Predicted Performance for Coherence Decoupled Systems and Rollback Estimations

Xiaoyang Gong xgong35@wisc.edu
Reese Kuper rkuper@wisc.edu
Jinglin Liang jliang54@wisc.edu
Naga Rahul Yarramsetty yarramsetty@wisc.edu
McKinley Sconiers-Hasan msconiershas@wisc.edu

I. Abstract

This project focuses on implementing a decoupled cache for approximating loads, making a predictor to recover accuracy, and estimating the benefit of a rollback to guarantee accuracy for when the predictor is wrong. These goals were all aimed to mitigate the costs of false sharing. This occurs when the cache coherence protocol invalidates a block when a different cache wants exclusive access, but the shared caches being invalidated want to load data from separate sections of that same block. False sharing hurts performance as loading the block again would become a cache miss to get the updated values. Our research looked into reducing these performance penalties by speculatively loading the stale values based on the predictor and rolling back if the stale value was modified by the cache with exclusive access to the block. Missed predictions are fine as the predictor's purpose is to only predict if a rollback is necessary.

Our findings showed that the predictor's accuracy was sufficient to implement a rollback by a significant margin. From finding the average GetS request times, the predictor only needed to be 30% accurate on our tested systems and given assumptions. The tests we ran were simulated in gem5 using MESI and MSI directory-based coherence protocols and two separate false sharing scripts - critical-fs and fs-reduction.

II. Introduction

Approximate computing is a paradigm of computer architecture that is becoming well researched, especially in recent years. Approximations allow for a percentage of error in program output, but it can yield increases energy efficiency and performance. The trade off is obvious with the main goal of minimizing error, while still aiming for the benefits of energy and time. One recent paper [1] proved a coherence modification enabling approximate computing and observed notable speedups as a result.

An area of improvement on their design would be to use a predictor to mitigate the loss in accuracy. The predictor's purpose would be to try to accurately guess whether or not the approximate load would produce an error. Knowing this, a coherence protocol would then be able to run through its standard protocol if the error is predicted, while the non-error prediction would allow caches to use the old, stale values instead of waiting for the guaranteed accurate data.

III. Implementation

A. Read Invalid Line (RIL) Predictors

The default RIL scheme described in [1] always reads data from invalid cache lines, as long as that cache line exists in cache. This always-taken invalid cache line approach would work well if the workload has many false sharing instances, but for general programs that have both true shares and false shares, this approach may fail.

To increase prediction accuracy for the general workload, the first improvement is to use prediction FSM to dynamically decide and update RIL prediction result for each address or memory region. Figure 1 demonstrates the setup for one-level predictor with a predictor table. Our RIL predictor table is indexed by memory address and contains entries that store taken/non-taken decisions. Once the cache coherence FSM receives data acknowledgement from either the directory or sharers, it knows whether there is a true-share or false-share and updates the prediction table accordingly using one of the five FSMs [4] shown in Figure 1. In addition, the prediction table does not need to store RIL predictions for every address. It can take lower bits of address as don't care bits and thus reduce the size of table. For example, a cache with 64 bytes cache line size could set last 6 bits of address as don't cares. The prediction table would be index by cache lines instead of individual addresses.

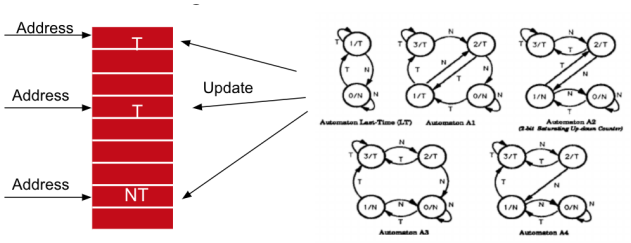

Fig. 1. Possible FSMs and their use for updating prediction data for various addresses.

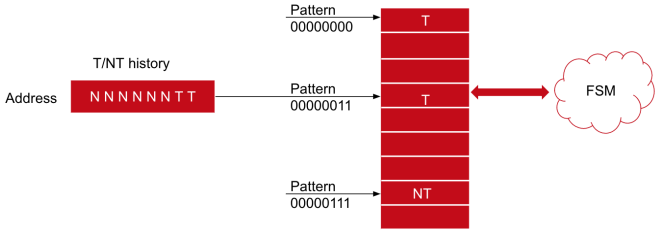

Fig. 2. Possible FSMs and their use for updating prediction data for various addresses.

