Enhancing Memory-Level Parallelism via Recovery-Free Value Prediction

Huiyang Zhou, *Member*, *IEEE*, and Thomas M. Conte, *Fellow*, *IEEE*

Abstract—The ever-increasing computational power of contemporary microprocessors reduces the execution time spent on arithmetic computations (i.e., the computations not involving slow memory operations such as cache misses) significantly. Therefore, for memory-intensive workloads, it becomes more important to overlap multiple cache misses than to overlap slow memory operations with other computations. In this paper, we propose a novel technique to parallelize sequential cache misses, thereby increasing memory-level parallelism (MLP). Our idea is based on value prediction, which was proposed originally as an instruction-level parallelism (ILP) optimization to break true data dependencies. In this paper, we advocate value prediction in its capability to enhance MLP instead of ILP. We propose using value prediction and value-speculative execution only for prefetching so that not only the complex prediction validation and misprediction recovery mechanisms are avoided, but better performance can also be achieved for memory-intensive workloads. The minor hardware modifications that are required also enable aggressive memory disambiguation for prefetching. The experimental results show that our technique enhances MLP effectively and achieves significant speedups, even with a simple stride value predictor.

Index Terms—Single data stream architectures.

1 Introduction

The trend in contemporary microprocessor design, including fast clock speed, deep pipelines [25], large instruction window sizes [15], [16], aggressive out-of-order execution, and wide fetch/issue bandwidth [22], results in tremendous capability in performing arithmetic computations (i.e., the computation not involving slow memory operations such as cache misses). Therefore, for memory-intensive workloads, it becomes more important to parallelize multiple cache misses than to overlap cache misses with arithmetic computations.

In this paper, we propose a novel technique to parallelize *sequential* cache misses speculatively. The target workload is memory-intensive workloads with heavy pointer chasing. The idea is developed upon value prediction [9], [18], [19], which was originally proposed as an instruction-level parallelism (ILP) optimization to break true data dependencies in computations. Since the data dependence between pointer-chasing loads enforces sequential execution, value prediction has the capability of parallelizing these loads, thereby increasing memory-level parallelism (MLP). We advocate that *for memory-intensive applications, the largest performance potential of value prediction lies in its capability to enhance MLP instead of ILP*.

Since we focus on using value prediction to increase MLP, the hardware overhead to support value prediction and value-speculative execution can be significantly reduced. In this paper, we propose using value prediction only for prefetching so that not only the complex value prediction validation and misprediction recovery mechanisms are avoided, but *higher* performance improvement can also be achieved. Unlike previously proposed value prediction schemes, where the speculative results are committed when a correct prediction is made, the speculative results in our scheme are only used for prefetching and will not be committed. In a different point of view, one can think of the speculative execution in our approach as a speculative preexecution scheme, which requires neither explicit preexecution thread generation nor multithreading support. Another important aspect is that the same hardware changes to support such value-speculative execution also enable aggressive memory disambiguation to break alias (i.e., load-after-store) dependencies. Such disambiguation is used for prefetching and is also recovery-free.

The experimental results, based on a set of SPEC2000 benchmarks [13] and Olden benchmarks [5] including both computation-intensive and memory-intensive benchmarks, show significant speedups resulting from breaking both true dependencies and alias dependencies between memory operations. Such speedups also scale well with the current trends in microprocessor design.

The remainder of the paper is organized as follows: Section 2 addresses the related work. Section 3 illustrates the performance potential of using value prediction to enhance MLP. Section 4 presents the details of our proposed approach. The experimental methodology is contained in Section 5 and the results are in Section 6. Two simple techniques are proposed in Section 7 to reduce the hardware overhead as well as useless speculation at runtime. The limitations of our proposed scheme are highlighted in Section 8. Finally, Section 9 concludes the paper and discusses future work.

- *H. Zhou is with the School of Computer Science, University of Central Florida, Orlando, FL 32816-2362. E-mail: zhou@cs.ucf.edu.*
- *T.M. Conte is with the Center for Embedded Systesm Research (CESR), Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695-7911. E-mail: conte@ncsu.edu.*

Manuscript received 18 Mar. 2004; revised 18 Jan. 2005; accepted 26 Jan. 2005; published online 16 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0101-0304.

2 Related Work

Due to the speed gap between the processor core and the memory, hiding memory latency has been an active research topic. One well-established solution is memory prefetching and the majority of work is based on address prediction [2], [14]. One recently proposed scheme [7], named *stateless, content-directed prefetch*, improves upon prior techniques by examining the prefetched data to check whether the data could potentially contain a pointer-dereference address. If so, the content will be used as the address for the next prefetch. Compared to it, our proposed technique uses the fetched data to compute pointer-chasing loads' addresses based on program semantics, thereby having fewer chances to fetch wrong data to pollute the caches.

Another promising way to hide memory latency is based on the concept of preexecution/precomputation. Both hardware-based and software-based schemes [6], [20], [23], [27], [33] have been proposed for this purpose. As will be discussed in Section 4, our recovery-free value prediction scheme is similar to the preexecution paradigm, although our approach requires neither explicit thread generation nor multithreaded support. Also, as pointed out in [27], a precomputation thread is more effective when used to prefetch the critical pointer-chasing loads in loop control than to prefetch loads in a loop body. A similar observation can be made for our proposed scheme since predicting pointer-chasing loads in loop control can overlap the execution of multiple iterations and result in better latency hiding. Runahead execution [8], [21] is another form of preexecution without multithreaded support. During the execution, if the processor is stalled due to a cache miss, the current execution state will be checkpointed and the processor enters the runahead mode to preexecute the independent instructions following the blocking instruction. The purpose of the preexecution is to prefetch future data into cache. When the cache miss is repaired, the processor returns to the normal mode and reexecutes these preexecuted instructions. In an out-of-order processor, runahead execution can achieve similar performance of a much larger instruction window. Our proposed scheme and runahead execution can be mutually beneficial as our scheme tries to preexecute the *dependent* operations of a blocking instruction. Also, as discussed in Section 3, large instruction windows achieved by runahead execution provide better chances for our scheme to improve MLP.

Value prediction was proposed originally as an ILP optimization technique [9], [18], [19], [24]. Using value prediction to hide load forward latencies is studied in [4]. By correctly predicting the value of a load instruction, dependent instructions can avoid stalling during the time that the load executes. Address prediction for prefetching is proposed in [10]. Based on address prediction, the data is prefetched and saved in a special buffer (called Memory-Prefetch Table) and used as the value prediction of the load. Our proposed approach is different from these previous works in that we use value prediction only for prefetching, thereby avoiding complex validation and recovery hardware while achieving higher performance for memory-intensive workloads (see Section 4.4 for a detailed discussion). Furthermore, our approach also leverages aggressive memory disambiguation for prefetching. As pointed out in Section 3, it is very important to break *both* true and alias dependencies in order to increase MLP.

3 Using Value Prediction to Enhance Memory-Level Parallelism

Values produced by individual instructions exhibit localities [24] and different value prediction schemes are proposed to exploit such localities to break true data dependencies [9], [18], [19]. In a typical value prediction/speculation scheme proposed for a superscalar processor, the prediction of an instruction enables its dependent instructions to be executed speculatively. If the prediction turns out to be correct, these instructions will commit their speculative results so that the processor makes faster forward progress by hiding the latency of speculative computations. If the prediction is wrong, however, a recovery scheme is necessary to squash the speculative results and to reexecute those affected instructions with correct data.

For memory-intensive workloads with heavy pointer chasing, sequential cache misses resulting from pointer-chasing code dominate the overall execution time. These cache misses form a memory dependence chain since one missing load's address depends on the previous missing load's value. Taking a frequently executed code segment from the benchmark *mcf* as an example,[1] shown in Fig. 1, the profile information shows that the pointer-chasing codes "*node->child*," "*node->sibling*," "*node->basic_arc->cost*," and "*node->pred->potential*" result in many cache misses. From the code segment, it can be seen that the "*node*" traversing can happen in more than one direction (through its "*child*" or "*sibling*" field). After examining the dynamic execution behavior, it is found that the most frequently followed traversing path is through the "*sibling*" field and the traversing load through the field "*child*" always fetches a value of zero, although it causes many L2 cache misses. Therefore, the dynamic memory dependence relationship among those missing loads can be modeled as a dependence chain, as shown in Fig. 2, in which each node represents a cache-missing load. In Fig. 2a, the dependence chain is based on a single iteration of the outer *while* loop in Fig. 1, where nodes 1 and 2 correspond to two dependent missing loads from "*node->basic_arc->cost*." Nodes 3 and 4 correspond to "*node->pred->potential*." Nodes 5 and 6 correspond to "*node->child*" and "*node->sibling*," respectively, and node 0 is the same load "*node->sibling*" from the previous iteration. Fig. 2b shows the dependence chain when the loop is executed multiple times. The solid arrow in Fig. 2 represents true data dependencies and the dashed arrow represents alias dependencies between missing loads. Alias dependence exists between a store and subsequent load instructions. Here, we use the same term to model the dependence between two cache-missing loads when one or more stores exist between them and one of these stores is dependent on the first missing load. It needs to be pointed out that alias dependencies span multiple iterations, though not shown in Fig. 2b for conciseness. Also, note that, in the memory dependence chain, only cache-missing loads are included as other instructions, such as stores, adds, branches, and loads, that hit in caches are not long latency operations.

From this example, we can see that both true data dependencies and alias dependencies enforce sequential

1. A simplified version of the code example is used in [30] to illustrate the concept more concisely.

```
typedef struct node {          //data structure of the variable 'node' in the code
   long number; char * ident;
   struct node *pred, *child, *sibling, *sibling_prev;
   long depth, orientation;
   struct arc *basic_arc, firstout, firstin;
   cost_t potential;
   flow_t flow;
   long mark, time;
} node_t;
 ……
tmp = node = root->child;
while( node != root ){                    //outer while loop
      while( node ){
           if( node->orientation == UP )
         node->potential = node->basic_arc->cost + node->pred->potential;      //nodes 1-4 in Fig. 2
      else{ /* == DOWN */
         node->potential = node->pred->potential - node->basic_arc->cost;      //nodes 1-4 in Fig. 2
         checksum++;
      }
      tmp = node;
      node = node->child;                 //node 5 in Fig. 2, a traversing load
   }
   node = tmp;
   while( node->pred )  {
      tmp = node->sibling;                //node 6 in Fig. 2, a traversing load (the most frequently used traversing direction)
      if( tmp ){                          //mostly taken
         node = tmp;
         break;
      }
      else
         node = node->pred;               //traversing load
   }
}
```

Fig. 1. A code segment in the benchmark *mcf* (in the function *refresh_potential*) resulting in many cache-misses.

