CAVA: Using Checkpoint-Assisted Value Prediction to Hide L2 Misses[*]

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

Modern superscalar processors often suffer long stalls due to load misses in on-chip L2 caches. To address this problem, we propose hiding L2 misses with Checkpoint-Assisted VAlue prediction (CAVA). On an L2 cache miss, a predicted value is returned to the processor. When the missing load finally reaches the head of the ROB, the processor checkpoints its state, retires the load, and speculatively uses the predicted value and continues execution. When the value in memory arrives at the L2 cache, it is compared to the predicted value. If the prediction was correct, speculation has succeeded and execution continues; otherwise, execution is rolled back and restarted from the checkpoint. CAVA uses fast checkpointing, speculative buffering, and a modest-sized value prediction structure that has about 50% accuracy. Compared to an aggressive superscalar processor, CAVA speeds up execution by up to 1.45 for SPECint applications and 1.58 for SPECfp applications, with a geometric mean of 1.14 for SPECint and 1.34 for SPECfp applications. We also evaluate an implementation of Runahead execution — a previously-proposed scheme that does not perform value prediction and discards all work done between checkpoint and data reception from memory. Runahead execution speeds up execution by a geometric mean of 1.07 for SPECint and 1.18 for SPECfp applications, compared to the same baseline.

1 Introduction

Load misses in on-chip L2 caches are a major cause of processor stall in modern superscalars. A missing load can take hundreds of cycles to be serviced from memory. Meanwhile, the processor keeps executing and retiring instructions. However, the missing load instruction eventually reaches the head of the Reorder Buffer (ROB), dependences clog the ROB, and the processor stalls.

Performance can be improved if processors find better ways to overlap L2 misses with useful computation and even with other L2 misses. Currently implemented techniques to address this problem include aggressive out-of-order execution to support more instructions in flight, hardware prefetching (e.g., [2, 5, 10, 11]), and software prefetching (e.g., [4, 17]). Unfortunately, with out-of-order execution, significant further improvements can only come with high implementation costs.

[*]This work extends “CAVA: Hiding L2 Misses with Checkpoint-Assisted Value Prediction”, which appeared in the IEEE Computer Architecture Letters (CAL) in December 2004. This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel. Luis Ceze is supported by an IBM PhD Fellowship.

Moreover, while prefetching typically works well for scientific applications, it often has a hard time with irregular applications.

Past research has shown that it is possible to use history to successfully predict data values [3, 8, 16, 23]. Load value prediction [16] has been used in the context of conventional superscalar processors to mitigate the effect of memory latency and bandwidth. In this case, however, a very long latency load (such as one that misses in the L2) will eventually wait at the head of the ROB, even if its value has been predicted using techniques such as the one in [16].

We address this problem by using a checkpoint to be able to speculatively retire the long latency load and unclog the ROB. This way, when a long latency load reaches the head of the ROB, the following happens: the processor state is checkpointed, a predicted value is provided to the missing load, the missing load is retired and execution proceeds speculatively. When the processor is executing speculatively, the state produced has to be buffered. If the prediction is later determined to be correct, execution continues normally. Otherwise, execution is rolled back to the checkpoint. We call this idea Checkpoint-Assisted VAlue Prediction (CAVA).

Both processor checkpointing and speculative buffering mechanisms have been described elsewhere. For example, hardware-based checkpoint and fast rollback has been used in the context of branch speculation, recycling resources early [19], aggressively increasing the number of in-flight instructions [1, 6, 25], or prefetching data and training the branch predictors on an L2 miss [21]. Speculative state is buffered in the processor [1, 6, 21, 25] or in the cache [19].

We describe several key design issues in CAVA systems, including multiprocessor aspects. Then, we present a microarchitectural implementation that is built around a *Ready Buffer (RDYB)* in the processor's load functional unit and an *Outstanding Prediction Buffer (OPB)* in the L2 MSHR. Our design includes a confidence estimator to minimize wasted work on rollbacks due to mispeculations. If the confidence on a new value prediction is low, the processor commits its current speculative state and then creates a new checkpoint before consuming the new prediction. In our evaluation, we perform an extensive characterization of the architectural behavior of CAVA, as well as a sensitivity analysis of different architectural parameters.

CAVA is related to Runahead execution [21] and the concurrently-developed Clear scheme [12]. Specifically, Runahead also uses checkpointing to allow processors to retire missing loads and continue execution. However, Runahead and CAVA differ in three major ways. First, in Runahead there is no prediction: the destination register of the missing load is marked with an invalid tag, which is propagated by dependent instructions. Second, in Runahead, when the data arrives from memory, execution is *always* rolled back; in CAVA, if the prediction is correct, execution is not rolled back. Finally, while Runahead buffers (potentially incomplete) speculative state in a processor structure called Runahead cache, CAVA buffers the whole speculative state in L1. We evaluate Runahead without and with value prediction.

Compared to Clear, our implementation of CAVA offers a simpler design. Specifically, the value prediction engine is located close to the L2 cache, off the critical path, and is trained only with L2 misses. In Clear, prediction and validation mechanisms are located inside the processor core. Moreover, to simplify the design, CAVA explicitly chooses to support only a single outstanding checkpoint at a time, and terminates the current speculative section when a low-confidence prediction is found. Clear supports multiple concurrent checkpoints, which requires storing several register checkpoints at a time, and recording separately in the speculative buffer the memory state of each checkpoint. Finally, we show how to support CAVA in multiprocessors, an area not considered by Clear. A longer discussion on how CAVA and Clear compare is presented in Section 7.

Our simulations show that, relative to an aggressive conventional superscalar baseline, CAVA speeds up execution by up to 1.45 for SPECint applications and 1.58 for SPECfp applications, with a geometric mean of 1.14 for SPECint and 1.34 for SPECfp. Compared to the same baseline, Runahead obtains geometric mean speedups of 1.07 and 1.18 in SPECint and SPECfp applications, respectively.

This paper is organized as follows: Section 2 presents background information; Section 3 describes design issues in CAVA; Section 4 presents our microarchitectural implementation; Section 5 presents our evaluation methodology; Section 6 evaluates our implementation and variations; and Section 7 discusses related work.

2 Background

2.1 Miss Status Holding Registers (MSHRs)

Miss Status Holding Registers (MSHRs) [14] hold information about requests that miss in the cache. Typically, an MSHR is allocated when a miss occurs and is deallocated when the data is finally obtained. Multiple concurrent misses on the same line share the same MSHR — each of them uses a different subentry in the MSHR. However, only the first request is propagated down the memory hierarchy.

There are many possible organizations for MSHRs. In this paper, we use the Explicitly-Addressed MSHR organization of Farkas and Jouppi [7]. In such an organization, each subentry in the MSHR contains the explicit address of the word requested. This allows multiple outstanding misses to the same word address, each one allocating a new subentry.

2.2 Checkpointing and Buffering for Undo

Low-overhead, hardware-based register checkpointing is widely used in processors to support branch speculation. Recent work has used it to support speculative execution of long code sections (which overflow the ROB) in other environments. Examples of such work include early resource recycling [19], data prefetching and branch prediction training through runahead execution [21], aggressive support for a large number of in-flight instructions [1, 6, 25], or in Thread-Level Speculation (TLS) [9, 13, 24, 26]. We will use such support in CAVA.

In these checkpointed architectures, where the processor can run speculatively for a long time, the speculative memory state generated by the processor has to be buffered. Such state can be buffered in the store queue (e.g. [1, 6, 25]). Alternatively, it can be stored in a dedicated speculative buffer (e.g., [21]) or in the L1 as long as the lines with speculative data are marked (e.g., [19]). Without loss of generality, in CAVA, we use the L1 cache. In this approach, when the speculative section commits, such marks are reset and the lines are allowed to remain in the cache; if the section is squashed, the lines with the mark are invalidated. While a line has the mark set, it cannot be displaced from the cache.

3 Hiding L2 Misses with Checkpoint-Assisted Value Prediction

We propose Checkpoint-Assisted Value Prediction (CAVA) to hide long-latency L2 misses and minimize processor stalls due to load misses in on-chip L2 caches. Figure 1 illustrates the concept. Figure 1(a) shows the timeline of an L2 miss in a conventional processor. Figures 1(b) and 1(c) show the actions under CAVA. When an L2 load miss is detected, a prediction of the requested data's value is passed on to the CPU. When the missing load reaches the head of the ROB, the CPU checkpoints, uses the predicted value, and continues execution by speculatively retiring the load. Since the processor may execute for a long time before the data is received from memory, the processor can retire program state to both registers and L1 cache. When the data is finally received from memory, its value is compared to the prediction. If the prediction was correct (Figure 1(b)), the checkpoint is discarded and no action is taken. If the prediction was incorrect (Figure 1(c)), the register state is restored from the checkpoint and the cache lines generated since the checkpoint are discarded. This rolls back the processor to the state at the checkpoint. Execution resumes from there.

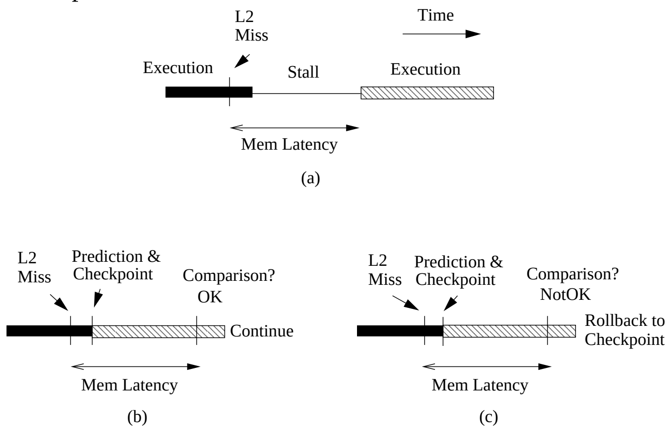


Figure 1: Example of execution with conventional (a) and CAVA (b and c) support.

