Constant Load Value Prediction

Prajyot Gupta
Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI, USA
pgupta54@wisc.edu

Satvik Maurya
Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI, USA
smaurya@wisc.edu

Robert Viramontes
Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI, USA
rviramontes@wisc.edu

***Abstract*— Locality is a crucial technique that drives performance and energy optimizations in the modern computing system, particularly as the performance of CPU cores and memory devices diverge. Value locality is a form of locality that occurs when a value at a memory address does not change [1]. Our project takes advantage of value locality by implementing constant value prediction. In constant value prediction, we add hardware that monitors the memory access stream and determines memory addresses where the value is highly stable. When a highly stable load is identified, our prediction hardware provides the value and allows the load instruction to skip accessing the memory hierarchy. We implement constant value prediction in the Minor in-order and DerivO3 out-of-order CPU models of the gem5 simulator to understand the impact of constant value prediction.**

We conduct experiments utilizing two benchmark suites, SPEC2006 and Perfect. Our experiments explore several variations of the prediction hardware at different table sizes and under different replacement policies. An encouraging result of our simulations is a maximum of 50% reduction in data cache accesses for a relatively small hardware configuration. While this does not translate to significant performance increase via IPC, it may lead to energy savings by replacing portions of cache traffic with a lightweight prediction structure.

I. Introduction

The disparity between the increase in speed of a microprocessor and the access speed of an off-chip memory is one of the major aspects of microprocessor design. Loads that miss the last level cache can take hundreds of cycles to fetch the data, leading to long stalls in the processor pipelines. This problem is caused because the differential rate of change in speeds between the processor and memory is known as the "*Memory Wall",* wherein processor speed improvements range 35-55% and on the other hand, memory latency improvements amount to about 7% [2]. Processors make the use of data and instruction level parallelism techniques to hide the stalls. However, this solution still suffers from a fundamental bottleneck – limits in the off-chip data transfer bandwidth in case of accessing main memory. Modern-day workloads from several prevalent and emerging application domains, such as image processing, computer vision, machine learning, and data analytics, often have input datasets with significant repetition of values. This presents us an opportunity to leverage the inherent predictability of data values in these applications to tackle the bottleneck. This paper exploits this opportunity.

In our work, we explore the concept of load value locality specifically through the lens of constant value prediction. The concept of value locality was discussed by Lipasti et al. [1] as the "likelihood of the influence of previously seen program values on the newly occurring ones". In ECE 752, we discussed spatial and temporal locality, which attempted to improve the probability of finding a given memory location in the cache by reducing the distance between successive accesses to that location. Value locality is a different approach to tackle the same problem. Using this concept, the processor can speculate what value will be returned by a load instruction, and have it execute the instructions speculatively using this value. When the load completes, the actual value returned by it can be compared with the speculated value. If the speculation is correct, the processor increases the confidence of its predictions and continues; but if the guess is incorrect, the processor rolls back the computation and recomputes using the actual value obtained via the load instruction.

Contemporary work, like Lipasti [1], Seznec [5] show that value prediction can result in high accuracies for certain workloads using relatively simple prediction schemes. One important point to note in the work here is that when we talk about value prediction, we always discuss the prediction of value for a load operation. A store operation that updates a base register is quite unlikely to exhibit any locality. On the other hand, a load is very likely to exhibit locality because any value stored in memory will likely be accessed more than once.

In constant value prediction, we introduce hardware to monitor all load operations. Then, loads that have a very consistent value are predicted as *Constant* and skip accessing the memory system. We have re-implemented the ideas from [1] and run them on more modern workloads to see how the concepts of value locality hold for modern workloads. These workloads include the SPEC2006 [3] and Perfect [4] benchmark suites. We modify the gem5 in-order MinorCPU and out-of-order DerivO3 models [5] to incorporate our load value prediction hardware.

Our code is available on GitHub: https://github.com/robertviramontes/ece752_loadvaluepredictor

II. Implementation

Our load value prediction hardware follows the design of the hardware proposed in [1]. As such, we will keep the discussion of the hardware attributes brief and here focus on the specifics of our implementation methodology and lessons learned during implementation. We have implemented the load value prediction features in a single Load Value Prediction Unit, LVPU, that serves as the interface between the CPU pipeline and the subunits that implement constant load value prediction, which are described below.

A. *Load Value Prediction Table*

The load value prediction table, also referred to as LVPT, stores the value to predict for loads. It takes in the PC value of the load instruction and outputs the predicted data whose value was previously loaded by the instruction. Its architecture is similar to a directly mapped cache – it is indexed by the load instruction address and is untagged. Because of this, both constructive and destructive interference can occur between the load instructions with the same index value, and hence can contain aliases. It can have a history depth of *n,* which provides a mechanism for selecting the value to predict from a set of *n* unique values for a given instruction.

The LVPT stores a valid tag to indicate that the storage location contains valid data from runtime observations. This is cleared at processor startup and set true when an instruction indexes in to the row to update the stored value. If the LVPT determines that an instruction value is invalid, then the lookup in the load classification table is skipped because an invalid value is always *Strongly Unpredictable*.

B. *Load Classification Table*

The load classification table, LCT, generates a classification for a load. The LCT is implemented as an array of saturating counters, which allowed us to base the software design on the similar Simple branch predictor available in gem5. As in [1], the saturating counters are indexed by the instruction address. The possible classifications in our LCT include *Strongly Unpredictable*, *Weakly Unpredictable*, *Predictable*, and *Constant*. A classification of *Strongly Unpredictable* is generated when all counter bits are 0, *Weakly Unpredictable* when the most significant bit is 0 and at least one other bit is 1, *Predictable* when the most significant bit is 1 and at least one other bit is 0, and *Constant* when all bits are 1. Although our current system implementation only utilizes a *Constant* classification, the remaining classification types are kept for statistic collection and future incorporation of modifications that can take advantage of other classifications.

The LCT is updated when the LVPU receives a request to verify a prediction as a memory transaction is committed. If the prediction was correct, the corresponding counter is incremented, and if incorrect, the counter is decremented. We choose this policy because it is straightforward to implement and reason about. However, we did discuss an alternate policy that would reset the counter to zero on a misprediction. The intuition behind this policy is that if a value is changing, particularly one that was previously *Constant*, this may represent a change in the instruction pattern where the previous value's predictability is no longer applicable. The tradeoff between these policies would be interesting to explore in future work, particularly in a system that utilizes the *Predictable* classification.

C. *Constant Verification Unit*

The CVU is responsible for maintaining coherency between the value of a constant load stored in the LVPT and the actual value in memory. The CVU achieves this by using the following mechanism- When a load instruction graduates from being *Predictable* to *Constant* the LCT, a new entry is allocated in the Content Addressable Memory (CAM) of the CVU. This CAM is a fully associative cache in which every entry consists of the effective virtual address of a constant load and the corresponding index of that constant load in the LVPT. Thus, if a load address is present in the CAM of the CVU, that load is constant and the value in the LVPT is coherent with memory. The CVU then monitors all store instructions that reach the commit stage of the pipeline. If an entry in the CVU CAM has a memory address which matches that of a committing store instruction, the CVU invalidates that entry in its CAM which means that that load address is no longer coherent with the entry in the LVPT. Loads are classified in the frontend of the pipeline, but the verification of the classified loads takes place later. This is primarily because the effective address of the load instruction must be calculated before it can be verified whether the load value predicted by the LVPT is coherent with memory.

The CVU is also responsible for checking loads that are marked as *Predictable* by the LCT. *Predictable* loads execute in the pipeline normally and after they finish execution, the actual value of the load instruction is compared with the value predicted by the LVPT. If these values match, the LCT increments its saturating counter and if the values do not match, the LCT decrements its saturating counters.

Since the CVU consists of a fully associative CAM, the replacement policy for its entries, once the memory is full, is an important aspect to investigate. Fully associative memories are expensive, fully associative CAMs even more so. Hence, the number of entries available in the CVU CAM is limited which implies that the entries should be managed as efficiently as possible. Thus, we have explored five different replacement policies in the CVU CAM-

- FIFO: This is a simple queue-based policy where the oldest entry is ejected from the CAM.
- Most Recently Used (MRU): This policy ejects the most recently used entry from the CAM.
- Not-Most Recently Used (NMRU): This is a simpler policy than the MRU policy implementation-wise. This ejects any entry from the CAM which has not been used most recently.
- Least Recently Used (LRU): This policy ejects the least recently used entry from the CAM.
- Not-Most Recently Used (NLRU): Like the NMRU policy, this is a simpler implementation, and it ejects any entry from the CAM which is not the least recently used entry.

D. *Minor CPU Modifications*

We modify the gem5 Minor CPU model to understand the impact of the constant value prediction in an in-order CPU. The key modifications include modifications to the dynamic instruction class, the decode stage, the load store queue and the commit stage. We also disable the early memory reference option to ensure the Minor CPU processes memory references in program order.

The dynamic instruction is modified to contain additional metadata about the instruction, including the classification from the LCT and the value predicted by the LVPT. We also add the virtual address as a property of the dynamic instruction, which was already present in the DerivO3 dynamic instruction.

The decode stage is modified so that all load instructions check for a prediction in the LVPU. The dynamic instruction's metadata is updated with the classification and value that is returned from the LVPU at this stage.

Memory operations proceed normally into the load store queue. When the instruction enters the queue, its virtual address is available, so the instruction metadata is updated to carry the virtual address through the rest of the execution pipeline, similar to the approach already implemented in the DerivO3 CPU. When a load instruction is at the head of the queue, we check if it was classified as *Constant*. If it was, then we call the LVPU to verify with the CVU that the value is still valid. If the constant load is still valid, the memory request is immediately fulfilled with the predicted value from the dynamic instructions metadata. This allows the constant predictions to avoid memory access. When a store instruction reaches the head of the queue, its PC and virtual address are forwarded to the CVU to invalidate subsequent constant loads.

The commit stage is modified so that all load instructions, regardless of classification, are sent to update predictions in the LCT and LVPT.

The modifications to the Minor CPU model are summarized in Figure 1.

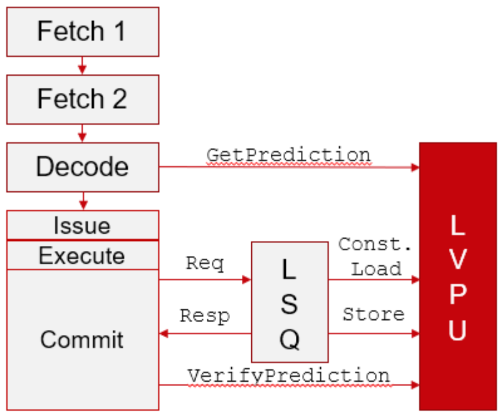


Figure 1 Graphical depiction of modificatons to Minor CPU

E. DerivO3 CPU Modifications

The DerivO3 CPU has a pipeline which is summarized in Figure 2. The integration of the load value prediction unit in the out-of-order CPU supported in gem5 faced the following challenges-

- Register renaming: Because the out-of-order CPU renames architectural registers as arbitrary physical registers before issuing instructions to eliminate false dependencies, the load value prediction unit cannot forward values of constant loads before register renaming is done.
- Load-Store ordering violations: The out-of-order CPU also has a memory dependency checker, which enables load instructions to execute out-of-order with stores. This adds an additional overhead of correcting ordering violations in case a younger load is executed before an older store instruction. Thus, the load value prediction unit will have to be mindful of this nuance since a constant load may reach the execute stage (where the fun stuff happens, to be described below) before an older store. This implies that the safest way to ensure functional sanity is by tightly coupling the load value prediction unit with the load-store queue unit.

These challenges were thus overcome by the following implementation steps-

- Load instruction addresses looked up in the LVPT and LCT during dispatch – instructions are still in-order when they reach dispatch. At this stage, the destination registers of all instructions have been renamed. Once the LCT and LVPT return the classification and the predicted value respectively, the load instruction is pushed into the load-store queue and the instruction queue normally. The predicted load value and the classification (not predictable, predictable, constant) are attached to the instruction itself.
- Irrespective of their classification, all load instructions flow through the issue stage of the pipeline normally. After a load instruction starts execution, the CVU lookup for a load classified as constant starts as soon as the effective load address has been calculated. If the CVU verification is successful, the load instruction does not trigger a memory read and is sent to the writeback stage of the pipeline. For all other cases, the load instructions will access memory.
- At writeback, the constant value predicted by the LVPT is written into the destination register of the load instruction. This is safe to do here because all memory ordering violations would have been detected earlier. The writeback stage of the pipeline is also the stage where other loads update the LCT. Loads classified as predictable are verified by the CVU and the result of the verification is used to update the LCT.
- The invalidation of CVU entries by store instructions occurs after stores have executed and have been sent for commit. Since these stores will commit at some point in the future, any constant loads with the same effective memory address in the LVPT will be incoherent with memory.

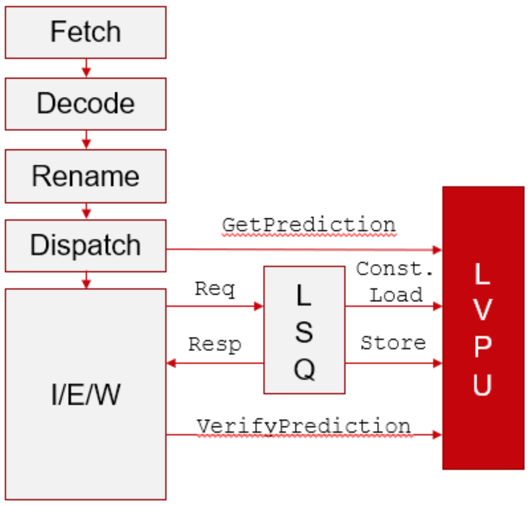


Figure 2 Graphical depiction of DerivO3 modifications

III. Results

In this section, we will present the results of our simulations and highlight some notable datapoints. We save our discussion of the results for section IV.

A. Simulator Configuration Overview

Our implementation of LVPU is simulated in gem5 [5] using X86 ISA using in-order Minor and out-of-order DerivO3 CPU. The simulator configurations can be observed in *Table I* which are consistent with Seznec's implementation of value predictor in [6].

B. LVPU Variations

In our work, we explore 8 variations of the LVPU hardware to understand the effects of various parameters on the system. These variations are described in TABLE *II*TABLE III. The parameters can be set via command line arguments to the gem5 system emulation script (`se.py`).

C. Coverage and Accuracy

Two metrics of interest for load value prediction are the coverage and accuracy of the predictor [6]. The coverage is the number of loads that the predictor classifies as predictable. The accuracy is the number of loads classified as predictable that are correctly classified. Figure 3 shows the coverage and accuracy for selected benchmarks in the Minor CPU and Figure 4 shows this for the DerivO3 CPU.

In the minor CPU, the `lbm` benchmark has the highest coverage with a peak of 56.65% for Variation-0 while `sjeng` exhibits low value locality with a peak coverage of only 1.36%. The accuracy varies significantly, with `libquantum` reaching the highest accuracy of 99.90% in Variation-1. Predictably, the accuracy is worst for the `specrand` benchmark, which heavily relies on random number generation [3], with the minimum accuracy of 1.55%.