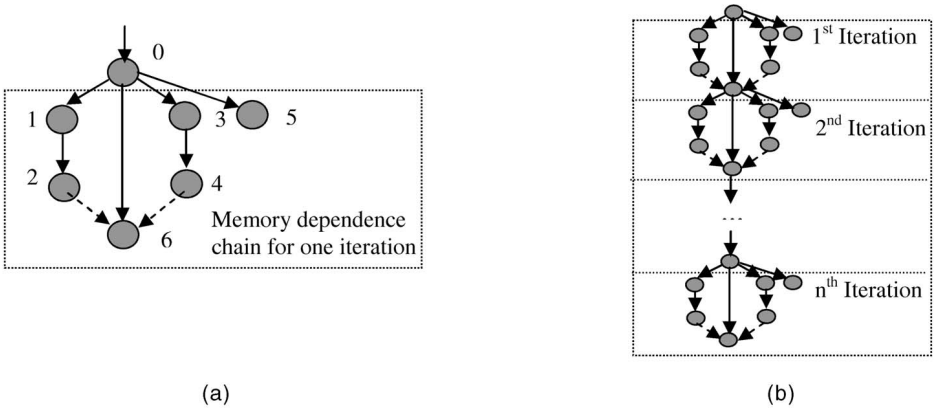

Fig. 2. The memory dependence chain based on the code in Fig. 1 (each node represents a cache-missing load as labeled in Fig. 1). (a) The dependence chain for a single iteration. (b) The dependence chain for multiple iterations (alias dependences among different iterations are not shown for conciseness).

execution of the missing loads, resulting in long execution time. In order to process these cache misses in parallel (i.e., to increase MLP), both dependencies need to be broken. While aggressive memory disambiguation can minimize alias dependencies, value prediction can be used to break true data dependencies. In this example, memory disambiguation removes the dependence of node 6 on nodes 2 and 4 in Fig. 2a, thus exposing the critical path of executing the loop as chasing the pointer "*node->sibling*" (i.e., node 6). If a correct prediction can be made for this load, the execution of multiple iterations of the loop can be overlapped, as shown in Fig. 3, where predicting the value of the pointer-chasing load (node 6′ in Fig. 3) in the second iteration

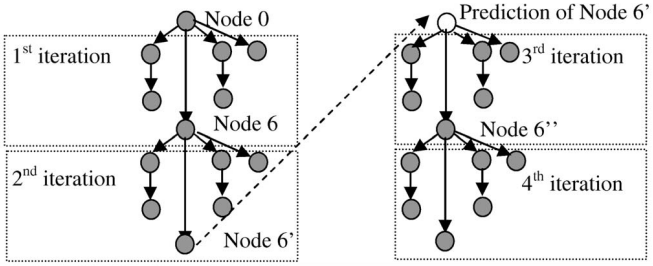

Fig. 3. Predicting the value of Node 6' enables overlapping of cache misses in different iterations.

enables the third and the fourth iterations to be executed speculatively so that their miss latencies are overlapped with the first and the second iterations. As a result, the long miss latencies in the third and the fourth iterations can be completely hidden if the correct value prediction is made.

The example in Fig. 3 illustrates the effectiveness of value prediction in breaking a memory dependence chain: Sequential cache misses can then be processed in parallel and MLP can be enhanced. Such effectiveness is affected by several characteristics of this memory dependence chain. The first is the length of a memory dependence chain. In the example in Fig. 3, the instruction window size determines how many iterations of the loop can be unrolled dynamically. If an instruction window can only hold two iterations of the loop, the speculative execution of the third and the fourth iterations is impossible when they are not fetched into the pipeline. The second is which cache-missing load along this dependence chain is predicted. In the example in Fig. 3, it can be seen that predicting the value of Node 6′ can overlap more cache misses than predicting Node 6 or Node 6″. The third is the predictability of these missing loads' values since more accurate prediction will result in more useful speculative executions. In [32], these characteristics are examined using an analytical model of value prediction in enhancing MLP. It is found that value prediction can be more effective than traditional address-prediction-based prefetching techniques for the same predictability model. The main reason is that, while prefetching techniques only bring the data close to the processor (e.g., the L1 D-cache), value prediction takes one step further by using the fetched data to drive the dependent load instructions to be executed early. In the example in Fig. 3, it can be seen that predicting the value of Node 6′ is equivalent to predicting the address of the dependent loads (e.g., Node 6″) since the only difference is a constant offset. So, using an address-prediction-based prefetching, the miss latency of Node 6″ can be hidden if the prefetch is triggered early enough. Value prediction, on the other hand, not only fetches the data of Node 6″, but also uses the fetched data to execute other dependent instructions (i.e., the cache-missing loads in the fourth iteration) even if their addresses/values are not predictable. As a result, value prediction is capable of hiding much more miss latencies. The analytical model also shows that the effectiveness of value prediction is proportional to the length of a memory dependence chain, value prediction accuracies, and cache miss latencies. Since the chain length scales with the effective instruction window size and miss penalties scale with fast processor clock speed, we argue that value prediction is a very powerful technique to improve MLP for future high performance microprocessors.

4 Recovery-Free Value Prediction

In this section, we discuss our proposed recovery-free value prediction in detail. The core idea is developed in Section 4.1 and a complexity-effective implementation is proposed in Section 4.2. Section 4.3 highlights an interesting observation that recovery-free value prediction is an implicit form of preexecution without multithreaded support. A detailed performance comparison between traditional value prediction and recovery-free value prediction is presented in Section 4.4 to explain why the latter is capable of achieving better performance for our target memory-intensive workloads.

4.1 Core Idea

As discussed in Section 3, value prediction has great potential to enhance MLP by overlapping otherwise sequential cache misses. To implement such a technique, however, complex hardware support is necessary to validate value predictions and to perform recovery from mispredictions. As discussed in Section 1, current microprocessors can perform computations very fast as long as slow memory operations (e.g., cache misses) are not involved. So, unlike previously proposed value prediction schemes [9], [18], [19], we propose using value prediction only for prefetching so that there is no need to validate a prediction or to perform recovery from mispredictions. Using the example in Fig. 3, based on the prediction of Node 6′, the third and the fourth iterations of the loop are executed speculatively. Unlike traditional value prediction schemes, these speculative results won't be committed in our approach and the only purpose of such speculative execution is to bring the data to the caches. As a result, even if the prediction is correct, the third and the fourth iterations of the loop will be executed again (unspeculatively) in our proposed scheme. We expect that such an execution will be very fast since the cache accesses in these iterations will hit in the L1 data cache (as the data have already been fetched during speculative execution if the prediction is correct). So, compared to traditional value prediction schemes, our technique *apparently* trades a small reexecution penalty in the case of correct value predictions for much simpler hardware overhead. In the case of a value misprediction, both traditional schemes and our proposed scheme will result in polluting the data caches, while our scheme associates *no* recovery penalties. A detailed discussion in Section 4.4 reveals that, in most cases, the (unspeculative) reexecution in recovery-free value prediction does not incur additional latencies and misprediction recovery in traditional value prediction schemes will have severer penalties for deeper pipelines. Another interesting point is that the same hardware changes required in our scheme also enable aggressive, recovery-free memory disambiguation for prefetching as a byproduct, therefore being capable of delivering even higher performance improvement.

4.2 A Proposed Implementation

To support recovery-free value prediction, only minor hardware changes are necessary. We present our proposed design based on a MIPS R10000 style microarchitecture [29], which has a 7-stage pipeline, as shown in Fig. 4. For memory operations, the execution (EXE) stage is replaced with an address generation (AGEN) stage and two memory access stages (MEM1 and MEM2). There are four key changes to the hardware, presented as follows.

First, a value predictor is included in the front-end of the processor and is indexed with *pc*, as shown in Fig. 4. The design of a high accuracy value predictor is out of the scope of this paper and we use a simple stride value predictor [9], [18], [24] to show the effectiveness of our technique, though a more powerful predictor [26], [31] can potentially lead to a higher performance improvement (see Section 6.4). To filter out incorrect predictions, a confidence mechanism based on prediction validation is necessary. This validation logic can

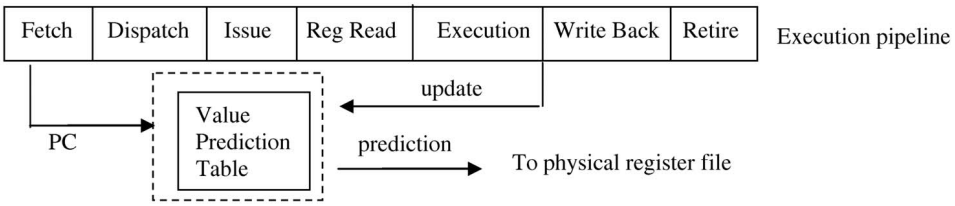

Fig. 4. The execution pipeline.

be built either in the predictor update unit or in the execution core, but off the critical path since such validation is not needed for program execution.

Second, two flag bits are added to control value-speculative execution. One flag bit, called value prediction speculative (*vp*), is added to every entry of issue queue (IQ) (or RUU) and load/store queue (LSQ). The other flag bit, called value prediction ready (*vp_ready*), is added for each register in the physical register file. When a confident value prediction is made at the dispatch stage, the *vp_ready* bit is set for the destination register and the predicted value is written to the physical register file. At the issue stage, if the source registers of an instruction are ready, it will be issued unspeculatively and the execution result will be used to update the value predictor. If source registers are not ready, but the *vp_ready* bits for these source registers are set (i.e., the values of these physical registers are either predicted or computed using previous predictions), the instruction is issued speculatively provided there are unused issue bandwidth and function units. When an instruction is issued speculatively, the corresponding *vp* flag in the IQ/LSQ is set to prevent the same instruction from being issued speculatively more than once since we do not need the same data to be prefetched multiple times. Speculatively issued instructions will remain in the IQ until they are issued unspeculatively later with (unspeculatively) ready source registers. When a speculatively issued instruction finishes, it writes back the speculative results to the physical register file and sets the corresponding *vp_ready* bit to enable dependent instructions to be executed speculatively. Writing the speculative results to the physical register file won't affect the correctness of the program execution since the physical register will be overwritten by the unspeculative execution of the same instruction. In the case when a misprediction leads to a wrong load address, which results in a cache miss, the speculative result may arrive later than the unspeculative result. Such a speculative result is simply dropped as the corresponding LSQ entry has been updated with the correct address, indicating that the load has been executed unspeculatively.

Third, the instruction selection logic is modified so that it prioritizes the issue of unspeculative instructions and prohibits the speculative execution of *store* and *branch* instructions. In such a way, the speculative execution will not compete with normal execution for resources and it only affects the normal execution through the data caches.

