CAVA: Hiding L2 Misses with Checkpoint-Assisted Value Prediction

Luis Ceze, **Karin Strauss**, **James Tuck**, **Jose Renau**[†] and **Josep Torrellas**

University of Illinois at Urbana-Champaign
{luisceze, kstrauss, jtuck, torrellas}@cs.uiuc.edu

[†]University of California, Santa Cruz
renau@soe.ucsc.edu

***Abstract*— Load misses in on-chip L2 caches often end up stalling modern superscalars. To address this problem, we propose hiding L2 misses with Checkpoint-Assisted VAlue prediction (CAVA). When a load misses in L2, a predicted value is returned to the processor. If the missing load reaches the head of the reorder buffer before the requested data is received from memory, the processor checkpoints, consumes the predicted value, and speculatively continues execution. When the requested data finally arrives, it is compared to the predicted value. If the prediction was correct, execution continues normally; otherwise, execution rolls back to the checkpoint. Compared to a baseline aggressive superscalar, CAVA speeds up execution by a geometric mean of 1.14 for SPECint and 1.34 for SPECfp applications. Additionally, CAVA is faster than an implementation of Runahead execution, and Runahead with value prediction.**

I. Introduction

Load misses in on-chip L2 caches are a major source of processor stall in modern superscalars, since each miss can take hundreds of cycles to complete. Often, the missing load reaches the head of the reorder buffer (ROB) before the data is received, dependences clog the pipeline, and the processor stalls.

To increase performance, processors must find better ways to overlap an L2 miss with useful computation and other misses. Popular techniques include very aggressive out-of-order execution, hardware prefetching, and software prefetching. Unfortunately, out-of-order execution appears able to provide significant additional improvements only at high implementation costs. Moreover, while prefetching typically works well for regular applications, it often has a hard time in irregular codes.

Past research has shown that it is possible to successfully predict data values (e.g., [7]). Moreover, past work has used hardware-based processor checkpointing in a variety of contexts, including early recycling of resources [8], increasing the number of in-flight instructions [1], [3], and warming up caches and branch predictors on L2 misses [9].

Based on these ideas, this paper suggests a new approach to hide the latency of L2 misses. We call the new approach Checkpoint-Assisted VAlue Prediction (CAVA). As soon as a load misses in the L2 cache, a predicted return value is passed to the CPU. If the missing load reaches the head of the ROB before the data is received from main memory, the CPU performs a checkpoint, consumes the predicted value, and continues execution. The missing load and the subsequent instructions are speculatively retired. The speculative state that they generate is kept buffered in the L1 cache. If the prediction is later determined to be correct, the speculative state commits and execution continues normally — no re-execution of any speculatively retired instruction is needed. Otherwise, the speculative state is discarded and execution rolls back to the checkpoint.

Manuscript submitted: September 15, 2004. Manuscript accepted: November 16, 2004. Final manuscript received: December 14, 2004.

In this paper, we describe the novel CAVA microarchitecture. We use value prediction with confidence estimation. Based on the confidence, the processor performs different actions. To simplify our first implementation, we only support a single outstanding checkpoint at a time.

A checkpointed scheme related to CAVA is Runahead execution [9]. Runahead checkpoints the processor and continues execution after an L2 miss. However, in Runahead: (1) The destination register of the missing load is marked with an invalid tag, and dependent instructions that propagate this tag are not used to warm up the branch predictor or prefetch into the cache; (2) The processor always rolls back execution when the requested data arrives from memory, irrespective of its value; (3) The processor buffers some incomplete speculative state in a buffer rather than the complete state in L1. For completeness, we compare the performance of CAVA to Runahead, and to Runahead with value prediction. We discuss other related schemes in Section V.

Relative to an aggressive conventional superscalar, CAVA delivers average speedups of 1.14 and 1.34 for SPECint and SPECfp applications, respectively. Compared to the same baseline, an implementation of Runahead produces average speedups of 1.07 and 1.18 for SPECint and SPECfp applications, respectively.

