Enhancing Memory Level Parallelism via Recovery-Free Value Prediction

Huiyang Zhou Thomas M. Conte
Department of Electrical and Computer Engineering
North Carolina State University
1-919-513-2014
{hzhou, conte}@eos.ncsu.edu

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 the 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 to use value prediction and value speculative execution only for prefetching so that the complex prediction validation and misprediction recovery mechanisms are avoided and only minor changes in the microarchitecture are needed. The same hardware modifications 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.

Categories and Subject Descriptors
C.1.1 [**Processor Architecture**]: Single Data Stream Architectures.

General Terms
Performance, Design.

Keywords
Recovery-free value prediction, memory level parallelism, prefetching, memory disambiguation.

1. INTRODUCTION
The trend in contemporary microprocessor design, including fast clock speed, deep pipeline [23], large instruction window size [13],[14], aggressive out-of-order execution, and wide fetch bandwidth [20], 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],[16],[17], 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 the sequential execution, value prediction has the capability to parallelize these loads, thereby increasing the 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 to use value prediction only for prefetching so that the complex value prediction validation and misprediction recovery mechanisms are avoided and only minor changes in the hardware are necessary. Unlike the traditional value prediction schemes, where the speculative results are committed when the 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 pre-execution scheme, which requires neither explicit pre-execution thread generation nor multi-threading support. Another important aspect is that the same hardware changes in our scheme also enable aggressive memory disambiguation to break alias (i.e., the 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 [11] 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 trend in microprocessor design.

The remainder of the paper is organized as follows. Section 2 addresses the related work. Section 3 illustrates the performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'03, June 23-26, 2003, San Francisco, California, USA.

Copyright 2003 ACM 1-58113-733-8/03/0006…$5.00.

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. Section 7 discusses the limitations of our proposed scheme. Section 8 concludes the paper and discusses the future work.

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 the address prediction [2],[12]. 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 be a pointer de-reference 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 load addresses based on the code semantics, thereby having fewer chances to fetch the wrong data to pollute the cache.

Another way to hide memory latency is based on the concept of pre-execution/pre-computation. Both hardware-based and software-based schemes [6], [18], [21], [25], [30] have been proposed for this purpose. As will be discussed in Section 4, our recovery-free value prediction scheme is similar to the pre-execution paradigm although our approach requires neither explicit thread generation nor multi-threaded support. Also, as pointed out in [25], the pre-computation thread is more effective when used to prefetch the critical pointer chasing loads in the loop control than to prefetch loads in the 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 more latency hiding. Runahead execution [8],[19] is another form of pre-execution 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 pre-execute the *independent* instructions following the blocking instruction. The purpose of the pre-execution is to prefetch the (future) data into cache. When the cache miss is repaired, the processor goes back to the normal mode and re-executes these pre-executed 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 pre-execute the *dependent* operations of a blocking instruction. Also, as discussed in Section 3, the large instruction window achieved by runahead execution provides better chances for our scheme to improve MLP.

Value prediction was proposed originally as an ILP optimization technique [9],[16],[17],[22]. 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 and the associated recovery penalties. Also, our approach 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 MLP

Values produced by individual instructions exhibit localities [22] and different value prediction schemes are proposed to exploit such localities to break true data dependencies [9],[16],[17]. 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 value speculative computation in the un-speculative computations. If the prediction is wrong, however, a recovery scheme is necessary to squash the speculative results and to re-execute these affected instructions with correct data.

For memory intensive workloads with heavy pointer chasing, sequential cache-misses resulting from pointer chasing code structures dominate the overall execution time. These cache-misses form a memory dependence chain since one missing load's address is dependent on the previous missing load's value. Taking a frequently executed code segment from the benchmark *mcf* as an example, shown in Figure 1, the profile information shows that the pointer chasing codes '*node->child*', '*node->basic_arc->cost*', and '*node->pred->potential*' result in many cache misses. The memory dependence chains formed by these missing loads are shown in Figure 2.

```
while( node )
{
  if( node->orientation == UP )
    node->potential = node->basic_arc->cost + node->pred->potential;    // (Nodes 1,2,3,4)
  else /* == DOWN */
  {
    node->potential = node->pred->potential - node->basic_arc->cost;
    checksum++;
  }
  tmp = node;
  node = node->child;        // (Nodes 0, 5)
}
```