Fourth, to break alias (i.e., load-after-store) dependencies, the *vp* flag is set for the load instructions that are stalled due to prior unresolved store addresses. Then, those load instructions can be issued speculatively as if they were based on predicted values. Therefore, no alias dependencies are enforced. This aggressive memory disambiguation requires no recovery since the same load instructions and their dependent instructions will be executed again unspeculatively after the prior store addresses are resolved and the speculative execution is used only for prefetching. We call this *recovery-free speculative memory disambiguation*.

The proposed changes are relatively minor and are unlikely to affect the critical path of the processor. Using the physical register file to keep the value predictions and speculative execution results enables our approach to utilize the otherwise unused machine resources and does not require additional ports to the register file.

4.3 Recovery-Free Value Prediction versus Preexecution

One interesting observation is that our proposed recovery-free speculative execution could be viewed as a simple, yet efficient form of preexecution. As each predicted value (or a presumably disambiguated load instruction) enables a set of dependent instructions to be executed speculatively, these speculatively executed instructions can be viewed as a preexecution thread triggered by the prediction, though there is no explicit multithread support. Such preexecution threads are constructed dynamically for each predicted value based on the data dependence relationship from the fetched instruction stream, thus taking advantage of dynamic branch prediction. The preexecution is terminated when the normal execution catches up with the preexecution thread at the same instruction. The reason is that, when the source registers of an instruction are ready, normal execution is performed and the *vp_ready* flag is not propagated anymore. The purpose of such preexecution is to prefetch the data and the preexecution thread executes only if there are unused resources, thus avoiding resource competition with the main thread. These implicit preexecution threads are part of the main thread. Therefore, unlike other preexecution schemes using explicit multithreaded support, such preexecution does not incur any additional requirement on critical resources such as the register file or instruction window.

4.4 Performance Comparison between Recovery-Free Value Prediction and Traditional Value Speculation

In Section 4.1, the following initial observation is made: Compared to traditional value prediction schemes, recovery-free value prediction incurs some reexecution latencies in the case of correct value predictions and avoids recovery penalties in the case of value mispredictions. A detailed examination on these latencies using our proposed implementation reveals very interesting and somewhat unexpected insights.

Correct value predictions can hide memory access latencies as well as computation latencies in traditional prediction schemes. One such example is shown in Fig. 5, in

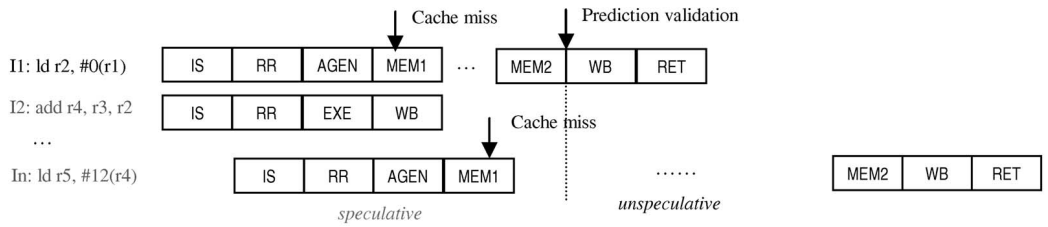

Fig. 5. Pipeline flowchart for correct value prediction (*r2* in *I1*) in traditional value speculation schemes.

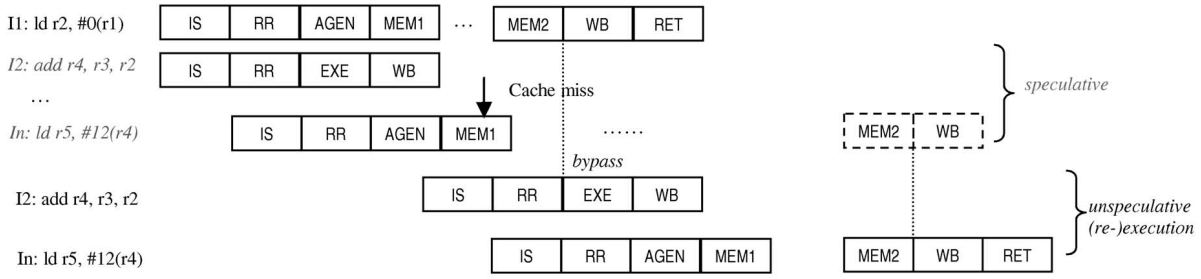

Fig. 6. Pipeline flowchart for correct value prediction (*r2* in *I1*) in recovery-free value prediction.

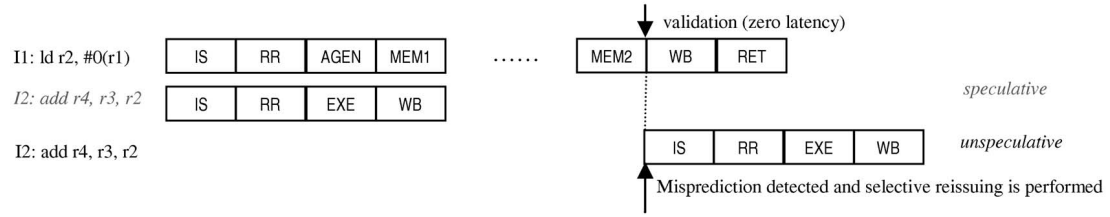

Fig. 7. Pipeline flowchart for a value misprediction (*r2* in *I1*) in traditional value speculation schemes.

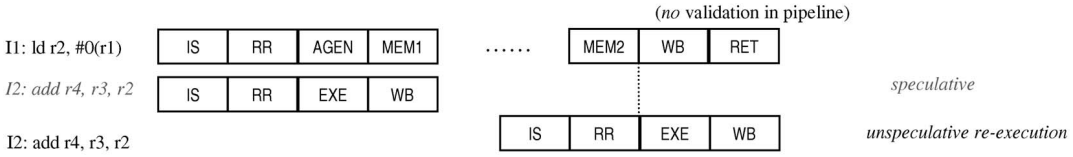

Fig. 8. Pipeline flowchart for a value misprediction (*r2* in *I1*) in recovery-free value prediction.

which the value (*r2*) of a missing load (*I1*) is predicted. With the predicted value, many dependent instructions (*I2* to *In*) can be issued and executed speculatively. Once *r2* is available (at the MEM2 stage of *I1*), the prediction is validated. As the prediction is correct, the speculative results are committed as unspeculative and the miss latency of instruction *I1* is successfully hidden when *In* is a long latency operation, such as another cache miss.

Considering the same case for recovery-free value prediction, as shown in Fig. 6, we can see that, if the prediction leads (directly or indirectly) to a long latency operation (*In*), the early service of the cache miss (*In*) during the speculative execution effectively reduces its latency in the unspeculative execution so that the unspeculative execution catches up with the speculative execution (i.e., the main thread catches up with the preexecution thread). Therefore, there is no performance penalty for reexecution in recovery-free value prediction compared to traditional value prediction. As cache misses are common for our target memory-intensive workloads, recovery-free value prediction has almost no performance loss due to reexecution compared with traditional value-prediction schemes. Our experiments show that, for both computation-intensive and memory-intensive benchmarks, up to 92 percent and 76 percent, on average, of all preexecution threads that lead to a cache miss will be caught up by the main thread. The difference between computation-intensive and memory-intensive workloads is that there are many fewer preexecution threads leading to a cache miss in computation-intensive workloads. Consequently, in the case of correct value predictions, recovery-free value prediction is less effective than traditional value prediction for these workloads.

In the case of value mispredictions, traditional value speculation schemes have to perform recovery and incur considerable performance penalties even with zero-cycle validation and recovery (reissuing) delay, as shown in Fig. 7. Due to the misprediction of the value of *r2*, the dependent instructions (*I2*) have to squash the speculative results and then to be reissued and reexecuted. Since the misprediction is detected at the end of the MEM2 stage of *I1*, the reissuing can only happen as early as the next cycle, even with zero cycle validation and recovery latency. In recovery-free value prediction, in contrast, the unspeculative execution is never delayed and the back-to-back execution is never disrupted, as shown in Fig. 8. Comparing to recovery-free

TABLE 1
Base Processor Configuration

Instruction Cache	Size = 64 kB; Associativity = 4-way; Replacement = LRU; Line size = 16 instructions (64 bytes); Miss penalty = 10 cycles.
Data Cache	Size = 32 kB; Associativity = 2-way; Replacement = LRU; Line size = 64 bytes; Miss penalty = 10 cycles; 32 MHSRs; Data cache ports: 4
Unified L2 Cache	Size = 512 kB; Associativity = 8-way; Replacement = LRU; Line size = 128 bytes; Miss penalty = 80 cycles; 64 MHSRs.
Branch Predictor	64K entry G-share; 32K entry BTB
Superscalar Core	Reorder buffer: 64 entries; Dispatch/issue/retire bandwidth: 4-way superscalar; 4 fully-symmetric function units
Execution Latencies	Address generation: 1 cycle; Memory access: 2 cycles (hit in data cache); Integer ALU ops = 1 cycle; Complex ops = MIPS R10000 latencies
Memory Disambiguation	Load stalls when there is a pending store with unresolved address; *oracle* memory disambiguation is used in Section 6.3 only.

TABLE 2
Baseline Results of the Benchmarks

Benchmarks	Computation-Intensive					Memory-Intensive				
	bzip2	gap	gcc	gzip	perl	mcf	parser	twolf	health	mst
IPC	1.68	1.31	2.11	1.46	1.46	0.51	0.85	0.83	0.32	0.21
L1 D-cache miss rate (misses per 1K insn.)	2.14% (4.88)	0.45% (0.95)	5.29% (14.08)	6.88% (16.24)	1.98% (8.61)	46.6% (166.3)	9.12% (33.04)	14.1% (45.23)	16.3% (66.08)	55.3% (175.1)
L2 Cache miss rate (misses per 1K insn.)	28.5% (1.39)	68.3% (0.65)	46.0% (6.48)	46.6% (7.57)	40.2% (3.46)	67.5% (112.3)	48.0% (15.84)	62.2% (28.12)	85.0% (56.20)	96.4% (168.8)

value prediction, there is at least a 2-cycle performance penalty for each value misprediction in traditional value speculation schemes if such a penalty cannot be overlapped with other useful execution. For deeper pipelines, e.g., a two-cycle register read stage, such a penalty will increase, which makes recovery-free prediction more preferable.

In summary, although it apparently seems that recovery-free value prediction wastes a chance to better utilize the speculative results (i.e., to commit them), the detailed discussion above shows that, for memory-intensive workloads, committing speculative results cannot further improve the performance for most cases, while incurring significant penalties for misprediction recovery.