We need four components to support CAVA. A first module predicts the return value for each L2 load miss and passes it to the processor. It also keeps the predicted value for later comparison to the correct data coming from memory. We call this module Value Predictor and Comparator (VP&C). Second, we need support for fast register checkpointing. Third, we need an L1 cache that marks lines with speculative data, and prevents their displacement until the prediction is proven correct. Finally, when a prediction is incorrect, we need a rollback mechanism that restores the checkpoint and invalidates the speculative cache lines.

In this paper, we support the four components in hardware. We place the VP&C module close to the L2 cache controller, where it can easily observe L2 misses.

3.1 Design Issues

Key design issues in a CAVA processor include where to place the value predictor, when to use the predicted value, when to checkpoint, the number of outstanding checkpoints, when to finish speculation, and how to correctly support it in multiprocessor systems.

3.1.1 Where to Place the Value Predictor

In CAVA, we predict the value of data for loads that miss in the L2 cache. Two possible locations for the value predictor are: the processor core and by the L2 cache controller. The first location is attractive because all memory requests are visible, regardless of their hit/miss outcome, and information such as program counter and branch history is readily available. On the other hand, the value predictor occupies precious real state in the core, and may make it harder to design the processor for high frequency. If the value predictor is placed by the L2 cache controller, not all memory requests and processor information are visible to it (see Section 6.3.1). However, it is a simpler overall design, as it removes the necessary storage and logic from time-critical core structures. For this reason, CAVA places the value predictor by the L2 cache controller.

Even though we have placed the value predictor at the L2, we expose all L1 load misses to the L2 cache to see if they miss and need a prediction. This includes secondary misses in the L1, which are requests to L1 lines for which there is already a pending miss. For this reason, CAVA slightly modifies the L1 MSHR organization to send information about all secondary misses to the L2. This ensures that the value predictor can return predictions for all L2 load misses. Note, however, that the L2 only returns the line with the correct data to L1 once (Section 4.1.1). Overall, for the predictor CAVA uses, we find that the accuracy is higher when we train it with only L2 misses rather than with all processor accesses (Section 6.3.1).

3.1.2 When to Use the Predicted Value

A processor can use the predicted value as soon as it is available (Immediate use), or it can buffer it and consume it only when the missing load is at the head of the ROB and memory has not yet responded (Delayed use). Alternatively, it can use it only when both the load is at the ROB's head and the processor is stalled because the ROB is full.

While the third choice is unattractive because it is likely to have less potential, there are trade-offs between Immediate and Delayed use. Immediate use may enable faster execution and does not require buffering the predicted value in the processor. Moreover, as soon as a misprediction is detected, the VP&C module can trigger a rollback immediately, as the processor has surely consumed the incorrect prediction. A Delayed use has the opposite characteristics. In particular, every misprediction requires a check to see if the processor has actually consumed the incorrect value. However, it has the advantage that the VP&C module may be able to confirm or reject many predictions before they are actually used. This can reduce the number of potentially unnecessary checkpoints and, especially, the number of rollbacks. Consequently, CAVA uses the Delayed scheme. We estimate its benefits compared to the Immediate scheme in Section 6.3.3.

3.1.3 When to Checkpoint

There are two choices of when to checkpoint: at the missing load or at the first instruction that uses the predicted value. Strictly speaking, the checkpoint can be delayed until the first use. However, doing so complicates the design. For example, as the predicted value reaches the processor, the first use may be unknown, or may be an instruction in a mispredicted branch path. Consequently, CAVA checkpoints at the load retirement. We expect little performance difference because the distance between load and use (typically only a few instructions, as shown in Section 6.3.4) is much smaller than the latency of an L2 miss (typically equivalent to hundreds of instructions).

3.1.4 Number of Outstanding Checkpoints

The issue of how many outstanding checkpoints to support at a time is one of performance versus complexity. Since several L2 misses may overlap in time, one approach is to start a new checkpoint at each missing load. This would enable the processor to roll back only up to the first misprediction. However, multiple checkpoints increase hardware complexity, since the processor needs to keep several register checkpoints at a time and also record separately in the L1 cache the state generated after each checkpoint. Consequently, to simplify the hardware, CAVA supports only one checkpoint at a time. If several misses overlap, CAVA takes a checkpoint only at the first one, and associates all misses with the one checkpoint. If any predicted value happens to be incorrect, CAVA rolls back to the checkpoint.

3.1.5 When to Finish Speculation

Speculation ends with a commit when the last outstanding memory response for a group of overlapping, correctly-predicted misses arrives at the processor. Assuming an average prediction accuracy a, the probability of rolling back a checkpoint after consuming n predictions is $P_{rollback} = 1 - a^n$. Since a is significantly smaller than 1 (a little over 0.5), $P_{rollback}$ increases very fast with n. Consequently, CAVA uses a confidence estimator to detect when a possible bad prediction is produced. When the processor is about to consume a low-confidence prediction, CAVA stops predicting values and treats misses conventionally, so that the current speculative section can eventually commit and decrease the chance of wasting work. A new speculative section may then be started. In practice, we observe that good predictions tend to cluster in time (Section 6.2).

There are three other cases where CAVA also stops value prediction and starts treating misses conventionally to speed up eventual commit. One is when the number of outstanding predictions reaches the limit that can be buffered in the hardware structure that records predicted requests. A second case is when the amount of speculative state stored in the L1 cache reaches a certain threshold — if the state were about to overflow, the program would need to stall in order to prevent polluting the lower level of the memory hierarchy with speculative data[1]. Finally, the third case is when the number of instructions retired speculatively since the most recent checkpoint reaches a certain threshold T_{chk}. At that point, we stop new predictions until after all current predictions are confirmed or rejected, and the current speculative execution is committed or rolled back. We do this to reduce the chances that a misprediction will wipe out a considerable amount of work.

3.1.6 Multiprocessor Issues

Supporting CAVA in a multiprocessor environment with a relaxed consistency model requires three special considerations: termination of speculation on fences, consistency of value predictions, and transferring speculative data across threads. We consider them in turn.

The first consideration is simple: when a thread reaches a fence, it has to stall until all its previous memory operations

[1] In our experiments, we have observed that stall due to overflow almost never happens because speculative execution tends to be relatively short.

have completed. This includes outstanding loads for which a prediction has been used. Consequently, executing a fence effectively implies terminating the current speculative section. Codes with very frequent synchronization are less likely to benefit from CAVA.

The second issue was pointed out by Martin *et al.* [18]: unless care is taken, an inconsistency may arise if, between the use of a value prediction and its confirmation, a second thread updates the corresponding location. To see the problem, we slightly change the example suggested by Martin *et al*. Consider a two-element array *A[]* and a pointer *P*. For each element of the array, a producer thread: (i) first initializes it and (ii) then sets *P* to point to it. Consider the time when the producer thread has just initialized *A[0]* and set *P* to point to it. A consumer thread reads *P*, misses in the cache, and predicts that it points to *A[1]*. With naive value prediction, it proceeds to access *A[1]* and reads an un-initialized value. Later, the producer thread initializes *A[1]* and sets *P* to point to it. After that, the cache miss by the consumer completes and finds that the prediction that it made (*P* pointing to *A[1]*) was correct. Unfortunately, the consumer does not realize that it read un-initialized data!

To eliminate this problem, CAVA uses a scheme that follows the guidelines in [18]. Specifically, the hardware needs to determine if, between the use of a prediction on the contents of address *A*, and its confirmation, any other processor updated *A*. If another processor did, the hardware conservatively squashes and rolls back the speculative section.

To identify if another processor wrote to *A*, we use the cache coherence protocol. However, since the line that contains *A* may be evicted from the cache and might even have been retired already, we cannot rely on checking invalidations to the L1 cache or the load queue. Instead, CAVA adds a buffer that holds the addresses of cache lines for which at least one prediction has been consumed by the processor. This buffer snoops invalidations coming from the memory bus. If an invalidation is received for a line whose address is held in the buffer, the processor rolls back. The buffer is cleared at every rollback or checkpoint commit.

This buffer is unnecessary in traditional load value prediction schemes (without checkpointing). The reason is that the load instructions that used a predicted value would still be in flight in the processor, and the load queue could be snooped. With CAVA, such instructions could be retired already.

The final issue is transferring speculative data from a thread to another one. If this were allowed, the receiver must become speculative, and be ready to roll back if and when the supplier thread rolls back. To reduce complexity, our CAVA implementation does not allow the coherence protocol to provide speculative data to another cache.

4 Implementation

Based on the previous discussion, we outline CAVA's implementation. We first describe the microarchitectural structures and then the key operations.

4.1 Microarchitectural Structures

4.1.1 Basic Buffers

CAVA is built around two buffers: the *Outstanding Prediction Buffer (OPB)*, which extends the MSHRs of the L2 cache controller, and the *Ready Buffer (RDYB)*, which buffers the predictions inside the processor's load functional unit.

In conventional processors, the L2 cache controller allocates an MSHR entry for every L2 miss, to keep a record of its pending status. In CAVA, the structure (now called OPB) also obtains a predicted data value for the load from a value predictor, sends the prediction to the processor, and stores it locally. Note that predictions are made at the granularity requested by the processor (e.g., word, byte, etc)[2]. When the requested cache line arrives from memory, the OPB compares the line's data against all the predictions made for words of the line. The OPB deallocates the corresponding entry and 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.

