing by half the number of *work-items* doing a useful job. If, for the current *work-item*, the expression $iLI < iPos$ becomes true, then the expression in the last line will be interpreted as $scratch[iLI] + = (1) * (scratch[iLI + (1) * iPos])$, ensuring that the value stored in position $iLI + iPos$ will be added to the value in position iLI. On the other hand, if the expression becomes false, it will be interpreted as $scratch[iLI] + = (0) * (scratch[iLI + (0) * iPos])$, ensuring that the value in position iLI will not be considered. Since all *work-items* are always in the same step of computation – doing exactly the same job (useful or not), independently of being in the same wavefront – sync barriers are unnecessary.

Listing 1.4. Avoiding Divergences

```
for (iPos = iLocalSize/2; iPos > 0; iPos >>= 1)
{
 bFlag = iLI < iPos;
 scratch[iLI] += (bFlag)*(scratch[iLI + (bFlag)*iPos]);
}
```

5 Computational Experiments

In this section we compare the new approach against the proposals of Catanzaro, Harris and Luitjens. Their original codes were publicly available and, therefore, used in the current study. All algorithms, the existing ones and our proposal, are in C++ language. Catanzaro's code uses OpenCL, as previously described. Harris and Luitjens are in CUDA. We performed experiments for comparing the performance of the codes in two GPU platforms: an AMD GPU and a NVIDIA GPU. For this aim, two versions of our approach were implemented: one in OpenCL and the other in CUDA.

For the first platform, we used a computer with an AMD FX-9590 Black Edition Octa Core CPU, with clock ranging from 4.7 GHz to 5.0 GHz, 32 GB of RAM and running Ubuntu 16.04 64-bits operating system. The computer had a Radeon SAPPHIRE R9 290X GPU video card, with 4 GB of memory. The architecture of the video card provides 2816 stream processing units and an enhanced engine clock of up to 1040 Mhz. Its memory is clocked at 1300 MHz (5.2 GHz effectively). At such speed, the theoretical GPU memory bandwidth is 332.8 GB/sec. The programming codes for this platform used OpenCL 1.2 with the Software Development Kit 2.9.1.

mazalves@inf.ufpr.br

For the second platform, we used an Intel Xeon CPU E5-2650 computer, with 20 Cores (40 visible cores using hyper thread, each clocked at 2.3 GHz), with 128 GB of RAM and running Ubuntu 16.04 64-bits operating system. The computer was equipped with a NVidia Tesla K40m GPU with 11 GB of memory. This GPU architecture has 15 Multiprocessors with 192 CUDA Cores per MP, summing up 2880 CUDA cores. Each core has a clock of 745 MHz and the GPU's memory is clocked at 3.0 GHz. We ran only CUDA codes at this hardware, which were compiled using CUDA SDK 7.5.

For both platforms, we used the GNU g++ compiler version 4.8.2 and the compilation parameters "`-O3 -mcmodel=medium -m64 -g -W -Wall`".

All tests were run on two types of vectors, one of integers and one of single precision floating points. There were no measurable differences, regarding the execution times, between these types. We refer to vectors with 5,533,214 elements[3], except when another size is explicitly mentioned in the text.

Table 2 and Figs. 1 and 2 depict the speedup gains achieved by our OpenCL code against the code presented by Catanzaro at [2]. The performance of Catanzaro's approach is shown at the first data line of the Table. The results for our code appear in the next lines, for increasing values of the unrolling factor F. The values were obtained with the OpenCL CodeXL profiler version 2.0.12400.0, and are the averages of five consecutive executions for each test case.

As can be seen in the table, our implementation for $F = 1$ is already faster than Catanzaro's code. This is due to the optimization strategies implemented at steps 1 and 3, as described in Sect. 4. Our approach performs even better when the unrolling factor increases, reaching a *speedup* close to 2.8x when $F = 8$. It can also be noted that the *speedup* stabilizes around this value (for $F = 16$, the gain is around 1.5% when compared to the result for $F = 8$)[4].