5 Methodology

We implemented the proposed technique in a detailed timing simulator using the Simplescalar [3] toolset. The underlying processor organization is based on the MIPS R10000 processor, configured as indicated in Table 1. In our experiments, we vary the D-cache configurations, the ROB size (or the instruction window size), and the memory disambiguation model of the base configuration to evaluate our proposed technique in a range of processor models. Both computation-intensive and memory-intensive benchmarks are selected from the SPEC2000 integer benchmark suite and the Olden benchmark suite. The benchmarks *bzip2*, *gap*, *gcc*, *gzip*, and *perl* are computation intensive and the benchmarks *mcf*, *parser*, *twolf*, *health*, and *mst* are memory intensive as they exhibit much higher data cache miss rates. The reference input data are used for the SPEC2000 benchmarks. We fastforward 800M instructions and simulate the next 200M instructions. For the benchmark *health*, the input is "*max_level = 5 and max_time = 500*" and it runs into completion. For the benchmark *mst*, 3,407 nodes are used as input and the first 1,950M instructions are skipped and the next 200M instructions are simulated. The baseline performance results of these benchmarks using the base processor model are shown in Table 2 and it can be seen that the execution bandwidth of the pipeline is much underutilized for memory-intensive benchmarks, even with the aggressive memory hierarchy (i.e., four D-cache ports and sufficiently large number of MHSRs), indicating the memory subsystem is still the performance bottleneck.

As described in Section 4, a simple stride value predictor (tag-less 4K-entry) is used in our experiments to generate value predictions. The prediction table is indexed with *pc* and each entry in the table has three fields, as shown in Fig. 9. The field "*last value*" holds the most recent execution result and the field "*stride*" keeps the difference between the last two execution results. The 3-bit confidence counter is used to filter out the likely incorrect predictions. For each successful prediction, the confidence counter is increased by 2 and is decreased by 1 for each misprediction [26]. A prediction with the confidence counter larger than 4 is viewed as a confident prediction. A speculative update scheme similar to that proposed in [17] is also used to improve the prediction accuracy.

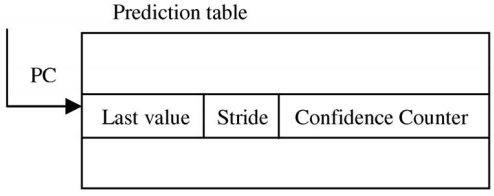

Fig. 9. A stride value prediction table.

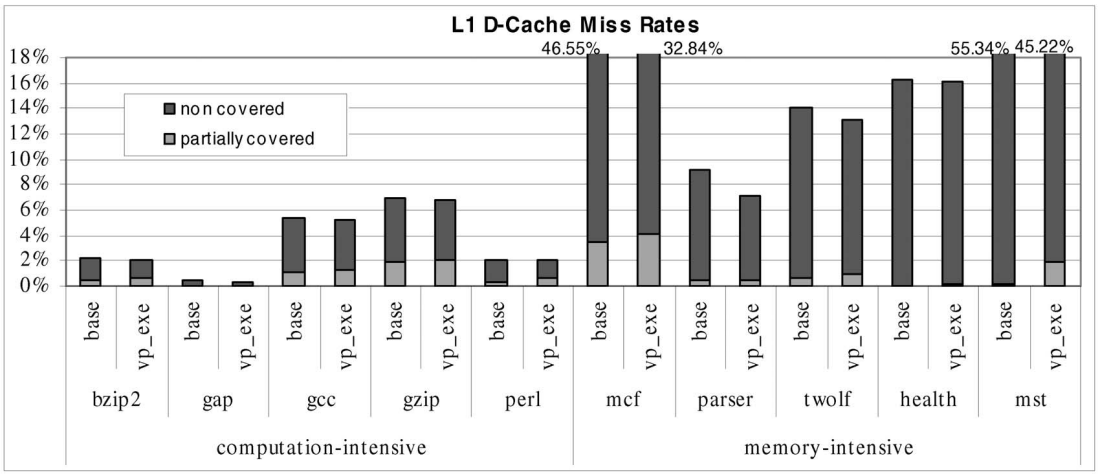

Fig. 10. The L1 D-cache miss rates.

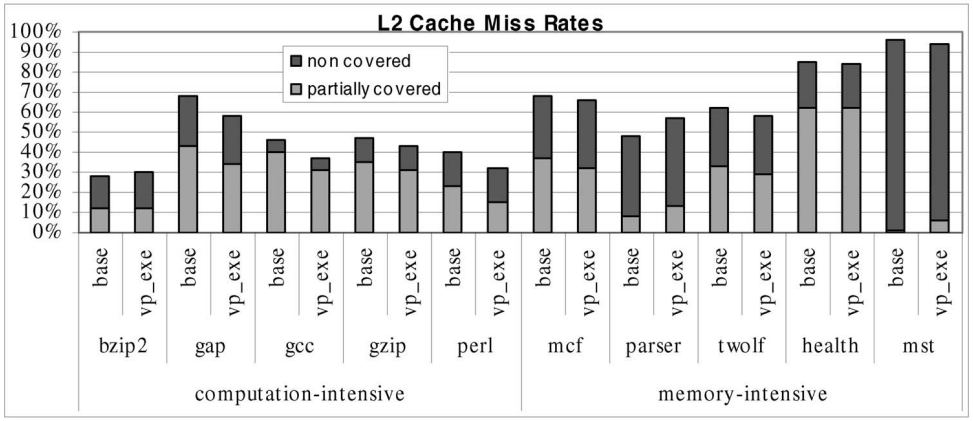

Fig. 11. The L2 cache miss-rates.

6 Experimental Results

In this section, we first evaluate the effectiveness of our proposed technique in reducing data cache miss rates, increasing MLP, and achieving performance gains. We then analyze where the performance gains come from in Section 6.2. In Section 6.3, we perform a sensitivity analysis by applying the proposed technique to a range of processor models. A limit study in Section 6.4 examines the performance potential of the proposed technique using an ideal value predictor. Section 6.5 addresses the interaction between prefetching schemes and recovery-free value prediction. Using recovery-free value prediction for early detection of branch mispredictions is discussed in Section 6.6.

6.1 Performance Evaluation

As discussed in Section 4, our proposed technique breaks both true data dependencies and alias dependencies between missing loads so that many otherwise stalled loads can be executed speculatively in parallel with prior unspeculative missing loads. These speculatively executed loads warm up the caches so that the unspeculative execution will experience fewer cache misses. We first examine the effect of this technique in reducing data cache miss rates, as shown in Fig. 10 and Fig. 11. In Fig. 10 and Fig. 11, the cache misses during speculative execution are not counted since they are used as prefetch. For each benchmark in Fig. 10, the L1 D-cache miss-rate results are reported for both the baseline processor (labeled "*base*") and the processor with recovery-free value prediction (labeled "*vp_exe*"). Also, the cache misses are further divided into partially covered misses (i.e., a miss request for a cache line that is already being repaired from the L2 cache or memory) and noncovered misses. Partially covered cache misses have less impact on overall performance compared to noncovered cache misses. Fig. 10 shows that, for memory-intensive benchmarks, the proposed technique reduces the L1 D-cache miss rate significantly, ranging from 14 percent (from 47 percent to 33 percent in the benchmark *mcf*) to 0.5 percent (from 16.5 percent to 16 percent for the benchmark *health*) and increases the ratio of partially covered misses for most benchmarks. For computation-intensive benchmarks, a slight reduction in the L1 D-cache miss rate is shown for the benchmarks *bzip2, gap*, and *gzip*, although the baseline miss rates are relatively small for these benchmarks.

Fig. 11 shows the cache miss rate effect on the L2 caches. It can be seen that the large reduction in the L1 D-cache miss rates resulting from our proposed approach does not increase the L2 cache miss rate for most benchmarks, which shows that the speculative execution not only brings the data that are already in the L2 cache into the L1 D-cache, but also reduces many L2 cache misses. For those benchmarks that exhibit increased miss rate in the L2 cache, for example, the benchmark *parser*, when considering the

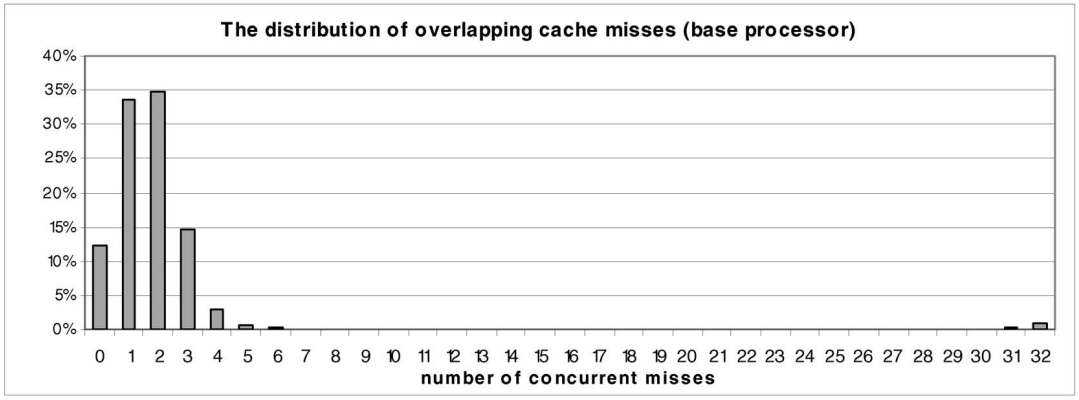

Fig. 12. The baseline MLP for the benchmark *mcf* (overall execution time = 390M cycles).

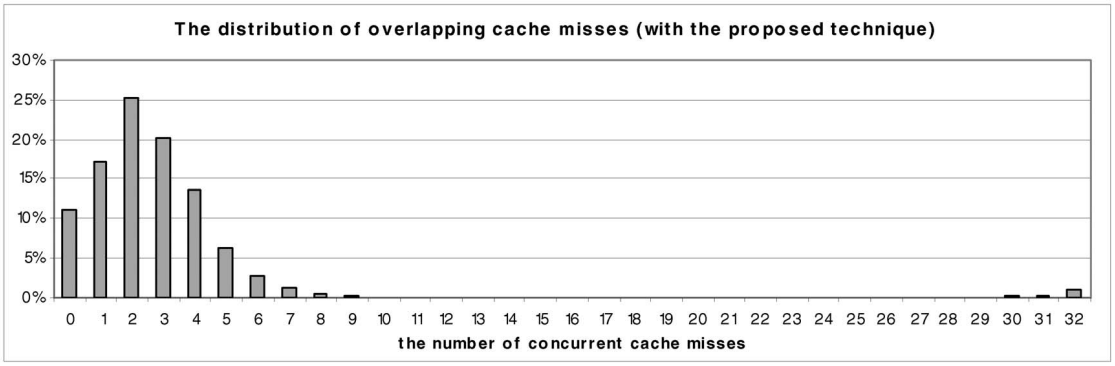

Fig. 13. Improved MLP for *mcf* using recovery-free value prediction (overall execution time = 327M cycles).