II. Hiding L2 Misses with CAVA

To support CAVA, we need four components. First, we need a module that, as soon as an L2 miss occurs, predicts a value for the requested data and passes it to the processor. That same module can keep the prediction for later comparison with the correct data coming from memory. Second, we need support to perform fast register checkpointing. Third, after the checkpoint, the L1 cache has to mark and buffer all updated cache lines, preventing their displacement to lower levels of the memory hierarchy until the prediction is proved correct. Finally, if the prediction is incorrect, we restore the checkpoint and invalidate the updated cache lines; if the prediction is correct, the updated cache lines are committed.

In this paper, we implement the four components in hardware. We place the value predictor close to the L2 cache controller. We do this to minimize the modifications to time-critical modules like the processor core and L1 cache. In addition, it is easier to train the value predictor with L2 misses (the objective of CAVA), rather than with all loads.

A. Basic Buffers

CAVA is built around two buffers: one that extends the Miss Status Holding Registers (MSHRs) [5] of the L2 cache, and one that buffers predictions in the processor's load functional unit. We call them *Outstanding Prediction Buffer (OPB)* and *Ready Buffer (RDYB)*, respectively.

In conventional systems, each L2 MSHR keeps the record of an L2 miss. In CAVA, the OPB entry also obtains a predicted data value for the requested data from a value predictor, sends the prediction to the processor, and stores it locally. Predictions are made at the granularity requested by the processor (e.g., word, byte, etc). When the requested cache line arrives from memory, the OPB compares the line's data against all the predictions made for data in that line. The OPB forwards the line upstream to the L1, including in the message a *confirmation* or *rejection* tag for each of the predictions made. These tags and data will eventually reach the processor. The OPB then deallocates the entry.

Figure 1-(a) shows the OPB, which is an enhanced MSHR structure in L2. As a reference, Figure 1-(b) shows the unmodified MSHR structure in L1. For both the L1 MSHRs and the L2 OPB, we use the Explicitly-Addressed organization in [4]. A conventional MSHR in L2 keeps line address information. An OPB entry extends it with additional information to support several predicted words in that line. For each such word, the OPB contains the word offset, the destination register, and the predicted value sent to the processor.

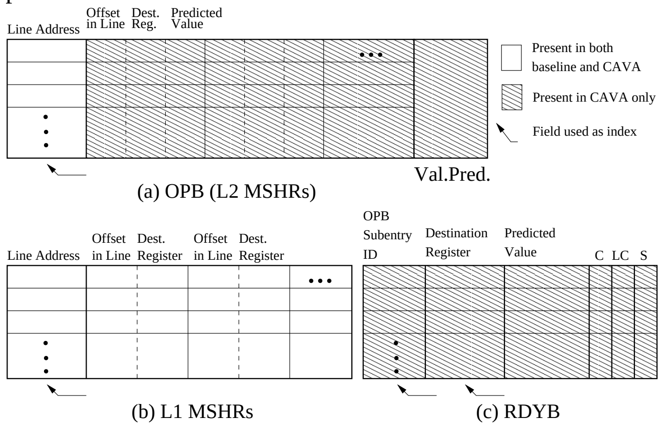

Fig. 1. Buffers used in CAVA. The shaded areas are the fields added to support CAVA. In the figure, "register" means a physical register name.

The L1 MSHR structure (Figure 1-(b)) is unmodified, although its control logic is changed slightly. Specifically, all loads that miss in the L1 are propagated to the L2, including loads on words whose line is already requested by L1. The reason is that the OPB has to observe all the L1 load misses to be able to supply predictions if they miss in the L2 as well.

On the processor side, the RDYB temporarily buffers the value predictions forwarded by the OPB. A new RDYB entry is allocated when the processor receives a prediction. When the processor receives the final value with a confirmation or rejection tag, the entry is deallocated.