TABLE I SIMULATOR CONFIGURATION

Front-End	**Fetch through Rename width:** 8 insts/cycle **Branch Pred.:** TournamentBP, size 8K **BTB:** 2-way 4k-entry, Tag size 16 **RAS:** size 16

Cache	**L1 D$**: Size = 32kB, Associativity = 8;
	L1 I$: Size = 32kB, Associativity = 8;
	L2$: Size = 256kB, Associativity = 8;

Main Memory	**Type** = DDR3_2133_8x8, **Size**=16GB

TABLE II *LVPU VARIATIONS*

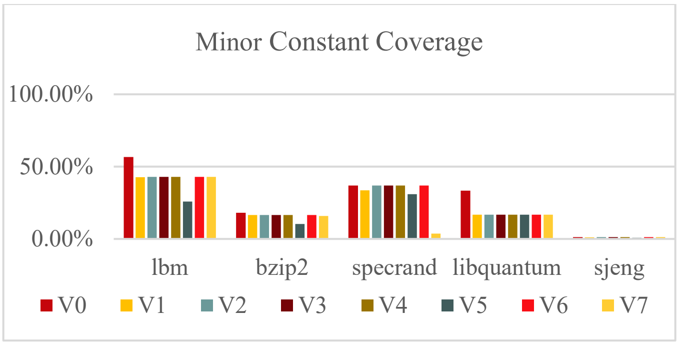
Var.	**LCT**		**LVPT**		**CVU**	
	Entries	Counter	Entries	Hist. depth	entries/ thread	Rep. policy
0	512	2-bit	1024	1	8	FIFO
1	512	4-bit	1024	1	8	FIFO
2	1024	2-bit	4096	16	32	FIFO
3	1024	2-bit	4096	16	32	LRU
4	1024	2-bit	4096	16	32	NLRU
5	1024	2-bit	4096	16	32	MRU
6	1024	2-bit	4096	16	32	NMRU
7	1024	4-bit	4096	16	32	FIFO

In the DerivO3 CPU, we see the continuation of some general trends, such as `libquantum` demonstrating high value locality and `sjeng` demonstrating low locality. The highest coverage achieved is 49.81% for `libquantum` Variation-0 and 2-6 while the lowest coverage is 2.51% for bzip2 on Variation-1. The highest accuracy of ~100% is achieved on the `libquantum` benchmarks for Variation-1 and Variation-7 while the lowest is 1.19% for `specrand` on Variation-5.

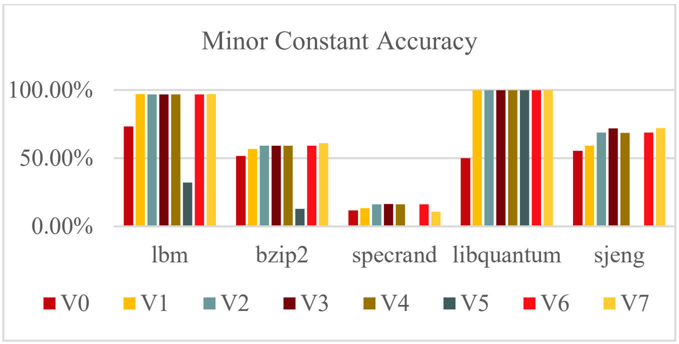
D. D$ accesses

L1 data cache accesses for our selected benchmarks, normalized to the number of L1 data cache accesses in the unmodified processor are shown in Figure 5 and Figure 6 for the Minor and DerivO3 CPU, respectively. For the Minor CPU, our simulations show that in a benchmark with high value locality like `libquantum`, the L1 data cache accesses are, in the best case, reduced to 50.0306% of the unmodified Minor CPU accesses in variations V2-V6. In the DerivO3 CPU for the `libquantum` benchmark, the best case reduces the L1 data caches accesses to 74.9090% of the unmodified CPU accesses for variations V2-V4.

Our data verifies that, even in the worst case with benchmarks with poor locality, the LVPU does not increase data accesses. This agrees with our theoretical understanding

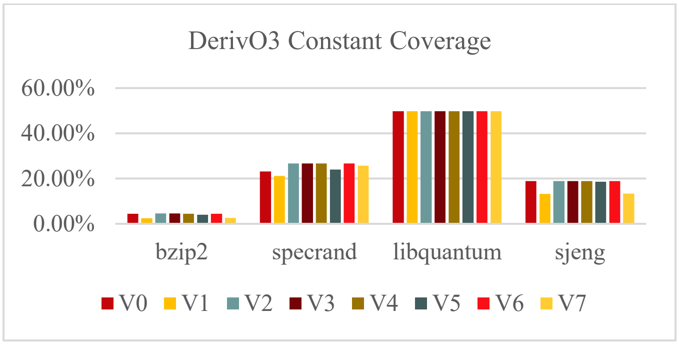


(a) Constant Prediction Coverage

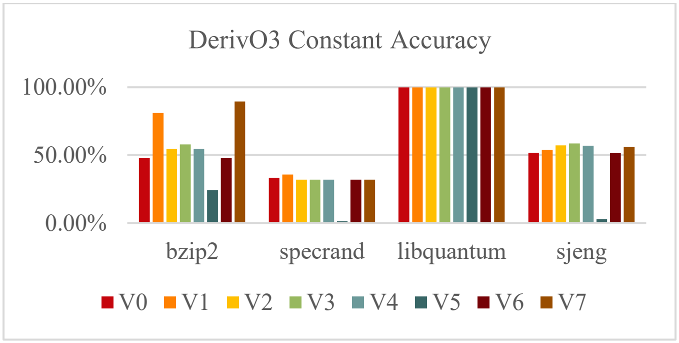


(b) Constant Prediction Accuracy

Figure 3 Constant Coverage and Accuracy for Minor CPU



(a) Constant Prediction Coverage



(b) Constant Prediction Accuracy

Figure 4 Constant Coverage and Accuracy for DerivO3 CPU

that constant prediction acts as a highly specialized cache but does not modify program execution.

E. *Instructions Per Cycle*

We present results for IPC on selected benchmarks in Figure 7 for Minor CPU and Figure 8 for DerivO3 CPU. The IPC is collected from the default gem5 output and normalized to the unmodified processor to clearly show the impact of the change. We find that constant value prediction does not have a significant effect on the overall instructions per cycle, IPC, of a program. We observe that the IPC may have more potential for improvement in the out-of-order CPU than an in-order CPU. In the `libquantum` benchmark, the out-of-order CPU reaches a normalized IPC of 1.01 while the in-order reaches a normalized IPC of 1.006 even though the in-order skips more data accesses. We see a similar, though not as significant, trend in the `specrand` benchmark (peak 1.008 vs 1.007).

F. *Occurrence of Zeros and Ones*

We collect statistics about the actual values that are predicted as constant, specifically tracking constant predictions that are 0 and 1. We collect these statistics at during the lookup process at the decode/dispatch stage for instructions classified as *Constant*, so these represent the values predicted for constant loads regardless of if they are committed or not.

Figure 9 shows the predicted values of Variation-0 in the Minor CPU. The `sjeng` benchmark has the highest proportion of values predicted as zeros and ones, at 21.9% and 7.6%, respectively, in Minor and 30.2% and 30.2%, respectively, in DerivO3. However, for most benchmarks the “other” category of values represents a significant proportion of the values that are predicted as constant. Only the `sjeng` benchmark on the DerivO3 CPU demonstrates a pattern where zeros and ones constitute most of the constant values.

While our data shows that zeros and ones constitute only a small proportion of the constant loads, we would need further study to understand the frequency of individual values which may show that zero and one occur more frequently than any other individual value. However, our data does suggest that there is a benefit to general value prediction and limiting the predictions to zeros and ones would eliminate significant opportunity for value prediction.

G. *Additional Results*

In addition to the SPEC2006 benchmark results described above, we also ran the following benchmarks from the Perfect suite [4] of image and signal processing benchmarks with our basic LVPU (LVP_V0)- `sar-backprojection`, `sar-pfa-interp1`, `stap-inner-product`, `stap-outer-product`, `wami-change-detection`.

These benchmarks did not show any significant change in the IPC, so we have omitted the IPC chart for the sake of brevity. However, the constant coverage and accuracy charts are shown in Figures 11 and 12 for the Minor and DerivO3 CPUs. The coverage for both CPUs is generally high (compared to SPEC06) for almost all the chosen workloads. However, the constant accuracy graphs in Figure 12 show that despite having a relatively high coverage, the accuracy of predictions was

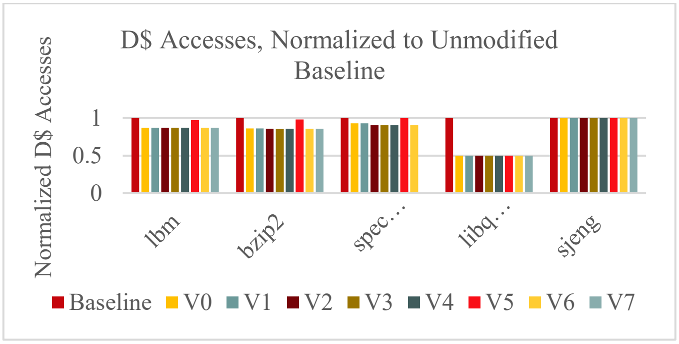


Figure 5 Minor CPU data cache accesses, normalized to unmodified CPU for select SPEC2006 benchmarks

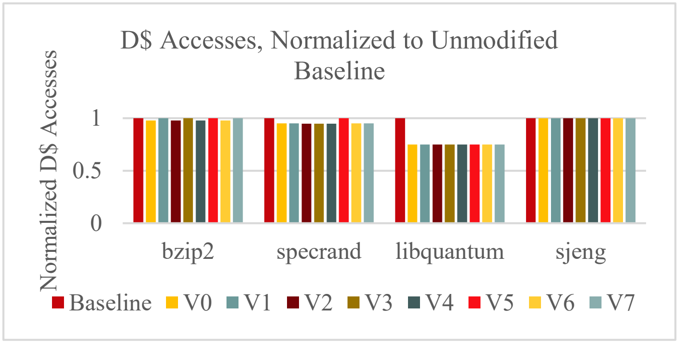


Figure 6 DerivO3 CPU data cache accesses, normalized to unmodified CPU for select SPEC2006 benchmarks

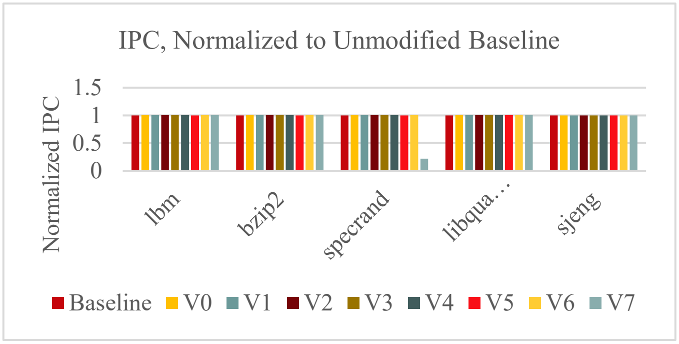


Figure 7 Minor CPU IPC, normalized to unmodified CPU for select SPEC2006 benchmarks

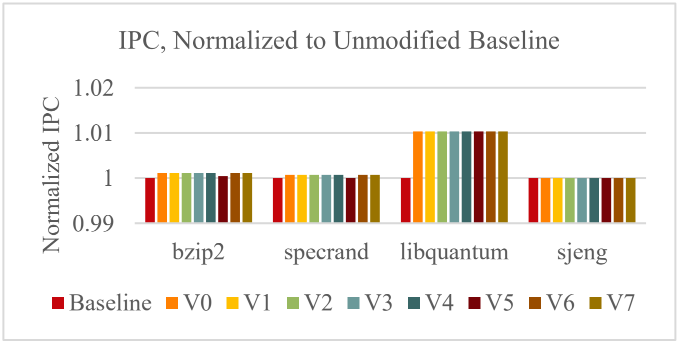


Figure 8 DerivO3 CPU IPC, normalized to unmodified CPU for select SPEC2006 benchmarks

mediocre. We attribute this to the fact that there are a lot of invalidations in the CVU CAM due to store instructions.

The decrease in data cache accesses was as expected from both CPUs as shown in Figure 13.

While we have not shown the results here, an analysis of the distribution of load *values* for the five benchmarks showed that the percentage of constant loads with values of one was very small (< 1%). However, for constant loads with values of zero, there was a slightly different trend, with the `wami-change-detection` workload reporting that about 25% of constant loads had a value of 0.

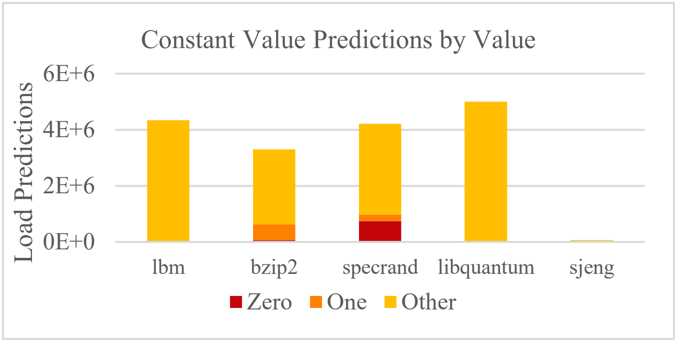


Figure 9 Minor CPU constant value predictions categorized as either zero, one or other for select SPEC2006 benchmarks

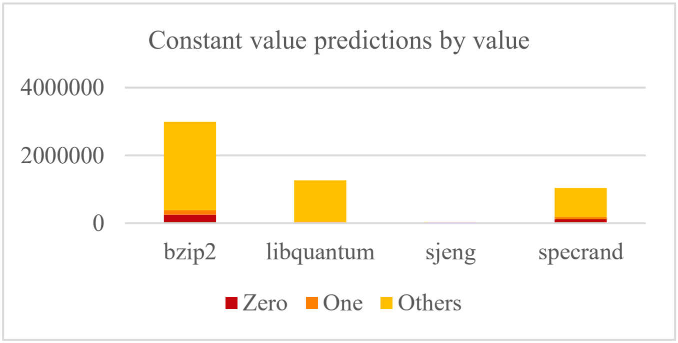


Figure 10 DerivO3 CPU constant value predictions categorized as either zero, one or other for select SPEC2006 benchmarks