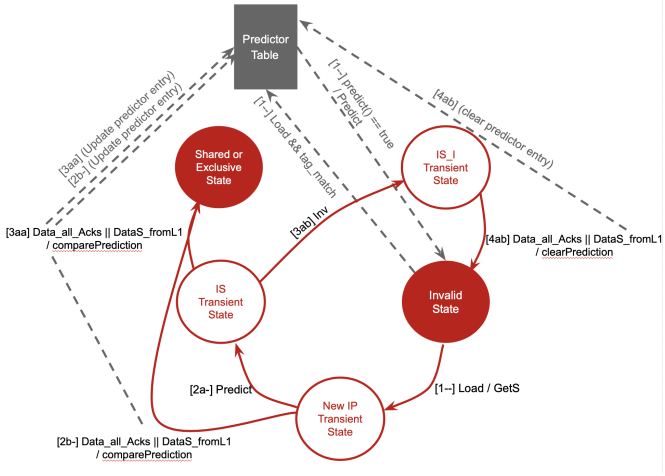

Fig. 4. All changes made for approximate loads using a predictor. Introduces a new IP transient state, which would not stall instructions when in this state. The number in the event notation [#XX] mean the relative time the event took place compared to another. The first letter would be 'a' when there would be a predicted rollback and would continue the protocol as normal, while 'b' would be using RIL. The second letter is shown for a comparePrediction route, 'a', or a clearPrediction route, 'b'.

To further improve prediction accuracy, a two-level branch predictor mentioned in [4] is adopted for RIL prediction. Instead of using a history table, a pattern table is used to store RIL prediction decisions. Figure 2 shows the structure of two-level predictor. On the second level, each address/memory region would be assigned a local history buffer to store RIL histories. Using local histories as indexes, the two-level predictor makes RIL predictions and updates RIL predictions in the same manner as the one-level predictor.

B. Coherence protocol transition modifications

In order to allow for predictions to be used within coherence protocols, new actions need to be created and used in specific transitions. In our implementation, we added three new actions to the MESI and MSI coherence protocols: predict(), comparePrediction(), and clearPredict().

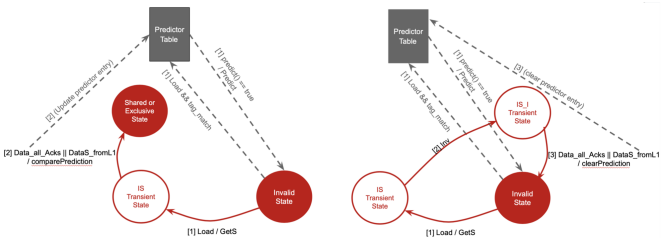

Fig. 3. Implementation of comparePrediction() (left) and clearPrediction() (right) modifications made for the coherence state machines.

predict() checked to see if the address passed in as a parameter is found within the L1 cache. This is where modifications to predicting on memory regions could take place, but we chose not to as it most likely would not lead to a significant performance increase and we left that for future work.

comparePrediction() would be called when receiving accurate data from a GetS request. As shown in Figure 3, comparePrediction() would serve as an 'update' to the predictor, reinforcing or weakening predictions for that block based on the new data received.

In the event an invalid message is sent, clearPredict() would remove the prediction entry for that address from the prediction table. This is shown in Figure 3, where the invalid event would force the prediction table to remove entries related to that address when exiting the IS_I state, which occurs when the block in question becomes invalidated by another cache. This is to ensure that new entries can be added without having to remove other potentially useful entries.

C. Coherence protocol approximation modifications

Modifications had to be made to the coherence protocol so instructions would not be stalled in the event

of a stale cache read. In our implementation, we added a single new transient state in the MESI directory-based coherence protocol, called ”IP”, for this single purpose. The basis of this state was to serve as an intermediate state between the Invalid state and the IS transient state, while also allowing for instructions to run using stale values.