Figure 1-(c) shows the RDYB structure. The first field contains the *OPB Subentry ID*, which is an ID sent by the OPB together with the data value prediction at the time the RDYB entry is allocated. It identifies the location in the OPB that holds the prediction. When the OPB sends the final data to the processor with a confirmation or rejection tag, it includes the OPB Subentry ID. The latter is used to index the RDYB – the physical register number cannot be used because the register may have been reused. Missing loads that are about to speculatively retire obtain predicted data by indexing the RDYB with the destination physical register.

The RDYB also stores the Consumed (C), LowConfidence (LC), and Stale (S) bits. C is set when the entry is consumed. LC is set when the value predictor sends to the processor a low-confidence prediction. S is set when the processor is rolled back. S entries are retained as stale until a value prediction confirmation or rejection message arrives from memory and deallocates them.

B. Value Predictor

The L2 controller contains a Value Prediction module that is closely coupled with the OPB. When an L2 miss occurs, the value predictor predicts the value of the requested word. We use a hybrid value predictor organization, with one global and one local value predictor, along with a selector. The predictor also estimates the confidence of each prediction. In our configuration, the total size of the predictors, selector and estimator is 72 Kbits.

C. Additional Processor Support

The processor supports fast, hardware-based register checkpointing as in [1], [3], [8], [9]. Checkpoints include architectural registers and branch history.

A Status Register indicates in which mode the processor is currently running. It can run speculatively under a checkpoint generated by a high-confidence prediction (*Chk-High Mode*) or a low-confidence one (*Chk-Low Mode*), or it can run non-speculatively (*Non-Chk Mode*).

D. Additional Cache Hierarchy Support

Following a checkpoint, the processor generates speculative memory state that needs to be buffered separately. We can use a special buffer as in [1], [3], [9] or the L1 cache as in [8]. Without loss of generality, this paper uses the L1.

All L1 lines updated speculatively are marked with a *Speculative (S)* bit in the tag (this is the same as the *Volatile* bit in [8]). If the line was dirty before the speculative update, the line is written back to memory before accepting the update. Lines with the S bit set cannot be evicted.

When all value predictions are confirmed and the processor transitions to Non-Chk mode, the cache clears the S bits. If, instead, a prediction fails, all the lines with a set S bit are invalidated. These two operations are done as in [8], with a hardware signal that takes a few cycles.

If the L1 runs out of space for speculative lines, it signals the processor. The latter then stalls until either it rolls back due to a prediction rejection or all outstanding predictions are confirmed. We note that this event is extremely rare because execution in speculative mode is usually short (Table II). It almost never happened in our simulations.

E. Overview of Operation

When the L2 detects a miss, the OPB allocates a free subentry in an existing or new OPB entry. A prediction is then generated and sent to the processor, together with the confidence in the prediction. In the processor, the prediction is stored in a newly allocated RDYB entry. If appropriate, the LowConfidence bit in the entry is set.

When a load reaches the head of the ROB, it waits until either its destination register is loaded with actual data, or an entry with a predicted value for that register is found in the RDYB. If the latter occurs, the processor checkpoints (if it is in Non-Chk mode), the value is forwarded to the destination register, and the Consumed bit in the RDYB entry is set. At this point, the load speculatively retires. The execution mode becomes Chk-Low or Chk-High, depending on the LowConfidence bit. If the processor was in Chk-High mode and the LowConfidence bit is set, the processor waits until it can commit the current speculative section; then it performs a new checkpoint and resumes in Chk-Low mode.

Eventually, the cache hierarchy replies with the requested data, together with a confirmation or rejection (prediction was incorrect) tag. If the corresponding entry is found in the RDYB with its Consumed bit clear, the incoming data is sent to the destination register and the RDYB entry is deallocated.

If the response has a rejection tag and finds the entry in the RDYB with the Consumed bit set, it means that the processor has consumed incorrect data. The RDYB entry is then deallocated, all the other valid RDYB entries set their Stale bit, and the processor rolls back.

If the response has a confirmation tag and finds the entry in the RDYB with its Consumed bit set, it means that the processor has consumed correct data. The RDYB entry is deallocated. When all the non-stale RDYB entries are deallocated, a hardware signal triggers a move to Non-Chk mode and the speculative data is committed.