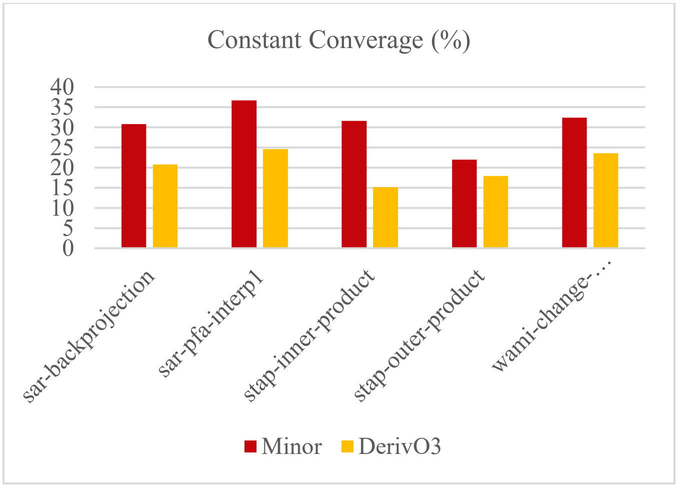


Figure 11 Constant coverage for the perfect-suite workloads

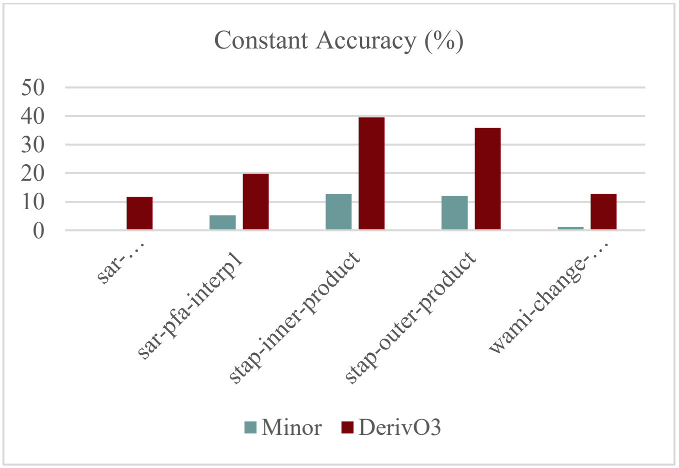


Figure 12 Constant accuracy for the perfect-suite workloads

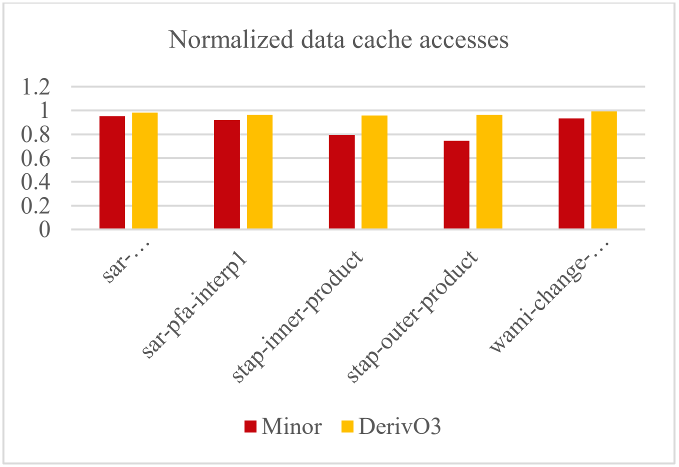


Figure 13 Normalized D$ accesses for the perfect-suite workloads

IV. Discussion

A. *Effect of structure size*

Our results show that the smallest version of the hardware, Variation-0, performs almost as well as the other variations at reducing data accesses. The values for the normalized data accesses of the `specrand` benchmark are reproduced in Table III and show that there is little variation as the hardware size changes. The standard deviation for the normalized accesses in the Minor CPU is 3.0×10^{-6} and for the DerivO3 CPU is 1.2×10^{-5}, providing an empirical measurement of the small variation in data cache accesses across LVPU variations.

This is a useful result because one of the primary motivations of our predictor is to reduce the data cache accesses which may help reduce the power consumed in cache transactions. Our results show that large structures are not required to effectively implement constant prediction and suggest that smaller, more power-efficient structures are suitable. Further study would be required to determine a lower bound on the effective sizes and physical simulation to determine the actual energy cost of LVPU structures.

Table III Normalized D$ Accesses for `specrand`

Variation	Minor CPU	DerivO3 CPU
0	0.500307173	0.749091022
1	0.500312572	0.749122977
2	0.500306173	0.749088838
3	0.500306173	0.749088838
4	0.500306173	0.749088838
5	0.500306173	0.749095587
6	0.500306173	0.749091022
7	0.500312972	0.749099159

B. *Effect of replacement policy*

We see that overall, the least recently used (LRU) policy performs the best, even though the difference between this policy and other policies like NLRU is very slight. Since the NLRU policy is easier to implement in hardware in terms of area and complexity, this policy is the best for the CVU fully associative CAM.

The use of the MRU policy shows the most drastic degradation in performance and this degradation is more prominent in the out-of-order core than the in-order core. Using the MRU policy results in a direct violation of the principles of value and temporal locality because it discards the most local values. Thus, the use of this replacement policy serves as a proof-of-concept for our load value prediction scheme.

C. *Minimal impact on IPC*

We found that constant value prediction had a very small impact on the IPC of the program, at most achieving a ~2.5% gain in IPC. This result is not surprising as the requirements for constant value prediction are multiple accesses to the same memory address with an unchanged value. This access pattern means this data is likely in the data cache, so we are only skipping a hit in the data cache which has a small (1 or 2-cycle) latency in our simulations. This analysis is based on our understanding of the systems and further study would be required to verify this analysis.

Furthermore, our implementation is also conservative as far as value forwarding for constant loads is concerned. The value of a prediction for a constant load is forwarded only during the writeback and commit stages for the O3 and Minor CPUs, respectively. If the value forwarding were done earlier and more aggressively, we might be able to see better results for the IPC, especially for the O3 CPU since there could be multiple dependent instructions waiting in the reservation stations for the load value.

Our results also show that the out-of-order CPU can improve more than the in-order CPU. We attribute this difference to the out-of-order CPUs ability to take advantage of the reduced pressure on the load queue by reordering instructions to take advantage of slots freed by constant prediction.

D. *Coverage and accuracy of in-order vs. out-of-order cores*

When examining the coverage of the constant predictions, we observe and interesting difference between the in-order and out-of-order cores. For in-order, the coverage of bzip2 is generally higher than that of `sjeng` (15.90% vs 1.18% by arithmetic mean of coverage) while the out-of-order shows the opposite trend (3.94% vs 17.48% by arithmetic mean of coverage). When looking at the absolute values of the number of loads processed by the two CPUs for `sjeng`, we see that there is a 50x difference between the two CPUs. The opposite is true for `bzip2`. This difference in total loads processed is the reason we see different coverage for the two benchmarks across the two CPUs and we think that the number of loads being processed is different because of the different scheduling policies of the two CPUs. While the actual number of loads would still be the same, either CPU could get caught in a loop where a branch is mispredicted and numerous loads are predicted and later squashed. We cannot verify this at this point since we have not collected statistics about load instructions that we classified as constant and later squashed in the pipeline.

V. Future Work

There are several improvements or variations we believe would be useful to explore in future works. One improvement we believe is possible is to optimize the timing of the value forwarding, making constant results available earlier in the pipeline. In our current implementation, the instructions are issued and executed as normal even though the value is already available in the decode stage. This would require more extensive modifications to the execution pipeline to allow the *Predictable* and *Constant* to appear as if they have executed normally while structures like the Load Store Queue. In fact, this was our original approach but were unable to fully complete this approach after significant programming and debugging effort. We also note that this optimization is limited by the availability of the virtual address, which is not computed until the execute phase.

A feature we would like to implement is the utilization of the *Predictable* classification. This would allow the pipeline to speculatively execute instructions after a load with a predicted value. We expect that this approach would lead to a more significant improvement in IPC, as instructions that previously stalled waiting for a load could proceed speculatively. Implementing this functionality was part of our original plan and we made steps towards this implementation, but we unfortunately ran out of time to debug the squashing mechanism on value mispredictions.

We also note that our implementation only considers a single-CPU model, which may not be realistic for many modern workloads. Because we are effectively implementing a specialized cache, we believe future works should explore the implications of cache coherence requirements on load value prediction. For instance, the CVU must consider stores that happen in other processors to matching memory addresses and the cost of additional snooping hardware must be outweighed by reduced data accesses. Our implementation also considers associating values and classifications with specific threads, but we do not extensively study the effect of multiple threads on our implementation.

VI. Conclusions

In this report, we have investigated the principles of value locality and value prediction. By using a load value prediction unit, we can see a notable drop in memory accesses for a few benchmarks which could be useful for devices that require a high degree of power efficiency. Overall, the addition of the load value prediction unit does not degrade the IPC and it helps reduce memory accesses for frequently used constants. However, we believe that by implementing a more aggressive value forwarding scheme and by executing instructions based on speculative values, we might be able to see more improvement in the IPC and this is something we would like to pursue in the future.

VII. References

[1] M. H. Lipasti, C. B. Wilkerson and J. P. Shen, "Value locality and load value prediction," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, New York, NY, 1996.

[2] G. Sohi, *Unit 8: Storage Hierarchy I: Caches,* Madison, WI, 2021.

[3] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News,* vol. 34, no. 4, pp. 1-17, 2006.

[4] K. Barker, T. Benson, D. Campbell, E. David, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent and A. Tumeo, "PERFECT (Power Efficiency Revolution For," December 2013. [Online]. Available: http://hpc.pnnl.gov/projects/PERFECT/.. [Accessed 4 May 2021].

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. H. J. Basu, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News,* pp. 1-7, May 2011.

[6] A. Perais and A. Seznec, "Practical data value speculation for future high-end processors," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Orlando, 2014.

Appendix

We provide the code written to model the Load value prediction unit as well as the diffs that show how the LVPU was integrated into the DerivO3 and Minor CPU models. Section I shows the code for the load value prediction unit, Section II shows the integration of the LVPU with the DerivO3 CPU model, and Section III shows the integration of the LVPU with the Minor CPU model.

I. Load value prediction unit

A. Load Value Prediction Table

```
/******************************************************************************/
// Author   : Prajyot Gupta
// Department   : Grad Student @ Dept. of Electrical & Computer Engineering
// Contact      : pgupta54@wisc.edu
// Project      : ECE 752
/******************************************************************************/

#ifndef __CPU_LVP_LOADVALUEPREDICTIONTABLE_HH__
#define __CPU_LVP_LOADVALUEPREDICTIONTABLE_HH__

#include "base/types.hh"
#include "cpu/static_inst.hh"
#include "sim/sim_object.hh"

#include "enums.hh"
#include "params/LoadValuePredictionTable.hh"

#include "arch/types.hh"
#include "base/logging.hh"
#include "config/the_isa.hh"

/** Creating a default Load Value Prediction Table entry
 *  which will have below attributes
 *  tag   : Specifies the Opcode of the Load instruction
 *  taget : Specifies the Load value associated with the tag
 *  valid : Specifies if the value loaded is valid
 */

class LoadValuePredictionTable : public SimObject
{
  private:
    struct LVPTEntry
    {
        LVPTEntry()
            : tag(0), target(0), valid(false)
        {}

        /** The entry's tag. */
        Addr tag;

        /** The entry's target. */
        RegVal target;

        /** The entry's thread id. */
        ThreadID tid;

        /** Whether or not the entry is valid. */
        bool valid;
    };

  public:
    /** Creates a LVPT with the given number of entries, number of bits per
     *  tag, and instruction offset amount.
```

```
      *  @param numEntries Number of entries for the LVPT.
      *  @param tagBits Number of bits for each tag in the LVPT.
      *  @param instShiftAmt Offset amount for instructions to ignore alignment.
      */
     LoadValuePredictionTable(const LoadValuePredictionTableParams *params);

     void reset();

     /** Looks up an address in the LVPT. Must call valid() first on the address.
      *  @param inst_PC The address of the branch to look up.
      *  @param tid The thread id.
      *  @return Returns the predicated load value.
      */
     RegVal lookup(ThreadID tid, Addr instPC, bool *lvptResultValid);

     /** Checks if the load entry is in the LVPT.
      *  @param inst_PC The address of the branch to look up.
      *  @param tid The thread id.
      *  @return Whether or not the branch exists in the LVPT.
      */
     bool valid(Addr instPC, ThreadID tid);

     /** Updates the LVPT with the latest predicted Load Value.
      *  @param inst_PC The address of the branch being updated.
      *  @param target_PC The predicted target data.
      *  @param tid The thread id.
      */

     void update(Addr instPC, const RegVal target, ThreadID tid);

     /** Returns the index into the LVPT, based on the branch's PC.
      *  @param inst_PC The branch to look up.
      *  @return Returns the index into the LVPT.
      */

     unsigned getIndex(Addr instPC, ThreadID tid);

     /** Returns the tag bits of a given address.
      *  @param inst_PC The branch's address.
      *  @return Returns the tag bits.
      */
     inline Addr getTag(Addr instPC);

   private:

     /** The actual LVPT declaration */
       std::vector<LVPTEntry> LVPT;

       /** The number of entries in the LVPT. */
       const unsigned numEntries;

       /** Depth of data history kept in the LVPT*/
       const unsigned historyDepth;

       //prajyotg :: No sure if below will be used
       /** The index mask. */
       unsigned idxMask;

       /** The number of tag bits per entry. */
       unsigned tagBits;

       /** The tag mask. */
       unsigned tagMask;

       /** Number of bits to shift PC when calculating index. */
       unsigned instShiftAmt;
```

```

    /** Number of bits to shift PC when calculating tag. */
    unsigned tagShiftAmt;

    /** Log2 NumThreads used for hashing threadid */
    unsigned log2NumThreads;
};

#endif // __CPU_LVP_LOADVALUEPREDICTIONTABLE_HH__