Figure 2(c) shows the OPB structure. As a reference, Figure 2(b) shows the MSHR in the L1 cache, whose structure is unmodified. For both the L1's MSHR and the L2's OPB, we use an Explicitly-Addressed organization [7] (Section 2.1). The control logic of the L1's MSHR is changed slightly, so that secondary misses in L1 are propagated to the L2 (Section 3.1.1).

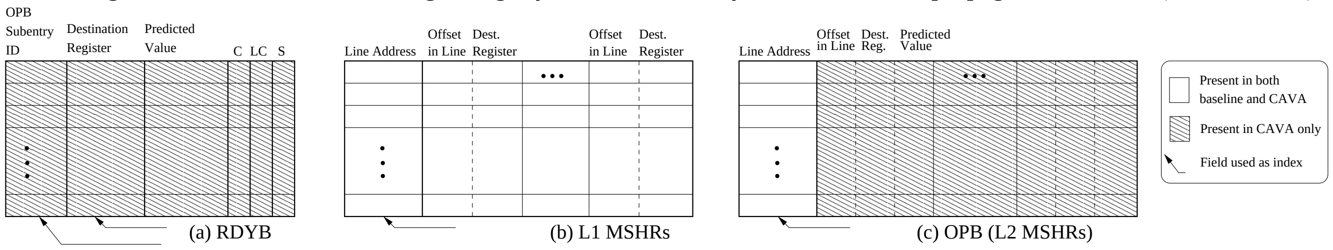


Figure 2: Main microarchitectural structures in CAVA. In the figure, register refers to a physical register name.

A conventional MSHR in L2 only keeps line address information. The OPB extends it with additional information for 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.

On the processor side, the purpose of the RDYB is to temporarily buffer the value predictions forwarded by the OPB until they are confirmed or rejected. A new RDYB entry is allocated when a prediction is received; the entry is deallocated when the processor finally receives the value from memory with a confirmation or rejection tag.

Figure 2(a) shows the RDYB structure. The first field (*OPB Subentry ID*) contains 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. This is used to index the RDYB — the physical register number cannot be used because the register may have been recycled and reused (by the time the prediction confirmation/rejection comes back from memory, the load may have already retired). The second field (destination register) is checked by missing loads right before they retire, in order to obtain the predicted

[2]For simplicity, we use the term word to refer to any fine-granularity data size.

data (third field), which is then copied to the load's destination register. At that point the register is guaranteed not to have been recycled, since the load is still in the ROB.

From this discussion, we see that the RDYB is indexable by both the OPB Subentry ID and the physical register number. This can be accomplished without unreasonable hardware cost given that the RDYB is not a time-critical structure. Since it is typically accessed by L2-missing loads right before they retire, and by confirmation/rejection messages coming from memory, the RDYB can afford to have a multicycle access time.

In addition to a valid bit, the RDYB also stores three additional bits: Consumed (C), LowConfidence (LC), and Stale (S). The Consumed bit is set when the entry is consumed. The LowConfidence bit is set when the value prediction received from the OPB is of low confidence. The Stale bit is set for entries that are still allocated when the processor is rolled back. The reason is that a RDYB entry is only deallocated on reception of a value prediction confirmation or rejection message, while a rollback can occur before some entries receive such a message. When an entry with a Stale bit set finally receives the confirmation or rejection message, it is silently deallocated.

4.1.2 Value Predictor

The L2 controller contains a Value Prediction module that is closely associated with the OPB and trained only with L2 cache misses. When an L2 miss occurs, the value predictor predicts the value of the requested word. The value is stored in one of the OPB Subentries (Figure 2(c)) and returned to the processor, together with a high or low confidence code. The processor allocates a RDYB entry to store the prediction and associated information.

CAVA uses a hierarchical value predictor, which contains a global and a local value predictor, along with a selector. The value predictor also has a confidence estimator that estimates the confidence in each prediction. The structure of the value predictor, selector, and confidence estimator is discussed in Section 5. The L1 acts as a filter to reduce the pressure of updates on the value predictor.

4.1.3 Additional Processor Support

In addition to the RDYB and the hardware-based register checkpointing of Section 2.2, a CAVA processor needs three small modifications. The first one is a Status Register (Figure 3), which indicates if the processor is currently running speculatively (*Chk-Low* or *Chk-High* mode, depending on the prediction confidence) or not (*Non-Chk* mode). The Status Register is used to decide whether a checkpoint is needed when a retiring load consumes a predicted value.

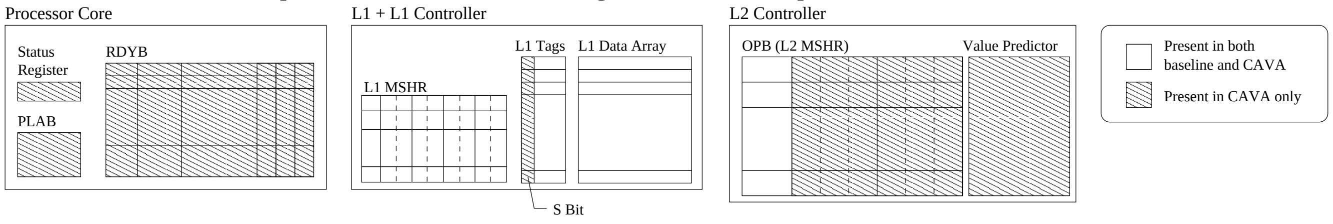


Figure 3: Overall microarchitectural support in CAVA.

The second modification involves passing some bits of the PC and branch history from the processor to the value predictor in the L2 controller. The value predictor we choose (Section 5) requires these bits.

The third modification is the *Predicted Line Address Buffer (PLAB)*. It holds the addresses of cache lines for which at least one prediction has been consumed by the processor during the current checkpointed run. As indicated in Section 3.1.6, this buffer ensures prediction consistency under multiprocessing.

4.1.4 Additional Cache Hierarchy Support

As indicated in Section 2.2, the L1 cache is modified to also buffer memory state generated by the processor as it runs speculatively following a checkpoint. If the speculative section succeeds, such state is merged with the architectural state of the program; otherwise, it is discarded.

More specifically, when the processor speculatively updates a location, the corresponding cache line in L1 is updated and marked with a *Speculative (S)* bit in the tag (Figure 3). If the line was dirty before the update, the line is written back to memory before accepting the update and setting the S bit. In any case, the speculatively updated cache line cannot be displaced from L1.

When all value predictions are confirmed and the processor transitions to Non-Chk mode, the cache commits all the speculative lines by gang-clearing the S bits. The lines can now be displaced to L2 and main memory on demand. Instead, if a prediction fails and the processor needs to roll back to the previous checkpoint, all the lines with a set S bit get their Valid and S bits gang-cleared. These gang operations have been described elsewhere [9, 19]. They are typically done with a hardware signal and can take several cycles, as they occur infrequently.

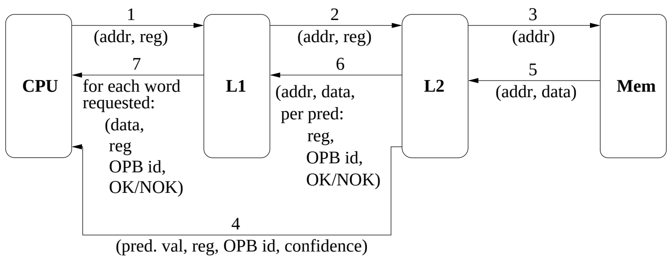
It is possible that, as the processor executes speculatively, the L1 runs out of space for speculative lines. In this case, the L1 signals the processor, which stalls until either it rolls back due to a misprediction or it terminates speculative execution due to confirmation of all outstanding predictions. In our simulations, this stall only occurs very rarely.

4.2 Detailed Operation

To understand the operation of CAVA, we overview a few aspects: communication between processor and caches, entering and committing a speculative section, and handling branch mispredictions and load replays.

4.2.1 Communication between Processor and Caches

Figure 4 shows the messages exchanged between processor and caches. When a load misses in L1 (message 1), the request is forwarded to L2 (message 2), including the fine-grain address and the destination register number. The relevant case is when the load also misses in L2. If the load is for a line already in the L2's OPB, the OPB simply uses a free OPB Subentry in the existing entry. Otherwise, a new OPB entry is allocated and the line is requested from memory (message 3). In either case, a prediction is returned to the processor, together with the register number, OPB Subentry ID, and level of confidence (high or low) in the prediction (message 4). At the processor, the prediction is stored in a newly-allocated RDYB entry and, if appropriate, the LowConfidence bit is set. When the missing load finally reaches the head of the ROB, if the destination register has not yet received the data from memory, the processor checkpoints (unless it is already speculating), consumes the value in the RDYB entry and sets the Consumed bit. The load then retires.



CPU
L1
L2
Mem
1
(addr, reg)
2
(addr, reg)
3
(addr)
7
for each word requested: (data, reg OPB id, OK/NOK)
6
(addr, data, per pred: reg, OPB id, OK/NOK)
5
(addr, data)
4
(pred. val, reg, OPB id, confidence)

Figure 4: Messaging between processor and caches in CAVA.

When the requested line arrives from memory (message 5), the OPB forwards the line to L1 (message 6). For each prediction made on the line's words, the message includes: a confirmation/rejection tag (OK/NOK in Figure 4), the destination register, and the OPB Subentry ID. As the L1 sends each of the requested words to the processor separately (messages 7), it also includes the confirmation/rejection tag, destination register, and OPB Subentry ID.

