Cost Effective Speculative Scheduling

Maninder Kumar, Adithya Pillai Ramesh, Harshal Pandit

I. Problem Statement

Modern-day processors have a significant latency between the issue and execute stages for activities like reading the register file and doing book-keeping duties. This latency is wasteful in terms of performance and dependents can be issued even before their respective source instructions are completed, this method is called **speculative scheduling**. This idea is lucrative because most of the instructions have a fixed latency nature, that is, they have very little or no variation in the amount of time it will take them to go from issue to writeback stage, giving us a potential for a significant performance improvement. The problem only arises with variable latency instructions (like loads) as they might (if it hits in L1 cache) or might not (if it misses in the L1 cache) finish in time, leading to a squash/replay of speculatively woken up instructions. In this project, we tried to analyze the problem, looked at different approaches in the design space of this problem and explored any potential improvements.

II. Mechanism for Speculative Scheduling in Gem5

The O3 processor in Gem5 is modelled on **Alpha 21264** out-of-order pipeline architecture. In the context of speculative scheduling, the pertinent pipeline stages are shown in *Fig.1* for non-memory and memory instructions.

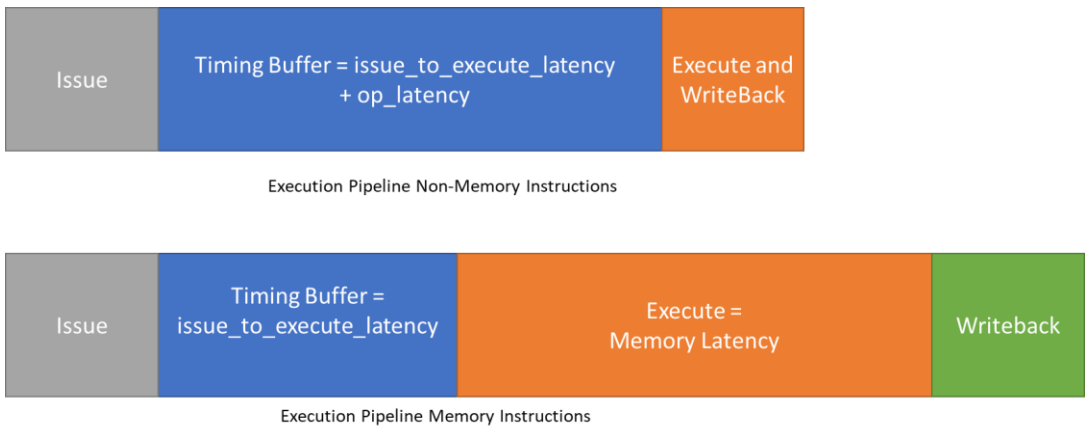
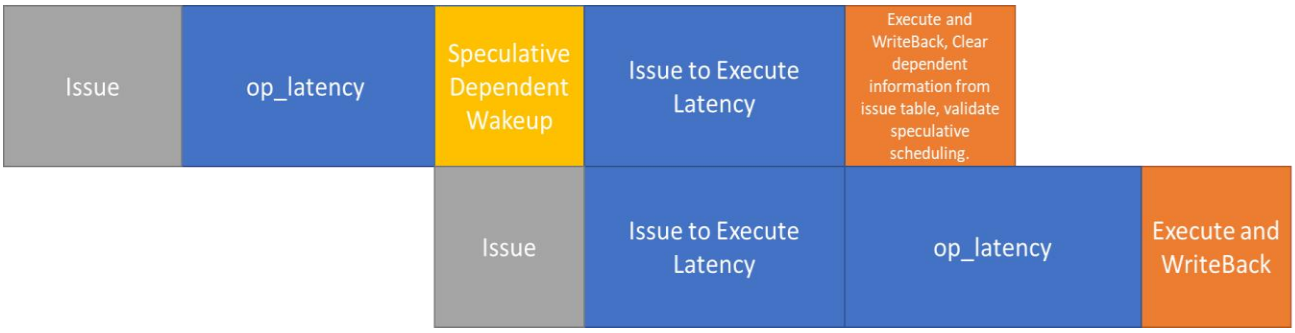
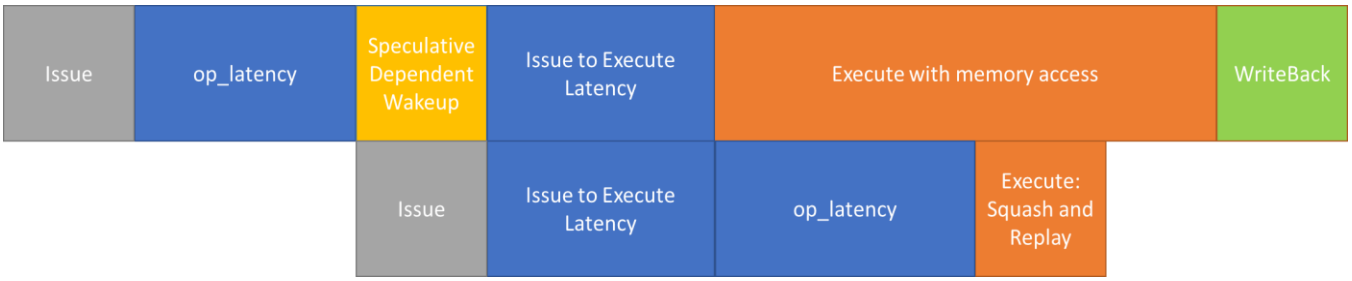