```

```
/*******************************************************************************/
// Author    : Prajyot Gupta
// Department    : Grad Student @ Dept. of Electrical & Computer Engineering
// Contact       : pgupta54@wisc.edu
// Project       : ECE 752
/*******************************************************************************/

#include "cpu/lvp/load_value_prediction_table.hh"

#include "base/intmath.hh"
#include "base/trace.hh"
#include "debug/LVPT.hh"

LoadValuePredictionTable::LoadValuePredictionTable(const LoadValuePredictionTableParams *params)
    : SimObject(params),
      numEntries(params->entries),
      historyDepth(params->historyDepth),
      idxMask(numEntries - 1),
      instShiftAmt(0)

{
    DPRINTF(LVPT, "LVPT: Creating LVPT object.\n");

    if (!isPowerOf2(numEntries)) {
        fatal("LVPT entries is not a power of 2!");
    }

    LVPT.resize(numEntries);

    DPRINTF(LVPT, "LVPT: Doing an initial reset \n");
    for (unsigned i = 0; i < numEntries; ++i) {
        LVPT[i].valid = false;
    }

    idxMask = numEntries - 1;
    tagMask = (1 << tagBits) - 1;
    tagShiftAmt = instShiftAmt + floorLog2(numEntries);
}

/* Reset API */
void
LoadValuePredictionTable::reset()
{
    DPRINTF(LVPT, "LVPT : Calling the Reset API \n");
    for (unsigned i = 0; i < numEntries; ++i) {
        LVPT[i].valid = false;
    }
}

/* APIs to get index and tag*/
unsigned
LoadValuePredictionTable::getIndex(Addr instPC, ThreadID tid)
{
    // Need to shift PC over by the word offset.
```

```cpp
  // Math: ((instPC >> instShiftAmt)^(tid<<(tagShiftAmt-instShiftAmt-log2NumThreads))&idxMask;
  DPRINTF(LVPT, "LVPT : Getting Index \n");
  return ((instPC >> instShiftAmt)
      ^ (tid << (tagShiftAmt - instShiftAmt - log2NumThreads)))
      & idxMask;
}

inline
Addr
LoadValuePredictionTable::getTag(Addr instPC)
{
  DPRINTF(LVPT, "LVPT : Getting Tag \n");
  return (instPC >> tagShiftAmt) & tagMask;
}

/** Checks if the load entry is in the LVPT.i **/
bool
LoadValuePredictionTable::valid(Addr instPC, ThreadID tid)
{
  unsigned LVPT_idx = getIndex(instPC, tid);


  // Making sure index doesn't go out of bounds
  assert(LVPT_idx < numEntries);

  // Check if: (a) LVPT entry is valid
  // (b) index matches
  // (c) tag matches
  if (LVPT[LVPT_idx].valid
    && LVPT[LVPT_idx].tid == tid) {
    return true;
  } else {
    DPRINTF(LVPT, "LVPT : Checking if LVPT entry is valid \n");
    return false;
  }
}

// data = 0 represent invalid entry.
RegVal
LoadValuePredictionTable::lookup(ThreadID tid, Addr instPC, bool *lvptResultValid)
{
  unsigned LVPT_idx = getIndex(instPC, tid);

  assert(LVPT_idx < numEntries);

  if (valid(instPC, tid)) {
    DPRINTF(LVPT, "Found valid entry for tid: %d at pc %#x : %d \n",
      tid, instPC, LVPT[LVPT_idx].target);
    *lvptResultValid = true;
    return LVPT[LVPT_idx].target;
  } else {
    DPRINTF(LVPT, "Did not find valid entry for tid: %d at address %#x \n",
      tid, instPC);
    return 0;
  }
}

void
//prajyotg :: updated :: LoadValuePredictionTable::update(Addr instPC, const TheISA::PCState &target, ThreadID tid)
LoadValuePredictionTable::update(Addr instPC, const RegVal target, ThreadID tid)
{
  unsigned LVPT_idx = getIndex(instPC, tid);
  DPRINTF(LVPT, "LVPT : Updating the value in the LVPT at index %ld \n", LVPT_idx);

  assert(LVPT_idx < numEntries);
```

```

    LVPT[LVPT_idx].tid = tid;
    LVPT[LVPT_idx].valid = true;
    LVPT[LVPT_idx].target = target;
    LVPT[LVPT_idx].tag = getTag(instPC);
}

LoadValuePredictionTable*
LoadValuePredictionTableParams::create()
{
    return new LoadValuePredictionTable(this);
}

```

B. Load Classification Table

```
/*
 * Robert Viramontes
 * Created March 25, 2021
 * Based on the 2bit_local branch predictor provided with the gem5 source.
 *
 *
 */

#ifndef __CPU_LVP_LOADCLASSIFICATIONTABLE_HH__
#define __CPU_LVP_LOADCLASSIFICATIONTABLE_HH__

#include <vector>

#include "base/sat_counter.hh"
#include "base/types.hh"
#include "cpu/static_inst.hh"
#include "sim/sim_object.hh"

#include "enums.hh"
#include "params/LoadClassificationTable.hh"

/**
 * Implements a local predictor that uses the PC to index into a table of
 * counters.  Note that any time a pointer to the bp_history is given, it
 * should be NULL using this predictor because it does not have any branch
 * predictor state that needs to be recorded or updated; the update can be
 * determined solely by the branch being taken or not taken.
 */
class LoadClassificationTable : public SimObject
{
  public:
    /**
     * Build load classfication table.
     */
    LoadClassificationTable(const LoadClassificationTableParams *params);

    /**
     * Looks up the given instruction address in the LCT and returns
     * a LctResult that indicates unpredictable, predictable, or constant.
     * @param inst_addr The address of the instruction to look up.
     * @param bp_history Pointer to any bp history state.
     * @return Whether or not the branch is taken.
     */
    LVPType lookup(ThreadID tid, Addr inst_addr);

    /**
     * Updates the branch predictor with the actual result of a branch.
     * @param tid ThreadID of the thread to predict.
     * @param inst_addr The address of the instruction to update.
     * @param prediction The prediction that was associated with this instruction on decode.
     * @param prediction_correct Whether or not the load prediction was correct.
     * @return Updated LVPType of the location
     */
    LVPType update(ThreadID tid, Addr inst_addr, LVPType prediction, bool prediction_correct);

    // void squash(ThreadID tid, void *bp_history)
    // { assert(bp_history == NULL); }

    /**
     * @brief Resets all saturating counters in LCT.
     */
    void reset();

  private:
    /**
     *  Returns the unpredictable/predictable/constant prediction given
     *  the value of the counter.
     *  @param count The value of the counter.
     *  @return The prediction based on the counter value.
     */
    inline LVPType getPrediction(uint8_t &count);

    /** Calculates the local index based on the PC. */
    inline unsigned getLocalIndex(Addr &PC);

    /** Resets the ctr at the specified index. */
    inline void resetCtr(unsigned local_predictor_idx);

    /** Size of the local predictor. */
    const unsigned localPredictorSize;

    /** Number of bits of the local predictor's counters. */
    const unsigned localCtrBits;

    /** Number of sets. */
    const unsigned localPredictorSets;

    /** Array of counters that make up the local predictor. */
    std::vector<SatCounter> localCtrs;
    std::vector<ThreadID> localCtrThreads;

    /** Mask to get index bits. */
    const unsigned indexMask;

    /** Number of bits to shift instructions by for predictor addresses. */
    const unsigned instShiftAmt;

    const bool invalidateConstToZero;
};

#endif // __CPU_LVP_LOADCLASSIFICATIONTABLE_HH__

```

```
/*
 * Copyright (c) 2004-2006 The Regents of The University of Michigan
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met: redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer;
 * redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution;
 * neither the name of the copyright holders nor the names of its
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#include "cpu/lvp/load_classification_table.hh"

#include "base/intmath.hh"
#include "base/logging.hh"
#include "base/trace.hh"
#include "debug/LCT.hh"

LoadClassificationTable::LoadClassificationTable(const LoadClassificationTableParams *params)
    : SimObject(params),
      localPredictorSize(params->localPredictorSize),
      localCtrBits(params->localCtrBits),
      localPredictorSets(localPredictorSize / localCtrBits),
      localCtrs(localPredictorSets, SatCounter(localCtrBits, 0)),
      localCtrThreads(localPredictorSets, 0),
      indexMask(localPredictorSets - 1),
      invalidateConstToZero(params->invalidateConstToZero),
      instShiftAmt(0) // TODO what is correct???
{
    if (!isPowerOf2(localPredictorSize)) {
        fatal("Invalid LCT local predictor size!\n");
    }

    if (!isPowerOf2(localPredictorSets)) {
        fatal("Invalid number of LCT local predictor sets! Check localCtrBits.\n");
    }

    DPRINTF(LCT, "LCT index mask: %#x\n", indexMask);

    DPRINTF(LCT, "LCT size: %i\n",
            localPredictorSize);

    DPRINTF(LCT, "LCT counter bits: %i\n", localCtrBits);

    DPRINTF(LCT, "instruction shift amount: %i\n",
            instShiftAmt);
}

LVPType
LoadClassificationTable::lookup(ThreadID tid, Addr inst_addr)
{
    unsigned local_predictor_idx = getLocalIndex(inst_addr);

    if (tid != localCtrThreads[local_predictor_idx])
    {
        return LVP_STRONG_UNPREDICTABLE;
    }

    DPRINTF(LCT, "Thread ID: %d", tid);

```

```cpp
    uint8_t counter_val = localCtrs[local_predictor_idx];

    DPRINTF(LCT, "During lookup index %#x had value %i\n",
            local_predictor_idx, (int)counter_val);

    return getPrediction(counter_val);
}

LVPType
LoadClassificationTable::update(ThreadID tid, Addr inst_addr, LVPType prediction, bool prediction_correct)
{
    unsigned local_predictor_idx;

    // No state to restore, and we do not update on the wrong
    // path.
    // if (squashed) {
    //     return;
    // }

    // Update the local predictor.
    local_predictor_idx = getLocalIndex(inst_addr);

    if (tid == localCtrThreads[local_predictor_idx])
    {
        if (prediction_correct) {
            DPRINTF(LCT, "Load classification updated as correct.\n");
            localCtrs[local_predictor_idx]++;
        } else {
            DPRINTF(LCT, "Load classification updated as incorrect.\n");
            if (prediction == LVP_CONSTANT && invalidateConstToZero) {
                resetCtr(local_predictor_idx);
            } else {
                localCtrs[local_predictor_idx]--;
            }
        }
    }
    else
    {
        // Destructive interference with a different thread, reset this index and update the thread
        localCtrThreads[local_predictor_idx] = tid;
        resetCtr(local_predictor_idx);
    }

    uint8_t counter_val = localCtrs[local_predictor_idx];
    return getPrediction(counter_val);
}

void
LoadClassificationTable::reset()
{
    auto zeroCounter = new SatCounter(localCtrBits, 0);
    for (int c = 0; c < localPredictorSets; c++)
    {
        localCtrs[c] = *zeroCounter;
    }

    delete zeroCounter;
}

void
LoadClassificationTable::resetCtr(unsigned local_predictor_idx)
{
    while (localCtrs[local_predictor_idx] != LVP_STRONG_UNPREDICTABLE) {
        localCtrs[local_predictor_idx]--;
    }
```

```
}

inline
LVPType
LoadClassificationTable::getPrediction(uint8_t &count)
{
    // If MSB is 0, value is unpredictable
    // If counter is saturated, value is constant
    // Otherwise, the value is predictable
    return (count == 0) ? LVP_STRONG_UNPREDICTABLE
            : (count >> (localCtrBits - 1)) == 0 ? LVP_WEAK_UNPREDICTABLE
            : count == (power(2, localCtrBits) -1) ? LVP_CONSTANT
            : LVP_PREDICTABLE;
}

inline
unsigned
LoadClassificationTable::getLocalIndex(Addr &inst_addr)
{
    return (inst_addr >> instShiftAmt) & indexMask;
}

LoadClassificationTable*
LoadClassificationTableParams::create()
{
    return new LoadClassificationTable(this);
}

```

C. Constant Verification Unit

```
/**
 * @defgroup   CONSTANT_VERIFICATION_UNIT constant verification unit
 *
 * @brief      This file the implements class of the constant verification unit
 *                                 for a load value predictor.
 *                                 The constant verification unit has the following functions:
 *                                 - Keep track of load addresses which have been deemed "constant"
 *                                 by the LCT
 *                                 - Monitor all committed stores and invalidate a constant load
 *                                 address if a store that is being committed has the same address
 *                                 as the constant load
 *                                 - Compare a load value for address that had been deemed
 *                                 "predictable" by the LCT with the actual load value retrieved
 *                                 from memory
 *                                 - Provide feedback to the LCT for all predictable loads
 *                                 - Possible implementation of this class could also include a
 *                                 mechanism to pass load values from the commit stage back to the
 *                                 LVPT (Since this will be needed if an instruction is to be
 *                                 promoted from unpredictable to predictable and so on.)
 *
 * @author     Satvik
 * @date       2021
 */

#ifndef __CPU_LVP_CONSTANT_VERIFICATION_UNIT__
#define __CPU_LVP_CONSTANT_VERIFICATION_UNIT__

#include "sim/sim_object.hh"
#include "base/types.hh"
#include "params/ConstantVerificationUnit.hh"
#include "base/statistics.hh"

#include <vector>
#include <map>
```

```
#include <iterator>
#include <algorithm>
#include <chrono>

// Replacement Policies
#define RP_FIFO 1        // CAM is implemented as a queue
#define RP_LRU  2               // The least recently used entry is kicked out
#define RP_NLRU 3               // The not least recently used entry is kicked out
#define RP_MRU  4        // The most recently used entry is kicked out
#define RP_NMRU 5        // The not most recently used entry is kicked out

#define DURATION std::chrono::duration_cast<std::chrono::duration<double, std::milli>>
typedef std::chrono::high_resolution_clock::time_point TimePoint;

/**
 * @brief      The CVU stores the load address and the index of the LVPT in a
 *             concatenated form in its CAM. This is searched by loads that are
 *             predicted as constant and by store addresses. Thus, the search of
 *             the CAM will be different for each case. The search for a load
 *             predicted as "constant" will be done on the entire entry in the
 *             CAM while a store address will only need to be compared with the
 *             load address stored in the CAM.
 */

/**
 * @brief      Data stored in every entry of the CVU CAM
 */
struct CAMEntry {
  Addr lvpt_index;
  Addr pc;
  Addr load_address;
  TimePoint entry;
  TimePoint access;
};

class ConstantVerificationUnit : public SimObject {
public:
  ConstantVerificationUnit(ConstantVerificationUnitParams *p);

  ~ConstantVerificationUnit();

  /**
   * @brief      A store address will be provided to the CVU which will then
   *                              search the CAM and invalidate every entry which matches the
   *                              input address. This will also generate the control signals
   *                              required for communicating with the LCT.
   *
   * @param[in]  address  The store address
   */
  void processStoreAddress(ThreadID tid, Addr address);

  /**
   * @brief      Check if a load address classified as constant is present in
   *                     the CVU CAM.
   *
   * @param[in]  pc         The PC for this load instruction
   * @param[in]  lvptIndex  The lvpt index corresponding to that load address
   * @param[in]  tid        Thread ID
   *
   * @return     True if the load address, LVPT index pair exist in the CAM
   *                              False otherwise
   */
  bool processLoadAddress(Addr load_address, Addr lvptIndex, ThreadID tid);

  /**
             * @brief      Inserts info of a new constant load into the CVU CAM
```

```
          *
          * @param[in]  pc         Instruction address
          * @param[in]  address    The load address
          * @param[in]  lvptIndex  The lvpt index
          * @param[in]  tid        The tid
          *
          * @return     True if the update is successful
          */
         bool updateConstLoad(Addr pc, Addr address, Addr lvptIndex, ThreadID tid);

         /**
          * @brief      Replaces an entry in the CVU CAM with a new one according to
          *             a replacement policy
          *
          * @param[in]  new_entry  The new entry
          * @param[in]  tid        The tid
          */
         void replaceBlock(struct CAMEntry new_entry, ThreadID tid);

         /**
          * @brief      Print stats
          */
         void regStats() override;

private:
         /**
          * The CVU Content Addressable Memory;
          * If a store address is found in this memory, the corresponding entry is
          * invalidated and the invalidation also triggers an update routine which
          * tells the LCT that this load address is no longer constant.
          * The vector stores an ordered pair (load address, LVPT index)
          */
         std::list<struct CAMEntry> _cvuCAM[64];

         /**
          * Number of entries in the CVU CAM
          */
         uint32_t _numEntries;

         /**
          * Number of loads marked "constant" which were incorrectly predicted.
          */
         Stats::Scalar _numConstantHits;

         /**
          * Number of loads marked "constant" that were correctly predicted.
          */
         Stats::Scalar _numConstantMiss;

         /**
          * Number of store addresses which hit in the CVU CAM
          */
         Stats::Scalar _numStoreHits;

         /**
          * Number of store addresses which missed in the CVU CAM.
          */
         Stats::Scalar _numStoreMiss;

         /**
          * Replacement policy of the CAM
          */
         uint8_t _replacementPolicy;

         /**
          * Number of CAM blocks replaced.
```

```
             */
            Stats::Scalar _numReplacements;

            /**
             * Total stores processed by the CVU
             */
            Stats::Formula _numStoreAccesses;

            /**
             * Total Loads processed by the CVU
             */
            Stats::Formula _numLoadAccesses;
  };


  #endif