Every time the processor receives one of these messages, it finds the corresponding RDYB entry. If the Consumed bit is clear, the incoming data is sent to the destination register. If the Consumed bit is set and the message contains a rejection tag, the hardware sets the Stale bit of all the valid RDYB entries, and the processor initiates a rollback. If either the Consumed bit is set and the message contains a confirmation flag, or the Stale bit is set (the load was canceled in a rollback), no action is taken. In all cases, the corresponding RDYB entry is deallocated.

Note that, after a rollback, the OPB continues to send messages with rejection or confirmation tags that reach the processor. As a message matches its RDYB entry, it finds the Stale bit set and, therefore, the RDYB entry is simply deallocated.

4.2.2 Entering and Committing a Speculative Section

As indicated in Section 4.2.1, a processor may enter a speculative section when a load waiting for memory reaches the head of the ROB and finds a valid RDYB entry for its destination register. At this point, three cases are possible. First, if the processor is in Non-Chk mode, it performs the checkpoint, consumes the prediction, and enters a speculative section. The execution mode becomes Chk-Low or Chk-High depending on the LowConfidence bit in the RDYB entry.

Secondly, if the processor is in Chk-High mode and the LowConfidence bit in the RDYB entry is set, the processor waits until all the pending predictions are confirmed and the current speculative section commits. Then, it performs a new checkpoint, consumes the prediction, and starts a new speculative section in Chk-Low mode.

Finally, in all other cases, the processor simply consumes the prediction and remains in the speculative mode it used to be.

On the other hand, a speculative section commits only when the last non-Stale RDYB entry is deallocated. At that point, a hardware signal triggers the changes discussed above on the L1 cache and Status Register. This condition can happen naturally. Alternatively, the processor can gently enable it to occur sooner by stopping the prediction of values for new misses and treating them conventionally. The processor may choose to do this for the reasons discussed in Section 3.1.5.

4.2.3 Branch Mispredictions and Load Replays

In conventional processors, there are cases when the processor issues loads to the cache hierarchy that will not commit. Examples are loads that follow the wrong path of a branch, or loads that need to be replayed to satisfy the memory consistency model or other conditions.

Under CAVA, it is possible that some of these loads miss in L2 and the OPB provides predictions, therefore allocating RDYB entries. However, correctness is not compromised. In conventional processors, when a load is squashed (e.g. on a branch misprediction), the hardware voids the load's load queue entry. In CAVA, this process involves marking the load's RDYB entry as Stale so that, when the correct data arrives from memory, it is not forwarded to the register.

5 Experimental Setup

We evaluate CAVA using execution-driven simulations with a detailed model of a state-of-the art processor and memory subsystem. Due to limited space in this paper, we only evaluate a uniprocessor system; we leave the evaluation of multiprocessor issues for future work. The processor modeled is a four-issue dynamic superscalar with two levels of on-chip caches. Other parameters are shown in Table 1. We set T_{chk} to a value that experimentally appears to work well.

Processor		Cache	I-L1	D-L1	L2	
Frequency: 5.0 GHz	Fetch/issue/comm width: 6/4/4	Size:	16KB	16KB	1MB	Hardware Prefetcher:
Branch penalty: 13 cyc (min)	I-window/ROB size: 60/152	RT:	2 cyc	2 cyc	10 cyc	16-stream stride prefetcher
RAS: 32 entries	Int/FP registers: 104/80	Assoc:	2-way	4-way	8-way	hit delay: 8 cycles
BTB: 2K entries, 2-way assoc.	LdSt/Int/FP units: 2/3/2	Line size:	64B	64B	64B	buffer size: 16 KB
Branch predictor (spec. update):	Ld/St queue entries: 54/46	Ports:	1	2	1	Memory: DDR-2
bimodal size: 16K entries	Checkpoint ovhd(hidden): 5 cycles	D-L1 and L2 MSHR: 128 entries each				FSB frequency: 533MHz
gshare-11 size: 16K entries	Rollback ovhd: 11 cycles	CAVA specific:				FSB width: 128bit
		RDYB: 128 entries				DRAM bandwidth: 8.528GB/s
		OPB: 128 entries with 4 subentries each				RT: 98ns
		Val. pred. table size: 2048 entries				(approx. 490 processor cyc)
		Max. inst. ret. spec. (T_{chk}): 3000				

Table 1: Architecture modeled. In the table, RAS, FSB and RT, stand for Return Address Stack, Front-Side Bus, and minimum Round-Trip time from the processor, respectively. Cycle counts refer to processor cycles.

We compare six different architectures: a plain superscalar (*Base*), CAVA with a realistic value predictor (*CAVA*), CAVA with a 100% accurate value predictor (*CAVA Perf VP*), Runahead modified by storing the speculative state in the L1 cache rather than in the Runahead cache [21] (*Runahead/C*), Runahead/C that uses predicted values for missing loads rather than marking their destination register as invalid (*Runahead/C w/ VP*), and Base with a perfect L2 cache that always hits (*Perf Mem*).

In our model of Runahead, we store the speculative state in L1 rather in the original Runahead cache [21]. We do so to make the comparison to CAVA more appropriate. Note that we also use different architectural parameters: a 4-issue processor and a 1 MB L2, rather than the 3-issue processor and 512 KB L2 used in [21]. Moreover, we use different applications. However, our results for *Runahead/C* are in line with those in [21]: the mean speedup for the six (unspecified) SPECint applications reported in [21] is 1.12, while the mean speedup of Runahead/C for our six top-performing SPECint applications is 1.11.

All architectures including *Base* use an *aggressive 16-stream stride prefetcher*. This prefetcher is similar to the one in [22], with support for 16 streams and non-unit stride. The prefetcher brings data into a buffer that sits between the L2 and main memory.

Our value predictor is composed of a single-entry global last value predictor, and a last-value predictor indexed by the PC hashed with some branch history bits. A 2-bit saturating counter selector predicts, based on the PC, which prediction to take. In addition, we have a confidence estimator to estimate the confidence degree of the prediction. The confidence estimator is a 2-bit saturating counter indexed by the PC. The last-value predictor, the selector, and the confidence estimator use 2048-entry tables. As shown in Section 6.3.1, we choose this configuration because it gives high accuracy for a reasonable area.

Overall, CAVA requires modest additional storage: approximately 7Kbits for the RDYB, 4Kbits for the confidence estimator, 24Kbits for the OPB, and 68 Kbits for the value predictor, for a total of 103Kbits. All structures except the RDYB are placed outside the processor core.

For the evaluation, we use most of the SPECint and some SPECfp applications. These codes are compiled into MIPS binaries using `gcc` 3.4 with `-O3`. The only SPECint application missing is *eon*, which we do not support because it is written in C++. Some SPECfp applications are not used because they are written in Fortran 90 (which our compiler cannot handle) or use system calls unsupported by our simulator. We run the codes with the *ref* input set. We first skip the initialization, as signalled by semantic markers that we insert in the code. The initialization corresponds to 10 million to 4.8 billion instructions, depending on the application. Then, we graduate at least 600 million correct instructions.

6 Evaluation

6.1 Overall Performance

Figure 5 shows the speedups of the different architectures described in Section 5 over *Base*. If we compare *CAVA* to *Base*, we see that CAVA delivers an average speedup of 1.14 for SPECint applications and 1.34 for SPECfp. In addition, no application is slowed down by CAVA.

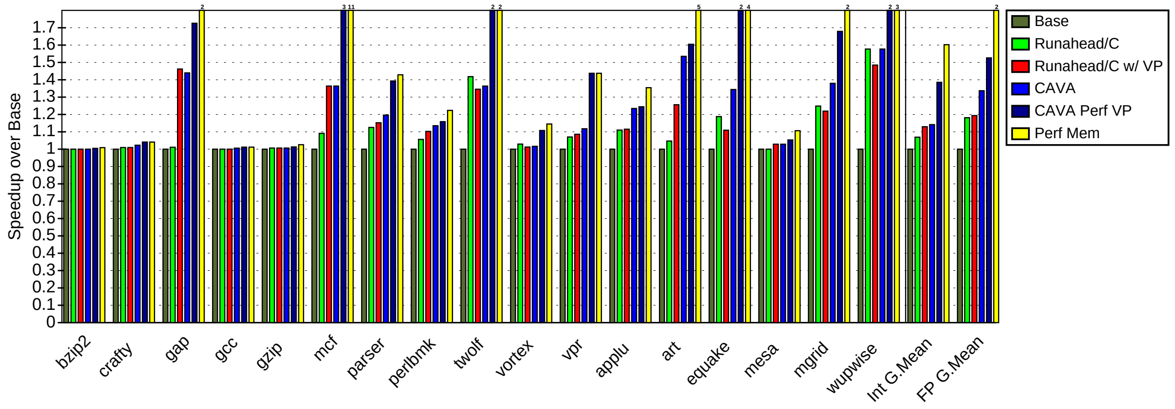


Figure 5: Speedups of the different architectures described in Section 5 over *Base*.

Comparing *CAVA* and *CAVA Perf VP*, we see that the performance of CAVA can be significantly improved with better value prediction. However, even with perfect value prediction (*CAVA Perf VP*), the performance is still far off from the case of a perfect L2 cache (*Perf Mem*). The reason is that *Perf Mem* does not suffer from off-chip memory latency since it never misses in the L2 cache, and it never suffers from stalls due to filling up MSHR structures. On the other hand, in *CAVA Perf VP*, the request still goes to memory to validate the prediction. This means that *CAVA Perf VP* may sometimes suffer from stalls due to lack of resources (e.g., lack of MSHRs due to too many outstanding misses) and from the latency of going off-chip.