Fig.1: Execution Pipeline of Gem5 O3 Processor

The speculative scheduling mechanism that we implemented is modelled after the speculative scheduling architecture used in **Alpha 21264** [1][2]. In this mechanism, after being issued, an instruction is queued to wake up its dependents after a delay of *op_latency-1* in case of non-memory instructions and *op_latency-1 + L1_hit_latency* for memory instructions. On a speculative wakeup, the dependent instructions proceed towards the execute stage with their entry still in the issue queue and scoreboard. If it reaches the execute stage before all the source instructions have reached the writeback stage, the instruction is squashed and marked for replay. If all the source instructions have already reached the writeback stage before a dependent instruction reaches the execute stage, the instruction is allowed to continue. All the data structures for dependent tracking like scoreboard and issue queue are updated only when a source instruction reaches the writeback stage. A dependent instruction is removed from the issue queue when the last source instruction reaches the writeback stage. *Fig.2* depicts this process.



Speculative Scheduling with Correct Speculation



Speculative Scheduling with Mis-speculation

Fig.2: Speculative Scheduling Mechanism in Gem5 O3 Processor

III. Load L1 Hit Predictor

The problem with loads is that they might or might not finish in time depending on whether they hit in the L1 cache or not. For this purpose, to fine tune the mispredictions, a simple load predictor as mentioned in [1] can be employed. If a load is predicted to be a L1 hit, it is enqueued for speculative dependent wakeup.