```

```
/**
 * @defgroup   CONSTANT_VERIFICATION_UNIT constant verification unit
 *
 * @brief      This file implements constant verification unit.
 *
 * @author     Satvik
 * @date       2021
 */
#include "cpu/lvp/constant_verification_unit.hh"

#include "base/intmath.hh"
#include "base/logging.hh"
#include "base/trace.hh"
#include "debug/CVU.hh"

ConstantVerificationUnit::ConstantVerificationUnit(ConstantVerificationUnitParams *params) :
    SimObject(params), _numEntries(params->entries), _numConstantHits(0),
    _numConstantMiss(0), _numStoreHits(0), _numStoreMiss(0),
    _replacementPolicy(params->replacementPolicy), _numReplacements(0)
{}

ConstantVerificationUnit::~ConstantVerificationUnit() {}

void ConstantVerificationUnit::processStoreAddress(ThreadID tid,

             Addr address) {
  DPRINTF(CVU, "[TID %d]: Store address: 0x%x being searched in CVU CAM\n", tid, address);
  // Only the load address needs to be compared with the store address
  bool found = false;
  auto itr = _cvuCAM[(uint16_t)tid%64].begin();
  while (itr != _cvuCAM[(uint16_t)tid%64].end()) {
            struct CAMEntry temp = *itr;
            if (temp.load_address == address) {
                      DPRINTF(CVU, "[TID %d]: Found store address: 0x%x in CVU CAM\n", tid, address);
                      itr = _cvuCAM[(uint16_t)tid%64].erase(itr);
                      ++_numStoreHits;
                      found = true;
                      if (itr == _cvuCAM[(uint16_t)tid%64].end()) break;
            }
            itr++;
  }
  if (!found) {
            DPRINTF(CVU, "[TID %d]: Address 0x%x not found in CVU CAM\n", tid, address);
            ++_numStoreMiss;
  }
}

```

```
bool ConstantVerificationUnit::processLoadAddress(Addr load_address,
            Addr lvptIndex,
            ThreadID tid) {
  // Both load address and LVPT index have to be searched
  auto itr = _cvuCAM[(uint16_t)tid%64].begin();
  while (itr != _cvuCAM[(uint16_t)tid%64].end()) {
            struct CAMEntry temp = *itr;
            if (temp.load_address == load_address && temp.lvpt_index == lvptIndex) {
                        DPRINTF(CVU, "[TID %d] Load address: 0x%x matched in CVU CAM\n", tid,
load_address);
                        ++_numConstantHits;
                        // Update the end time of this entry (for replacement purposes)
                        itr->access = std::chrono::high_resolution_clock::now();
                        return true;
            }
            itr++;
  }
  ++_numConstantMiss;
  DPRINTF(CVU, "[TID %d] Load address: 0x%x not found in CVU CAM\n", tid, load_address);
  return false;
}

bool ConstantVerificationUnit::updateConstLoad(Addr pc, Addr address,
Addr lvptIndex, ThreadID tid) {
  DPRINTF(CVU, "[TID %d]: Adding load address: 0x%x with PC: 0x%x to CVU CAM\n", tid, address, pc);
  struct CAMEntry temp;
  temp.pc = pc;
  temp.lvpt_index = lvptIndex;
  temp.load_address = address;
  temp.entry = std::chrono::high_resolution_clock::now();
  temp.access = temp.entry;
  if(_cvuCAM[(uint16_t)tid%64].size() >= _numEntries) {
            // CAM is full, replace a block according to a set replacement policy
            DPRINTF(CVU, "[TID %d]: No space for load address: 0x%x in CVU CAM\n", tid, address);
            this->replaceBlock(temp, tid);
            return true;
  }
  _cvuCAM[(uint16_t)tid%64].push_back(temp);
  return true;
}

void ConstantVerificationUnit::replaceBlock(struct CAMEntry new_entry, ThreadID tid) {
  _numReplacements++;
  if(_replacementPolicy == RP_FIFO) {
            // Kick the oldest entry from the CAM
            auto itr = _cvuCAM[(uint16_t)tid%64].begin();
            auto itr_replace = itr;
            TimePoint current_time = std::chrono::high_resolution_clock::now();
            double time_diff = DURATION(current_time - itr->entry).count();
            itr++;
            while(itr != _cvuCAM[(uint16_t)tid%64].end()) {
                        if(time_diff < DURATION(current_time - itr->entry).count()) {
                                    itr_replace = itr;
                                    time_diff = DURATION(current_time - itr->entry).count();
                                    }
                                    itr++;
                        }
                        DPRINTF(CVU, "Entry with PC: 0x%x being ejected from CVU CAM\n", itr_replace->pc);
                        _cvuCAM[(uint16_t)tid%64].erase(itr_replace);
                        _cvuCAM[(uint16_t)tid%64].push_back(new_entry);
                        return;
            }
            if(_replacementPolicy == RP_LRU) {
```

```
		// Kick the least recently used (compare the access)
		auto itr = _cvuCAM[(uint16_t)tid%64].begin();
		auto itr_replace = itr;
		TimePoint current_time = std::chrono::high_resolution_clock::now();
		double time_diff = DURATION(current_time - itr->access).count();
		itr++;
		while(itr != _cvuCAM[(uint16_t)tid%64].end()) {
			if(time_diff < DURATION(current_time - itr->access).count()) {
				itr_replace = itr;
				time_diff = DURATION(current_time - itr->access).count();
			}
			itr++;
		}
		DPRINTF(CVU, "Entry with PC: 0x%x being ejected from CVU CAM\n", itr_replace->pc);
		_cvuCAM[(uint16_t)tid%64].erase(itr_replace);
		_cvuCAM[(uint16_t)tid%64].push_back(new_entry);
		return;
	}
	if(_replacementPolicy == RP_NLRU) {
		// Kick the not least recently used (compare the access)
		auto itr = _cvuCAM[(uint16_t)tid%64].begin();
		auto itr_replace = itr;
		TimePoint current_time = std::chrono::high_resolution_clock::now();
		double time_diff = DURATION(current_time - itr->access).count();
		itr++;
		while(itr != _cvuCAM[(uint16_t)tid%64].end()) {
			if(time_diff < DURATION(current_time - itr->access).count()) {
				itr_replace = itr;
				time_diff = DURATION(current_time - itr->access).count();
			}
			itr++;
		}
		itr = _cvuCAM[(uint16_t)tid%64].begin();
		while(itr == itr_replace && itr != _cvuCAM[(uint16_t)tid%64].end()) {
			itr++;
		}
		itr_replace = itr;
		DPRINTF(CVU, "Entry with PC: 0x%x being ejected from CVU CAM\n", itr_replace->pc);
		_cvuCAM[(uint16_t)tid%64].erase(itr_replace);
		_cvuCAM[(uint16_t)tid%64].push_back(new_entry);
		return;
	}
	if(_replacementPolicy == RP_MRU) {
		// Kick the least recently used (compare the access)
		auto itr = _cvuCAM[(uint16_t)tid%64].begin();
		auto itr_replace = itr;
		TimePoint current_time = std::chrono::high_resolution_clock::now();
		double time_diff = DURATION(current_time - itr->access).count();
		itr++;
		while(itr != _cvuCAM[(uint16_t)tid%64].end()) {
			if(time_diff > DURATION(current_time - itr->access).count()) {
				itr_replace = itr;
				time_diff = DURATION(current_time - itr->access).count();
			}
			itr++;
		}
		DPRINTF(CVU, "Entry with PC: 0x%x being ejected from CVU CAM\n", itr_replace->pc);
		_cvuCAM[(uint16_t)tid%64].erase(itr_replace);
		_cvuCAM[(uint16_t)tid%64].push_back(new_entry);
		return;
	}
	if(_replacementPolicy == RP_NMRU) {
		// Kick the not least recently used (compare the access)
		auto itr = _cvuCAM[(uint16_t)tid%64].begin();
		auto itr_replace = itr;
		TimePoint current_time = std::chrono::high_resolution_clock::now();
```

```
                    double time_diff = DURATION(current_time - itr->access).count();
                    itr++;
                    while(itr != _cvuCAM[(uint16_t)tid%64].end()) {
                             if(time_diff > DURATION(current_time - itr->access).count()) {
                                       itr_replace = itr;
                                       time_diff = DURATION(current_time - itr->access).count();
                             }
                             itr++;
                    }
                    itr = _cvuCAM[(uint16_t)tid%64].begin();
                    while(itr == itr_replace && itr != _cvuCAM[(uint16_t)tid%64].end()) {
                             itr++;
                    }
                    itr_replace = itr;
                    DPRINTF(CVU, "Entry with PC: 0x%x being ejected from CVU CAM\n", itr_replace->pc);
                    _cvuCAM[(uint16_t)tid%64].erase(itr_replace);
                    _cvuCAM[(uint16_t)tid%64].push_back(new_entry);
                    return;
          }
}

void ConstantVerificationUnit::regStats() {
          SimObject::regStats();

          _numConstantHits.name(name() + ".constLoadHits")
                                                 .desc("Number of loads marked constant that hit in the CVU CAM");
          _numConstantMiss.name(name() + ".constLoadMisses")
                                                 .desc("Number of loads marked constant that missed in the CVU CAM");
          _numStoreHits.name(name() + ".numStoreHits")
                                        .desc("Number of stores that hit in the CVU CAM");
          _numStoreMiss.name(name() + ".numStoreMisses")
                                        .desc("Number of stores that missed in the CVU CAM");
          _numReplacements.name(name() + ".numCAMReplacements")
                                           .desc("Number of CVU CAM entries replaced");
          _numStoreAccesses.name(name() + ".numStoreAccesses")
                                                  .desc("Number of store instructions which accessed the CVU");
          _numLoadAccesses.name(name() + ".numLoadAccesses")
                                                 .desc("Number of loads marked constant which accessed the CVU");

          _numLoadAccesses = _numConstantMiss + _numConstantHits;
          _numStoreAccesses = _numStoreHits + _numStoreMiss;
}

ConstantVerificationUnit*
ConstantVerificationUnitParams::create()
{
    return new ConstantVerificationUnit(this);
}
```