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

In Figure 2(a), the dependence chain is based on a single iteration of the *while* loop in Figure 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*'. Node 5 corresponds to '*node->child*' and node 0 is the same load '*node->child*' from the previous iteration. Figure 2(b) shows the dependence chain when the loop is unrolled multiple times. The solid arrow in Figure 2 represents the true data dependence and the dashed arrow represents the alias dependence between missing loads. Although the alias

dependence exists between a store and a following load instruction, we use the same term to model the dependence between two missing loads when one or more store instructions exist between them and one of these stores is dependent on the first missing load. Here, it needs to be pointed out that alias dependencies span multiple iterations, e.g., there exists alias dependence between the node 2 in the first iteration and all the loads in later iterations, though not shown in Figure 2(b) for conciseness. Also, note that in the memory dependence chain, only missing loads are included as other instructions such as stores, adds, branches, and loads that hit in cache are not long latency operations.

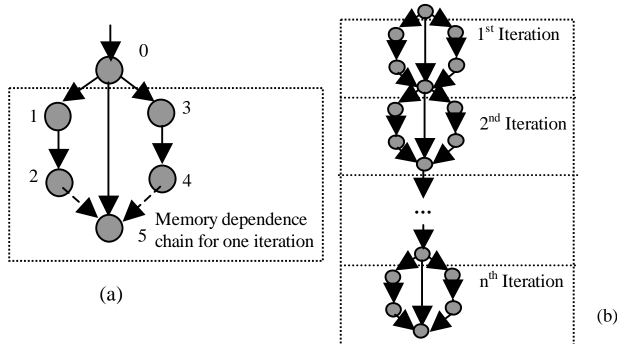

Figure 2. The memory dependence chain based on the code in Figure 1. (a) The dependence chain for a single iteration. (b) The dependence chain for multiple iterations (alias dependence among different iterations are not shown for conciseness).

From this example, we can see that both true data dependencies and alias dependencies enforce the sequential 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 5 on nodes 2 and 4 in Figure 2(a), thus exposing the critical path of executing the loop as chasing the pointer '*node->child*' (i.e., node 5). If a correct prediction can be made for this load, the execution of multiple iterations of the loop can be overlapped, as shown in Figure 3, where predicting the value of the pointer chasing load (node 5' in Figure 3) in the second iteration 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 Figure 3 illustrates that the effectiveness of value prediction in breaking the true memory dependence chain so that sequential cache misses can be processed in parallel and MLP can be enhanced. Such effectiveness is affected by several characteristics of the memory dependence chain. The first is the length of the memory dependence chain. In the example in Figure 3, the instruction window size determines how many iterations of the loop can be unrolled dynamically. If the 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 missing load along the dependence chain is predicted. In the example in Figure 3, it can be seen that predicting the value of Node 5' can overlap more cache misses than predicting Node 5 or Node 5''. The third is the predictability of these missing loads' values since more accurate prediction will result in more useful speculative executions. In [29], 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 other dependent load instructions to be executed early. In the example in Figure 3, it can be seen that predicting the value of Node 5' is equivalent to predicting the address of the dependent loads (e.g., Node 5'') since the only difference is a constant offset. So, using the address prediction based prefetching, the miss latency of Node5'' can be hidden if the prefetch is triggered early enough. Value prediction, on the other hand, not only fetches the data of Node 5'' but also uses the fetched data to execute other dependent instructions (i.e., the 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 memory dependence chain length, the value prediction accuracy, and the cache miss latencies. Since the chain length scales with the effective instruction window size and miss penalties scales with fast processor clock speed, we argue that value prediction is a very powerful technique to improve MLP for future high performance microprocessors.

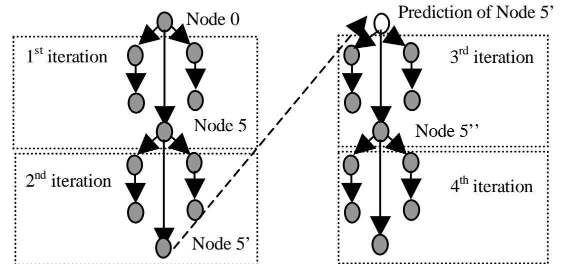

Figure 3. Predicting the value of Node 5' enables overlapping of cache misses in different iterations.