It is best to limit the duration of speculative execution — a miss-speculation after running speculatively for a long time will waste much work. Consequently, after running speculatively for T_{chk} cycles, no more predictions are made and misses are handled normally. This will result in the eventual termination of the speculation.

Finally, during or after a rollback, the OPB will continue to send messages with rejection or confirmation tags that will find RDYB entries with the Stale bit set. In this case, the RDYB entry is simply deallocated. This behavior seamlessly supports branch mispredictions and load replays. For example, consider loads that are in a wrong branch path. If they miss in L2 and the OPB provides predictions, RDYB entries will be allocated. When the processor knows that these entries are useless, it sets their Stale bit. When the correct value finally arrives from memory, the RDYB entry is deallocated.

III. Experimental Setup

We evaluate CAVA using execution-driven simulations with a detailed model of a processor and memory subsystem (Table I). The configurations modeled are: *Base* (plain superscalar), *Runahead/C* (Runahead that stores the speculative state in the L1 rather than in the smaller Runahead cache [9]), *Runahead/C with VP* (Runahead/C that uses the value predictor of CAVA rather than marking the destination register of the missing load as invalid [9]), *CAVA*, *CAVA Perf VP* (CAVA with a perfect value predictor), and *Perf Mem* (*Base* with a 100% hit-rate L2 cache). All configurations include an aggressive prefetcher (Table I) that supports multiple non-unit stride streams and uses the algorithm in [10]. Prefetched data goes to a separate buffer, instead of the L2 cache.

TABLE I

Simulated processor parameters.

Processor				
Frequency: 5.0 GHz with 70 nm				Fetch/issue/comm width: 6/4/4
Branch penalty: 13 cyc (min)				I-window/ROB size: 60/152
RAS: 32 entries				Int/FP registers: 104/80
BTB: 2K entries, 2-way assoc.				LdSt/Int/FP units: 2/3/2
Branch predictor (spec. update):				Ld/St queue entries: 54/46
bimodal size: 16K entries				
gshare-11 size: 16K entries				
Cache	I-L1	D-L1	L2	
Size:	16KB	16KB	1MB	HW Prefetcher:
Round Trip:	2 cyc	2 cyc	10 cyc	16-stream stride prefetcher
Assoc:	2-way	4-way	8-way	8KB prefetch buffer
Line size:	64B	64B	64B	
Ports:	1	2	1	Memory: DDR-2
MSHRs:	4	128	128	Bus frequency: 533MHz
CAVA specific:				Bus width: 128bit
OPB: 128 entries				DRAM bandwidth: 8.528GB/s
Val. pred. table size: 2048 entries				Round Trip: 98ns (490 cyc)
Max. ckpt. duration (T_{chk}): 1280 cyc				

We run most of the SPECint and some SPECfp codes. We do not use the whole SPEC suite because our compiler and simulator do not support C++, Fortran90, and some system calls. We use *gcc 3.4 -O3* to compile into MIPS binaries. We simulate 0.6-1 billion instructions after initialization.

IV. Evaluation

Figure IV presents speedups relative to *Base*. It shows that the geometric mean speedup of *CAVA* over *Base* is 1.14 for SPECint codes and 1.34 for SPECfp codes. *CAVA Perf VP* further improves the speedups, but it is not as fast as *Perf Mem*. The reason is that, unlike *Perf Mem*, *CAVA Perf VP* uses L1 MSHRs and OPB entries for a long time. As a result, it may run out of them and stall.

In contrast, *Runahead/C* delivers a geometric mean speedup of only 1.07 and 1.18 for SPECint and SPECfp codes, respectively. When we combine it with value prediction (*Runahead/C with VP*), its speedups get close to *CAVA* for SPECint, but not for SPECfp. The reason for the difference is that, in Runahead, even correctly-predicted speculative sections are re-executed.

Our Runahead numbers are not directly comparable to those in [9]. The reason is that the processor and memory subsystem modeled are different (4-issue and 1MB L2 for us vs. 3-issue and 512KB L2), and so are the codes used.