II. Modifications for the DerivO3 CPU

```
diff --git a/gem5/src/cpu/o3/O3CPU.py b/gem5/src/cpu/o3/O3CPU.py
index 51d9121..864d54e 100644
--- a/gem5/src/cpu/o3/O3CPU.py
+++ b/gem5/src/cpu/o3/O3CPU.py
@@ -46,6 +46,7 @@ from m5.objects.BaseCPU import BaseCPU
 from m5.objects.FUPool import *
 from m5.objects.O3Checker import O3Checker
 from m5.objects.BranchPredictor import *
+from m5.objects.LoadValuePredictionUnit import *

 class FetchPolicy(ScopedEnum):
     vals = [ 'SingleThread', 'RoundRobin', 'Branch', 'IQCount', 'LSQCount' ]
@@ -127,7 +128,7 @@ class DerivO3CPU(BaseCPU):
     LQEntries = Param.Unsigned(32, "Number of load queue entries")
     SQEntries = Param.Unsigned(32, "Number of store queue entries")
     LSQDepCheckShift = Param.Unsigned(4, "Number of places to shift addr before check")
-    LSQCheckLoads = Param.Bool(True,
+    LSQCheckLoads = Param.Bool(False,
         "Should dependency violations be checked for loads & stores or just stores")
     store_set_clear_period = Param.Unsigned(250000,
             "Number of load/store insts before the dep predictor should be invalidated")
@@ -174,6 +175,12 @@ class DerivO3CPU(BaseCPU):
     branchPred = Param.BranchPredictor(TournamentBP(numThreads =
                                                         Parent.numThreads),
                                        "Branch Predictor")
+    ##
+    ## SATVIK:
+    ## LVP object that comes from somewhere
+    ##
+    loadValPred = Param.LoadValuePredictionUnit(LoadValuePredictionUnit(),
+                                                  "Load value predictor")
     needsTSO = Param.Bool(buildEnv['TARGET_ISA'] == 'x86',
                           "Enable TSO Memory model")

diff --git a/gem5/src/cpu/o3/commit_impl.hh b/gem5/src/cpu/o3/commit_impl.hh
index 75d065f..61befa4 100644
--- a/gem5/src/cpu/o3/commit_impl.hh
+++ b/gem5/src/cpu/o3/commit_impl.hh
@@ -706,6 +706,9 @@ DefaultCommit<Impl>::tick()
             DPRINTF(Commit,"[tid:%i] Instruction [sn:%llu] PC %s is head of"
                     " ROB and ready to commit\n",
                     tid, inst->seqNum, inst->pcState());
+           if(inst->isLoad()) {
+               DPRINTF(Commit, "Inst ready to commit is a load:[sn:%llu]\n", inst->seqNum);
+           }

         } else if (!rob->isEmpty(tid)) {
             const DynInstPtr &inst = rob->readHeadInst(tid);
@@ -715,6 +718,18 @@ DefaultCommit<Impl>::tick()
             DPRINTF(Commit,"[tid:%i] Can't commit, Instruction [sn:%llu] PC "
                     "%s is head of ROB and not ready\n",
                     tid, inst->seqNum, inst->pcState());
+           for(int i = 0; i < inst->numSrcRegs(); i++) {
+               if(!inst->isReadySrcRegIdx(i)) {
+                   auto ptr = inst->renamedSrcRegIdx(i);
+                   DPRINTF(Commit, "Src reg %d for inst[%llu]: 0x%x not ready\n", ptr->index(),inst->seqNum, inst->instAddr());
+               }
+           }
+           if(inst->isConstPredictionCorrect() && !inst->readyToCommit()) {
+               DPRINTF(Commit, "Head instruction[%d] wasn't ready in ROB and was a const load\n", inst->seqNum);
+               //inst->setExecuted();
+               //inst->setResultReady();
+                //inst->setCanCommit();
+            }
         }

         DPRINTF(Commit, "[tid:%i] ROB has %d insts & %d free entries.\n",
diff --git a/gem5/src/cpu/o3/cpu.cc b/gem5/src/cpu/o3/cpu.cc
index 01938f1..c28c25f 100644
--- a/gem5/src/cpu/o3/cpu.cc
+++ b/gem5/src/cpu/o3/cpu.cc
@@ -126,6 +126,7 @@ FullO3CPU<Impl>::FullO3CPU(DerivO3CPUParams *params)
                   params->activity),

       globalSeqNum(1),
+      lvp(params->loadValPred),
       system(params->system),
       lastRunningCycle(curCycle())
 {
diff --git a/gem5/src/cpu/o3/cpu.hh b/gem5/src/cpu/o3/cpu.hh
index 0447275..cbf61ce 100644
--- a/gem5/src/cpu/o3/cpu.hh
+++ b/gem5/src/cpu/o3/cpu.hh
@@ -64,6 +64,8 @@
 #include "params/DerivO3CPU.hh"
 #include "sim/process.hh"

+#include "cpu/lvp/load_value_prediction_unit.hh"
+
 template <class>
 class Checker;
 class ThreadContext;
@@ -426,6 +428,26 @@ class FullO3CPU : public BaseO3CPU

     RegVal readArchFloatReg(int reg_idx, ThreadID tid);

+    void tagLVPDestReg(PhysRegIdPtr reg) {
+        regFile.insertPredictedLoadRegister(reg);
+    }
+
+    bool checkLVPTag(PhysRegIdPtr src_reg) {
+        return regFile.checkPredictedLoadRegister(src_reg);
+    }
+
+    void removeLVPTag(PhysRegIdPtr reg) {
+        regFile.clearPredictedLoadRegister(reg);
+    }
+
+    void addToMispredictList(PhysRegIdPtr reg) {
+        regFile.addToMispredictList(reg);
+    }
+
+    bool searchMispredictList(PhysRegIdPtr src_reg) {
+        return regFile.searchMispredictList(src_reg);
+    }
+
     const VecRegContainer& readArchVecReg(int reg_idx, ThreadID tid) const;
     /** Read architectural vector register for modification. */
     VecRegContainer& getWritableArchVecReg(int reg_idx, ThreadID tid);
@@ -690,6 +712,16 @@ class FullO3CPU : public BaseO3CPU
      */
     Checker<Impl> *checker;

+    /**
+     * SATVIK:
+     * Create an object for the LVP here?
+     * Makes sense since the LVP is a module that is shared across multiple
+     * stages of the CPU, unlike the bpred which belongs in fetch
+     * Any LVP API can then be called using inst->cpu->LVPAPI() since the
+     * cpu object is passed to the dyn_inst class.
+     */
+    LoadValuePredictionUnit *lvp;
+
     /** Pointer to the system. */
     System *system;

diff --git a/gem5/src/cpu/o3/dyn_inst.hh b/gem5/src/cpu/o3/dyn_inst.hh
index 5f2a588..8064f79 100644
--- a/gem5/src/cpu/o3/dyn_inst.hh
+++ b/gem5/src/cpu/o3/dyn_inst.hh
@@ -50,6 +50,7 @@
 #include "cpu/base_dyn_inst.hh"
 #include "cpu/inst_seq.hh"
 #include "cpu/reg_class.hh"
+#include "cpu/lvp/load_value_prediction_unit.hh"

 class Packet;

@@ -116,6 +117,16 @@ class BaseO3DynInst : public BaseDynInst<Impl>
     /** Number of destination misc. registers. */
     uint8_t _numDestMiscRegs;

+    /**
+     * The classification of this instruction as per the LCT (only for loads)
+     */
+    LVPType _classification;
+
+    RegVal _predictedVal;
+
+    bool _predictionCorrect;
+
+    bool _specExecOnLoad;

   public:
 #if TRACING_ON
@@ -423,6 +434,123 @@ class BaseO3DynInst : public BaseDynInst<Impl>
         this->cpu->setCCReg(this->_destRegIdx[idx], val);
         BaseDynInst<Impl>::setCCRegOperand(si, idx, val);
     }
+
+    std::pair<LVPType, RegVal>
+    predictLoad(ThreadID tid) {
+        std::pair<LVPType, RegVal> temp = this->cpu->lvp->predictLoad(tid,
+                                                      this->instAddr());
+        _classification = temp.first;
+        _predictedVal = temp.second;
+        return temp;
+    }
+
+    bool
+    verifyConstLoad(ThreadID tid) {
+        Addr lvpt_index = this->cpu->lvp->lookupLVPTIndex(tid,
+                                                  this->instAddr());
+        _predictionCorrect = this->cpu->lvp->processLoadAddress(tid,
+                                          this->instAddr(), this->effAddr, lvpt_index);
+        return _predictionCorrect;
+    }
+
+    bool
+    isConstPredictionCorrect() {
+        return _predictionCorrect;
+    }
+
+    RegVal
+    getPredictedValue() {
+        return _predictedVal;
+    }
+
+    bool isConstLoad() {
+        return _classification == LVP_CONSTANT;
+    }
+
+    void
+    tagLVPDestReg(int idx) {
+        this->cpu->tagLVPDestReg(this->_destRegIdx[idx]);
+    }
+
+    bool
+    checkLVPTag(int idx) {
+        return this->cpu->checkLVPTag(this->_srcRegIdx[idx]);
+    }
+
+    PhysRegIdPtr
+    getSrcRegPtr(int idx) {
+        return this->_srcRegIdx[idx];
+    }
+
+    PhysRegIdPtr
+    getDestRegPtr(int idx) {
+        return this->_destRegIdx[idx];
+    }
+
+    void
+    removeLVPTag(int idx) {
+        this->cpu->removeLVPTag(this->_destRegIdx[idx]);
+    }
+
+    void
+    addToMispredictList(int idx) {
+        this->cpu->addToMispredictList(this->_destRegIdx[idx]);
+    }
+
+    bool
+    searchMispredictList(int idx) {
+        return this->cpu->searchMispredictList(this->_srcRegIdx[idx]);
+    }
+
+    bool
+    verifyPrediction(int idx) {
+        RegVal temp = 0;
+        auto ptr = this->_destRegIdx[idx];
+        if(ptr->isIntPhysReg()) {
+            temp = this->cpu->readIntReg(ptr);
+        }
+        else if(ptr->isFloatPhysReg()) {
+            temp = this->cpu->readFloatReg(ptr);
+        }
+        else {
+           return true;
+        }
+        this->removeLVPTag(idx);
+        if(!this->effAddrValid()) panic("Virtual address not valid yet");
+        return this->cpu->lvp->verifyPrediction(this->threadNumber,
+                        this->instAddr(), this->effAddr, temp, _predictedVal,
+                        _classification);
+    }
+
+    void
+    lvpStoreAddressLookup() {
+        if(!this->effAddrValid()) {
```

```
+            panic("Virtual address of store not valid yet");
+        }
+        else {
+            this->cpu->lvp->processStoreAddress(this->threadNumber, this->effAddr);
+        }
+    }
+
+    void
+    speculativeExecOnLoad() {
+        _specExecOnLoad = true;
+    }
+
+    void
+    resetSpeculativeExecOnLoad() {
+        _specExecOnLoad = false;
+    }
+
+    bool
+    isExecOnSpecLoad() {
+        return _specExecOnLoad;
+    }
+
+    bool
+    isSpeculatedLoad() {
+        return (_classification == LVP_PREDICTABLE);
+    }
 };

 #endif // __CPU_O3_ALPHA_DYN_INST_HH__
diff --git a/gem5/src/cpu/o3/dyn_inst_impl.hh b/gem5/src/cpu/o3/dyn_inst_impl.hh
index 8a6a434..c1c1411 100644
--- a/gem5/src/cpu/o3/dyn_inst_impl.hh
+++ b/gem5/src/cpu/o3/dyn_inst_impl.hh
@@ -106,6 +106,11 @@ BaseO3DynInst<Impl>::initVars()

     _numDestMiscRegs = 0;

+    _classification = LVP_STRONG_UNPREDICTABLE;
+    _predictedVal = 0;
+    _predictionCorrect = false;
+    _specExecOnLoad = false;
+
 #if TRACING_ON
     // Value -1 indicates that particular phase
     // hasn't happened (yet).
diff --git a/gem5/src/cpu/o3/iew_impl.hh b/gem5/src/cpu/o3/iew_impl.hh
index 497c532..0435928 100644
--- a/gem5/src/cpu/o3/iew_impl.hh
+++ b/gem5/src/cpu/o3/iew_impl.hh
@@ -57,6 +57,7 @@
 #include "debug/Activity.hh"
 #include "debug/Drain.hh"
 #include "debug/IEW.hh"
+#include "debug/LVP.hh"
 #include "debug/O3PipeView.hh"
 #include "params/DerivO3CPU.hh"

@@ -957,6 +958,11 @@ template <class Impl>
 void
 DefaultIEW<Impl>::dispatchInsts(ThreadID tid)
 {
+    /**
+     * SATVIK:
+     * LVP will forward the predicted load value to this API
+     */
+
     // Obtain instructions from skid buffer if unblocking, or queue from rename
     // otherwise.
     std::queue<DynInstPtr> &insts_to_dispatch =
@@ -964,6 +970,7 @@ DefaultIEW<Impl>::dispatchInsts(ThreadID tid)
         skidBuffer[tid] : insts[tid];

     int insts_to_add = insts_to_dispatch.size();
+    DPRINTF(IEW, "Insts to dispatch: %d\n", insts_to_add);

     DynInstPtr inst;
     bool add_to_iq = false;
@@ -1087,21 +1094,179 @@ DefaultIEW<Impl>::dispatchInsts(ThreadID tid)
             DPRINTF(IEW, "[tid:%i] Issue: Memory instruction "
                     "encountered, adding to LSQ.\n", tid);

-            // Reserve a spot in the load store queue for this
-            // memory access.
-            ldstQueue.insertLoad(inst);
+            /**
+             * SATVIK:
+             * A load classified as "constant" by the LCT need not be pushed in
+             * the LDST queue?
+             * -> In case the constant value has changed, this load will need to
+             * be issued again. A CVU lookup can be initiated here and the
+             * result of the lookup can be checked during writeback. This will
+             * give more meaningful results since the lookup will take atleast
+             * 1 cycle. In case the constant load is invalid, the load will be
+             * reissued from writeback.
+             *
+             * Implementation note-
+             * The PC can be sent to the LCT and LVPT for prediction during
+             * fetch. However, it makes sense to store the predicted value and
+             * forward it to other dependents only during dispatch. Since there
+             * is already a delay of more than 1 cycle between fetch and
+             * dispatch, it won't make a difference if the prediction and the
+             * CVU lookup are done here itself. The CVU lookup for a constant
+             * load and the prediction itself will happen for every load
+             * instruction in all cases so it doesn't make a difference where
+             * the two methods are called from.
+             * Inserting the load in the ldstQueue for a load that is correctly
+             * predicted as constant will waste a slot in the ldstQueue. It is
+             * thus better to verify the constant load here and only add it to
+             * the ldstQueue if the verification fails.
+             *
+             * TODO: Not too sure about the wakeDependents call below-
+             */
+            std::pair<LVPType, RegVal> prediction = inst->predictLoad(tid);
+
+            if(prediction.first == LVP_CONSTANT) {
+                // Trigger a CVU lookup of the lvpt index and the load address
+                bool const_valid = false; //inst->verifyConstLoad(tid);
+                if (!const_valid) {
+                    // This prediction failed
+                    // The CVU will have already incremented the misprediction
+                    // counter
+                    // Push this load instruction in the ldstQueue
+                    ldstQueue.insertLoad(inst);
+
+                    ++iewDispLoadInsts;
+
+                    add_to_iq = true;
+
+                    toRename->iewInfo[tid].dispatchedToLQ++;
+                }
+                else {
+                    // This load was predicted correctly
+                    // A correct prediction for a constant load need not update
+                    // the LCT.
+                    // Write the predicted value to the allocated register and
+                    // forward all values
+                    // Mark this load as executed and ready to commit.
+                    //toRename->iewInfo[tid].dispatchedToLQ++;
+                    //toRename->iewInfo[tid].dispatched++;
+                    //insts_to_dispatch.pop();
+                    //inst->setIssued();
+                    //inst->setExecuted();
+                    //inst->setCanCommit();
+                    //inst->setResultReady();
+                    //instToCommit(inst);
+                    //activityThisCycle();
+                    //add_to_iq = false;
+                    DPRINTF(LVP, "IEW: Const load [sn: %d] found by LVP: 0x%x\n", inst->seqNum, inst->instAddr());
+                    ldstQueue.insertLoad(inst);
+
+                    ++iewDispLoadInsts;
+
+                    add_to_iq = true;
+
+                    toRename->iewInfo[tid].dispatchedToLQ++;
+
+                    // Pass the load value to the destination register
+                    /*if(inst->numDestRegs() == 1) {
+                        auto ptr = inst->renamedDestRegIdx(0);
+                        if(inst->isInteger()) {
+                            inst->setIntRegOperand(inst->staticInst.get(),
+                                                   0, prediction.second);
+                            //instQueue.wakeDependents(inst);
+                            scoreboard->setReg(inst->renamedDestRegIdx(0));
+                            //DPRINTF(LVP, "LVP Const Inst[%llu]: ox%x getting reg %d as ready: %d\n", inst->seqNum, inst->instAddr(), ptr->index(), scoreboard->getReg(ptr));
+                        }
+                        else if(inst->isFloating()) {
+                            inst->setFloatRegOperandBits(inst->staticInst.get(),
+                                                   0, prediction.second);
+                            //instQueue.wakeDependents(inst);
+                            scoreboard->setReg(inst->renamedDestRegIdx(0));
+                            //DPRINTF(LVP, "LVP Const Inst[%llu]: ox%x getting reg %d as ready: %d\n", inst->seqNum, inst->instAddr(), ptr->index(), scoreboard->getReg(ptr));
+                        }
+                        else {
+                            // This isn't supposed to happen
+                        }
+                    }
+                    else {
+                        // This isn't supposed to happen (except maybe for