4. RECOVERY-FREE VALUE PREDICTION

As discussed in Section 3, value prediction has great potentials to enhance MLP by overlapping otherwise sequential cache misses. To implement such a technique, however, complex hardware support is necessary to validate the prediction and to perform recovery from value mispredictions. As discussed in Section 1, current microprocessors can execute computations very fast as long as slow memory operations (e.g., cache misses) are not involved. So, unlike previously proposed value prediction schemes [9],[16],[17], we propose to use 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

Figure 3, based on the prediction of node 5', the third and the fourth iterations of the loop are executed speculatively. Unlike the traditional value prediction schemes, the speculative results won't be committed in our approach and the only purpose of such speculative execution is to bring the data to L1 data cache. As a result, even if the prediction is correct, the third and the fourth iterations of the loop will be executed again (un-speculatively) in our proposed scheme. We expect that such 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 trades a small penalty of re-execution in the case of correct value prediction for much smaller hardware overhead. In the case of a value misprediction, both traditional schemes and our proposed scheme will result in polluting the data cache while our scheme associates *no* recovery penalties. 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 is capable of delivering higher performance improvement.

To support recover-free value prediction, only minor hardware changes are necessary. We present our proposed design based on a MIPS R10000 style microarchitecture [27], which has a 7-stage pipeline as shown in Figure 4. There are four key changes to the hardware, presented as follows.

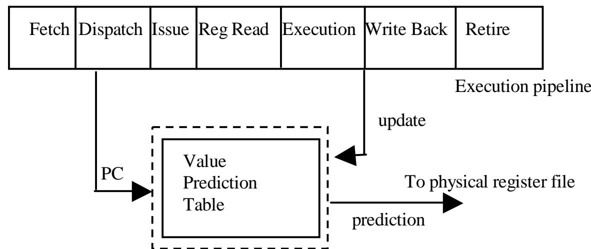

Figure 4. The execution pipeline.

First, a value predictor is included in the front-end of the processor and is indexed with *pc*, as shown in Figure 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],[16],[22] to show the effectiveness of our technique though a more powerful predictor [24],[28] can lead to a higher performance improvement.

Secondly, 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 window or RUU. 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 un-speculatively 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 issue queue 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 more than once. Speculatively issued instructions will remain in the issue queue until they are issued un-speculatively later with (un-speculative) ready source registers. When a speculatively issued instruction finishes, it writes back the speculative results to the physical register file and set 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 un-speculative execution of the same instruction. In the case when the speculative result arrives later than the un-speculative result, it is simply dropped.

Thirdly, the instruction selection logic is modified so that it prioritizes the issue of un-speculative 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 cache.

Fourthly, to break the 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, these 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 un-speculatively after the prior store addresses are resolved and the speculative execution is used only for prefetching. We call this as *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 the speculative execution results enables our approach to utilize the otherwise unused machine resources and does not require additional ports to the register file.

Here, one interesting observation is that our proposed recovery-free speculative execution could be viewed as a simple, yet efficient form of pre-execution. 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 pre-execution thread triggered by the prediction, though there is no explicit multi-thread support. Such pre-execution threads are constructed for each predicted value based on the data dependence relationship dynamically from the fetched instruction stream, thus taking advantage of dynamic branch prediction. The pre-execution is terminated when the normal execution catches up with the pre-execution 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 pre-execution is to prefetch the data and the pre-execution thread executes only if

there are unused resources, thus avoiding resource competition with the main thread.

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 and the ROB size (or the instruction window size) 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 Olden benchmark suite. Benchmarks *bzip2*, *gap*, *gcc*, *gzip*, and *perl* are computation-intensive and benchmarks *mcf*, *parser*, *twolf*, *health*, and *mst* are used as memory-intensive as they exhibit much higher data cache miss rates. The reference input data are used for SPEC2000 benchmarks. We fast-forward 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*, 3407 nodes are used as input and the first 2B 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.

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.
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; Data cache ports: 4
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.

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 Figure 5. 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 potential incorrect predictions. For each successful prediction, the confidence counter is increased by 2 and is decreased by 1 for each misprediction [24]. The prediction with the confidence counter larger than 4 is viewed as a confident prediction. The speculative update similar to what proposed in [15] is also used to improve the prediction accuracy.

