Investigating the Use of the Store-Buffer as a Filter-Cache

Ricardo Alves
Uppsala University
ricardo.alves@it.uu.se

Alberto Ros
Universidad de Murcia
aros@ditec.um.es

Stefanos Kaxiras
Uppsala University
stefanos.kaxiras@it.uu.se

David Black-Schaffer
Uppsala University
david.black-schaffer@it.uu.se

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**; **Pipeline computing**; **Multicore architectures**;

ACM Reference Format:
Ricardo Alves, Alberto Ros, Stefanos Kaxiras, and David Black-Schaffer. 2018. Investigating the Use of the Store-Buffer as a Filter-Cache. In *Proceedings of International Symposium on Microarchitecture (MICRO'18).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROBLEM STATEMENT

Long latency stores can have a significant impact in performance by delaying the committing of instructions and putting pressure on CPU resources. This problem is exacerbated on architectures with *TSO* (Total Store Order) memory ordering, since a long latency store at the head of the store-queue will delay committing of all subsequent stores [2]. A store-buffer mitigates this problem, by gathering long latency stores and allows them (and subsequent ones) to retire immediately [1]. The write-backs to L1 are later performed in program order as early as possible. Since this buffer needs to be accessed on every load (for correctness), and is faster and more energy efficient than an L1, it functions similarly to a filter-cache[3]. However, low store-buffer hit-ratios (figure 1) hinders performance and energy benefits expected of a filter-cache. Moreover, to minimize memory access latency, the store-buffer and L1 are probed in parallel (since the load is likely to miss in the store-buffer), eliminating any load dynamic energy improvement, independently of store-buffer hit-ratio.

In this work we investigate the possibility of using the store-buffer as a filter-cache without compromising its original purpose of hiding store-miss latency. This objective poses us with 3 challenges: (1) determine if and by how much can the store-buffer hit-ratio be improved (section 2); (2) how to minimize L1 cache accesses without increasing average load latency (section 3); and (3) how to delay stores in the store-buffer in order to maximize store-buffer hits without increasing CPU stalls (section 3).

MICRO'18, October 2018, Fukuoka, Japan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

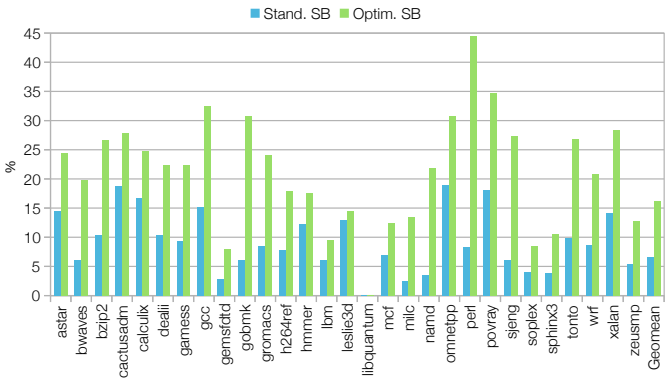

Figure 1: Percentage of loads that have their data forwarded (hit-ratio) from a standard store-buffer implementation and our optimal store-buffer (higher is better)

2 EVALUATING THE POTENTIAL – OPTIMAL IMPLEMENTATION

Given the store-buffer small size (42 to 44 entries)[1] one expects that to be the culprit of the low hit-ratios, but an occupancy analysis of the store-buffer (figure 2) shows that, while the store-buffer is stressed on all benchmarks at some point in their execution, the majority of the time the buffer remains underutilized (61.6% of the time, only 20% or less of the buffer is being used on average). The low occupancy is the result of stores being performed as early as possible. While this policy is optimal to minimize CPU stalls (the primary purpose of the store-buffer), it is detrimental to store-buffer hits. The low store-buffer hit-ratio thus seems to be associated with its write-back policy and not with its small size.