L1 miss rate reduction, we can see that the overall L2 misses are also reduced, 14.5 L2 misses per 1k instruction compared to 15.8 L2 misses originally.

Next, we use the benchmark *mcf* as an example to show the MLP improvement (i.e., the overlapping of multiple cache misses) achieved by the proposed technique for a typical heavy pointer-chasing workload. Fig. 12 shows the distribution of how many L1 D-cache misses are overlapped in the baseline processor. The x-axis of Fig. 12 is the number of the overlapping misses and the y-axis is the time during execution that the overlapping happens. From Fig. 12, we can see that the processor spends 12 percent of overall execution time on computations that do not involve a cache miss. In 33 percent of the time during the execution, a single missing load is accessing the L1 D-cache (i.e., low MLP since no overlapping happens) and, in 35 percent of the time, two missing loads are accessing the L1 D-cache. The maximum number of overlapping cache misses is determined by the MSHRs used in the cache (32 used in our experiment). It can be inferred from this distribution that the benchmark *mcf* has many sequential cache misses, resulting in low MLP and MSHR utilization, thereby resulting in long execution time.

With recovery-free value prediction, the overall execution time is significantly reduced and MLP is much improved, as shown in Fig. 13. Compared to Fig. 12, a significant amount of sequential cache misses are now processed in parallel. One important consequence of MLP exploration is the increased pressure on memory hierarchy bandwidth. This experiment shows that such pressure on MSHRs is not overwhelming as sequential cache misses are rarely converted into more than six concurrent cache-misses by recovery-free value prediction and the number of concurrently employed MSHRs is less than eight most of the time.

Fig. 14 shows the speedups of the proposed recovery-free value prediction and it can be seen that our proposed technique achieves significant speedups for memory-intensive benchmarks, up to 24 percent for *mst* and 11 percent on average (average speedup is computed based on the harmonic means of the IPCs, labeled "*H_mean*"). For the well-known pointer-chasing benchmark, *mcf*, the speedup is 19.6 percent. Considering the low hardware overhead required by this technique, the performance gains are impressive. Moreover, much higher speedups can be achieved with better prediction accuracy and larger instruction windows, as discussed in Sections 6.3 and 6.4. For computation-intensive benchmarks, smaller speedups (average of 0.5 percent) are observed, which is expected since the reduction in the D-cache miss rate for these benchmarks is small. The only benchmark that shows a negative speedup (-0.7 percent) is *gcc*, which will be discussed further in Section 6.3 and Section 7.

6.2 Performance Analysis

To analyze why the proposed technique achieves significant speedups, we first examine the stride value predictor to see how well it predicts a value and how often a missing load is correctly predicted. It is observed in previous studies [9], [18], [24] that many instructions exhibit stride locality and a more recent work [28] also showed that stride locality exists in the address stream for many load instructions in irregular programs. As pointed out in Section 3, the predictability of load addresses is equivalent to the load value predictability for pointer-chasing codes. Our results, shown in Fig. 15,

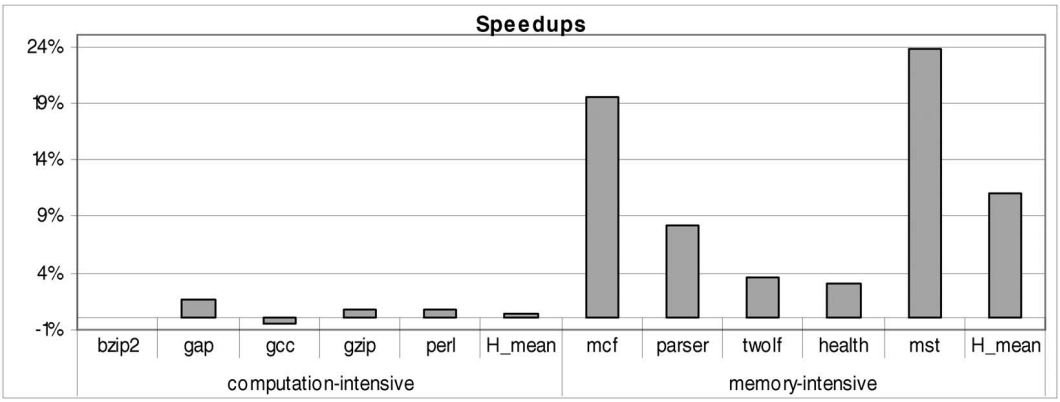

Fig. 14. The speedups of using recovery-free value prediction (H_mean stands for harmonic mean).

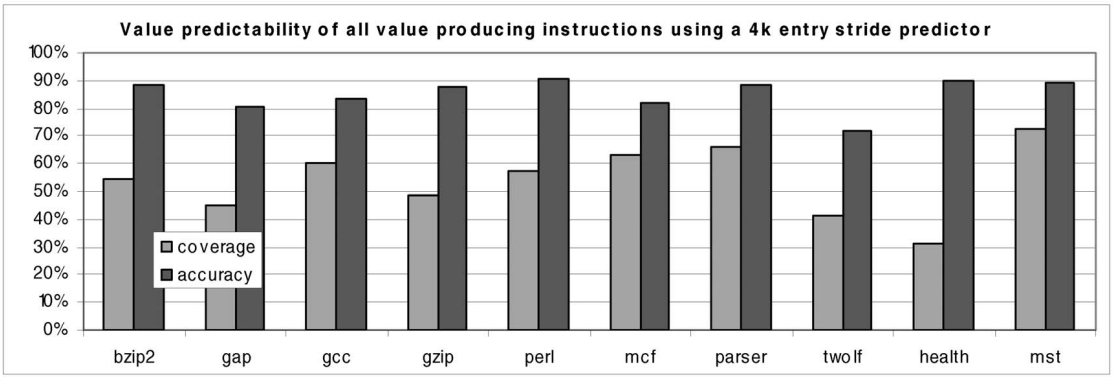

Fig. 15. Value predictability for all value producing instructions using a 4k-entry stride predictor.

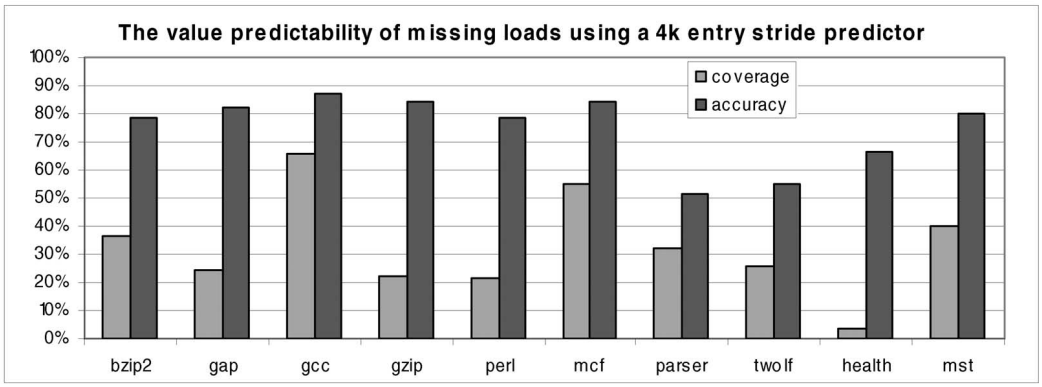

Fig. 16. Value predictability for missing loads using a 4k-entry stride predictor.

confirm these observations. For each benchmark, both the value prediction coverage (i.e., the ratio of confident predictions over all predictions) and the value prediction accuracy (i.e., the ratio of the correct predictions over confident predictions) are shown in Fig. 15 for all value-producing instructions using a 4k-entry stride value predictor. It can be seen that most benchmarks, especially the benchmarks *mcf*, *parser*, and *mst*, exhibit significant stride type of value locality and this small value predictor achieves decent prediction coverage and accuracy.

Since value predictions are used to break memory dependence chains, the predictability of the missing loads is of special interests and is examined in Fig. 16. From Fig. 16, it can be seen that the value of missing loads exhibits different degrees of stride locality for different benchmarks. For the heavy pointer-chasing benchmarks *mcf* and *mst*, the value predictor achieves good prediction coverage and high accuracy. Given their high cache miss rate and pointer-chasing characteristics, this explains why these benchmarks enjoy significant speedups. For another pointer-chasing benchmark, *health*, the missing loads show very limited stride type of locality. As we will see next, the speedup for this benchmark is mainly from speculative memory disambiguation instead of breaking true memory dependencies. Again, if a more powerful predictor (e.g., context-based) is used to explore the locality in its address stream, higher speedup can be expected for this particular benchmark (see Section 6.4).

As discussed in Section 3, both true data dependence and alias dependence between missing loads prevent these loads from being executed in parallel. Recovery-free value prediction breaks both dependencies during the speculative

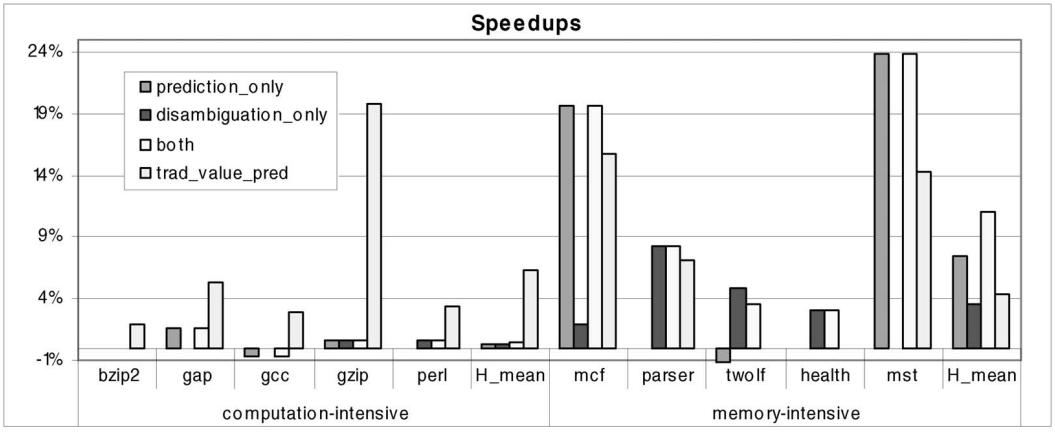

Fig. 17. The speedups resulting from breaking different dependencies and traditional value speculation.