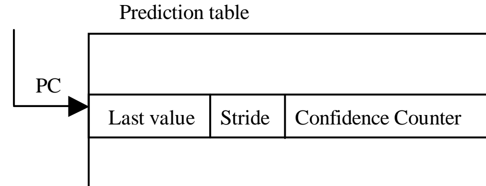

Figure 5. The stride value prediction table.

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.

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 the otherwise stalled loads can be executed speculatively in parallel with the un-speculative missing loads. These speculatively executed loads perform the functionality of prefetching the data into the cache so that the un-speculative execution will experience fewer cache misses. We first examine the effect of this technique in reducing data cache miss rates, as shown in Figures 6 and 7. Here, the cache misses during the speculative execution are not counted since they are used as prefetch. For each benchmark in Figure 6, the L1 D-cache miss-rate results are reported for both the base 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 non-covered misses. Partially cover cache misses have less impact on overall performance compared to non-covered cache misses. Figure 6 shows that for memory intensive benchmarks, the proposed technique reduces the L1 D-cache miss rate significantly ranging from 14% (from 47% to 33% in the benchmark *mcf*) to 0.5% (from 16.5% to 16% for the benchmark *health*) and increases the ratio of partially covered misses for most benchmarks. For computational intensive benchmarks, a

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)

visible 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.

Figure 7 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 does not only bring 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 L1 miss rate reduction, we can see that the overall L2 misses are also reduced, 14.5 L2 misses per 1k instruction comparing to 15.8 L2 misses originally.

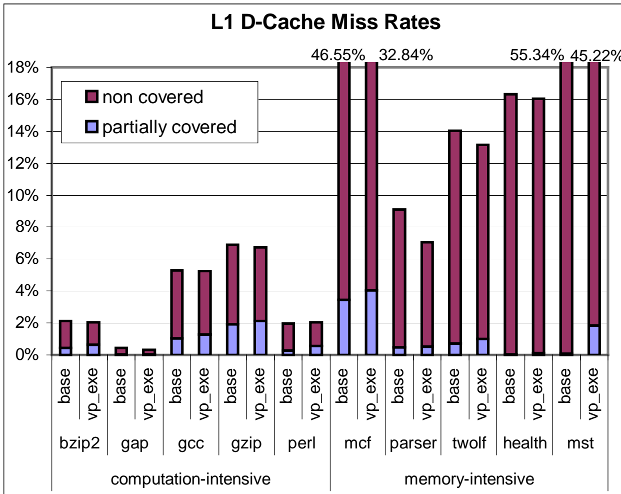

Figure 6. The L1 D-cache miss-rates.

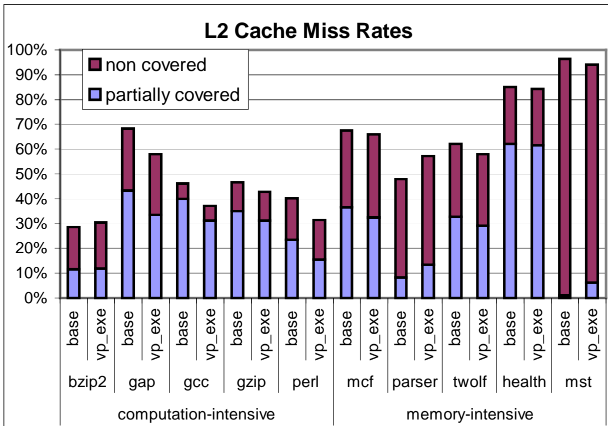

Figure 7. The L2 cache miss-rates.

Next, we use the benchmark *mcf* as an example to show the MLP improvement (i.e., overlapping the cache misses) achieved by the proposed technique for a typical heavy pointer chasing workload. Figure 8 shows the distribution of how many L1 data cache misses are overlapped in the base processor. The *x*-axis of Figure 8 is the number of the overlapping misses and the *y*-axis is the time during execution that the overlapping happens. From Figure 8, we can see that the processor spends 12% of overall execution time on computations that do not involve a cache miss. In 33% 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% of the time two missing loads are accessing the L1 D-cache. The maximum number of overlapping cache misses are determined by the MSHRs used in the cache and our experiment uses 32 MSHRs for the L1 D-cache. It can be inferred from this distribution that the benchmark *mcf* has many sequential cache misses, resulting in low MLP and MSHR utilization, and therefore long execution time.