Demonstrated in Figure 4, there are two new paths the cache can take given this new addition. A Predict event sent by the predictor table will notify the cache if it would potentially rollback or not. If there is a predicted rollback, the block's state will simply transition immediately to the IS state and follow the unmodified MESI protocol. Otherwise, the block will remain in the new IP state and continue running the programs instructions with the stale data from the local cache. Once the accurate data arrives, the block will transition to the shared state (S) and allow for an update of the prediction table. While not implemented, a rollback at this point would replay all instructions ran if the comparison showed that the stale value did not match the data obtained from the GetS request.

IV. Evaluation

In this project, our team made use of the gem5 architecture simulator for testing the MSI and MESI directory-based coherence protocols on the x86 architecture. The configuration for our tests used a cache line size of 64 bits, a two set-associative 64KB L1I and L1D caches, and a two set-associative L2 cache size of 8MB. The two primary scripts used, critical-fs and fs-reduction, were to evaluate the accuracy of the predictor and the average time for GetS requests.

A. Prediction accuracy measurement

As described in Section 3B, three new actions are added to coherence protocol state machine files. To measure prediction accuracy without a recovery mechanism on misprediction, new actions should not disturb the program with misprediction information. For the ”Predict” action, subsequent GetS request would be sent so that the cache still reads new data upon a coherence miss. When the cache receives data from directory/sharers, ”comparePrediction” action simply checks if the prediction is correct or not and the program still writes data from the GetS request to cache. Finally, ”clearPrediction” action only cleans up external prediction data structure but does not modify any cache lines. With the above setup, we were able to measure prediction accuracy metrics without affecting correctness of the program.

Two Level - Critical-fs:

Iterations	Total Predicted	Correctly Predicted	Predictor Accuracy
100	2,324	808	34.8%
1,000	2,488	1,171	47.1%
10,000	12,201	10,717	87.8%
100,000	52,239	50,757	97.2%

Two Level - fs-reduction:

Iterations	Total Predicted	Correctly Predicted	Predictor Accuracy
100	2,289	866	37.8%
1,000	3,133	1,691	54%
10,000	11,687	10,245	87.7%
100,000	176,280	175,969	99.3%

Fig. 5. Table for two-level predictor results using MESI: critical-fs = top and fs-reduction = bottom

B. Approximation and rollback estimate measurements

Estimating a rollback proved to be a relatively straightforward process, but may not be precise. The most important measurement needed was an average time in cycles for a GetS request as this could provide insights in approximation and replay costs/savings. To collect the times, a debug flag was placed in a few key areas. The action for sending a GetS request was one such location, along with the transition from the IS transient state to the shared state (S). The output also ensured that the pair of GetS request traces were for the same address.

C. False sharing scripts

Two scripts, Critical-fs and fs-reduction, were used in testing the effectiveness of the predictor and obtaining the average of GetS requests. Critical-fs invalidates neighboring data accesses often using an OpenMP directive to statically schedule the threads and divide each iteration of a large loop into chunks of size 1. fs-reduction results in the opposite, reducing the amount of false sharing by utlizing OpenMP's reduction directive.

V. Results and analysis

Using the project setup described above, we have collected data on the predictors accuracy, estimated savings for approximations, and calculated the feasibility of rollbacks improving performance.

A. RIL predictor accuracy

From the results shown in Figure 5, the predictors accuracy improved with increasing the number of iterations. At a small number of iterations (100), the predictor

still maintained a prediction accuracy above 30% for both critical-fs and fs-reduction. For a large number of iterations (100,000) both tests were nearing an accuracy of 100% with both being over 97% accurate.

B. Approximation savings estimates

Despite the work put into implementation and testing in the MESI and MSI approximation coherence modifications, we were unable to provide data on the accuracy and performance of approximate computing benchmarks. Estimates can be made using the prediction estimates and the percent of cycles spent of GetS requests within a program (7.89% in the case of critical-fs with 100,000 iterations and 2 threads). Based on the critical-fs tests, there can be up to a 7.86% improvement when using 100,000 iterations and up to a 2.98% when using 100 iterations.