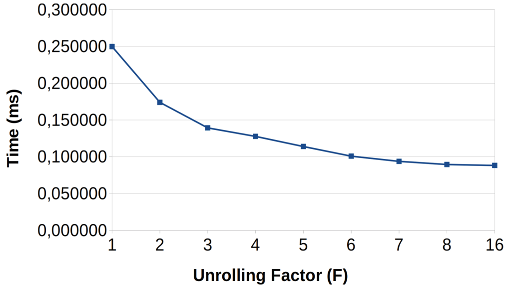


Fig. 1. Chart of the parallel reduction execution time.

[3] Reduction operations on vectors with millions of values are common when computing cost function in some real-life optimization problems. For instance, traffic assignment computation for large urban networks involves such vectors.

[4] The ideal unrolling factor strongly depends on the GPU model. In preliminary tests, performed in older hardware not listed here, the highest gains were obtained when F was defined as 6.

mazalves@inf.ufpr.br

Table 2. Parallel reduction execution times. New approach compared against Catanzaro's original code.

F	Time (ms)	Speedup	Memory bandwidth (GB/s)	Bandwidth usage (%)
–	0.294679	–	75.1083585868	22.57
1	0.249780	1	88.6094002722	26.63
2	0.173930	1.4360949807	127.2515149773	38.24
3	0.139260	1.7936234382	158.9318971708	47.76
4	0.127700	1.955990603	173.3191542678	52.08
5	0.113930	2.1923988414	194.2671464935	58.37
6	0.100810	2.4777303839	219.5502033528	65.97
7	0.093740	2.6646042245	236.1089822914	70.95
8	0.089490	2.7911498491	247.3221142027	74.32
16	0.088160	2.8332577132	251.0532667877	75.44

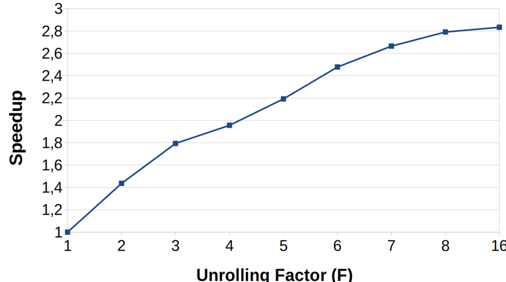


Fig. 2. Chart of the parallel reduction speedup.

Furthermore, the version of our approach implemented in CUDA was compared against Harris' Kernel 7 and Luitjens' code, in the second platform. The experiments also employed the two aforementioned vectors. Several values of the unrolling factor (F) were tried in our code, in order to find the optimal value for the K40m video board. It was determined that up to $F = 6$ the performance gains are substantial and, with $F \geq 8$, the gains are very discrete. According to this, all experiments were conducted using $F = 8$, including the one presented by [11].

Table 3 shows the running time (in milliseconds) of the three approaches and the percentage of performance (given by the formula $100 * (1 - \frac{T}{T_H})$, where T is the execution time of the method in the given line and T_H is the execution time of Harris' code, as a reference. Our CUDA implementation runs in almost the same time of Harris' code. Luitjens approach, using the SHFL instruction in CUDA for the K40m GPU, outperforms Harris' code by 25.98%.

mazalves@inf.ufpr.br

Table 3. Parallel reduction execution times in the K40m.

Approach	Exec. time (ms)	% of Gain
Harris (Kernel 7)	0.19032	0
New Approach (CUDA. F = 8)	0.18009	5.37
Luitjens	0.14086	25.98

Table 4 illustrates how the running time of the algorithms (also in milliseconds) change as the size of the vector doubles. For all cases the new approach is slightly better than Harris' method. Luitjens outperforms both methods, constrained to use a hardware with the SHFL instruction.

Table 4. Evolution of the execution times in the K40m.