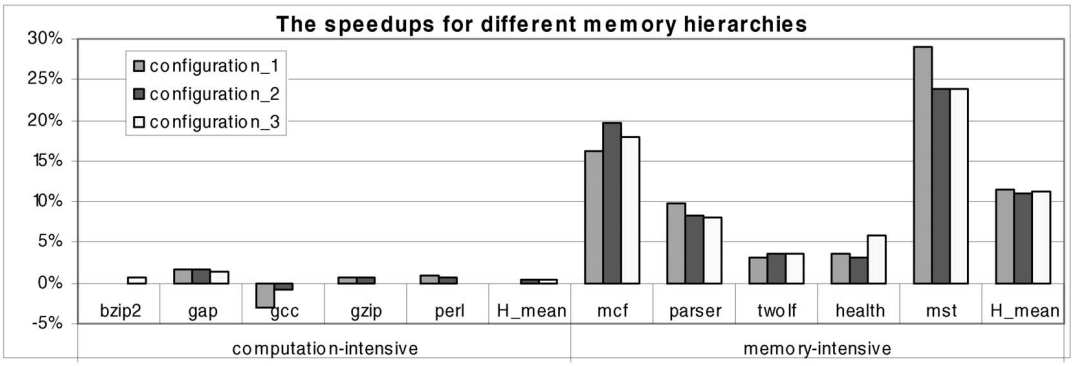

Fig. 18. The speedups of recovery-free value prediction for different memory hierarchies.

execution. Next, we examine the impact of breaking either of these two dependencies in enhancing MLP. In the next experiment, we isolate the performance impact by breaking only one type of dependency at a time. Fig. 17 shows the speedup results for breaking true data dependency only (labeled "*prediction_only*"), breaking alias dependence only (labeled "*disambiguation_only*"), and breaking both dependencies (i.e., the same results as in Fig. 14, labeled "*both*"). We also include the speedup results using traditional value prediction (labeled "*trad_value_pred*") in Fig. 17. In the traditional value prediction scheme, the same stride value predictor is used and an idealistic validation and selective reissuing (1-cycle latency) mechanism is incorporated into the execution pipeline. From Fig. 17, it can be seen that, for computation-intensive benchmarks, aggressive memory disambiguation has slightly better speedups than performing value prediction only. For memory-intensive benchmarks, breaking true dependencies results in much higher speedups for *mcf* and *mst*, but fewer speedups for other benchmarks compared to breaking alias dependencies. The reason is that, for these benchmarks, many critical memory dependencies are due to alias dependencies. For these benchmarks, increasing the instruction window size and performing speculative memory disambiguation can improve MLP effectively. Also, our value predictor only exploits the stride locality, limiting the opportunity to break true memory dependence more aggressively. The benchmarks *mcf* and *mst*, on the other hand, feature heavy pointer chasing and exhibit strong stride locality in their value streams. So, breaking true dependencies becomes more profitable. Fortunately, when both true dependencies and alias dependencies are broken at the same time using our proposed approach, higher speedups are achieved. This mutually beneficial effect confirms our observation in Section 3 that both memory dependencies need to be broken to improve MLP and similar results are also reported in a study [4] of the interaction between value prediction and memory dependence speculation.

Comparing our proposed recovery-free scheme with traditional value prediction, we can see that traditional value prediction achieves higher speedups for computation-intensive benchmarks. For memory-intensive workloads, recovery-free prediction has much higher speedups as it avoids misprediction penalties and benefits from speculative disambiguation, as discussed in Section 4.4.

6.3 Sensitivity Analysis

In this experiment, we evaluate the proposed technique in different memory hierarchies, 16kB direct-mapped L1 D-cache and 256kB 4-way L2 cache (labeled "*configuration 1*"), 32kB 2-way L1 D-cache and 512kB 8-way L2 cache (the same as in our baseline processor, labeled "*configuration 2*"), and 64kB 4-way L1 D-cache and 2,048kB 8-way L2 cache (labeled "*configuration 3*"). The speedups of the proposed technique in these configurations are show in Fig. 18.

Interesting observations can be made from Fig. 18. First, for the small D-cache of 16kB, the memory problem becomes more evident. As a result, more speedups are achieved by hiding miss latencies using recovery-free value prediction, as we can see from the benchmarks, *mst* and *parser*. On the other hand, however, a small cache can tolerate less cache pollution resulting from value mispredictions. So, the miss rate can

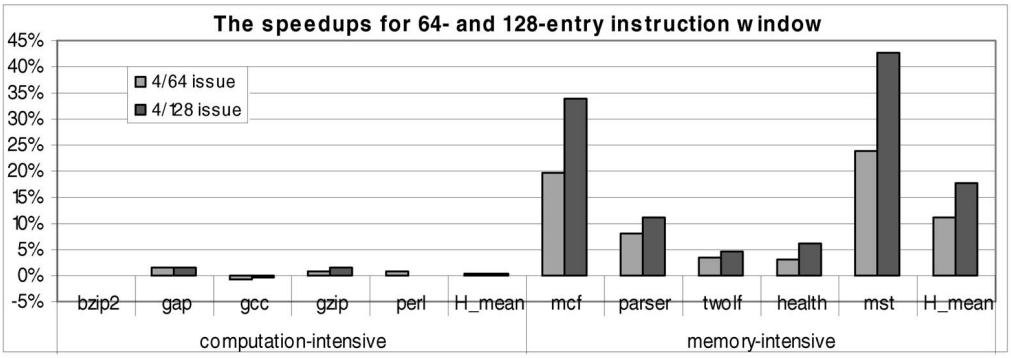

Fig. 19. The speedups of recovery-free value prediction for different instruction window sizes.

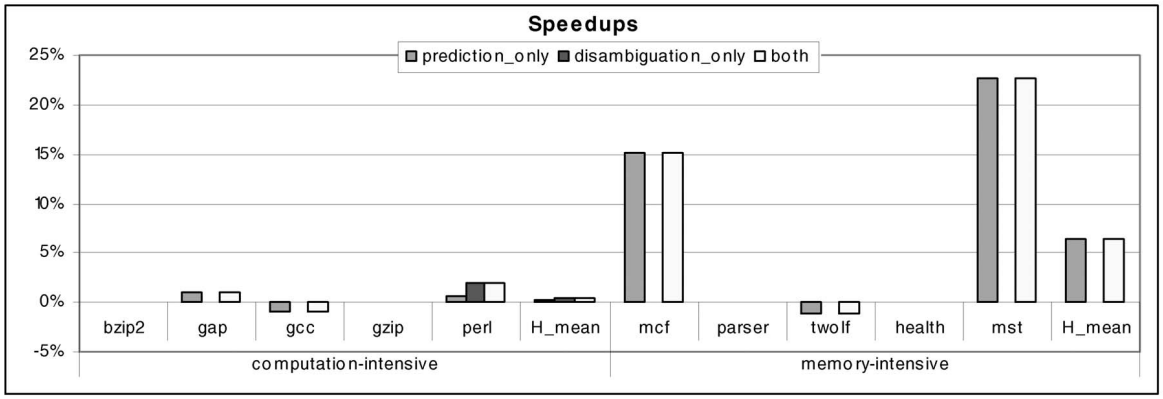

Fig. 20. The speedups resulting from breaking different dependencies with *oracle* memory disambiguation.

actually increase if the value misprediction rate is high and the speedups are reduced, as in the benchmarks *gcc* and *twolf*. Large caches, such as 64kB, are more tolerant on the cache pollution problem, while the criticality of memory operations is reduced if they hit in the caches more often.

In the next experiment, we increase the instruction window size to 128 to allow it to be more tolerant to L1 D-cache misses. The same 32kB 2-way L1 D-cache and 512kB 8-way L2 cache are used as in the baseline 4/64 issue model. The results are shown in Fig. 19. From this experiment, we can see that much higher speedups are reported for the 128-entry instruction window in all memory-intensive benchmarks using our proposed recovery-free value prediction. There are two major reasons accounting for this trend. First, a large instruction window size of 128 holds a longer memory dependence chain. As discussed in Section 3, breaking a longer chain can overlap more cache misses, resulting in higher performance improvement. Second, a larger instruction window enables more instructions to be fetched into the window under a long-latency cache miss, thereby enabling those instructions to be predicted sooner than in a small instruction window. As a result, speculative loads (or prefetches) can be issued earlier to hide more memory access latencies.

Oracle memory disambiguation removes aliasing dependencies completely. Consequently, only a few loads stall due to earlier stores accessing the same address. Issuing these loads speculatively (i.e., recovery-free memory disambiguation) only helps when the memory content is not changed by those stores (i.e., stores writing the same values) and such a benefit is quite limited due to the store-load forwarding mechanism. On the other hand, the side effect of such speculative execution with stale data is also minimal since the number of such stalled loads is too small to make an impact on performance. This observation is confirmed by Fig. 20, which shows the speedups of recovery-free disambiguation and recovery-free value prediction over the baseline model with oracle memory disambiguation. All benchmarks have no performance changes except the benchmark *perl* (a 2 percent speedup) using recovery-free disambiguation. Comparing to the results in Fig. 17, one interesting point is that the proposed recovery-free disambiguation has the capability of adapting to the employed memory disambiguation scheme: less speculation for more aggressive memory disambiguation and more speculation for less aggressive disambiguation. As a result, less performance improvement is achieved for more aggressive memory disambiguation and vice versa. Another observation made from Fig. 20 is that, although aliasing dependencies have been eliminated by oracle memory disambiguation, there are still significant speedups (up to 22 percent speedup and 6.5 percent on average) achieved from breaking true dependencies among load instructions for memory-intensive workloads.