To determine the potential of the store-buffer as a cache, we implemented an optimal store-buffer policy that keeps the stores in the buffer as long as possible, only preforming the write-back to L1 when store-buffer entries are needed. To avoid any negative impact in performance, when a new entry in the buffer is needed and one can not be allocated due to a long latency store at the head of the buffer, this head-store is completed immediately to not cause a CPU stall. While unrealistic, this implementation should give us a ceiling on what to expect from a realistic implementation. The optimal solution showed an average store-buffer hit-ratio of 16.2%

[1] In modern X86_64 architectures, store-queue and store-buffer are unified and share the same physical structure. Distinction between the two is purely logical – uncommitted and committed stores.

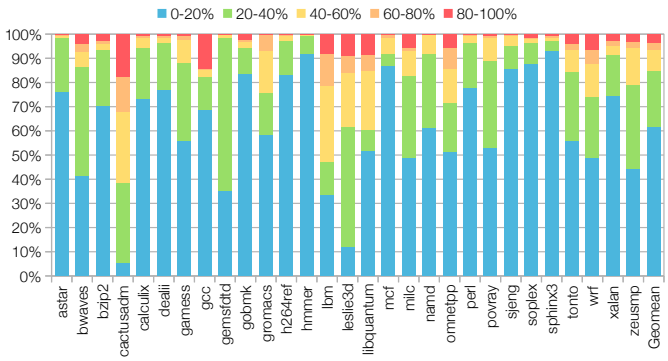

Figure 2: Unified store-queue/store-buffer occupancy

(vs the 6.6% of the standard implementation), and up to 44.5% in *perl* (figure 1).

3 FUTURE WORK – REALISTIC IMPLEMENTATION

The optimal implementation shows substantial potential for performance and energy efficiency improvement. However, an equivalent realistic implementation would have to overcome two obstacles: (2) when to probe the store-buffer or the L1, without increasing average access latency; and (3) when and for how long to delay stores in the store-buffer without causing any extra CPU stalls.

To overcome the first obstacle one cannot simply serialize the access to the store-buffer and L1 because, despite the improvement in store-buffer hit-ratio, the majority of the loads still miss in the store-buffer. A better alternative would be to predict where we expect the load to hit. Modern CPUs are already capable to indirectly do this prediction by using a *forward-predictor* [5]. Such a predictor is used to estimate if the data of a load is going to be provided by the memory hierarchy (store-buffer miss) or forwarded into the pipeline from the store-buffer (store-buffer hit) in order to properly schedule load-dependent instructions [4]. An academically established strategy is to use *store-distances* [6] to do the prediction.

The overcome the second obstacle one needs to (A) predict when a store is expected to miss, (B) make the prediction sufficiently early to have enough time to free entries in the store buffer, and (C) predict how many slots are required in the store-buffer so not to unnecessarily remove stores from the buffer. Unlike the *forward-predictor* no existing CPU predictor can be used, so a new one is required. At this time, we have not implemented any such predictor with a high enough accuracy.

REFERENCES

[1] Ravi Bhargava and Lizy K John. 2000. Issues in the design of store buffers in dynamically scheduled processors. In *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*. IEEE, 76–87.

[2] Yuan Chou, Lawrence Spracklen, and Santosh G Abraham. 2005. Store memory-level parallelism optimizations for commercial applications. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*. IEEE, 12–pp.

[3] Johnson Kin, Munish Gupta, and William H Mangione-Smith. 1997. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 184–193.

[4] Arthur Perais, André Seznec, Pierre Michaud, Andreas Sembrant, and Erik Hagersten. 2015. Cost-effective speculative scheduling in high performance processors. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 247–259.

[5] Sam S Stone, Kevin M Woley, and Matthew I Frank. 2005. Address-indexed memory disambiguation and store-to-load forwarding. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 171–182.

[6] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. 1999. Speculation techniques for improving load related instruction scheduling. In *ACM SIGARCH Computer Architecture News*, Vol. 27. IEEE Computer Society, 42–53.