Table II shows more details on the behavior of CAVA. The distance between a checkpoint and the termination of speculative execution is called a Checkpointed Run. From left to right, the table shows the distance between checkpoints, the duration of a checkpointed run, the number of value-predicted L2 misses in a checkpointed run, the fraction of failed checkpointed runs, the fraction of instructions wasted due to failed checkpointed runs, the value prediction accuracy, the confidence estimation accuracy (how often high-confidence

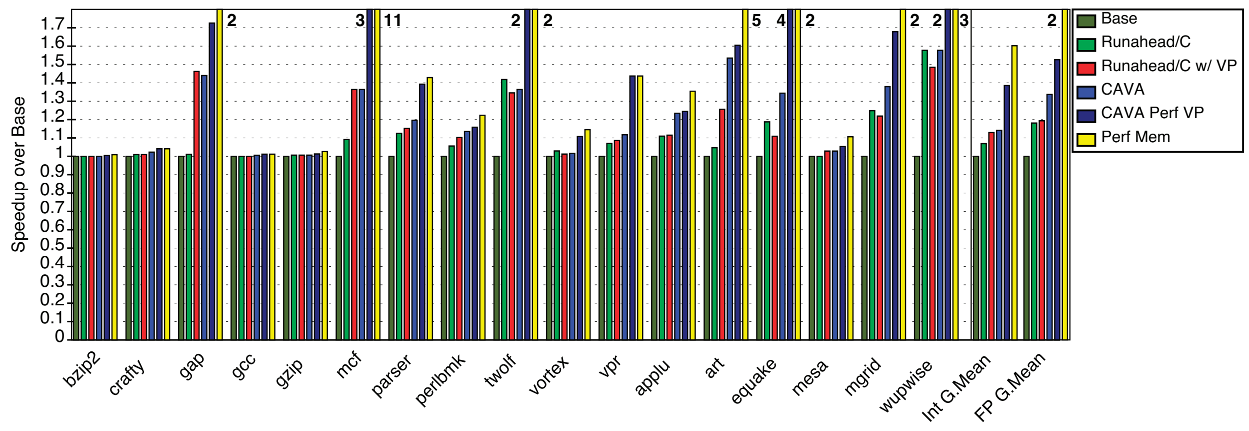

Fig. 2. Speedups normalized to *Base*.

predictions are correct and low-confidence ones incorrect), the L2 miss rate in *Base*, and the IPC of three different systems: *Base* without prefetching (*Nopref*), *Base*, and *CAVA*. Space limitations prevent us from further discussion.

TABLE II

CHARACTERIZATION OF *CAVA* EXECUTION.

App.	Checkpointed Run				Frac. inst wasted	Val. pred. acc. (%)	Conf. est. acc. (%)	Base L2 miss rate (%)	IPC Nopref,Base,CAVA
	Avg. dist. (inst)	Avg. size (inst)	Avg. # of L2 misses	Frac. fail					
bzip2	48702	167	1.7	0.94	0.00	58.4	96.0	0.0	1.97, 2.24, 2.24
crafty	27877	1054	2.3	0.48	0.02	61.6	81.9	0.0	2.23, 2.19, 2.24
gap	645	215	6.4	0.87	0.23	50.1	85.4	1.4	0.61, 0.91, 1.31
gcc	26939	178	1.2	0.56	0.00	48.4	76.7	0.0	1.69, 1.72, 1.73
gzip	25559	270	52.7	0.97	0.01	3.9	97.1	0.1	1.66, 1.56, 1.57
mcf	211	182	9.2	0.67	0.78	59.8	80.5	14.8	0.09, 0.11, 0.15
parser	1557	326	3.3	0.61	0.16	51.2	81.0	0.4	0.91, 1.12, 1.34
perlbmk	4583	162	3.0	0.73	0.03	74.1	90.1	0.2	2.02, 2.15, 2.44
twolf	646	395	4.3	0.64	0.45	39.1	73.1	0.9	0.58, 0.55, 0.75
vortex	6730	633	13.1	0.72	0.08	61.8	85.0	0.1	2.39, 2.42, 2.46
vpr	2295	708	3.9	0.71	0.24	21.3	88.4	1.2	1.21, 1.28, 1.43
applu	16985	1043	45.6	0.55	0.03	59.0	99.4	0.2	1.72, 2.09, 2.58
art	412	308	30.6	0.56	0.38	54.7	92.6	30.4	0.35, 0.43, 0.66
equake	250	191	13.1	0.53	0.55	47.1	76.6	3.5	0.33, 0.64, 0.86
mesa	11147	445	3.1	0.69	0.03	31.5	81.1	0.2	2.49, 2.44, 2.51
mgrid	695	464	15.1	0.62	0.27	75.1	99.2	1.2	0.53, 1.37, 1.89
wupwise	1580	778	33.2	0.76	0.28	46.1	88.7	1.2	1.22, 1.30, 2.05
Int Avg	13249	390	9.2	0.71	0.18	48.2	85.0	1.6	1.40, 1.48, 1.61
FP Avg	5178	538	23.4	0.61	0.25	52.3	89.6	6.1	1.34, 1.38, 1.76