6.4 A Limit Study: The Performance Potential of Recovery-Free Value Prediction

In our experiments in Section 6, the performance improvements are achieved using a stride value predictor (4k entry, tagless) and Fig. 15 and Fig. 16 indicate that such a simple value predictor has limited capability to predict values correctly, especially the values of missing loads, for certain benchmarks. In the next experiment, a limit study is performed to illustrate the performance potential of

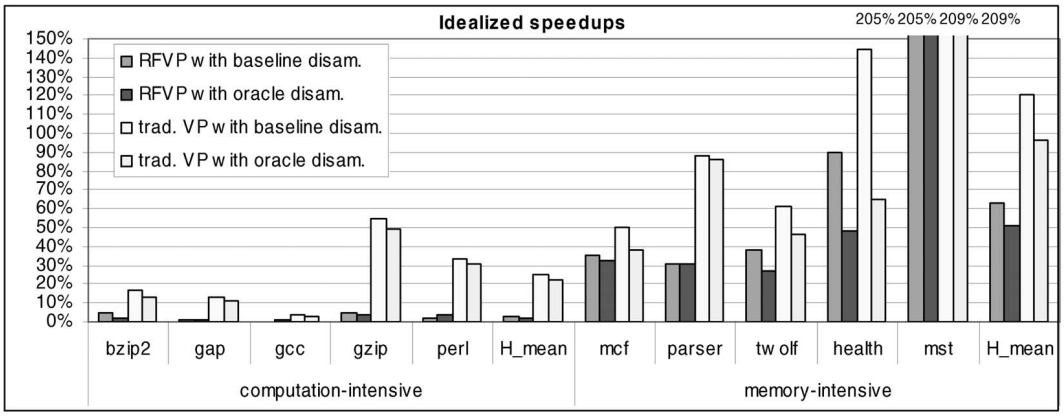

Fig. 21. Idealized speedups of recovery-free value prediction with baseline and oracle memory disambiguation.

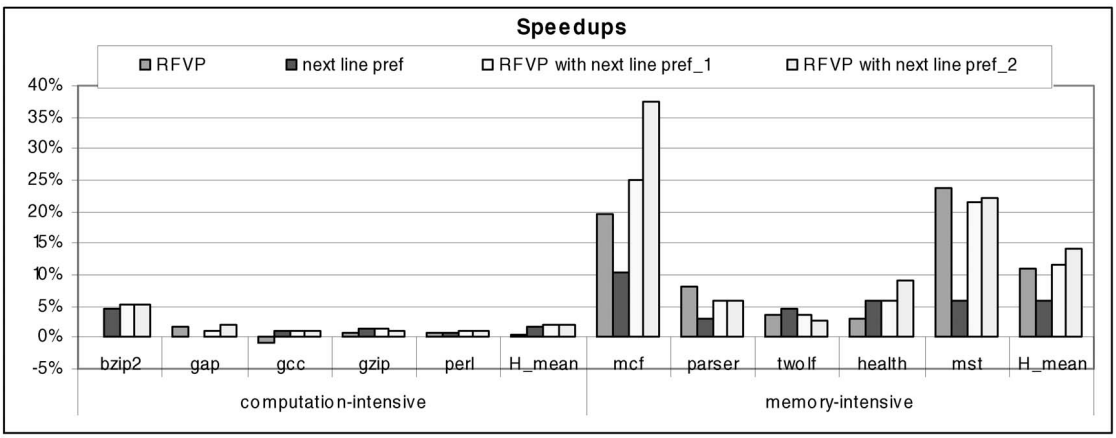

Fig. 22. The speedups of next-line prefetch and different combinations with recovery-free value prediction.

recovery-free value prediction assuming an ideal value predictor. Such idealized speedups are shown in Fig. 21 for both recovery-free value prediction and traditional value prediction with the baseline and oracle memory disambiguation models.

From Fig. 21, it can be seen that, for computation-intensive benchmarks, the performance improvement of the recovery-free scheme is quite limited, even with ideal value prediction, as the memory access is not their performance bottleneck. For our target workloads, memory-intensive benchmarks, the performance potential of recovery-free value prediction is highly impressive (up to 205 percent and 62 percent on average for baseline disambiguation; up to 205 percent and 50 percent on average for oracle disambiguation) and it is evident that a more powerful value predictor than a simple stride value predictor would achieve higher performance improvement. Also, it should be noted that the idealized speedups in this experiment are based on a 64-entry instruction window and a larger window would have even higher idealized speedups, as discussed in Section 6.3. Traditional value prediction, on the other hand, benefits more from ideal predictions as misprediction recovery penalties are eliminated completely.

6.5 The Interaction between Recovery-Free Value Prediction and Next-Line Prefetching

In the baseline processor model configured as Table 1, no prefetching mechanism has been included. In this experiment, the following next-line prefetching scheme is implemented on all data caches (L1-D and L2 caches): For every cache miss, a prefetch for the next cache line is invoked. Then, considering whether such a prefetch is allowed to be initiated for a miss during speculative execution, there are two design choices to combine recovery-free value prediction and next-line prefetch. Fig. 22 shows the speedups of the next-line prefetch (labeled "*next line pref*"), original recovery-free value prediction (same results as in Fig. 14, labeled "*RFVP*"), next-line prefetch for unspeculative cache misses but not for speculative misses (labeled "*RFVP with next line pref_1*"), and next-line prefetch for both speculative and unspeculative cache misses (labeled "*RFVP with next line pref_2*"). From Fig. 22, it can be seen that, although next-line prefetching improves performance for all benchmarks, mixed interactions with recovery-free value prediction can be observed. For the benchmark *mcf*, both recovery-free value prediction and next-line prefetch enhance the memory performance. Combining both schemes further enhances the performance, especially when the prefetches can be initiated for speculative misses. For the benchmark *twolf*, on the other hand, such combination leads to inferior performance improvement. The reason is that the misses that are encountered in speculative execution either pollute the caches (due to value misprediction) or have low spatial locality. On average, the best performance improvement (14.1 percent) is achieved by initiating prefetches for both speculative and unspeculative cache misses (i.e., the

positive interaction between recovery-free value prediction and next-line prefetch).

6.6 Early Branch Misprediction Detection Using Recovery-Free Value Prediction

Using value prediction to help branch prediction has been proposed in [11], [12]. A slight hardware modification is needed to enable recovery-free value prediction for early detection of branch mispredictions. In the proposed implementation scheme described in Section 4.2, both store and branch instructions are not allowed to be issued speculatively based on value predictions. We can simply remove such a restriction on branches so that the speculative execution of branches can validate branch predictions *early* and *speculatively*. The unspeculative reexecution of the branch afterward using unspeculative values will then guarantee the correctness of the program.

In this naive application of recovery-free value prediction, we found that 12 to 38 percent of branch mispredictions can be detected early during speculative execution. However, due to value mispredictions, the speculative execution also introduces incorrect branch resolutions (20 to 150 percent) for some correctly predicted branches. As a result, performance degradation is observed from speculatively executing branches, mostly due to the limited value prediction capability of the simple stride value predictor and the associated confidence mechanism. From this experiment, we conclude that, although there is significant performance potential to use recovery-free value prediction for control speculation, a simple naive application is not enough and a more careful study in this direction can be highly rewarding and is part of our future work.

7 Reducing Overheads of Recovery-Free Value Prediction

The performance improvement from recovery-free value prediction is achieved from speculation based on predicted values. To support such speculation, additional hardware is required, as described in Section 4.2. Moreover, the speculatively executed instructions will incur additional power consumption. In this section, we propose two simple techniques to reduce the number of speculatively executed instruction while not affecting or even improving the performance enhancement achieved from recovery-free value prediction.

As discussed in Section 3, memory-level parallelism is enhanced by breaking dependencies among missing loads. In the proposed implementation, all value producing instructions are predicted using a 4k-entry, tagless stride value predictor. As our focus is on breaking pointer chasing, i.e., breaking load-to-load flow dependencies, one evident optimization is to predict loads only instead of predicting all value producing instructions. In this way, the value prediction table needs to be tagged (or partially tagged), but the number of entries can be greatly reduced. In one experiment, a 1k-entry stride predictor is used and there is no observable performance variation from those reported in Fig. 14.

Another feature of pointer-chasing code is that the value of one load is used as an address for another load. We can thereby filter out those value predictions that are not likely

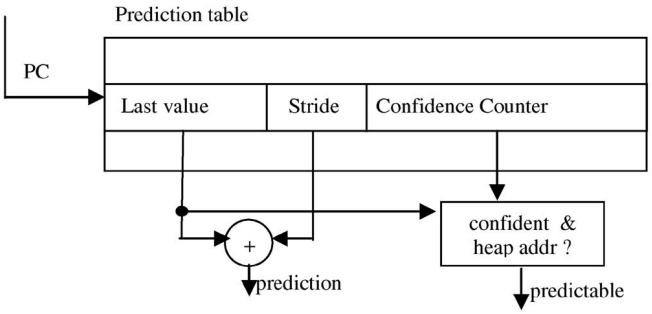

Fig. 23. A stride value predictor with value filtering.

to be an address for another load. Moreover, since the objective of recovery-free value prediction is to overlap multiple cache misses, we can further filter those predictions that fall into the stack range as most stack accesses usually hit in the cache. As a result, a prediction is only made when it is either a heap address or a global data segment address. Fig. 23 shows the simple modification in a stride value predictor to make such a check and it is unlikely that such simple value filtering will affect the access time of the value predictor. In addition to fewer value predictions, such a filtered value predictor also has beneficial performance impact on the benchmarks such as *gcc* since many value mispredictions are filtered out and the cache pollution effect is highly reduced. As a result, the original performance degradation for *gcc* using recovery-free value prediction is eliminated and the performance improvement for other benchmarks is not changed.

The two techniques discussed above reduce the number of speculatively executed instructions significantly, as shown in Fig. 24, which measures the ratio of the number of the speculatively issued instructions over the unspeculatively executed instructions. From Fig. 24, it can be seen that, in the initial implementation of recovery-free value prediction, an average of 38 percent and 11 percent of dynamic instructions are issued speculatively and then reexecuted unspeculatively for memory-intensive and computation-intensive benchmarks, respectively. If only the load instructions are predicted, the overheads drop to 18 percent and 4 percent. When focusing only the heap and global data segment addresses, the average resulting overheads are 10.6 percent and 2.6 percent. All of these overhead reductions are achieved with none or positive (*gcc*) performance improvement over the initial implementation.

As a final note, the 10.6 percent overhead of speculative issued instructions for memory-intensive workloads (or 2.6 percent overhead for computation-intensive) does not translate to an additional 10.6 percent power consumption. As the speculatively issued instructions remain in issue queue, the reexecution only involves issue, execution and write back activities, but not fetch, decode, renaming, or retirement. Moreover, considering the reduction in execution time shown in Fig. 14, recovery-free value prediction could be a more energy efficient design than the baseline processor.