On the other hand, for applications with low L2 miss rates (*bzip2*, *crafty*, *gcc* and *gzip*), no architecture makes much of a difference.

If we now compare *CAVA* and *Runahead/C*, we see that *CAVA* is faster: its average speedups of 1.14 and 1.34 on SPECint and SPECfp applications, respectively, are higher than *Runahead/C*'s 1.07 and 1.18. The gains come from two effects, which we can quantify by analyzing *Runahead/C w/ VP*.

Specifically, the difference between *Runahead/C* and *Runahead/C w/ VP* is the support for value prediction for missing loads. As a result, in *Runahead/C w/ VP*, speculative execution leads to execution that is more similar to correct execution. This improves data prefetching and branch training. We call this effect *execution effect*. It is most prominent in SPECint applications.

The difference between *Runahead/C w/ VP* and *CAVA* is that the former rolls back even on a correct value prediction. This wastes some useful work. We call this effect *commit effect*. It is most prominent in SPECfp applications.

We observe some cases where the bars have unusual behavior. For example, *CAVA* is slower than *Runahead/C* in *twolf* and slower than *Runahead/C w/ VP* in *gap*. These effects occur because the speculative sections are typically longer in *CAVA*. *Runahead/C* and *Runahead/C w/ VP* roll back immediately after the first long latency load is serviced. On the other hand, *CAVA* rolls back only after the first value misprediction is detected, which may happen much later, resulting in more wasted work. In addition, the cache may get more polluted.

A second unusual behavior is when the use of value prediction hurts performance. This occurs in *twolf*, *equake*, *mgrid*, and *wupwise*, where *Runahead/C w/ VP* is slower than *Runahead/C*. For these runs, we observe worse branch prediction accuracies and L1 miss rates in *Runahead/C w/ VP*. The predicted values unexpectedly train the branch prediction worse and pollute the cache more.

6.2 Characterization of CAVA

Table 2 characterizes the execution under *CAVA*. As a reference, Columns 2 to 4 show some characteristics of execution under *Base*: the IPC, L2 miss rate, and the percentage of L2 misses that find the requested data in the prefetch buffer, respectively. Note that the prefetcher is also present in *CAVA*.

Column 5 shows *CAVA*'s IPC. Compared to *Base* (Column 2), *CAVA* typically has a higher IPC. In applications with very low L2 miss rate such as *bzip2, crafty, gcc, gzip* and *vortex* (Column 3), the two IPCs are very similar.

Columns 6 and 7 show the major *CAVA* overheads. Specifically, Column 6 shows the fraction of instructions wasted in

App.	Base			CAVA					Checkpointed Run			
	IPC	L2 miss rate (%)	Prefetch coverage (%)	IPC	Instrs wasted (%)	Rollback overhead (% cycles)	Val. pred. accuracy (%)	Conf. est. accuracy (%)	Separation (instrs)	Duration (instrs)	Number preds	Failures (%)
bzip2	2.24	0.0	47.0	2.24	0.0	0.0	58.4	96.0	48702	167	1.7	94
crafty	2.19	0.0	0.1	2.24	2.0	0.0	61.6	81.9	27877	1054	2.3	48
gap	0.91	1.4	65.7	1.31	23.0	2.2	50.1	85.4	645	215	6.4	87
gcc	1.72	0.0	56.9	1.73	0.0	0.0	48.4	76.7	26939	178	1.2	56
gzip	1.56	0.1	97.3	1.57	1.0	0.0	3.9	97.1	25559	270	52.7	97
mcf	0.11	14.8	28.7	0.15	78.0	1.6	59.8	80.5	211	182	9.2	67
parser	1.12	0.4	51.5	1.34	16.0	0.6	51.2	81.0	1557	326	3.3	61
perlbmk	2.15	0.2	26.1	2.44	3.0	0.4	74.1	90.1	4583	162	3.0	73
twolf	0.55	0.9	0.7	0.75	45.0	1.3	39.1	73.1	646	395	4.3	64
vortex	2.42	0.1	36.6	2.46	8.0	0.2	61.8	85.0	6730	633	13.1	72
vpr	1.28	0.2	1.0	1.43	24.0	0.5	21.3	88.4	2295	708	3.9	71
applu	2.09	0.2	29.7	2.58	3.0	0.0	59.0	99.4	16985	1043	45.6	55
art	0.43	30.4	94.6	0.66	38.0	1.4	54.7	92.6	412	308	30.6	56
equake	0.64	3.5	80.5	0.86	55.0	3.9	47.1	76.6	250	191	13.1	53
mesa	2.44	0.2	78.7	2.51	3.0	0.1	31.5	81.1	11147	445	3.1	69
mgrid	1.37	1.2	93.4	1.89	27.0	2.2	75.1	99.2	695	464	15.1	62
wupwise	1.30	1.2	77.2	2.05	28.0	1.3	46.1	88.7	1580	778	33.2	76
Int Avg	1.48	1.6	37.4	1.61	18.2	0.6	48.2	85.0	13249	390	9.2	72
FP Avg	1.38	6.1	75.7	1.76	25.7	1.5	52.3	89.6	5178	538	23.4	62

Table 2: Characterizing *CAVA* execution.

rollbacks. Such a fraction is on average 18% and 26% for SPECint and SPECfp, respectively. Since these instructions are executed in the shadow of a miss, discarding them does not affect performance much. Furthermore, they may train branch predictors and prefetch data. Column 7 shows the total rollback overhead in percentage of program cycles. This number is only 0.6% and 1.5% of the program cycles for SPECint and SPECfp, respectively. This number can be computed by multiplying the 11 cycles of rollback overhead (Table 1) times the number of rollbacks. During this time, the processor stalls. There is also the overhead of checkpointing (5 cycles as shown in Table 1). However, such overhead is not visible to the application, since it is done completely in the background and overlapped with useful computation.

The value predictor used for *CAVA* has reasonable accuracy (Column 8). We compute accuracy as the ratio of correct predictions (both high and low confidence) over all predictions. On average, the accuracy is 48% for SPECint and 52% for SPECfp.

Column 9 lists the accuracy of the confidence estimation mechanism. We define the accurate cases for the estimator those when the prediction is declared high-confidence and is correct, or it is declared low-confidence and is incorrect. From the table, we see that the average accuracy of the confidence estimator is 85% and 90% for SPECint and SPECfp, respectively. Although not shown in the table, both the low- and high-confidence estimations have high accuracy. Specifically, 87.2% and 91.2% of all the high-confidence predictions are correct in SPECint and SPECfp, respectively. Moreover, 82.6% and 88.1% of all the low-confidence predictions are incorrect in SPECint and SPECfp, respectively. On average, 51.2% and 48% of the predictions are high-confidence for SPECint and SPECfp, respectively.

The last four columns of Table 2 (Columns 10-13) characterize the average behavior of a speculative section. Such a section, which we call a *Checkpointed Run*, starts when a checkpoint is created and finishes when a commit or a rollback occurs. Column 10 shows that the separation between consecutive checkpointed runs (from checkpoint creation to the next checkpoint creation) is on average slightly over 13K instructions for SPECint and 5K for SPECfp. SPECfp applications have more frequent checkpoints than SPECint applications because they have higher L2 miss rates.

Column 11 shows that the average checkpointed run lasts for 390 instructions for SPECint and 538 for SPECfp. These numbers are smaller than the latency of a memory access because sometimes a checkpoint finishes early due to low confidence predictions. Moreover, according to Column 12, a run contains on average 9.2 predictions for SPECint and 23.4 for SPECfp. In addition, Column 13 shows that the fraction of runs that terminate with a rollback is on average 72% for SPECint and 62% for SPECfp. It is interesting to note that, although SPECfp applications have more predictions per checkpointed run than SPECint codes, checkpointed runs fail less often. In both SPECfp and SPECint, however, it is clear that correct predictions are clustered in time: given the many predictions needed per run and the average value prediction accuracy (around 50%), if each prediction had the same probability of failure, practically all checkpointed runs would fail.

Finally, Figure 6 gives the intuition as to why *CAVA* delivers better performance than *Base*. The figure shows histograms of the number of outstanding L2 misses during program execution when there is at least one outstanding L2 miss. The data in the figure corresponds to *mcf* and *art*, both under *Base* and *CAVA*. Comparing the histograms, we observe that, under *CAVA*, the case of a high number of concurrent outstanding L2 cache misses occurs more often. This shows that *CAVA* enables more memory level parallelism.

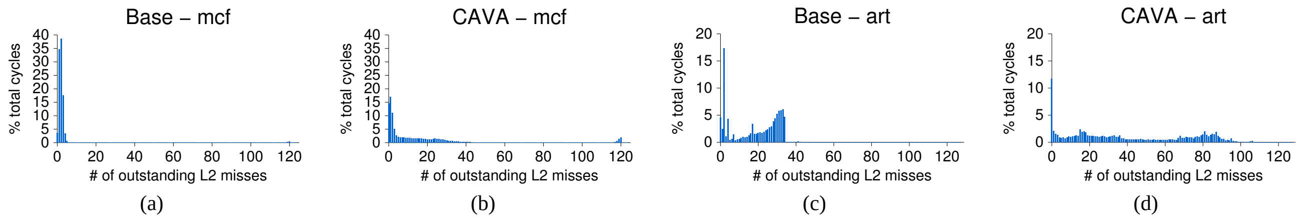


Figure 6: Distribution of the number of outstanding L2 misses when there is at least one outstanding L2 miss. This data corresponds to *Base* and *CAVA*, for the memory-bound applications *mcf* and *art*.