V. RELATED WORK

Runahead execution [9] checkpoints the processor and speculatively removes a long-latency load from the head of the ROB, marking its destination register as containing invalid data. Dependent instructions propagate the invalid mark and are also removed from the head of the ROB. When the data is received from memory, the processor rolls back and re-executes from the load. Hopefully, code independent of the missing load has warmed up caches and branch predictors.

Zhou and Conte [12] used value prediction on missing loads to continue executing (speculatively). Speculative instructions remain in the issue queue, since no checkpointing is made. When the actual data is received from memory, the speculative instructions are always re-executed. As in Runahead, speculative execution is employed for prefetching.

There are several works on value prediction (e.g., [2], [7], [11]). We have used their insights for our value predictor with confidence estimation.

Martinez *et al.* [8] proposed Cherry, where resources are recycled early by leveraging checkpoints. Akkary *et al.* [1] and Cristal *et al.* [3] used checkpoints to increase the number of in-flight instructions in a ROB-less processor. Lebeck *et al.* [6] increased the number of in-flight instructions by temporarily moving instructions dependent on a long-latency one out of the issue queue.

VI. CONCLUSION

This paper presented CAVA, a new technique that hides L2 cache misses by checkpointing the processor, using a predicted value, and speculatively retiring the load and subsequent instructions. When the memory response arrives, the prediction is validated: if correct, execution resumes; otherwise, the processor rolls back to the checkpoint. We support a single checkpoint at a time to make hardware simpler. Value prediction confidence estimation is used to make checkpointing decisions that improve performance. The microarchitecture necessary to implement CAVA requires a small chip area. Overall, CAVA delivers significant speedups: the geometric mean speedup is 1.14 and 1.34 for SPECint and SPECfp applications, respectively. CAVA significantly outperforms Runahead.

REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in *36th MICRO*, Nov. 2003.
[2] M. Burtscher and B. G. Zorn, "Exploring Last *n* Value Prediction," in *PACT*, Oct. 1999.
[3] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-Order Commit Processors," in *10th HPCA*, Feb. 2004.
[4] K. I. Farkas and N. P. Jouppi, "Complexity/Performance Tradeoffs with Non-Blocking Loads," in *21st ISCA*, April 1994.
[5] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," in *8th ISCA*, May 1981.
[6] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," in *29th ISCA*, May 2002.
[7] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *7th ASPLOS*, Oct. 1996.
[8] J. F. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors," in *35th MICRO*, Nov. 2002.
[9] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *9th HPCA*, Feb. 2003.
[10] S. Palacharla and R. E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," in *21st ISCA*, Apr. 1994.
[11] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," in *30th MICRO*, Dec. 1997.
[12] H. Zhou and T. Conte, "Enhancing Memory Level Parallelism via Recovery-Free Value Prediction," in *17th ICS*, June 2003.