8 Limitations

Two limitations exist with our proposed scheme. First, as we pointed out in Section 3, value prediction can

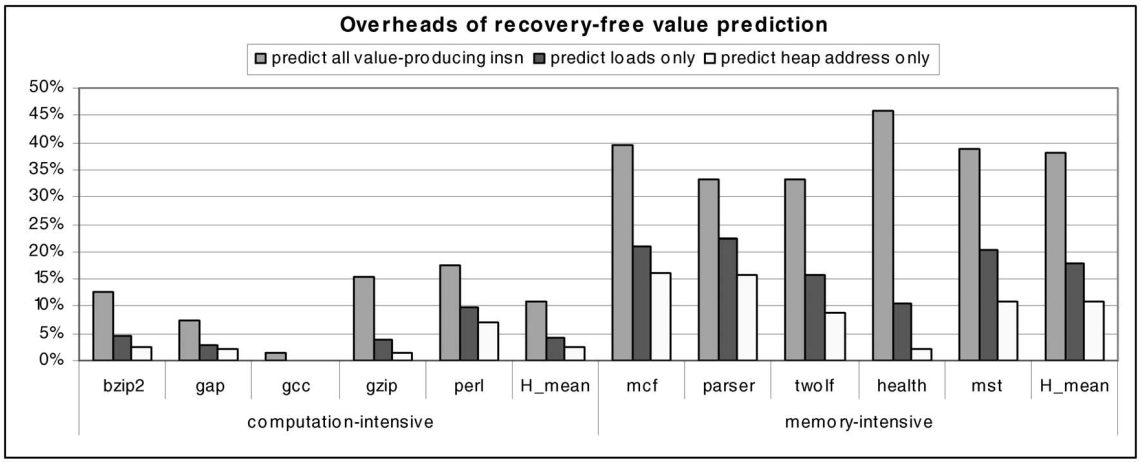

Fig. 24. The overheads of recovery-free value prediction measured by the ratio of speculatively issued instructions over the unspeculatively executed instructions.

hide memory access latencies by breaking memory dependencies, especially for long memory dependence chains. As a result, it is effective for memory-intensive workloads with heavy pointer chasing. If a workload does not exhibit such memory dependencies, for example, the cache misses due to randomly accessing large data arrays as featured in some floating-point benchmarks, our proposed scheme will have very limited capabilities to hide those cache-miss penalties.

Second, in our proposed recovery-free value prediction scheme, a prediction is made only after one instruction is fetched and the prediction is consumed only when the dependent instructions are in the instruction window. This implies that the earliest time for a speculative load to be executed is after the load instruction is dispatched into the issue queue. It limits the capability of exploring the far-flung MLP even if the correct prediction can be made. Experiments in Section 6.3 show the performance impacts of using a large instruction window to bring in instructions early into the instruction window. Another interesting way to explore the distant MLP is to combine with the run-ahead execution [8], [21] to preexecute/prefetch both independent and dependent memory accesses.

9 Conclusions

In this paper, we advocate using value prediction to enhance MLP for memory-intensive benchmarks with heavy pointer chasing. As current microprocessors can execute instructions very fast as long as long memory latency operations, such as cache misses, are not involved, we propose to use value prediction only for data prefetching so that complex prediction validation and misprediction recovery mechanisms are avoided and only minor hardware changes are necessary. Also, the same hardware changes enable aggressive recovery-free memory disambiguation for prefetching.

We present our design of recovery-free value prediction based on a MIPS R10000 processor model and the simulation results show that our technique enhances MLP effectively for a range of benchmarks and achieves significant speedups. Two simple techniques, predicting loads only and prediction filtering, are then proposed to reduce the associated runtime overhead.

As pointed out in [1], only a few static load instructions are responsible for the majority of dynamic cache misses. So, it would be very interesting to tune the value predictor to predict only the values leading to the address computation of these load instructions. This would further reduce the hardware overhead and the power consumption overhead due to useless speculation.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments. This work was supported by US National Science Foundation awards CCR-0208596 and CCR-0072926 and a hardware donation from Hewlett-Packard. This paper is an expanded version of the paper that appeared in the *Proceedings of the 2003 International Conference on Supercomputing*, pages 326-355, June 2003.

References

[1] S.G. Abraham, R.A. Sugumar, D. Windheiser, B.R. Rau, and R. Gupta, "Predictability of Load/Store Latencies," *Proc. 26th Int'l Symp. Microarchitecture (MICRO-26),* 1993.

[2] M. Bekerman, S. Jourdan, R. Ronen, G Kirshenboim, L. Pappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors," *Proc. 26th Int'l Symp. Computer Architecture (ISCA-26),* 1999.

[3] D. Burger and T. Austin, "The SimpleScalar Tool Set, v2.0," *Computer Architecture News,* vol. 25 June 1997.

[4] B. Calder and G. Reinman, "A Comparative Survey of Load Speculation Architectures," *J. Instruction-Level Parallelism,* 2000.

[5] M. Carlisle, "Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines," PhD thesis, Computer Science Dept., Princeton Univ., 1996.

[6] J.D. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J.P. Shen, "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads," *Proc. 28th Int'l Symp. Computer Architecture (ISCA-28),* 2001.

[7] R. Cooksey, S. Jourdan, and D. Grunwald, "A Stateless, Content-Directed Data Prefetching Mechanism," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-X),* 2002.

[8] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions under a Cache Miss," *Proc. 1997 Int'l Conf. Supercomputing,* 1997.

[9] F. Gabbay and A. Mendelson, "Speculative Execution Based on Value Prediction," Technical Report 1080, Electrical Eng. Dept., Technion-Israel Inst. of Technology, Nov. 1996.
[10] J. Gonzalez and A. Gonzalez, "Speculative Execution via Address Prediction and Data Prefetching," *Proc. 1997 Int'l Conf. Supercomputing,* 1997.
[11] J. Gonzalez and A. Gonzalez, "Control-Flow Speculation through Value Prediction for Superscalar Processors," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* 1999.
[12] T Heil, Z. Smith, and J.E. Smith, "Improving Branch Predictors by Correlating on Data Values," *Proc. 32nd Int'l Symp. Microarchitecture (MICRO-32),* 1999.
[13] J. Henning, "SPEC2000: Measuring CPU Performance in the New Millennium," *Computer,* July 2000.
[14] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *IEEE Trans. Computers,* vol. 48, no. 2, Feb. 1999.
[15] T. Karkhanis and J. Smith, "A Day in the Life of a Cache Miss," *Proc. Second Ann. Workshop Memory Performance Issues (WMPI 2002),* 2002.
[16] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," *Proc. 29th Int'l Symp. Computer Architecture (ISCA-29),* 2002.
[17] S. Lee and P. Yew, "On Some Implementation Issues for Value Prediction on Wide ILP Processors," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '00),* 2000.
[18] M.H. Lipasti and J.P. Shen, "Exceeding the Dataflow Limit via Value Prediction," *Proc. 29th Int'l Symp. Microarchitecture (MICRO-29),* 1996.
[19] M.H. Lipasti, C.B. Wikerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. Seventh Int'l Conf. Architectural Support for Programming Language and Operation Systems (ASPLOS-7),* Oct. 1996,
[20] C.K. Luk, "Tolerating Memory Latency through Soft-Ware-Controlled Preexecution in Simultaneous Multithreading Processors," *Proc. 28th Int'l Symp. Computer Architecture (ISCA-28),* 2001.
[21] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," *Proc. Ninth Int'l Symp. High Performance Computer Architecture (HPCA-9),* 2003.
[22] E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Int'l Symp. Microarchitecture (MICRO-29),* 1996.
[23] A. Roth and G. Sohi, "Speculative Data Driven Multithreading," *Proc. Seventh Int'l Symp. High Performance Computer Architecture (HPCA-7),* 2001.
[24] Y. Sazeides and J.E. Smith, "The Predictability of Data Values," *Proc. 30th Int'l Symp. Microarchitecture (MICRO-30),* Nov. 1997.
[25] E. Sprangle and D. Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines," *Proc. 29th Int'l Symp. Computer Architecture (ISCA-29),* 2002.
[26] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction Using Hybrid Predictors," *Proc. 30th Int'l Symp. Microarchitecture (MICRO-30),* Nov. 1997.
[27] P.H. Wang, H. Wang, J.D. Collins, E. Grochowski, R.M. Kling, and J.P. Shen, "Memory Latency-Tolerance Approaches for Itanium Processors: Out-of-Order Execution vs. Speculative Precomputation," *Proc. Eighth Int'l Symp. High Performance Computer Architecture (HPCA-8),* 2002.
[28] Y. Wu, "Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching," *Proc. ACM 2002 Conf. Programming Language Design and Implementation (PLDI-2002),* 2002.
[29] K.C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro,* 1996.
[30] H. Zhou and T. Conte, "Enhance Memory-Level Parallelism via Recovery-Free Value Prediction," *Proc. 2003 Int'l Conf. Supercomputing (ICS-03),* 2003.
[31] H. Zhou, J. Bodine, and T. Conte, "Detecting Global Stride Localities in Value Streams," *Proc. 30th Int'l Symp. Computer Architecture (ISCA-30),* 2003.
[32] H. Zhou and T. Conte, "Performance Modeling of Memory Latency Hiding Techniques," Technical Report, Electrical and Computer Eng. Dept., North Carolina State Univ., Dec. 2002.
[33] C. Zilles and G. Sohi, "Execution-Based Prediction Using Speculative Slices," *Proc. 28th Int'l Symp. Computer Architecture (ISCA-28),* 2001.

Huiyang Zhou (S'98-M'03) received the Bachelor of Electrical Engineering degree from Xian Jiaotong University, People's Republic of China, in 1992 and the PhD degree in computer engineering from North Carolina State University in 2003. He is currently an assistant professor in the School of Computer Science at the University of Central Florida. His research focuses on high-performance microarchitecture, low-power design, and backend compiler optimization. He is a member of the IEEE.

Thomas M. Conte (S'84-M'92-SM'99-F'05) received the Bachelor of Electrical Engineering degree from the University of Delaware in 1986. He went on to receive the MS and PhD degrees in electrical engineering from the University of Illinois at Urbana-Champaign in 1988 and 1992, respectively. He is currently a professor of electrical and computer engineering and director of the Center for Embedded Systems Research at North Carolina State University. While on leave from NC State in 2000-2001, he served as the chief microarchitect and manager of back end compiler development for DSP vendor BOPS, Inc. He is the editor-in-chief of the *Journal of Instruction-Level Parallelism* and an associate editor of both the *ACM Transactions on Architecture and Compiler Optimization* and the *ACM Transactions on Embedded Computer Systems*. He is also the chair of the IEEE CS Technical Committee on Microprogramming and Microarchitecture (TC-uARCH). HE currently directs several PhD students on the TINKER project in topics spanning advanced microarchitecture, compiler optimization, and performance analysis. His research is or has been supported by Compaq, the US Defense Advanced Research Projects Agency, Hewlett-Packard, IBM, Intel, Sun Microsystems, Texas Instruments, and the US National Science Foundation (NSF). He is the recipient of an NSF CAREER award and the IBM T.J. Watson Partnership Award for Faculty Development. He is a fellow of the IEEE.