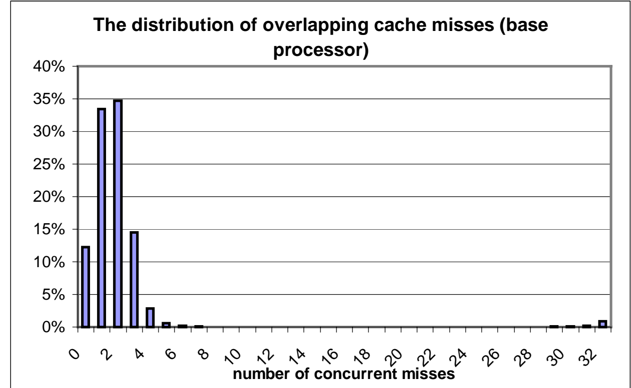

Figure 8. The baseline MLP for the benchmark *mcf* (overall execution time = 390M cycles).

With recover-free value prediction, the overall execution time is significantly reduced and MLP is much improved as shown in Figure 9. Compared to Figure 8, a significant amount of sequential cache misses are now processed in parallel. Another interesting observation is that the speculative execution does not increase the pressure on MSHRs since it rarely converts sequential cache misses into more than six concurrent cache-misses.

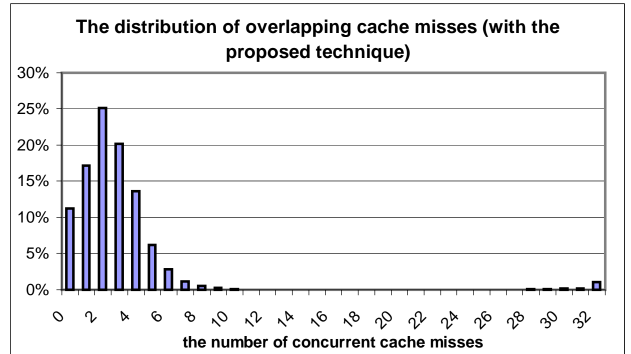

Figure 9. The improved MLP for the benchmark *mcf* with recovery-free value prediction (overall execution time = 327M cycles).

Figure 10 shows the speedups of the proposed recovery-free value prediction and it shows that our proposed technique achieves significant speedups for memory intensive benchmarks, from 3.2% for the benchmark *health* to 24% for the benchmark *mst*. For the well-known pointer-chasing benchmark *mcf*, the speedup is 19.6%. Considering the low hardware overhead required by this technique, the performance gains are impressive.

For computation intensive benchmarks, smaller speedups (average of 0.5%) result, 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%) is *gcc*, which will be discussed further in Section 6.3.

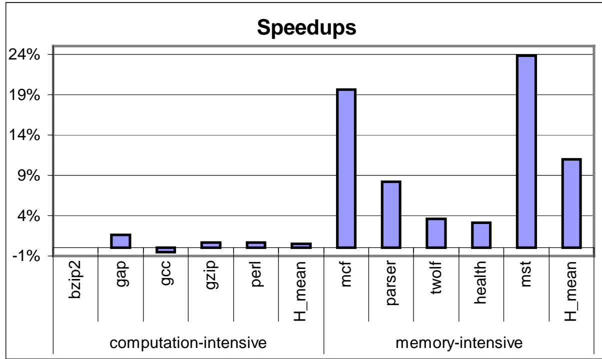

Figure 10. The speedups of using recovery-free value prediction.

6.2 Performance Analysis

To analyze why the proposed technique achieves such impressive 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],[16],[22] that many instructions exhibit stride locality and a more recent work [26] also showed that stride locality exists in the address stream for many load instructions in irregular programs. As pointed out in Section 3, the address predictability of load addresses is equivalent to load value predictability for pointer chasing codes. Our results, shown in Figure 11, 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 Figure 11 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.