C. Rollback timing estimation

Over 147 pairs of GetS requests, the average time was 12,380 cycles. Using the critical-fs test script, GetS requests accounted for 7.89% of the programs total cycle count (1,819,860 cycles out of 23,068,500 cycles). Assuming that 40% of the cycles within the 12,380 cycles of a GetS request must be replayed for full accuracy, the prediction accuracy would have to be 28.57% accurate to produce a performance improvement. The 40% assumption is fair as many of the cycles within the program would become memory-bound (stall) and would therefore not need to be replayed. Even at 80%, the predictor would need to be 44% accurate, which would be acceptable for most programs, except those with short run times/small number of iterations (i.e. 100 iterations of critical-fs). Prediction accuracy estimations can be made by calculating 1 - 1/(1 + X), where X would be the fraction of cycles needing to be replayed.

VI. Related work

Reading from invalid caches is a technique in approximate computing discussed in [1].This project extends their work and created a predictor to improve accuracy of approximate load. [3] proposed a coherence decoupling scheme that uses Speculative Cache Lookup to provide a speculative value upon coherence misses and a recovery mechanism on misprediction. To improve prediction accuracy, the paper uses PC-indexed confidence predictor to dynamically set prediction outcome.

Branch prediction and coherence protocol predictions are closely related to RIL predictors in this project. [4] proposed a classic two-level branch predictor and one-level predictors with various FSM setups. We adopted and modified their idea in branch prediction to our project. [5] extends the two-level predictor by Yeh & Patt for coherence protocol prediction. Local coherence message buffers for memory addresses are used to index into their prediction table to predict coherence messages.

VII. Future work

A. Advanced predictors

Implementing a predictor that would make use of the instruction's memory region could allow for more accurate predictions when one region exhibits frequent instances of false or true sharing. Neural predictors as described in [6] could also be adopted for RIL prediction. In addition to using local RIL history, global history of different length could be used as input to TAGE predictors in [7] and [8] to further improve prediction accuracy.

B. Rollback implementation and hardware analysis

Due to the likelihood of a rollback becoming worthwhile in terms of performance, an efficient rollback system should be explored along with an analysis of the rollback and predictors effect on power and area. While the performance may prove to be an improvement, the area and power needed for the extra hardware could outweigh the benefits achieved by implementing them.

C. Approximation performance and accuracy balance

Implementing a full approximate computing architecture and testing to find a balance between RIL (or SVC [1]) and accurate computing was a hurdle that this project was unable to demonstrate. A complete test and benchmarked system that did successfully implement and verify would be another future extension of the work done in this paper.

VIII. Conclusion

In conclusion, our results demonstrate how using both predictors and rollbacks would improve performance, despite the cost of replaying instructions. Our findings showed that the predictor would need to be at least 30% accurate to ensure a net positive trade off of performance and program accuracy in approximate computing assuming a misprediction cost of 140%. This assumption of misprediction cost is a fair assumption to make because of memory-bound stalling in large programs. Predicted benefits of approximate loads using our predictor could reach an improvement of up to 7.86% for large programs or 2.98% for small programs.

Team Member Contributions	
Xiaoyang Gong	Implemented predictors Modified coherence protocol Collected simulation results
Reese Kuper	MESI modifications Rollback cost estimations Predictor adjustments
Jinglin Liang	MSI modifications Implementation research

REFERENCES

[1] P. V. Rengasamy, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, 2015. "Exploiting Staleness for Approximating Loads on CMPs". In Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT). 343–354.

[2] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: making use of incoherence. In Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2004.

[3] Shubhendu S. Mukherjee and Mark D. Hill. 1998. Using prediction to accelerate coherence protocols. SIGARCH Comput. Archit. News 26, 3 (June 1998), 179–190.

[4] Tse-Yu Yeh and Yale N. Patt. 1991. Two-level adaptive training branch prediction. In Proceedings of the 24th annual international symposium on Microarchitecture (MICRO 24). Association for Computing Machinery, New York, NY, USA, 51–61.

[5] S. S. Mukherjee and M. D. Hill. Using Prediction to Accelerate Coherence Protocols. In Proc. of the 25th Annual International Symposium on Computer Architecture, June 1998.

[6] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7), pp. 197–206, January 2001.

[7] A. Seznec, "A 64-kbytes ittage indirect branch predictor," in Proceedings of the JWAC-2: Championship Branch Prediction, June 2011.

[8] A. Seznec, "A 256 kbits l-tage branch predictor," Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2), vol. 9, May 2007.