6.3 Sensitivity Analysis

6.3.1 L2 Value Prediction Accuracy

We examine the prediction accuracy of several value predictors. The predictors analyzed are: zero predictor (*Z*, always predict the value zero); single-entry global last-value predictor (*GLV*); last value predictor indexed by a hash of the PC (*LV*); last value predictor indexed by a hash of the PC and the branch history (*BHLV*); stride predictor (*S*); and finite context method predictor (*FCM*) [8]. We also analyze combinations of any two of them, where the prediction is selected by a 2-bit saturating counter selector indexed by a hash of the PC.

Figure 7 shows the prediction accuracy across all the applications for each predictor. There are two bars for each predictor. The first one is the accuracy when trained exclusively with L2 misses, as is the case in CAVA. The second one is the accuracy when trained with all memory accesses, as is the case, for example, in Clear [12]. In both cases, predictions are only generated on L2 misses. The labels on top of the bars show the total size of the corresponding predictor in bits. All predictors, including selectors, have 2048 entries except for *Z* and *GLV*.

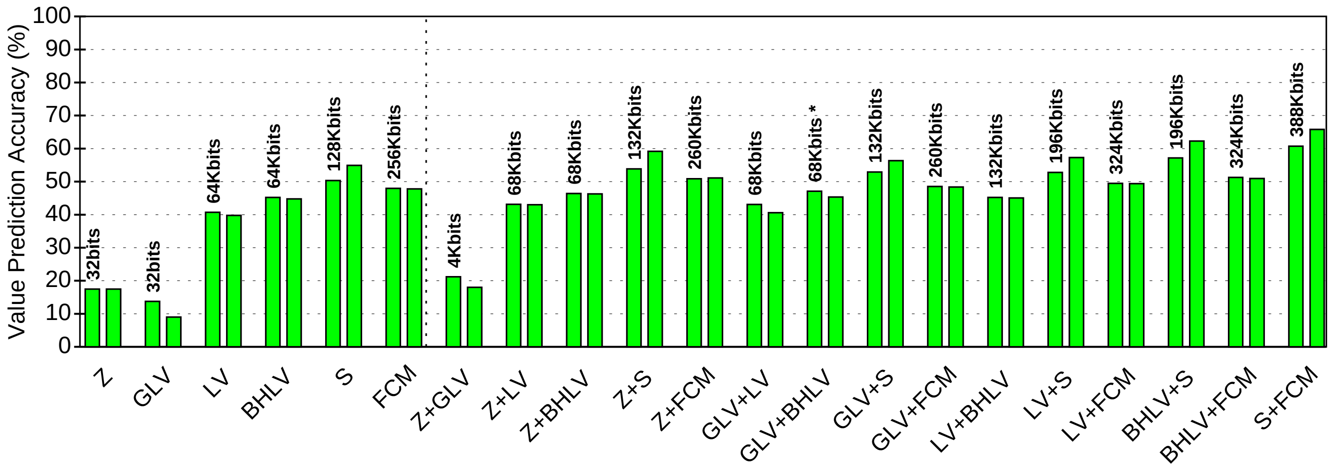


Figure 7: Value prediction accuracy and size for various predictors. The accuracy across applications is computed by weighing the accuracy in each application by the number of predictions in the application. There are two bars for each predictor: the first one is the accuracy when trained exclusively with L2 misses; the second one is the accuracy when trained with all memory accesses. The bar with an asterisk corresponds to the predictor used in this paper.

The figure shows that the prediction accuracy when training with only L2 misses is very similar to that when training with all memory accesses. In the predictors where they differ, the difference is small. While, intuitively, training with only L2 misses could see discontinuities in the value stream, the overall impact on the accuracy is minor. As a result, CAVA's approach is not at a disadvantage.

The predictor that exhibits the best size/accuracy tradeoff when trained only with L2 misses is *GLV+BHLV*. Consequently, this is the one that we use in this paper. It has a size of slightly over 8KB and an accuracy close to 50%. Because it is trained with only L2 misses, there is no need to place it close to the processor; it is placed in the L2 cache, where it can be more easily accommodated.

6.3.2 Number of Outstanding Checkpoints

The number of outstanding checkpoints supported at a time is a performance versus complexity trade-off. The complexity comes from the fact that additional checkpoints require additional logic and storage in the processor. Also, the hardware that buffers speculative data needs to support multiple speculative versions, and therefore needs more complicated logic to make data forwarding decisions. To keep the design simple, CAVA supports only a single checkpoint at a time. In contrast, other schemes like Clear [12] support multiple checkpoints.

To see the impact of this decision, Figure 8 shows the performance when multiple checkpoints are supported normalized to that when only one checkpoint is supported. The last bar for each application corresponds to an unlimited number of checkpoints. From the figure, we observe very little performance difference between the schemes. Only *mcf, art, equake, mgrid* and *wupwise* show any noticeable improvement with multiple checkpoints. Even with an unlimited number of checkpoints, the geometric mean speedup changes by only 1% for SPECfp and by even less for SPECint. For this reason, CAVA supports only a single checkpoint at a time.

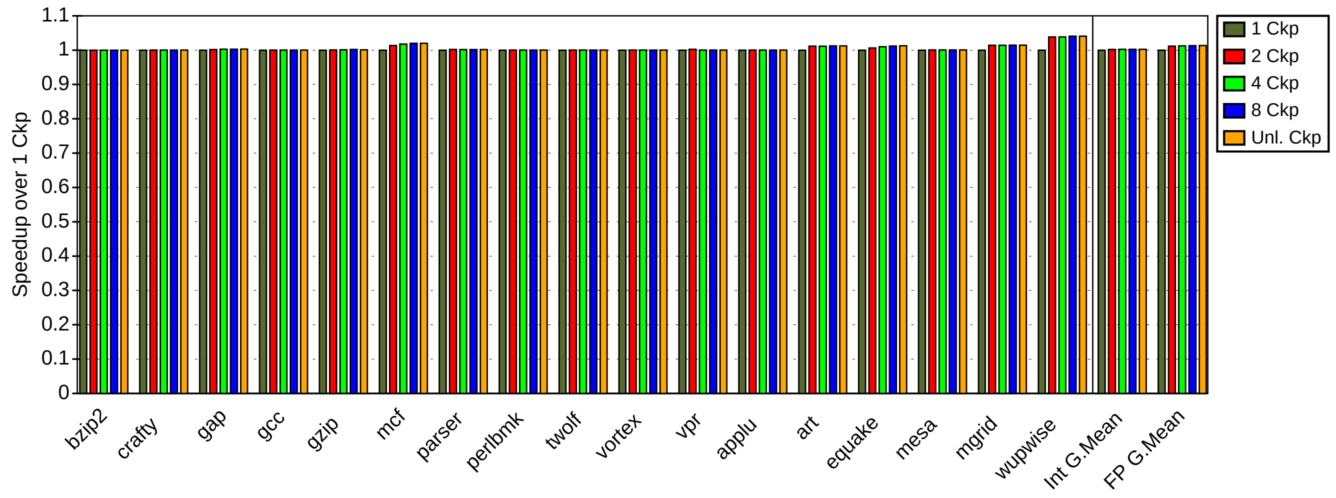


Figure 8: Performance of supporting multiple outstanding checkpoints at a time, normalized to that of supporting only one checkpoint (*1 Ckp*).

6.3.3 Immediate vs Delayed Value Consumption

As described in Section 3.1.2, there is a choice between consuming a value prediction as soon as the prediction arrives at the processor (Immediate use) or waiting until the missing load reaches the head of the ROB (Delayed use). *CAVA* employs Delayed use. The rationale is that a prediction may be rejected before its value is actually used, effectively avoiding a rollback.

To assess the impact of supporting the Delayed use in *CAVA*, we measure the percentage of predictions that have not been consumed by the time the value is confirmed or rejected. This is shown in Column 2 of Table 3. On average, over 5% of the predictions for SPECint and 11% of the predictions for SPECfp were not consumed before they were confirmed or rejected. Note that, in Immediate use, it takes only one of these predictions to be incorrect to cause a processor rollback. Since the hardware to implement Delayed use is simple, *CAVA* employs Delayed use.