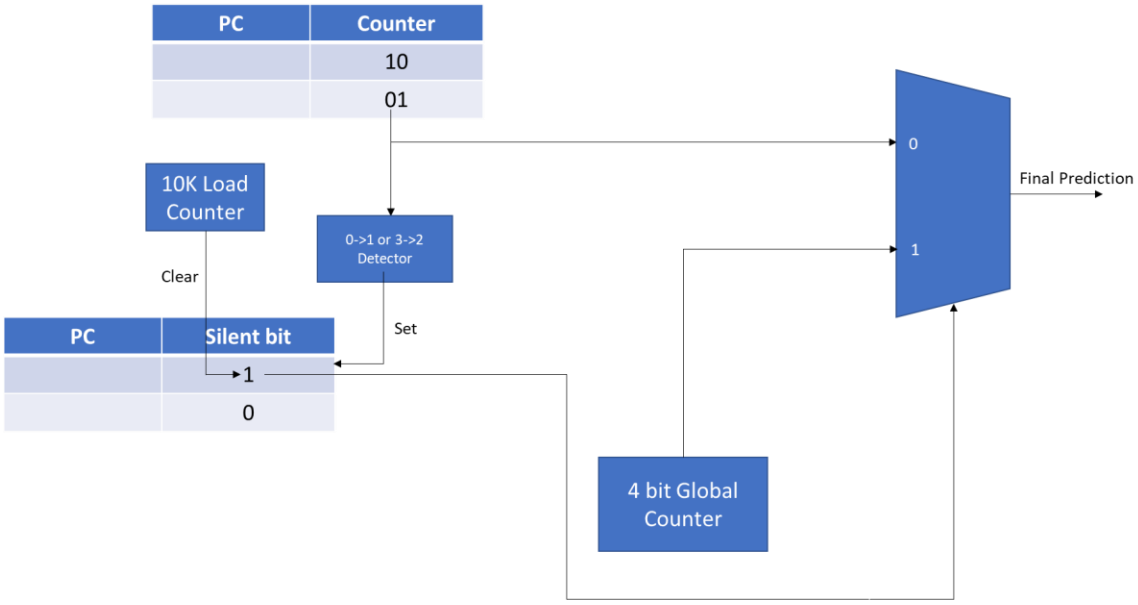


Fig.3: L1 Hit Predictor

The algorithmic structure of the predictor is shown in *Fig.3*. The predictor includes a 4-bit saturating global counter and a direct-mapped PC indexed 2-bit saturating history table for loads. For both the structures, the most significant bit tells the prediction. The mechanism to select between these two predictors is implemented through a silent bit table. Whenever an entry in the history table moves from a saturated state to a non-saturated state (0->1 or 3->2), the silent bit is set and the output is taken from the global predictor otherwise the output is taken from the history table entry. When the silent bit is set, the updates to that entry in the history table are also stopped. Further, all the silent bits are cleared at every 10K committed loads. The rationale behind using silent bits is the periodic change in the behavior of all the loads globally, like on a context switch or iteratively working on chunks of a large working set.

IV. Setup and Results

Due to some issues in our setup, we were able to run only 7 of the SPEC2006 benchmarks. Fig.4 shows the plot for %performance (IPC) gains for the benchmarks as compared to the baseline setup with *issue_to_execute_latency=4* and without *speculative scheduling.* We ran all the benchmarks for 50M instructions.

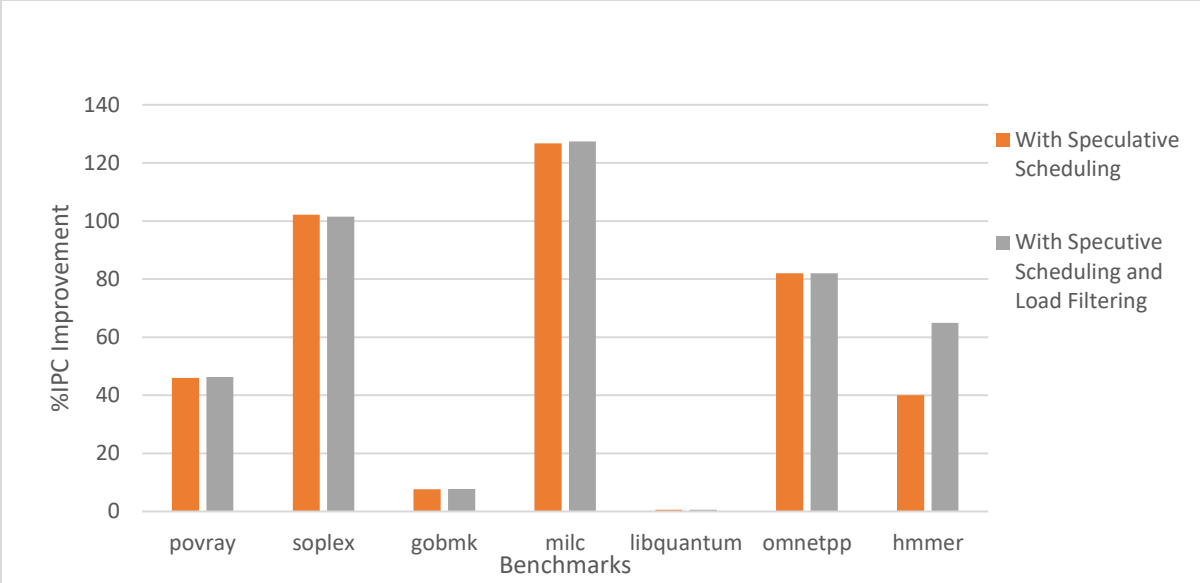


Fig.4: % IPC improvement by using Speculative Scheduling and Load Filtering (as compared to baseline processor with issue_to_execute_latency=4 and no speculative scheduling)

V. Analysis of Results

As mentioned in section IV, speculative scheduling leads to a significant performance improvement for most of the benchmarks. The performance enhancement by adding load filtering, however, is not that much. This observation can be imputed to the general latency cost analysis of making a misprediction in case of loads. If the predictor speculates a miss and the load hits in the cache, the cost of not waking up the dependents early is more than the cost of replay in case of predicting a hit and then the load missing in the cache as the later is overshadowed by the latency of fetching the load from the next level in the cache hierarchy. On the other hand, however, replayed instructions put a pressure on the limited issue width, thus the number of correct miss-miss predictions allow for a small performance improvement and significant energy improvement. This benefit can be seen in Fig.5 where the percentage decrease in the number of replayed instructions is plotted for each benchmark as the load filtering was used.