Vector size	Harris (Kernel 7)	New approach (CUDA. F = 8)	Luitjens
2766607	0.18515	0.17941	0.07626
5533214	0.19032	0.18009	0.14086
11066428	0.29604	0.28219	0.24311
22132856	0.57268	0.53652	0.46887
44265712	1.13165	1.05426	0.91617
88531424	2.25516	2.09966	1.81140
177062848	4.51196	4.20577	3.60514

6 General Remarks

All parallel reduction techniques currently in use suffer from some basic issues. Many of them only reach their peak performance by employing proprietary strategies/technologies. That limits their use to the platforms for which they were designed. Others, although generic, do not adopt certain procedures that could increase their performance without loss of generality.

In the present work, we explored a combination of strategies that could improve the performance of the original Catanzaro's code for parallel reduction. One highlight should be made to the employment of attributions with algebraic expressions instead of conditional statements, in order to minimize or eliminate the phenomenon of thread divergence and to avoid the use of synchronization barriers.

The strategies presented in this paper are generic enough to be used with both CUDA and OpenCL and can run on hardware of the two major GPU manufacturers with minimal changes, just being adapted to the particularities of each platform.

mazalves@inf.ufpr.br

The experiments that were carried out showed that the execution times of our approach and of Harris' are very similar. Therefore, we assume them to be equivalent. It was also possible to verify that Luitjens' proposal is more efficient than the other approaches. We advise to choose Luitjens' method if support to the SHFL instruction is available. For GPUs with no SHFL support, the new implementation described here is more advantageous, since it provides equivalent performance to Harris' approach [11] but with a code that is easier to implement and works for both CUDA and OpenCL.

It is worth mentioning that the discussed techniques are of general use, being reduction only one of its applications.

For future work, we intend to explore the benefits of SHFL-equivalent instructions that appear in recent AMD GPUs in our new OpenCL code.

References

1. Billeter, M., Olsson, O., Assarsson, U.: Efficient stream compaction on wide SIMD many-core architectures. In: Proceedings of the Conference on High Performance Graphics 2009, HPG 2009, pp. 159–166. ACM, New York (2009). https://doi.org/10.1145/1572769.1572795
2. Catanzaro, B.: OpenCL optimization case study: simple reductions, August 2014. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/. Published by Advanced Micro Devices. Accessed 05 Jan 2014
3. Chakroun, I., Mezmaz, M., Melab, N., Bendjoudi, A.: Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. Concurr. Comput.: Pract. Exp. **25**(8), 1121–1136 (2013)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
5. Fog, A.: Optimizing subroutines in assembly language: an optimization guide for x86 platforms. Technical University of Denmark (2013)
6. Fung, W.W.L., Sham, I., Yuan, G., Aamodt, T.: Dynamic warp formation and scheduling for efficient GPU control flow. In: 2007 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2007, pp. 407–420, December 2007. https://doi.org/10.1109/MICRO.2007.30
7. Goldberg, D.: What every computer scientist should know about floating-pointarithmetic. ACM Comput. Surv. **23**(1), 5–48 (1991). https://doi.org/10.1145/103162.103163
8. Khronos OpenCL Working Group, et al.: The OpenCL specification. Version 1(29), 8 (2008)
9. Gupta, K., Stuart, J.A., Owens, J.D.: A study of persistent threads style GPU programming for GPGPU workloads. In: Innovative Parallel Computing (InPar), pp. 1–14. IEEE (2012)
10. Han, T.D., Abdelrahman, T.S.: Reducing branch divergence in GPU programs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, pp. 3:1–3:8. ACM, New York (2011). https://doi.org/10.1145/1964179.1964184
11. Harris, M.: Optimizing Parallel Reduction in CUDA (2007). https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-parallel-reduction. Published by NVidia Corporation. Accessed 10 Sept 2018