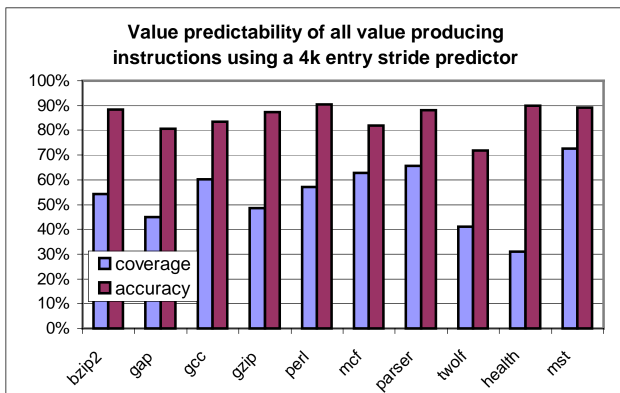

Figure 11. The value predictability for all value producing instructions using a 4k-entry stride predictor.

Since value predictions are used to break memory dependence chain, the predictability of the missing loads is of special interests and is examined in Figure 12. From Figure 12, it can be seen that the value of missing loads exhibit different degrees of stride locality for different benchmarks. For the heavy pointer chasing benchmarks *mcf* and *mst*, the value predictor achieves large 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 as well.

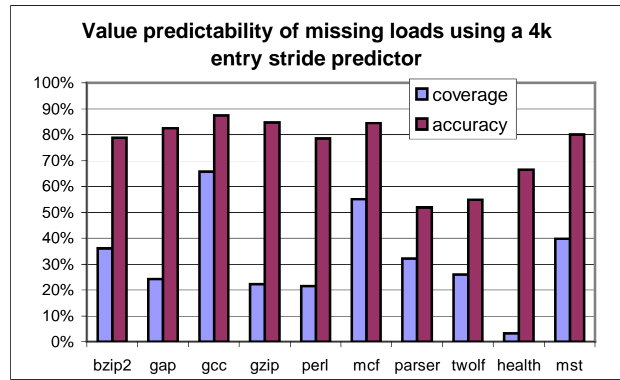

Figure 12. The value predictability for missing loads using a 4k-entry stride predictor.

As discussed in Section 3, both true data dependence and the alias dependence between missing loads prevent these loads from being executed in parallel. The recover-free value prediction scheme breaks both dependencies during the speculative 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. Figure 13 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 Figure 10, labeled '*both*'). We also include the speedup results using the traditional value prediction (labeled '*trad_value_pred*') in Figure 13. In the traditional value prediction scheme, the same stride value predictor is used and an idealistic validation and selective recovery (1 cycle penalty) mechanism is incorporated into the execution pipeline. From Figure 13, it can be seen that for computation-intensive benchmarks, the aggressive memory disambiguation has slightly better speedups than performing value prediction only. For memory-intensive benchmarks, breaking true dependencies results much higher speedups for *mcf* and *mst* but less 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 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.

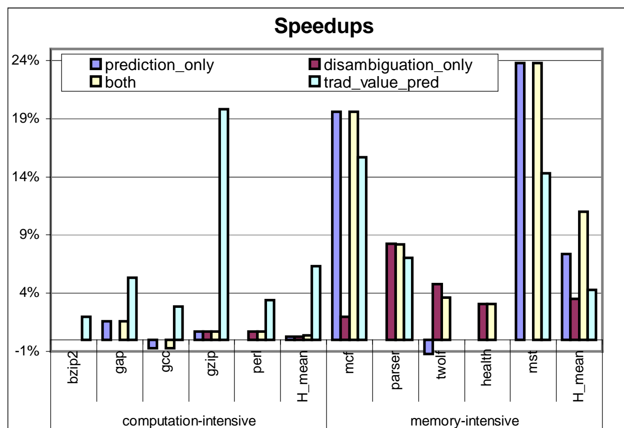

Figure 13. The speedups resulting from breaking different dependencies and traditional value speculation.

Comparing our proposed recovery-free scheme to the traditional value prediction, we can see that the traditional value prediction achieves higher speedups for computation intensive benchmarks. For memory-intensive benchmarks, our recovery-free prediction scheme has much higher speedups since it avoids the misprediction penalties and benefits from speculative memory disambiguation. For example, the recovery penalties (even with only 1 cycle penalty per misprediction) account for 2.6% of the overall execution time for the benchmark *mcf* while our recovery-free scheme totally removes such penalties. Moreover, in recovery-free value prediction, the value predictor is updated with un-speculative execution results (i.e., the computation results not involving direct/indirect predicted values), thereby being able to achieve higher prediction accuracies than the traditional value speculation scheme.