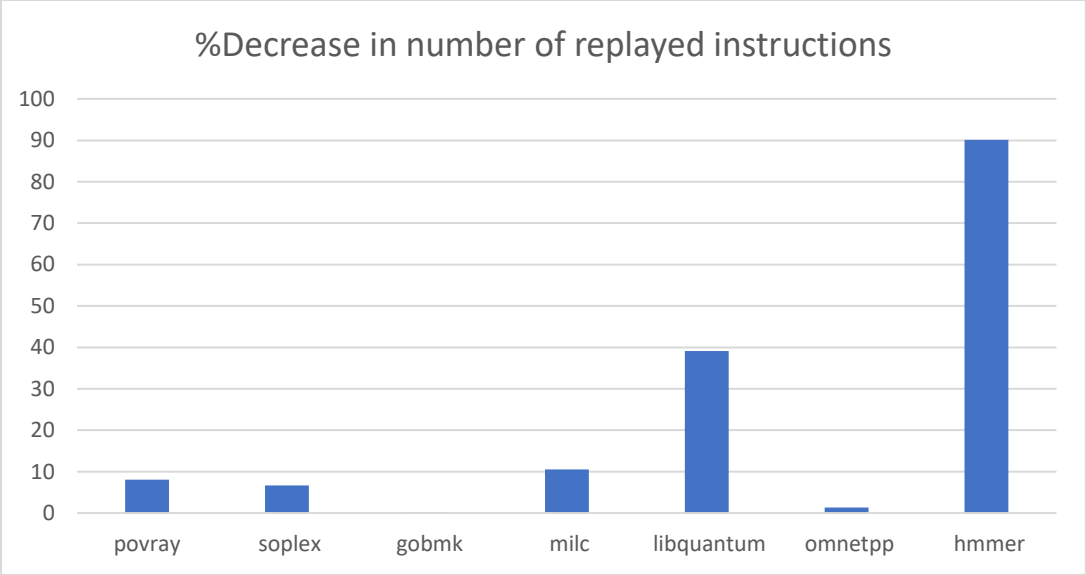


Fig.5: % Decrease in the number of replayed instructions with load filtering.

One of the notable observations here is that of *hmmer* which shows a significant reduction in the number of replayed instructions and thus has some good performance gained by using load filtering. *Table.1* notes the performance of predictor alone.

Benchmark	Instructions Replayed without load filtering	Instructions Replayed with load filtering	Correct Predictions	Incorrect Predictions Hit-Miss	Incorrect Predictions Miss Hit
povray	540109	496451	12496184	2280337	68557
soplex	10625882	9918455	9611312	3797602	49974
gobmk	818241	816714	6923026	1973216	15479
milc	123129078	110196814	12206399	971791	10801
libquantum	23	14	6339102	2958	637
omnetpp	174595	172200	5851499	596587	6231
hmmer	6251506	617942	13125416	5108404	11352

Table.1 Statistics for the Load Filter

There are some outliers like the *libquantum* which has a dismal performance gain even because of speculative scheduling. Also, the number of replayed instructions both with and without load filtering is quite low for *libquantum.* The % reduction in the number of replayed instructions is also quite low for *gobmk.*

VI. Conclusion and Future Work

We implemented speculative scheduling with load filtering in Gem5 and our initial analysis of the behavior of this setup draws the attention to two important overall conclusions:

1. There is a significant performance speedup by using speculative scheduling alone.
2. The design space for load filter has two major power and performance parameters:
    a. Hit-Miss Mispredictions: These mispredictions increase the number of replayed instructions thus lead to a small performance drop due to bandwidth pressure on issue stage. These can have, however, a significant impact on energy.
    b. Miss-Hit Mispredictions: These mispredictions have a significant impact on performance as the cost of predicting a miss and then hitting in the cache is significant.

Point number 2 gives us an interesting direction to work by analyzing the behavior of loads and designing an optimum predictor to yield significant performance gains. [1] also mentions another reason for load dependent replays which is bank conflicts. This problem is targeted in [1] by using shift scheduling. We also wanted to try this idea, but could not due to steep ramping curve on gem5 and time constraints. From the data collected by us, outliers like *libquantum* and *gobmk* can further be analysed to pinpoint the exact reason for their peculiar behavior. All these directions have a good potential for future work.

VII. Code Repository

https://github.com/mandius/gem5_experiments.git

A *readme.md* is present in the top directory of this repository.

References

[1] A. Perais, A. Seznec, P. Michaud, A. Sembrant and E. Hagersten, "Cost-effective speculative scheduling in high performance processors," *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, USA, 2015, pp. 247-259, doi: 10.1145/2749469.2749470.

[2] I. Kim and M. H. Lipasti, "Understanding scheduling replay schemes," 10th International Symposium on High Performance Computer Architecture (HPCA'04), Madrid, Spain, 2004, pp. 198-209, doi: 10.1109/HPCA.2004.10011.