mazalves@inf.ufpr.br

12. Higham, N.: Accuracy and Stability of Numerical Algorithms, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia (2002)
13. Huang, J.C., Leng, T.: Generalized loop-unrolling: a method for program speed-up. In: Proceedings of the IEEE Symposium on Application-Specific Systems and Software Engineering and Technology, pp. 244–248 (1997)
14. Kiefer, J.C.: Sequential minimax search for a maximum. Proc. Am. Math. Soc. **4**, 502–506 (1953)
15. Luitjens, J.: Faster parallel reductions on Kepler. White Paper, February 2014. http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/. Published by NVidia Inc., Accessed 25 July 2014
16. Meng, J., Tarjan, D., Skadron, K.: Dynamic warp subdivision for integrated branch and memory divergence tolerance. SIGARCH Comput. Archit. News **38**(3), 235–246 (2010). https://doi.org/10.1145/1816038.1815992
17. Muller, J., et al.: Handbook of Floating-Point Arithmetic. Birkhäuser, Boston (2009)
18. Narasiman, V., Shebanow, M., Lee, C.J., Miftakhutdinov, R., Mutlu, O., Patt, Y.N.: Improving GPU performance via large warps and two-level warp scheduling. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, pp. 308–317. ACM, New York (2011). https://doi.org/10.1145/2155620.2155656
19. Nasre, R., Burtscher, M., Pingali, K.: Data-driven versus topology-driven irregular computations on GPUs. In: 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), pp. 463–474 (2013). https://doi.org/10.1109/IPDPS.2013.28
20. Parhami, B.: Introduction to Parallel Processing Algorithms and Architectures. Plenum Series in Computer Science. Plenum Press, London (1999). http://books.google.com/books?id=ekBsZkIYfUgC
21. Sarkar, V.: Optimized unrolling of nested loops. Int. J. Parallel Program. **29**(5), 545–581 (2001). https://doi.org/10.1023/A:1012246031671
22. Steinberger, M., Kainz, B., Kerbl, B., Hauswiesner, S., Kenzel, M., Schmalstieg, D.: Softshell: dynamic scheduling on GPUs. ACM Trans. Graph. **31**(6), 161:1–161:11 (2012). https://doi.org/10.1145/2366145.2366180
23. Wilt, N.: The CUDA Handbook: A Comprehensive Guide to GPU Programming. Pearson Education, White Plains (2013)
24. Zhang, E.Z., Jiang, Y., Guo, Z., Shen, X.: Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, pp. 115–126. ACM, New York (2010). https://doi.org/10.1145/1810085.1810104

mazalves@inf.ufpr.br

Power and Energy

Evaluating Cache Line Behavior Predictors for Energy Efficient Processors

Rodrigo Machniewicz Sokulski(✉), Emmanuell Diaz Carreno, and Marco Antonio Zanata Alves

Federal University of Paraná, Curitiba, PR, Brazil
{rms16,edcarreno,mazalves}@inf.ufpr.br

Abstract. The ever-increasing size of cache memories, nowadays achieving almost half of the area for modern processors, and so essential to the performance of the systems, are leading into a crescent static energy consumption. In order to save some of this energy and optimize its component performance, many techniques were proposed. Cache line reuse predictors and dead line predictors are some examples. These mechanisms predict whenever a cache line shall be dead, in order to turn it off, also applying other policies on them, such as replacement prioritization or bypassing its installation inside the cache. However, not all mechanisms implement all these policies, that directly affect the cache behavior in different ways. This paper evaluates the impacts of the priority and bypass policies over two dead line predictors, the Dead Block and Early Write Back Predictor (DEWP) and the Skewed Dead Block predictor (SDP). Both mechanisms turn off dead cache lines using Gated-Vdd technique in order to save their static energy, thus analyzing how each policy (Priority replacement and cache Bypass) affects the energy savings and the system performance.