+                        // vectors)
+                    }
+                    */
+                }
+            }
+            else if(prediction.first == LVP_PREDICTABLE) {
+                // Need to mark this instruction as predictable so that the CVU
+                // can verify later. -> this has been done during the
+                // predictLoad() call.
+
+                // These loads will follow the normal execution flow: but the
+                // predicted value will be passed to all consumers.
+                // The destination register will also need to be tagged with the
+                // predictable flag so that instructions which consume this
+                // register are not flushed from the IQ.
+                if(inst->numDestRegs() == 1) {
```

```
+                     // Tag the destination register for subsequent dependent
+                     // instructions
+                     //inst->tagLVPDestReg(0);
+                     /*
+                     if(inst->isInteger()) {
+                         inst->setIntRegOperand(inst->staticInst.get(),
+                                                0, prediction.second);
+                         instQueue.wakeDependents(inst);
+                         scoreboard->setReg(inst->renamedDestRegIdx(0));
+                     }
+                     else if(inst->isFloating()) {
+                         inst->setFloatRegOperand(inst->staticInst.get(),
+                                                0, prediction.second);
+                         instQueue.wakeDependents(inst);
+                         scoreboard->setReg(inst->renamedDestRegIdx(0));
+                     }
+                     else {
+                         // This isn't supposed to happen
+                     }
+                     */
+                 }
+                 else {
+                     // This isn't supposed to happen (except maybe for
+                     // vectors)
+                 }

-             ++iewDispLoadInsts;
+                 // Reserve a spot in the load store queue for this
+                 // memory access.
+                 ldstQueue.insertLoad(inst);

-             add_to_iq = true;
+                 ++iewDispLoadInsts;

-             toRename->iewInfo[tid].dispatchedToLQ++;
+                 add_to_iq = true;
+
+                 toRename->iewInfo[tid].dispatchedToLQ++;
+             }
+             else {
+
+                 // Reserve a spot in the load store queue for this
+                 // memory access.
+                 ldstQueue.insertLoad(inst);
+
+                 ++iewDispLoadInsts;
+
+                 add_to_iq = true;
+
+                 toRename->iewInfo[tid].dispatchedToLQ++;
+             }
          } else if (inst->isStore()) {
              DPRINTF(IEW, "[tid:%i] Issue: Memory instruction "
                      "encountered, adding to LSQ.\n", tid);

              ldstQueue.insertStore(inst);

+             /**
+              * SATVIK:
+              * Stores could finish execution out of order with respect to other
+              * loads. Processing the store address after execution has completed
+              * could invalidate an older load which is constant.
+              */
+             //inst->lvpStoreAddressLookup();
+
              ++iewDispStoreInsts;
```

```

             if (inst->isStoreConditional()) {
@@ -1294,6 +1459,9 @@ DefaultIEW<Impl>::executeInsts()
             } else if (inst->isLoad()) {
                 // Loads will mark themselves as executed, and their writeback
                 // event adds the instruction to the queue to commit
+                if(inst->isConstLoad() && !inst->strictlyOrdered() && !inst->isInstPrefetch())
+                    inst->verifyConstLoad(inst->threadNumber);
+
                 fault = ldstQueue.executeLoad(inst);

                 if (inst->isTranslationDelayed() &&
@@ -1309,6 +1477,26 @@ DefaultIEW<Impl>::executeInsts()
                 if (inst->isDataPrefetch() || inst->isInstPrefetch()) {
                     inst->fault = NoFault;
                 }
+
+                /**
+                 * SATVIK:
+                 * Check the LVP prediction here
+                if(inst->isExecuted() && fault == NoFault) {
+                    if (inst->numDestRegs() == 1) {
+                        bool verify = inst->verifyPrediction(0);
+                        if(!verify && inst->isSpeculatedLoad()) {
+                            // Add the destination register to a list of
+                            // mispredicted registers so that the instns in the
+                            // IQ know that they have to execute again.
+                            //instQueue.addToMispredictList(inst, 0);
+                        }
+                    }
+                    else {
+                        // This shouldn't happen
+                    }
+                }
+                 */
+
             } else if (inst->isStore()) {
                 fault = ldstQueue.executeStore(inst);

@@ -1334,6 +1522,16 @@ DefaultIEW<Impl>::executeInsts()
                     activityThisCycle();
                 }

+                /**
+                 * SATVIK:
+                 * At this point, if there is no error, this store instruction
+                 * will commit at some point. So invalidating any CVU CAM
+                 * entries that have this store address should be done here.
+                 */
+                if(inst->isExecuted() && fault == NoFault) {
+                    inst->lvpStoreAddressLookup();
+                }
+
                 // Store conditionals will mark themselves as
                 // executed, and their writeback event will add the
                 // instruction to the queue to commit.
@@ -1352,6 +1550,9 @@ DefaultIEW<Impl>::executeInsts()
                     inst->forwardOldRegs();
             }

+            /**
+             * Do this only if the source operands were not speculated.
+             */
             inst->setExecuted();

             instToCommit(inst);
```

```
@@ -1488,6 +1689,41 @@ DefaultIEW<Impl>::writebackInsts()
         // are first sent to commit.  Instead commit must tell the LSQ
         // when it's ready to execute the strictly ordered load.
         if (!inst->isSquashed() && inst->isExecuted() && inst->getFault() == NoFault) {
+            if(inst->isLoad()) {
+                if(inst->isConstPredictionCorrect() && !inst->strictlyOrdered() && !inst->isInstPrefetch()) {
+                    // Pass the load value to the destination register
+                    //inst->setCanCommit();
+                    //checkMisprediction(inst);
+                    if(inst->numDestRegs() == 1) {
+                        auto ptr = inst->renamedDestRegIdx(0);
+                        if(ptr->isIntPhysReg()) {
+                            inst->setIntRegOperand(inst->staticInst.get(),
+                                                   0, inst->getPredictedValue());
+                            //instQueue.wakeDependents(inst);
+                            scoreboard->setReg(ptr);
+                            DPRINTF(LVP, "LVP Const Inst[%llu]: ox%x setting reg %d as ready\n", inst->seqNum, inst->instAddr(), ptr->index());
+                        }
+                        else if(ptr->isFloatPhysReg()) {
+                            inst->setFloatRegOperandBits(inst->staticInst.get(),
+                                                   0, inst->getPredictedValue());
+                            //instQueue.wakeDependents(inst);
+                            scoreboard->setReg(inst->renamedDestRegIdx(0));
+                            DPRINTF(LVP, "LVP Const Inst[%llu]: ox%x setting reg %d as ready\n", inst->seqNum, inst->instAddr(), ptr->index());
+                        }
+                        else {
+                            // This isn't supposed to happen
+                        }
+                    }
+                    else {
+                        // This isn't supposed to happen (except maybe for
+                        // vectors)
+                    }
+                }
+                else {
+                    if(inst->numDestRegs() == 1)
+                        inst->verifyPrediction(0);
+                }
+            }
             int dependents = instQueue.wakeDependents(inst);

             for (int i = 0; i < inst->numDestRegs(); i++) {
diff --git a/gem5/src/cpu/o3/inst_queue_impl.hh b/gem5/src/cpu/o3/inst_queue_impl.hh
index ff5b3be..c878485 100644
--- a/gem5/src/cpu/o3/inst_queue_impl.hh
+++ b/gem5/src/cpu/o3/inst_queue_impl.hh
@@ -765,6 +765,14 @@ InstructionQueue<Impl>::processFUCompletion(const DynInstPtr &inst, int fu_idx)
     instsToExecute.push_back(inst);
 }

+/**
+ * SATVIK:
+ * If an instruction reaches the head of the IQ, check if any of its source
+ * registers is tagged with the predicted load. If yes, do not erase the
+ * instruction from the IQ. Only after the CVU validates the load can the
+ * instruction be removed from the IQ.
+ */
+
 // @todo: Figure out a better way to remove the squashed items from the
 // lists.  Checking the top item of each list to see if it's squashed
 // wastes time and forces jumps.
@@ -908,6 +916,11 @@ InstructionQueue<Impl>::scheduleReadyInsts()
                 // complete.
```

```
                ++freeEntries;
                count[tid]--;
+               /**
+                * SATVIK:
+                * This will be conditional- if the source registers are not
+                * speculated values, then clear the entry from the IQ.
+                */
                issuing_inst->clearInIQ();
            } else {
                memDepUnit[tid].issue(issuing_inst);
diff --git a/gem5/src/cpu/o3/lsq_impl.hh b/gem5/src/cpu/o3/lsq_impl.hh
index c4cb45e..2584860 100644
--- a/gem5/src/cpu/o3/lsq_impl.hh
+++ b/gem5/src/cpu/o3/lsq_impl.hh
@@ -735,6 +735,12 @@ LSQ<Impl>::pushRequest(const DynInstPtr& inst, bool isLoad, uint8_t *data,
            inst->effSize = size;
            inst->effAddrValid(true);

+           //SATVIK
+           // CHeck the CVU CAM for this load
+           if(inst->isConstLoad()) {
+               inst->verifyConstLoad(0);
+           }
+
            if (cpu->checker) {
                inst->reqToVerify = std::make_shared<Request>(*req->request());
            }
diff --git a/gem5/src/cpu/o3/lsq_unit.hh b/gem5/src/cpu/o3/lsq_unit.hh
index 3d6e3f0..4a94969 100644
--- a/gem5/src/cpu/o3/lsq_unit.hh
+++ b/gem5/src/cpu/o3/lsq_unit.hh
@@ -683,6 +683,19 @@ LSQUnit<Impl>::read(LSQRequest *req, int load_idx)
        load_inst->recordResult(true);
    }

+   // SATVIK
+   if(load_inst->isConstLoad() && load_inst->isConstPredictionCorrect()) {
+       // This is basically forwarding values from a local resource
+       // The data will be written during IEW WB
+       // Here, we just schedule the writeback event of the load
+       DPRINTF(LSQUnit, "Const load[%llu]: 0x%x scheduling a WB event\n", load_inst->seqNum, load_inst->instAddr());
+       load_inst->memData = new uint8_t[MaxDataBytes];
+       PacketPtr main_pkt = new Packet(req->mainRequest(), MemCmd::ReadReq);
+       main_pkt->dataStatic(load_inst->memData);
+       WritebackEvent *wb = new WritebackEvent(load_inst, main_pkt, this);
+       cpu->schedule(wb, curTick());
+       return NoFault;
+   }
    if (req->mainRequest()->isLocalAccess()) {
        assert(!load_inst->memData);
        assert(!load_inst->inHtmTransactionalState());
diff --git a/gem5/src/cpu/o3/lsq_unit_impl.hh b/gem5/src/cpu/o3/lsq_unit_impl.hh
index 808a671..1e478e3 100644
--- a/gem5/src/cpu/o3/lsq_unit_impl.hh
+++ b/gem5/src/cpu/o3/lsq_unit_impl.hh
@@ -617,6 +617,14 @@ LSQUnit<Impl>::executeLoad(const DynInstPtr &inst)

    assert(!inst->isSquashed());

+   //if(inst->isConstPredictionCorrect() && !inst->strictlyOrdered() && !inst->isInstPrefetch()) {
+   //    DPRINTF(LSQUnit, "Load PC %s was correctly predicted as constant sn:%llu", inst->pcState(), inst->seqNum);
+   //    inst->setExecuted();
+   //    iewStage->instToCommit(inst);
+   //    iewStage->activityThisCycle();
+    //    iewStage->checkMisprediction(inst);
+    //    return NoFault;
+    //}
     load_fault = inst->initiateAcc();

     if (load_fault == NoFault && !inst->readMemAccPredicate()) {
@@ -669,6 +677,20 @@ LSQUnit<Impl>::executeLoad(const DynInstPtr &inst)
         }
     }

+    /**
+     * SATVIK:
+     * We can call the CVU verification API here to check if the load was
+     * judged "predictable" by the LCT and if so, whether the actual load value
+     * is equal to the predicted value. If there's a mismatch,
+     * all its dependents will have to be issued again, with the correct values
+     *
+     * If the load was deemed "predictable" -> reissue all dependent instructions
+     * -> Not sure how to do this
+     *
+     * UPDATE: It's better to verify the load in the IEW stage since that will
+     * give us direct access to the IQ
+     */
+
     return load_fault;
 }

@@ -1099,6 +1121,16 @@ LSQUnit<Impl>::writeback(const DynInstPtr &inst, PacketPtr pkt)
         return;
     }

+    // SATVIK
+    if(inst->isConstLoad() && inst->isConstPredictionCorrect()) {
+        DPRINTF(LSQUnit, "Load PC %s was correctly predicted as constant sn:%llu", inst->pcState(), inst->seqNum);
+        inst->setExecuted();
+        iewStage->instToCommit(inst);
+        iewStage->activityThisCycle();
+        iewStage->checkMisprediction(inst);
+        return;
+    }
+
     if (!inst->isExecuted()) {
         inst->setExecuted();

diff --git a/gem5/src/cpu/o3/regfile.hh b/gem5/src/cpu/o3/regfile.hh
index 922089c..624d262 100644
--- a/gem5/src/cpu/o3/regfile.hh
+++ b/gem5/src/cpu/o3/regfile.hh
@@ -43,6 +43,7 @@
 #define __CPU_O3_REGFILE_HH__

 #include <vector>
+#include <list>

 #include "arch/types.hh"
 #include "base/trace.hh"
@@ -131,6 +132,14 @@ class PhysRegFile
     /** Mode in which vector registers are addressed. */
     VecMode vecMode;

+    /**
+     * List containing physical register pointers for registers that contain a
+     * predicted load.
+     */
+    std::list<PhysRegIdPtr> regsWithPredictedLoad;
```

```
+
+    std::list<PhysRegIdPtr> mispredictList;
+
   public:
     /**
      * Constructs a physical register file with the specified amount of
@@ -149,6 +158,69 @@ class PhysRegFile
      */
     ~PhysRegFile() {}
 
+    /**
+     * SATVIK:
+     * Implement a map and an API here which tags a PhysRegIdPtr with a
+     * predicted load. Every time an instruction is to issue, this map will be
+     * checked to see if any source operand of the instn uses a register that
+     * has been tagged. If there's a match, the instruction is not popped from
+     * the IQ.
+     */
+
+    /**
+     * @brief      Checks whether the given source register contains a value
+     *             that has been predicted for a load but not verified yet
+     *
+     * @param[in]  src_reg  The source register
+     *
+     * @return     True if the value is a predicted one
+     */
+    bool checkPredictedLoadRegister (PhysRegIdPtr src_reg) {
+        auto itr = regsWithPredictedLoad.begin();
+        while (itr != regsWithPredictedLoad.end()) {
+            if (*itr == src_reg) return true;
+            itr++;
+        }
+        return false;
+    }
+
+    /**
+     * @brief      Inserts a physical register that contains a speculated value
+     *
+     * @param[in]  reg   The register
+     */
+    void insertPredictedLoadRegister (PhysRegIdPtr reg) {
+        regsWithPredictedLoad.push_back(reg);
+    }
+
+    /**
+     * @brief      Clears a physical register pointer from the list
+     *
+     * @param[in]  reg   The register
+     */
+    void clearPredictedLoadRegister (PhysRegIdPtr reg) {
+        auto itr = regsWithPredictedLoad.begin();
+        while(itr != regsWithPredictedLoad.end()) {
+            if (*itr == reg) break;
+            itr++;
+        }
+        if (itr != regsWithPredictedLoad.end())
+            regsWithPredictedLoad.erase(itr);
+    }
+
+    void addToMispredictList(PhysRegIdPtr reg) {
+        mispredictList.push_back(reg);
+    }
+
+    bool searchMispredictList(PhysRegIdPtr src_reg) {
+        auto itr = mispredictList.begin();
+        while(itr != mispredictList.end()) {
+            if(*itr == src_reg) return true;
+            itr++;
+        }
+        return false;
+    }
+
     /** Initialize the free list */
     void initFreeList(UnifiedFreeList *freeList);