The results in Figure 13 also suggest another interesting optimization: we can apply recovery-free value prediction selectively by monitoring the dynamic behavior of a workload. Only if the workload is memory intensive (e.g., the L1 D-cache miss rate is larger than 10%), the recovery-free value prediction is turned on. Otherwise, recovery-free value prediction is turned off or only the aggressive memory disambiguation is used for prefetching. Further exploration of this optimization is out of the scope of this paper and left as future work.

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 unified cache (labeled as '*configuration 1*'), 32kB 2-way L1 D-cache and 512kB 8-way L2 cache (same as base processor, labeled as '*configuration 2*'), and 64kB 4-way L1 D-cache and 2048kB 8-way L2 cache (labeled as '*configuration 3*'). The speedups of the proposed technique in these configurations are show in Figure 14.

Interesting observations can be made from Figure 14. First, for the small D-cache of 16kB, the memory problem becomes more evident. As a result, more speedups are achieved by hiding the miss latency 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 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 cache pollution problem while the criticality of memory operations is reduced if they hit in the cache.

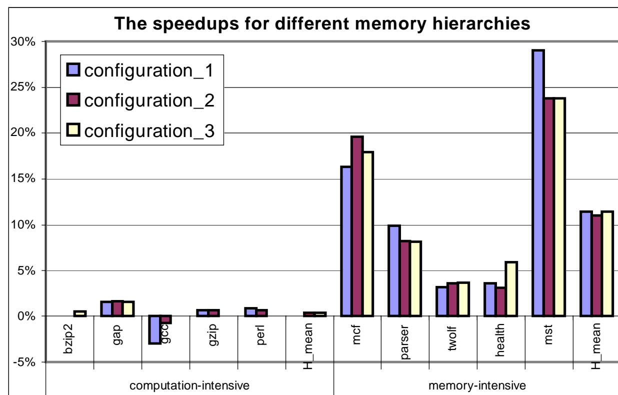

Figure 14. The speedups for different memory hierarchies.

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 4/64 issue model. The results are shown in Figure 15. 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. Secondly, 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.

7. LIMITATIONS

Two limitations exist with our proposed scheme. First, as we pointed out in Section 3, value prediction can hide memory access latencies by breaking the 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 accessing a large array, our proposed scheme will have very limited capability to hide these cache miss penalties.

Secondly, in our proposed recovery-free value prediction scheme, a prediction is made only after the 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 instruction window. It limits the capability to explore the far-flung MLP even 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], [19] to pre-execute/prefetch both independent and dependent memory accesses.

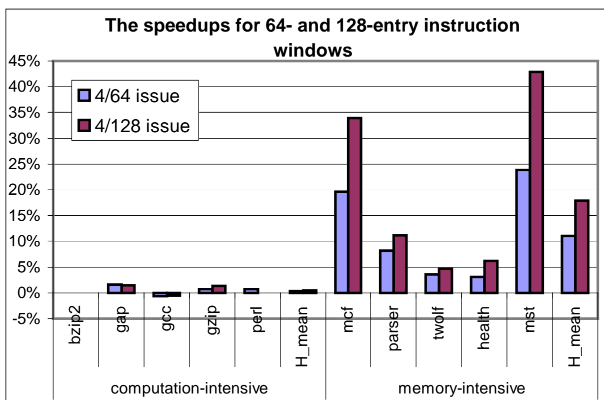

Figure 15. The speedups for different instruction window sizes.

8. CONCLUSION

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 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.

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 the useless speculation (i.e., the speculation not leading to useful prefetch).

9. ACKNOWLEDGMENTS

This work was supported by NSF awards CCR-0208596 and CCR-0072926 and a hardware donation from Hewlett-Packard.

10. REFERENCES

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta, “Predictability of load/store latencies”, *Proceeding of the 26th International Symposium on Microarchitecture (MICRO-26)*, 1993.

[2] M. Bekerman, S. Jourdan, R. Ronen, G Kirshenboim, L. Pappoport, A. Yoaz, and U. Weiser, “Correlated Load-Address Predictors”, *Proceeding of the 26th International Symposium on Computer Architecture (ISCA-26)*, 1999.