Keywords: Cache memory · Energy efficient · Cache usage predictor

1 Introduction

As technology advances and processors run faster, the performance gap between CPUs and DRAMs increases, further increasing the relevance of cache memories. These tiny and fast memories between the processor and the DRAM memory act attenuating the high delay of DRAMs, being critical for computers efficiency. In order for the processor to achieve higher performance, these cache memories need to maintain the necessary processor data for the majority of accesses, thus avoiding accessing to slower memory levels. Following this logic, the caches grew in size, and nowadays occupy about 50% from processors chip, also leading to increases in energy consumption.

The energy consumption in CMOS circuits such as SRAM and DRAM memories can be divided fundamentally into two sources. The dynamic energy used to perform circuits gating, and the static energy, also called leakage energy, which is

© Springer Nature Switzerland AG 2020
C. Bianchini et al. (Eds.): WSCAD 2018, CCIS 1171, pp. 185–197, 2020.
https://doi.org/10.1007/978-3-030-41050-6_12

mazalves@inf.ufpr.br

spent even when there are no interactions. This second source is directly related to this component size. The bigger its storage capacity, the more static energy it spends. Therefore, the cache memories growth leads to a higher static energy consumption, increasing the importance of methods to save it.

Many cache line behavior predictors have already been proposed [2,6,10,12,13,15]. Most of these mechanisms try to predict when a cache line content receives its last access, in order to apply a static energy saving technique over it. Regarding energy saving techniques, there are multiples proposals in the literature, where the most commonly used technique is called Gated-Vdd technique [22]. The Gated-Vdd consists in turning off a cache line in order to save almost all the static energy used to maintain it. This technique is applied whenever the processing core associated with a cache line is disabled, then we can disable the whole cache lines. Such gains may increase by using these techniques together with the cache line usage predictors, helping predicting the cache usage with cache line granularity. Despite the high savings, these shutdowns can directly increase the applications execution time.

Besides the cache line shutdowns, dead cache line predictors also allow cache line installation and replacement improvements, producing higher cache hit ratios. Whenever a cache line is predicted to be dead-on-arrival (i.e., the line is predicted to receive only a single access), that cache line can be bypassed, not being installed inside the cache, thus reducing the cache pollution [4]. For the cases where the cache line is predicted to have multiple accesses, whenever a cache line is predicted to be dead (i.e., it is not going to receive any further access until its eviction) the cache eviction policy can increase its eviction priority, in order to keep for longer the live ones. Nevertheless, during wrong predictions, these modifications in the installation and replacement policies may have a negative influence on energy consumption and the execution time from the programs, because data that should be kept in the cache is going to be fetched again from the main memory.

In this paper, our objective is to evaluate the influence caused by the early cache line eviction and the bypass techniques in the last level cache when using information from cache line predictors. We evaluate two cache dead line predictors, the Dead Block and Early Write Back Predictor (DEWP) [2] and the Skewed Dead Block Predictor (SDP) [13]. Our focus is to assess and understand the impact of these two techniques in the base implementation of each one of the cache line usage predictors evaluated, in terms of energy consumption and execution time. Our main contributions are:

Implementation: we present the idea of merging the DEWP and SDP mechanisms with the dead line priority eviction and the bypass policies.

Energy Impact: we evaluate different combinations of mechanism and policies that save from 30% to 60% of the energy consumed by a traditional cache and DRAM memory sub-system with performance variation of ±2%, offering an interesting trade-off between energy savings and performance.

mazalves@inf.ufpr.br

2 Motivation

The increase of static energy consumption in cache memories [1] leads to the research of many dead line predictors [2,8,13–15]. These mechanisms try to predict when a cache line receives its last access in order to turn it off. In addition to this, the predictions can be used to change the cache lines install and replacement policy, prioritizing dead lines and making bypass from dead-on-arrival lines. According to each application's memory access pattern and the predictor accuracy, these policy change can lead to gains or losses in an execution.