App.	Pred. not consumed (%)	Ld-to-use distance (instrs)	Impact of confidence estimation (relative)					Impact of T_{chk} (relative)			
			Chkpt duration (instrs)	# Success chkpts	# Failed chkpts	Execution speedup	Wasted instrs	$T_{chk} = 200$ instrs		$T_{chk} = 5000$ instrs	
								Execution speedup	Wasted instrs	Execution speedup	Wasted instrs
bzip2	13.8	2.1	0.96	1.22	0.99	1.00	0.95	1.00	0.87	1.00	1.01
crafty	0.7	3.9	0.89	1.25	1.00	1.00	0.98	1.00	0.94	1.00	1.01
gap	3.7	3.0	0.77	3.62	1.01	1.01	0.98	0.89	0.77	1.00	1.03
gcc	2.6	4.5	0.83	1.49	1.00	1.00	0.99	1.00	0.88	1.00	1.00
gzip	4.1	9.5	0.97	2.93	1.00	1.00	1.00	0.99	0.50	1.00	1.00
mcf	1.5	1.9	0.51	10.07	1.05	1.02	0.97	0.47	0.83	1.05	1.07
parser	2.4	3.5	0.72	2.87	1.02	1.01	0.98	0.97	0.63	1.00	1.00
perlbmk	3.0	1.5	0.54	12.63	0.85	1.00	0.99	0.96	0.78	1.00	1.04
twolf	6.0	2.5	0.44	4.87	1.12	1.00	1.01	1.00	0.81	1.00	1.00
vortex	22.2	5.9	0.84	1.64	1.01	1.00	1.00	1.00	0.97	1.00	1.00
vpr	2.7	1.8	0.76	2.47	0.98	1.01	0.96	1.00	0.87	1.00	1.00
applu	1.2	8.0	0.76	25.53	1.00	1.00	0.99	0.90	0.53	1.00	1.01
art	12.3	1.8	0.53	9.79	1.01	1.13	0.72	0.77	0.47	1.01	1.03
equake	7.8	5.4	0.61	3.30	1.00	1.03	0.91	0.58	0.52	1.00	1.00
mesa	22.1	3.0	0.72	3.03	1.02	1.00	0.98	0.99	0.58	1.00	1.00
mgrid	13.0	8.8	0.80	3.05	0.98	1.02	0.88	0.92	0.85	1.00	1.00
wupwise	10.0	3.9	0.69	4.24	1.13	1.03	0.89	0.91	0.39	1.00	1.05
Int Avg	5.7	3.6	0.75	4.10	1.00	1.01	0.98	0.93	0.80	1.00	1.01
FP Avg	11.1	5.2	0.69	8.16	1.02	1.04	0.89	0.84	0.56	1.00	1.02

Table 3: *CAVA* sensitivity analysis. The impact of the Confidence Estimator (CE) is shown as the ratio between measurements with CE and measurements without CE.

6.3.4 Checkpoint at Load vs at Use

Section 3.1.3 discusses whether to create a checkpoint at a missing load (as in *CAVA*) or at the first use of the corresponding predicted value. We have measured the average distance between a missing load and its first use for *Base*. The results presented in Column 3 of Table 3 show that the distance is small: there are no more than a few intervening instructions between load and use — typically 4-6. Consequently, we conclude that checkpointing at a missing load is good enough.

6.3.5 Confidence Estimation

Columns 4-8 of Table 3 show the impact of our Confidence Estimator (CE) for value prediction. The impact is shown as the ratio between measurements with CE and measurements without CE. Column 4 shows the ratio of the number of instructions per checkpointed run. The effect of the CE is to reduce the size of the checkpointed runs to 75% (SPECint) and 69% (SPECfp) of their original size. The reason is that the confidence estimator stops speculation on low-confidence predictions.

Columns 5 and 6 show the ratio of the number of successful and failed checkpointed runs, respectively. We see that, with CE, the number of successful checkpointed runs is much higher: 4 times for SPECint and 8 times for SPECfp. We also see that the number of failed checkpointed runs stays approximately the same. The reason for this effect is that the CE breaks long checkpointed runs that used to fail into shorter ones that succeed and shorter ones that still fail.

Column 7 shows the ratio of CAVA speedup with CE over CAVA speedup without CE. Furthermore, Column 8 shows the ratio of number of wasted instructions in CAVA with CE over number of wasted instructions in CAVA without CE. Using CE reduces the amount of wasted work in the applications and, therefore, increases the speedup. Specifically, the reduction in wasted work reaches up to 5% for SPECint applications (average of 2%), and up to 28% for SPECfp (average of 11%). The increase in speedup reaches up to 2% for SPECint (average of 1%) and up to 13% for SPECfp (average of 4%). Therefore, we conclude that using CE is useful.

6.3.6 Maximum Checkpointed Run Duration (T_{chk})

We have chosen the threshold for the maximum duration of a checkpointed run (T_{chk}) to be 3000 speculatively-retired instructions because, experimentally, it corresponds to a good performance point. If we do not enforce any threshold, or set it to a very high value, checkpointed runs are allowed to grow large. This increases the probability that an incorrect value prediction results in discarding a significant amount of work. On the other hand, if we set T_{chk} to a very low value, the potential performance improvements of CAVA are low.

Columns 9-12 of Table 3 show the impact of using a much lower and a much higher T_{chk}. The columns show the application execution speedup and wasted instructions due to rollback, all relative to the case for our default $T_{chk} = 3000$ instructions. We see that, for $T_{chk} = 200$ instructions, while the relative number of wasted instructions is low (on average 0.80 for SPECint and 0.56 for SPECfp), the relative speedup is also low (on average 0.93 for SPECint and 0.84 for SPECfp). For $T_{chk} = 5000$ instructions, the number of wasted instructions is higher (on average 1.01 for SPECint and 1.02 for SPECfp), while the speedups do not improve (they are on average 1.00 for both SPECint and SPECfp).

6.3.7 Number of MSHR and OPB Entries

We now vary the number of L1 MSHRs. Figure 9 shows the execution time of the applications for L1 MSHRs where the number of entries ranges from 16 to 8K. In all experiments, the number of L2 MSHRs (in *Base*) and OPB entries (in *CAVA*) are the same as the number of L1 MSHRs. For each size, we show four bars, corresponding to *Base* and *CAVA*, and for SPECint and SPECfp. Under each set of four bars, we show the number of L1 MSHRs. All bars are normalized to the performance for the same application set in *Base* for 128 L1 MSHRs.

We observe that the bars for *CAVA* level off at many more MSHRs than for *Base*. The reason is that *CAVA* can exploit a higher memory-level parallelism and, therefore, can use more MSHRs. Within *CAVA*, the saturation occurs at 128 MSHRs for SPECint and at 512 MSHRs for SPECfp. The reason is that SPECfp applications have a higher potential for memory-level parallelism. Overall, our *CAVA* design with 128 L1 MSHRs and OPB entries (Table 1) has as many as needed for SPECint, but not for SPECfp.

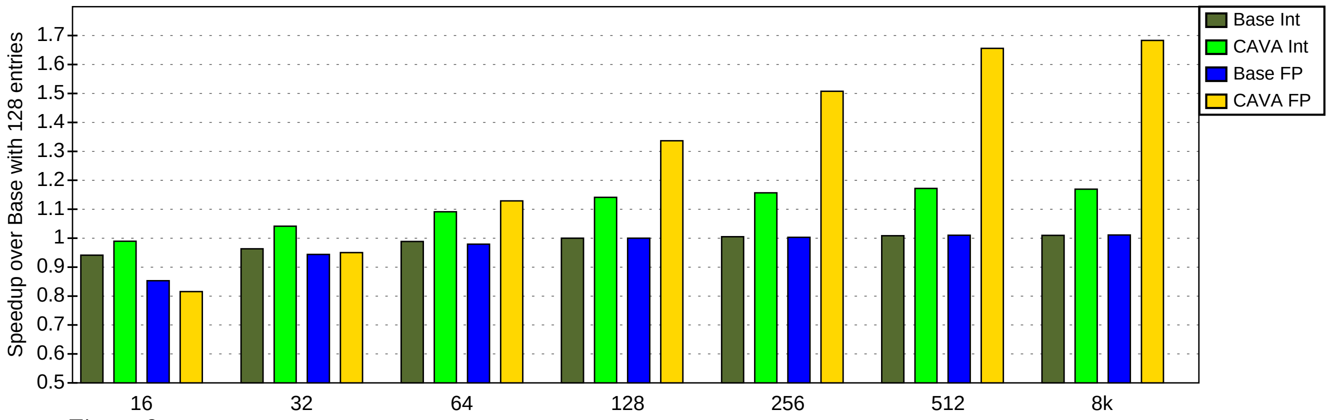


Figure 9: Impact of the number of L1 MSHRs. In all experiments, the number of L2 MSHRs (in *Base*) and in the OPB (in *CAVA*) are the same as the number of L1 MSHRs. All bars are normalized to *Base* for 128 L1 MSHRs.

6.3.8 Sensitivity to Other Overheads

Finally, we examine how certain timing changes affect the speedups delivered by CAVA. Columns 2-4 in Table 4 show the impact of increasing the checkpointing and the rollback overheads from 5 and 11 cycles (default in CAVA), to 10 and 22, 20 and 44, and 40 and 88, respectively. From the table we see that the speedups decrease slowly as we increase the size of these overheads. With 40-cycle checkpointing and 88-cycle rollback, the speedup of CAVA decreases by a geometric mean of 4% in SPECint and 10% in SPECfp.

Columns 5-6 in Table 4 show the impact of increasing the number of pipeline cycles between instruction fetch and rename. While our default CAVA design uses 13 cycles, we simulate machines with 16 and 20 cycles, respectively. From the table, we see that the speedups decrease very slowly. With 20 cycles, the speedup of CAVA decreases by a geometric mean of 1% in SPECint and 2% in SPECfp.

App.	Speedup relative to baseline CAVA				
	Checkpoint & rollback overheads (cycles)			# of stages between fetch and rename	
	10&22	20&44	40&88	16	20
bzip2	1.00	1.00	1.00	1.00	1.00
crafty	1.00	1.00	1.00	1.00	1.00
gap	0.98	0.94	0.86	0.99	0.96
gcc	1.00	1.00	1.00	1.00	1.00
gzip	1.00	1.00	0.99	1.00	1.00
mcf	0.99	0.97	0.94	0.99	0.98
parser	0.99	0.98	0.95	0.99	0.98
perlbmk	0.99	0.98	0.96	1.00	0.99
twolf	0.99	0.96	0.91	0.99	0.98
vortex	1.00	0.99	0.98	1.00	1.00
vpr	0.99	0.98	0.95	1.00	0.99
applu	1.00	1.00	1.00	1.00	1.00
art	0.99	0.98	0.95	0.99	0.99
equake	0.95	0.87	0.73	0.98	0.96
mesa	1.00	0.99	0.99	1.00	1.00
mgrid	0.98	0.94	0.87	0.99	0.98
wupwise	0.98	0.95	0.89	0.99	0.96
Int Geo. Mean	0.99	0.98	0.96	1.00	0.99
FP Geo. Mean	0.98	0.95	0.90	0.99	0.98

Table 4: Sensitivity to various overheads.