[3] D. Burger and T. Austin, “The SimpleScalar tool set, v2.0”, *Computer Architecture News (ACM SIGARCH newsletter)*, vol. 25, June 1997.

[4] B. Calder and G. Reinman, “A comparative survey of load speculation architecures”, *Journal of Instruction-Level Parallelism*, 2000.

[5] M. Carlisle, “Olden: parallelizing programs with dynamic data structures on distributed-memory machines”, *Ph.D. thesis, Princeton University Computer Science Department*, 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”, *Proceeding of the 28th International Symposium on Computer Architecture (ISCA-28)*, 2001

[7] R. Cooksey, S. Jourdan, and D. Grunwald, “A stateless, content-directed data prefetching mechanism”, *Proceeding of the 10th International Conference on 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”, *Proceeding of the 1997 International Conference on Supercomputing*, 1997.

[9] F. Gabbay and A. Mendelson, “Speculative execution based on value prediction,” *EE Department Tech Report 1080, Tachnion - Israel Institute of Technology*, Nov. 1996.

[10] J. Gonzalez and A. Gonzalez, “Speculative execution via address prediction and data prefetching”, *Proceeding of the 1997 International Conference on Supercomputing*, 1997.

[11] J. Henning, “SPEC2000: measuring CPU performance in the new millennium”, *IEEE Computer*, July 2000.

[12] D. Joseph and D. Grunwald, “Prefetching using Markov Predictors”, *IEEE Transactions on Computers.* Vol. 48, Feb 1999.

[13] T. Karkhanis and J. Smith, “A Day in the Life of a Cache Miss”, *Proceeding of the 2nd Annual Workshop on Memory Performance Issues (WMPI 2002)*, 2002.

[14] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, “A large, fast instruction window for tolerating cache misses”, *Proceeding of the 29th International Symposium on Computer Architecture (ISCA-29)*, 2002.

[15] S. Lee and P. Yew, “On some implementation issues for value prediction on wide ILP processors”, *Proceeding of*

the International Conference on Parallel Architectures and Compilation Techniques (PACT'00), 2000.

[16] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," *Proceeding of the 29th International Symposium on Microarchitecture (MICRO-29)*, 1996.

[17] M.H. Lipasti, C. B. Wikerson and J. P. Shen, "Value locality and load value prediction," *Proceeding of the 7th International Conference on Architectural Support for Programming Language and Operation Systems (ASPLOS-7)*, Oct, 1996.

[18] C. K. Luk, "Tolerating memory latency through soft-ware-controlled pre-execution in simultaneous multithreading processors", *Proceeding of the 28th International Symposium on Computer Architecture (ISCA-28)*, 2001.

[19] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors", *Proceeding of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003..

[20] E. Rotenberg, S. Bennett, and J. E. Smith. "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching", *Proceeding of the 29th International Symposium on Microarchitecture (MICRO-29)*, 1996.

[21] A. Roth and G. Sohi, "Speculative data driven multithreading", *Proceeding of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, 2001.

[22] Y. Sazeides and J. E. Smith, "The predictability of data values," *Proceeding of the 30th International Symposium on Microarchitecture (MICRO-30)*, Nov. 1997.

[23] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines", *Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29)*, 2002.

[24] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," *Proceeding of the 30th International Symposium on Microarchitecture (MICRO-30)*, Nov. 1997.

[25] 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", *Proceeding of the 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, 2002.

[26] Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching", *Proceeding of the ACM 2002 Conference on Programming Language Design and Implementation (PLDI-2002)*, 2002.

[27] K. C. Yeager, "The MIPS R10000 superscalar microprocessor", *IEEE Micro*, 1996.

[28] H. Zhou, J. Bodine, and T. Conte, "Detecting global stride localities in value streams", *Proceeding of the 30th Int'l Symp. on Computer Architecture (ISCA-30)*, 2003.

[29] H. Zhou and T. Conte, "Performance modeling of memory latency hiding techniques", *Technical Report, ECE Department, N. C. State University*, Dec. 2002

[30] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices", *Proceeding of the 28th International Symposium on Computer Architecture (ISCA-28)*, 2001.