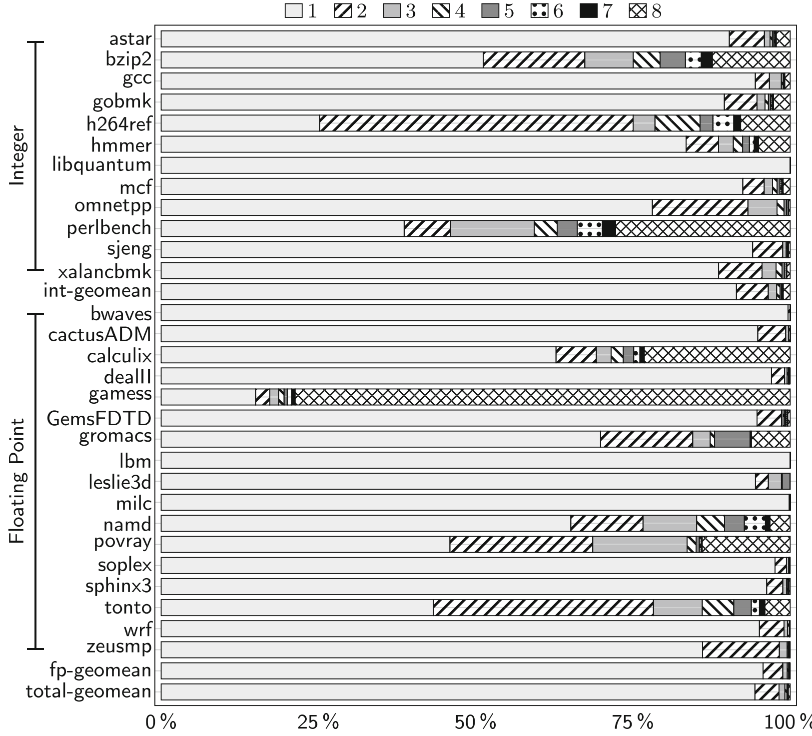
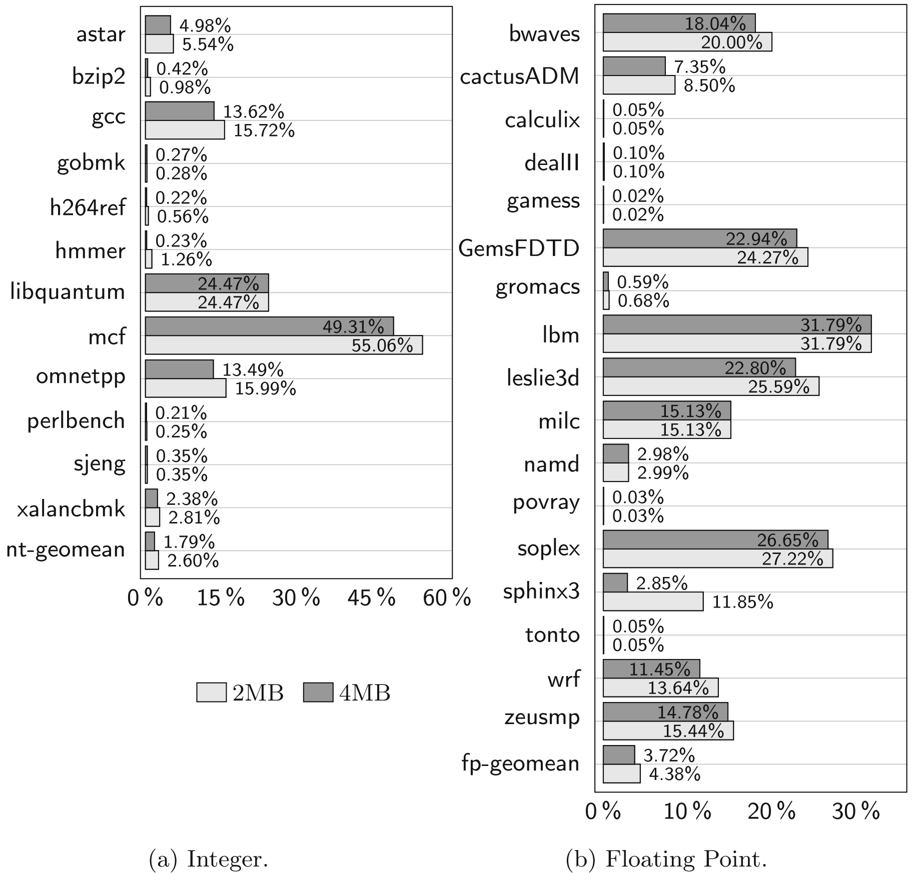