7 Related Work

Runahead execution [21] checkpoints the processor and prematurely retires a long-latency load before it completes, so that the processor can continue execution (speculatively). The goal of Runahead is to train branch predictors and to prefetch data into caches. Runahead and CAVA have three major differences. First, in Runahead there is no prediction: the destination register of the missing load is marked with an invalid tag, which is propagated by dependent instructions. Such instructions do not warm up branch predictor or caches. Second, in Runahead, when the data arrives from memory, execution is *always* rolled back; in CAVA, if the prediction is correct, execution is not rolled back. Finally, while Runahead buffers (potentially incomplete) speculative state in a processor structure called Runahead cache, CAVA buffers the whole speculative state in L1.

Clear [12] was *developed concurrently* with CAVA. Compared to Clear, CAVA presents a more simplified design for a few reasons. First, the Value Predictor and Comparator (VP&C) in CAVA is located close to the L2 controller and is trained only with L2 misses. In Clear, prediction and validation mechanisms are located inside the processor core. Second, the decision to take a checkpoint and speculate on a load is also simpler in CAVA. Because the value prediction is located at the L2 in CAVA, the hardware knows explicitly when a load is an L2 miss, whereas, it must be inferred in Clear by tracking timestamps for each pending load. We have shown in Section 6.3.1 that training the predictor only with L2 misses does not affect the prediction accuracy significantly. Third, to simplify the design, CAVA explicitly chooses to support only one outstanding checkpoint at a time, and forces the termination of a high-confidence speculative section when a low-confidence prediction needs to be made; Clear supports up to four outstanding checkpoints. We found that supporting multiple checkpoints increases the performance insufficiently compared to the complexity it requires (Section 6.3.2). Last, another difference between Clear and CAVA is the choice of where to buffer speculative state. Clear implicitly chooses to buffer state in the load/store queue, whereas CAVA uses a single-version cache. However, both CAVA and Clear could be implemented with either approach.

In addition to these issues, our paper described how to support CAVA in multiprocessors, an area not considered by

Clear, and provided a detailed evaluation. Importantly, our paper compares CAVA to Runahead with value prediction — an important design point left out by Clear. Moreover, we evaluated the effect of the number of MSHRs — a fundamental structure to support memory-level parallelism.

Also developed concurrently with CAVA is the work from Tuck and Tullsen on Multithreaded Value Prediction [27]. They propose taking a checkpoint on a long latency load, similar to Clear and CAVA, but, instead of using the same thread context, they spawn a speculative thread in another thread context of an SMT to execute past the load using a predicted value. With their hardware support, they can spawn more than one thread each with a different prediction for a single long latency load. In addition, they add a criticality predictor to their scheme to focus only on loads that appear in the critical path.

There are several other techniques to hide the latency of long-latency operations. For example, CPR [1] and Out-of-order Commit processors [6] remove the ROB and support many instructions in flight, which allows them to hide long-latency operations. They take frequent checkpoints so that on exceptions, branch mispredictions, or similar events, the processor can roll back to a checkpoint.

Lebeck et al [15] propose a design for the instruction window where instructions dependent on a long latency operation are moved from the conventional issue queue to another structure while the long latency operations is executed. Once the long latency operation completes, those instructions are moved back into the conventional issue queue and are executed. In the meantime, instructions not dependent on the long latency operation can be executed.

CFP [25] removes long latency loads and their dependent instructions (slice) from the execution window and places them in an off-critical path structure until the missing load is serviced. In the meantime, independent instructions execute, hiding the load latency. When the load is serviced, the slice is reintroduced in the execution window and is finally executed. Like CAVA, CFP uses checkpointing and is subject to failed speculation. However, the cause is different: on slice construction, some instructions are speculatively predicted by CFP to be dependent on other instructions already in the slice. A major difference between CAVA and CFP is that CFP hides the load latency with the execution of *independent* instructions, while CAVA hides it with both *dependent* (risking mispredictions) and *independent* instructions. There is some evidence that, at least some times, there may not be enough independent instructions to hide the latency of loads [20].

Zhou and Conte [28] use 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 discarded and re-executed. As in Runahead, speculative execution is employed for prefetching.

Several related schemes use register checkpointing and rollback to support the speculative execution of long code sections. For example, Cherry [19] checkpoints and then speculatively recycles resources early. TLS [9, 13, 24, 26] checkpoints and spawns a thread to run speculatively.

Finally, several authors have studied the prediction of register values [3, 8, 16, 23]. In our paper, we have reused some of their algorithms.

8 Conclusion

This paper presented a design and implementation of Checkpoint-Assisted VAlue Prediction (CAVA), a new technique that hides L2 cache misses by predicting their data values, checkpointing the state, and continuing execution. When the response with the value comes back from memory, the prediction is verified. If the prediction is correct, execution continues normally; if it is not, the hardware rolls back execution to the checkpoint. In either case, CAVA can increase performance. Specifically, if the prediction is correct, the processor has performed useful work. If the prediction is incorrect, CAVA has potentially prefetched good data into the caches and trained the branch predictor like Runahead.

CAVA delivers significant speedups for a variety of codes. Specifically, compared to a baseline aggressive superscalar processor, CAVA speeds up execution by up to 1.45 for SPECint applications and 1.58 for SPECfp applications, with a geometric mean of 1.14 for SPECint and 1.34 for SPECfp applications. These results substantially outperform Runahead, which does not use value prediction, and rolls back execution after every speculative section.

References

[1] AKKARY, H., RAJWAR, R., AND SRINIVASAN, S. T. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture* (Nov. 2003).

[2] BAER, J., AND CHEN, T. An Effective On-chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991International Conference on Supercomputing* (Nov. 1991).

[3] BURTSCHER, M., AND ZORN, B. G. Exploring Last *n* Value Prediction. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques* (Oct. 1999).

[4] CALLAHAN, D., KENNEDY, K., AND PORTERFIELD, A. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Feb. 1991).

[5] COOKSEY, R. *Content-Sensitive Data Prefetching*. PhD thesis, University of Colorado, Boulder, 2002.

[6] CRISTAL, A., ORTEGA, D., LLOSA, J., AND VALERO, M. Out-of-Order Commit Processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture* (Feb. 2004).

[7] FARKAS, K. I., AND JOUPPI, N. P. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *Proceedings of the 21st International Symposium on Computer Architecture* (April 1994).

[8] GOEMAN, B., VANDIERENDONCK, H., AND BOSSCHERE, K. D. Differential FCM: Increasing Value Prediction Accuracy by Improving Usage Efficiency. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture* (Jan. 2001).

[9] HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1998).

[10] JOSEPH, D., AND GRUNWALD, D. Prefetching using Markov Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture* (June 1997).

[11] JOUPPI, N. P. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture* (May 1990).

[12] KIRMAN, N., KIRMAN, M., CHAUDHURI, M., AND MARTINEZ, J. F. Checkpointed Early Load Retirement. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture* (Feb. 2005).

[13] KRISHNAN, V., AND TORRELLAS, J. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers* (September 1999), 866–880.

[14] KROFT, D. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th International Symposium on Computer Architecture* (May 1981).

[15] LEBECK, A. R., KOPPANALIL, J., LI, T., PATWARDHAN, J., AND ROTENBERG, E. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proceedings of the 29th International Symposium on Computer Architecture* (May 2002).

[16] LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. Value Locality and Load Value Prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1996).

[17] LUK, C.-K., AND MOWRY, T. C. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1996).

[18] MARTIN, M. M. K., SORIN, D. J., CAIN, H. W., HILL, M. D., AND LIPASTI, M. H. Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Multiprocessing. In *Proceedings of the 34th International Symposium on Microarchitecture* (Dec. 2001).

[19] MARTINEZ, J. F., RENAU, J., HUANG, M., PRVULOVIC, M., AND TORRELLAS, J. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture* (Nov. 2002).

[20] MUTLU, O., KIM, H., STARK, J., AND PATT, Y. N. On Reusing the Results of Pre-Executed Instructions in a Runahead Execution Processor. *Computer Architecture Letters* (Jan 2005).

[21] MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. N. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture* (Feb. 2003).

[22] PALACHARLA, S., AND KESSLER, R. E. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st International Symposium on Computer Architecture* (Apr. 1994).

[23] SAZEIDES, Y., AND SMITH, J. E. The Predictability of Data Values. In *Proceedings of the 30th International Symposium on Microarchitecture* (Dec. 1997).

[24] SOHI, G., BREACH, S., AND VIJAYAKUMAR, T. Multiscalar Processors. In *22nd International Symposium on Computer Architecture* (June 1995), pp. 414–425.

[25] SRINIVASAN, S. T., RAJWAR, R., AKKARY, H., GANDHI, A., AND UPTON, M. Continual Flow Pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 2004).

[26] STEFFAN, J. G., COLOHAN, C., ZHAI, A., AND MOWRY, T. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th International Symposium on Computer Architecture* (June 2000).

[27] TUCK, N., AND TULLSEN, D. Multithreaded Value Prediction. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture* (Feb. 2005).

[28] ZHOU, H., AND CONTE, T. Enhancing Memory Level Parallelism via Recovery-Free Value Prediction. In *Proceedings of the 17th International Conference on Supercomputing* (June 2003).