```

III. Modifications for Minor CPU

```
diff --git a/gem5/src/cpu/minor/MinorCPU.py b/gem5/src/cpu/minor/MinorCPU.py
index 1329dfb..29732bc 100644
--- a/gem5/src/cpu/minor/MinorCPU.py
+++ b/gem5/src/cpu/minor/MinorCPU.py
@@ -47,6 +47,8 @@ from m5.objects.DummyChecker import DummyChecker
 from m5.objects.BranchPredictor import *
 from m5.objects.TimingExpr import TimingExpr

+from m5.objects.LoadValuePredictionUnit import LoadValuePredictionUnit
+
 from m5.objects.FuncUnit import OpClass

 class MinorOpClass(SimObject):
@@ -273,7 +275,7 @@ class MinorCPU(BaseCPU):
     executeSetTraceTimeOnIssue = Param.Bool(False,
         "Set inst. trace times to be issue times")

-    executeAllowEarlyMemoryIssue = Param.Bool(True,
+    executeAllowEarlyMemoryIssue = Param.Bool(False,
         "Allow mem refs to be issued to the LSQ before reaching the head of"
         " the in flight insts queue")

@@ -283,6 +285,8 @@ class MinorCPU(BaseCPU):
     branchPred = Param.BranchPredictor(TournamentBP(
         numThreads = Parent.numThreads), "Branch Predictor")

+    loadValPred = Param.LoadValuePredictionUnit(LoadValuePredictionUnit(), "Load value predictor")
+
     def addCheckerCpu(self):
         print("Checker not yet supported by MinorCPU")
         exit(1)
diff --git a/gem5/src/cpu/minor/SConscript b/gem5/src/cpu/minor/SConscript
index 75cc940..2ef617a 100644
--- a/gem5/src/cpu/minor/SConscript
+++ b/gem5/src/cpu/minor/SConscript
@@ -60,6 +60,7 @@ if 'MinorCPU' in env['CPU_MODELS']:
     DebugFlag('MinorCPU', 'Minor CPU-level events')
     DebugFlag('MinorExecute', 'Minor Execute stage')
     DebugFlag('MinorInterrupt', 'Minor interrupt handling')
+    DebugFlag('MinorLoadPredictor', 'LVP in minor CPU')
     DebugFlag('MinorMem', 'Minor memory accesses')
     DebugFlag('MinorScoreboard', 'Minor Execute register scoreboard')
     DebugFlag('MinorTrace', 'MinorTrace cycle-by-cycle state trace')
diff --git a/gem5/src/cpu/minor/cpu.cc b/gem5/src/cpu/minor/cpu.cc
index a375e07..a051a51 100644
--- a/gem5/src/cpu/minor/cpu.cc
+++ b/gem5/src/cpu/minor/cpu.cc
@@ -47,7 +47,8 @@

 MinorCPU::MinorCPU(MinorCPUParams *params) :
     BaseCPU(params),
-    threadPolicy(params->threadPolicy)
+    threadPolicy(params->threadPolicy),
+    loadValuePredictor(params->loadValPred)
 {
     /* This is only written for one thread at the moment */
     Minor::MinorThread *thread;
diff --git a/gem5/src/cpu/minor/cpu.hh b/gem5/src/cpu/minor/cpu.hh
index 579a96b..24bdb6c 100644
--- a/gem5/src/cpu/minor/cpu.hh
+++ b/gem5/src/cpu/minor/cpu.hh
@@ -44,6 +44,8 @@
 #ifndef __CPU_MINOR_CPU_HH__
 #define __CPU_MINOR_CPU_HH__

+#include "cpu/lvp/load_value_prediction_unit.hh"
+
 #include "cpu/minor/activity.hh"
 #include "cpu/minor/stats.hh"
 #include "cpu/base.hh"
@@ -186,6 +188,9 @@ class MinorCPU : public BaseCPU
      *  already been idled.  The stage argument should be from the
      *  enumeration Pipeline::StageId */
     void wakeupOnEvent(unsigned int stage_id);
+
+    public:
+        LoadValuePredictionUnit* loadValuePredictor;
 };

 #endif /* __CPU_MINOR_CPU_HH__ */
diff --git a/gem5/src/cpu/minor/decode.cc b/gem5/src/cpu/minor/decode.cc
index b07ca4a..edee685 100644
--- a/gem5/src/cpu/minor/decode.cc
+++ b/gem5/src/cpu/minor/decode.cc
@@ -233,6 +233,15 @@ Decode::evaluate()
                 dynInstAddTracing(output_inst, parent_static_inst, cpu);
 #endif

+                // If this is a load instruction, check the LVP unit to see if
+                // it can be predicted and tag the instruction as such
+                if(output_inst->isInst() && output_inst->isMemRef() && output_inst->staticInst->isLoad())
+                {
+                    auto lvp_result = cpu.loadValuePredictor->lookup(tid, output_inst->pc.instAddr());
+                    output_inst->loadPredicted = lvp_result.taken;
+                    output_inst->loadPredictedValue = lvp_result.value;
+                }
+
                 /* Step to next sequence number */
                 decode_info.execSeqNum++;

diff --git a/gem5/src/cpu/minor/dyn_inst.hh b/gem5/src/cpu/minor/dyn_inst.hh
index b90e277..586228e 100644
--- a/gem5/src/cpu/minor/dyn_inst.hh
+++ b/gem5/src/cpu/minor/dyn_inst.hh
@@ -184,6 +184,25 @@ class MinorDynInst : public RefCounted
     /** Predicted branch target */
     TheISA::PCState predictedTarget;

+    /** Robert
+     * Track if the load instruction was predicted and, if so, what the predicted value is.
+     * Load prediction done in decode, acted upon by execute in scheduling and load value comparison. */
+
+    /** The classification of the load prediction for this instruction. Only valid for loads. */
+    LVPType loadPredicted;
+
+    /** The register value to be forwarded in the load prediction, depending on how loadPredicted set. */
+    RegVal loadPredictedValue;
+
+    /** Indicates that the processor executed this as a constant load, bypassing memory system */
+    bool executedAsConstant = false;
+
+    /** Stores the virtual address of the instruction */
+    Addr effAddr;
+
+    /** Indicates that the virtual address has been set and effAddr is valid */
+    bool effAddrValid = false;
+
     /** Fields only set during execution */

     /** FU this instruction is issued to */
diff --git a/gem5/src/cpu/minor/execute.cc b/gem5/src/cpu/minor/execute.cc
index 45ca002..321ed33 100644
--- a/gem5/src/cpu/minor/execute.cc
+++ b/gem5/src/cpu/minor/execute.cc
@@ -51,6 +51,7 @@
 #include "debug/ExecFaulting.hh"
 #include "debug/MinorExecute.hh"
 #include "debug/MinorInterrupt.hh"
+#include "debug/MinorLoadPredictor.hh"
 #include "debug/MinorMem.hh"
 #include "debug/MinorTrace.hh"
 #include "debug/PCEvent.hh"
@@ -374,6 +375,60 @@ Execute::handleMemResponse(MinorDynInstPtr inst,
                 static_cast<unsigned int>(packet->getConstPtr<uint8_t>()[0]));
         }

+        /** Robert
+         * Check if the value loaded from memory was correct and update the
+         * LVP prediction tables with the result. */
+        if (is_load)
+        {
+            bool load_predicted_correctly = false;
+            if (inst->loadPredicted != LVP_CONSTANT)
+            {
+                // Get the data from the packet (borrowed from memhelpers.hh)
+                uint64_t mem;
+
+                switch (packet->getSize()) {
+                    case 1:
+                        mem = packet->getConstPtr<uint8_t>()[0];
+                        break;
+                    case 2:
+                        mem = packet->getConstPtr<uint16_t>()[0];
+                        break;
+                    case 4:
+                        mem = packet->getConstPtr<uint32_t>()[0];
+                        break;
+                    case 8:
+                        mem = packet->getConstPtr<uint64_t>()[0];
+                        break;
+                    default:
+                        panic("Unhandled size in handleMemResponse.\n");
+                }
+                // DPRINTF(MinorLoadPredictor, "Verifying prediction: pc: %#x, predicted value: %#x, correct value: %#x\n",
+                //     inst->pc.instAddr(), inst->loadPredictedValue, mem);
+                load_predicted_correctly =
+                    cpu.loadValuePredictor->verifyPrediction(
+                        thread_id,
+                        inst->pc.instAddr(),
+            inst->effAddr,
+            mem,
+            inst->loadPredictedValue,
+            inst->loadPredicted);
+
+        /* Experimenting with predictable forwarding
+        if (!load_predicted_correctly && inst->loadPredicted == LVP_PREDICTABLE)
+        {
+          // TODO the instruction was incorrectly predicted, we need to rewind this instruction
+          // and any dependent instructions that used this value
+          DPRINTF(MinorLoadPredictor, "Predicted the wrong value, need to fix up!\n");
+          auto reg_id = thread->flattenRegId(inst->staticInst->destRegIdx(0));
+          thread->setIntRegFlat(reg_id.flatIndex(), mem);
+          BranchData &lvpResetTarget = *out.inputWire;
+
+          updateBranchData(thread_id, BranchData::UnpredictedBranch, inst, inst->pc.nextInstAddr(), lvpResetTarget);
+        }
+        */
+      }
+    }
+
     /* Complete the memory access instruction */
     fault = inst->staticInst->completeAcc(packet, &context,
       inst->traceData);
@@ -387,7 +442,9 @@ Execute::handleMemResponse(MinorDynInstPtr inst,
       /* Stores need to be pushed into the store buffer to finish
       *  them off */
      if (response->needsToBeSentToStoreBuffer())
+      {
        lsq.sendStoreToStoreBuffer(response);
+      }
     }
   } else {
     fatal("There should only ever be reads, "
@@ -464,7 +521,7 @@ Execute::executeMemRefInst(MinorDynInstPtr inst, BranchData &branch,
     DPRINTF(MinorExecute, "Initiating memRef inst: %s\n", *inst);

     Fault init_fault = inst->staticInst->initiateAcc(&context,
-      inst->traceData);
+        inst->traceData);

     if (inst->inLSQ) {
       if (init_fault != NoFault) {
@@ -930,6 +987,26 @@ Execute::commitInst(MinorDynInstPtr inst, bool early_memory_issue,
     bool completed_mem_inst = executeMemRefInst(inst, branch,
       predicate_passed, fault);

+    /* Experiments with forwarding load values
+    if (inst->staticInst->isLoad() && inst->loadPredicted == LVP_PREDICTABLE) {
+      bool could_forward = true;
+      switch (inst->flatDestRegIdx->classValue()) {
+        case IntRegClass:
+          thread->setIntRegFlat(inst->flatDestRegIdx->flatIndex(), inst->loadPredictedValue);
+          break;
+        case FloatRegClass:
+          thread->setFloatRegFlat(inst->flatDestRegIdx->flatIndex(), inst->loadPredictedValue);
+          break;
+        default:
+          could_forward = false;
+      }
+      if (could_forward)
+        scoreboard[inst->id.threadId].clearInstDests(inst, inst->isMemRef());
```

```
+
+            completed_mem_inst = could_forward;
+        }
+        */
+
         if (completed_mem_inst && fault != NoFault) {
             if (early_memory_issue) {
                 DPRINTF(MinorExecute, "Fault in early executing inst: %s\n",
@@ -1147,6 +1224,9 @@ Execute::commit(ThreadID thread_id, bool only_commit_microops, bool discard,

                 lsq.popResponse(mem_response);
             } else {
+                if(inst->executedAsConstant) {
+                    DPRINTF(MinorLoadPredictor, "Handling mem response of constant load %s.\n", *inst);
+                }
                 handleMemResponse(inst, mem_response, branch, fault);
                 committed_inst = true;
             }
diff --git a/gem5/src/cpu/minor/lsq.cc b/gem5/src/cpu/minor/lsq.cc
index 106b51b..c583db3 100644
--- a/gem5/src/cpu/minor/lsq.cc
+++ b/gem5/src/cpu/minor/lsq.cc
@@ -49,6 +49,7 @@
 #include "cpu/utils.hh"
 #include "debug/Activity.hh"
 #include "debug/MinorMem.hh"
+#include "debug/MinorLoadPredictor.hh"

 namespace Minor
 {
@@ -960,7 +961,7 @@ LSQ::StoreBuffer::minorTrace() const

 void
 LSQ::tryToSendToTransfers(LSQRequestPtr request)
-{
+{
     if (state == MemoryNeedsRetry) {
         DPRINTF(MinorMem, "Request needs retry, not issuing to"
             " memory until retry arrives\n");
@@ -1042,6 +1043,13 @@ LSQ::tryToSendToTransfers(LSQRequestPtr request)
             return;
         }
     } else {
+
+        /* Send store instructions to LVP to invalidate CVU entries. */
+        if (request->inst->staticInst->isStore() && request->request->hasVaddr()) {
+            DPRINTF(MinorLoadPredictor, "Processing a store to CVU.\n");
+            cpu.loadValuePredictor->processStoreAddress(request->inst->id.threadId, request->request->getVaddr());
+        }
+
         /* Store.  Can it be sent to the store buffer? */
         if (bufferable && !request->request->isLocalAccess()) {
             request->setState(LSQRequest::StoreToStoreBuffer);
@@ -1164,6 +1172,15 @@ LSQ::tryToSendToTransfers(LSQRequestPtr request)
             request->issuedToMemory = true;
         }

+        // deal with LVP stuff
+        request->inst->executedAsConstant = request->inst->staticInst->isLoad()
+            && request->inst->loadPredicted == LVP_CONSTANT
+            && cpu.loadValuePredictor->processLoadAddress(request->inst->id.threadId, request->inst->pc.instAddr(), request->request->getVaddr());
+
+        if(request->inst->executedAsConstant) {
```

```
+            DPRINTF(MinorLoadPredictor, "Found instruction %s that can execute as constant\n", *request->inst);
+        }
+
         if (tryToSend(request)) {
             moveFromRequestsToTransfers(request);
         }
@@ -1191,7 +1208,25 @@ LSQ::tryToSend(LSQRequestPtr request)
          *  so the response can be correctly handled */
         assert(packet->findNextSenderState<LSQRequest>());
 
-        if (request->request->isLocalAccess()) {
+        /* If the request can be executed as a constant, respond with the loadPredictedValue*/
+        if(request->inst->executedAsConstant) {
+            packet->setData((uint8_t *)&request->inst->loadPredictedValue);
+
+            DPRINTF(MinorLoadPredictor, "Forwarded values on constant load %s.\n", *request->inst);
+
+            request->stepToNextPacket();
+            ret = request->sentAllPackets();
+
+            if (!ret) {
+                DPRINTF(MinorLoadPredictor, "Constant load access has another packet: %s\n",
+                    *(request->inst));
+            }
+
+            if (ret)
+                request->setState(LSQRequest::Complete);
+            else
+                request->setState(LSQRequest::RequestIssuing);
+        } else if (request->request->isLocalAccess()) {
             ThreadContext *thread =
                 cpu.getContext(cpu.contextToThread(
                                 request->request->contextId()));
@@ -1654,6 +1689,12 @@ LSQ::pushRequest(MinorDynInstPtr inst, bool isLoad, uint8_t *data,
     inst->inLSQ = true;
     request->startAddrTranslation();
 
+    if (request->request->hasVaddr())
+    {
+        inst->effAddrValid = request->request->hasVaddr();
+        inst->effAddr = request->request->getVaddr();
+    }
+
     return inst->translationFault;
 }

```