Fig. 1. Number of LLC accesses before eviction

We designed two main experiments using all the 29 applications from the SPEC-CPU 2006 benchmark suite, to understand the possible impact when prioritizing dead lines and performing bypass from dead-on-arrival lines. Further methodology details are present in Sect. 4.

In the first experiment, we evaluate the number of accesses per cache line before it gets evicted from the LLC. Figure 1 shows a histogram that counts the number of accesses each cache line from a 2 MB LLC received before it was invalidated. We can observe that on average more than 95% of the last level

mazalves@inf.ufpr.br



(a) Integer. (b) Floating Point.

Fig. 2. Misses per kilo instruction (MPKI) showing potential improvements when increasing the cache size.

cache lines receive only single accesses and then they can be considered dead. From this first plot, we can learn it is possible to bypass multiple cache lines without harming the performance.

In the second experiment to motivate this work, we executed the same applications with two different LLC sizes, 2 MB, and 4 MB as can be seen in Fig. 2. Considering that both techniques, bypassing and prioritization of dead cache lines, will virtually increase the cache memory size, by reducing the number of dead cache lines inside the LLC, this second experiment aims to show the possible reduction in the MPKI (misses per kilo instruction) metric whenever the cache memory gets its size increased. We can see that MPKI reduces on average 14% when doubling the LLC size, showing the potential to achieve performance improvements.

mazalves@inf.ufpr.br

Our results from previous work [17,18] showed that for half of applications tested, more than 75% of the time, the circuit wasted static energy because the cache lines already received its last access and are waiting for its eviction. For most of the applications, a perfect mechanism could turn-off the cache lines for more than 40% of the execution time, with potential to save a high portion of static energy.

It is important to note that performance and energy consumption results will suffer significant influence from the dead cache line predictor accuracy. Whenever the mechanism correct predicts the cache line behavior, energy and time can be saved. On the other hand, during miss predictions, two different things can occur: (1) the system will consume extra time and energy to bring the cache line from the DRAM; or (2) the system will lack the opportunity to save energy and time.

Thus, we aim to evaluate the influence of dead lines priority and the bypass policy using two different dead cache line predictors, SDP and DEWP, in order to understand how each policy affects these dead line predictors.

3 Mechanisms and Techniques

This section details the dead cache line predictors DEWP and SDP, the gated-Vdd technique and the cache policies evaluated in this paper.

3.1 DEWP

The DEWP mechanism uses an access history table (AHT) with 512-entries, in order to store the number of reads and writes that a cache line content received before its last eviction. Based on historical data stored in the AHT, the predictor can determine when a cache line received its last access, declaring the line as dead. Moreover, it can reduce the memory controller pressure by detecting when the cache lines received their last write and performing early write-backs. Besides AHT, this predictor requires some additional information stored in each cache line, in order to make specific predictions to them.

Figure 3 illustrates the components requires by our DEWP implementation [17]. Unlike its original version, the read and write counters were replaced with an access counter, with the view to simplify the mechanism operation and analyze only its dead line predictions influence over the improved LRU and the bypass policy. The DEWP operations are triggered by some cache events; these events and its consequences are described below:

During a cache line hit:

- If the cache line and its *training* flag are on, the *access counter* from the AHT entry pointed by the *AHT pointer* in the cache line is incremented.
- If the cache line is on and the *training* flag is off, the *access counter* from the cache is decremented, denoting that one of the predicted accesses was performed.

mazalves@inf.ufpr.br