Performance Modeling of Memory Latency Hiding Techniques

Huiyang Zhou
Computer Science Department
University of Central Florida
zhou@cs.ucf.edu

Thomas M. Conte
Department of Electrical and Computer Engineering
North Carolina State University
conte@ncsu.edu

Abstract

Due to the ever-increasing computational power of contemporary microprocessors, the execution time spent on actual arithmetic computations (i.e., computations not involving slow memory operations such as cache misses) is significantly reduced. Therefore, for memory intensive workloads, it is more important to overlap multiple cache misses than to overlap slow memory operations with other computations. Based on the prediction of missing loads' addresses, data prefetching techniques have the capability to bring in the required data early so that the miss latency can be overlapped with prior memory operations or other computations. In this paper, we highlight the limitation of traditional prefetching techniques and advocate that value prediction, though proposed originally as an instruction-level-parallelism (ILP) optimization, is more capable of overlapping cache misses and increasing memory-level-parallelism (MLP) than traditional prefetching techniques for pointer-chasing *workloads. We develop an analytical model to examine performance potential of both data prefetching and value prediction. Important and somewhat unexpected insights are revealed and the code characteristics leading to the performance differences between these two techniques are identified. The observations based on the model provide sound theoretical backgrounds of recent proposals on hiding memory access latencies and highlight their performance potential.*

1. Introduction

The trends in contemporary microprocessor design, including fast clock speeds, deep pipelines [22], large window sizes [14],[15], aggressive out-of-order instruction execution, and wide fetch bandwidths [19], result in a tremendous ability to perform arithmetic computations (i.e., computation not involving slow memory operations such as cache misses). Therefore, for memory intensive workloads, especially those with heavy pointer chasing, it is more important to parallelize multiple cache misses than to

overlap cache misses with other computations. For example, assuming the pointer-chasing code shown in Figure 1 results in many cache misses when traversing the linked list, these cache misses form a memory dependence chain due to the dependencies among the missing loads.

```
while (a != NULL) {
    //Processing the fields of a
    a = a->next; //traverse the link list
}
```

Figure 1. A pointer-chasing code example.

As processing the linked list takes a relatively small amount of time compared to traversing it, the overall execution time is mainly determined by resolving such a memory dependence chain of missing loads. To reduce the time of serving these dependent cache misses, different techniques have been proposed. Data prefetching [1],[2],[8],[13],[21] based on address prediction of the missing loads, tries to bring the data close to the processor (e.g., L1 or L2 Cache) long before the missing load executes so that the miss latency can be overlapped either with other computation or with previous load misses. In the code example in Figure 1, every successful address prediction of the chasing load has the potential to eliminate one cache miss.

Value prediction [9],[16],[17], which relies on the predictability of the destination value of an instruction (e.g., the load value) rather than the load address, enables dependent computations to be executed speculatively while the missing load is being served. However, for *pointer-chasing codes*, the predictability of load values can be viewed as *at least equivalent* to the predictability of load addresses since one load address is simply the previous load's value plus a constant offset. Also, if the dependence between two missing loads is carried through several cache-hitting loads, these two misses can be overlapped when the first missing load's value is predictable or any of the hitting loads' value is predictable. The address prediction of the second missing load, however, does not benefit from any successful address prediction of those hitting loads. While value prediction was proposed originally to

break true data dependencies as an instruction-level-parallelism (ILP) optimization, we advocate that one of its most important merits lies in its ability to *enhance the memory-level-parallelism (MLP) by overlapping multiple outstanding load misses*. In the code example in Figure 1, assume that the instruction window can hold a certain number of iterations of the loop, say ten, and there are five pointer-chasing loads ('a->next') in these ten iterations that will miss in the data cache. Also, we assume that one of these five missing loads' values is predictable, say the second missing load. Predicting the value of the second missing load enables two of its dependent loads (the third and fourth missing loads in this example) to be overlapped with the first and second missing loads. As a result, a single value prediction can reduce the number of cache misses by 2 -- much better than what would be achieved using a prefetch with the same predictability.

The above simple pointer-chasing code illustrates that value prediction can be more effective in overlapping cache misses and increasing memory level parallelism (MLP). In this paper, we introduce a formal analytical model using performance bounds to evaluate and compare the performance potential of both prefetching and value prediction. The target workload is *memory intensive applications with heavy pointer-chasing*, e.g., programs involving traversing large linked lists, trees, and graphs. This analytical model reveals the capability of both data prefetching and value prediction in hiding cache miss latencies through MLP utilization. The important observations are drawn from the model: while prefetching is generally effective for short memory dependence chains, value prediction has *better* potential for long dependence chains. For a long memory dependence chain due to pointer-chasing, the performance difference between value prediction and prefetching scales proportionally with the prediction accuracy, the memory dependence chain length, and load miss penalties. Since the chain length scales with the effective instruction window size and miss penalties scale with fast

processor clock speed, the model shows that value prediction is a very powerful technique for improving MLP in high performance microprocessors.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 presents the performance modeling of memory prefetching. Section 4 contains the modeling of value prediction. The performance comparison of the two is in Section 5. A detailed case study based on the benchmark *mcf* is presented in Section 6. Finally, Section 7 summarizes and motivates the future work.

2. Related Work

As discussed in Section 1, value prediction is proposed originally as an ILP optimization technique. Lipasti et al. [17] proposed load value prediction to reduce the penalty of cache misses. Gabby et al. [10] performed an experimental study on value predictability in load misses and address predictability. They showed that value prediction can complement data prefetching since some loads show only predictable values and some loads exhibit only predictable addresses. In our study, we focus on pointer-chasing workloads, in which the predictability of load values is equivalent to the predictability of addresses, and we propose an analytical model to compare value prediction with prefetching directly in their capability to hide memory access latencies.

Recently, several schemes have been proposed to hide memory access latency using the value prediction principles. Address prediction for prefetching, proposed by Gonzales et al. [11], uses the predicted addresses to prefetch data into a special buffer, called Memory Prefetch Table, and the prefetched data are later on used as predicted values. Pointer cache structure, proposed by Collins et al. [5], stores the values of loads that correspond to pointer-chasing and then is used as a value predictor if a load hits in the pointer cache. Recovery-free value prediction [26] uses value prediction for prefetching, thereby avoiding the misprediction recovery and also being able to leverage aggressive memory disambiguation for prefetching. The analytical model developed in this paper provides both

sound theoretical backgrounds of these proposals and performance bounds that can be achieved by them.

A very large instruction window can be effective to hide long memory access latency [14],[15] although it incurs much higher branch misprediction penalty. Our analytical model show that performance potential of value prediction scales with the window size, thereby complementing the effectiveness of large windows in hiding long memory access latencies.

Another promising way to hide memory latency is based on the concept of pre-execution/pre-computation. Both hardware-based and software-based schemes have been proposed [6],[18],[20],[27]. As pointed out in [5], value prediction can help the speculative thread to run far ahead using data speculation. Also, as discussed in [26], speculative execution enabled by value prediction can be viewed as an implicit thread when it is used for prefetching.

3. Performance Modeling of Memory Prefetching

For workloads with heavy pointer-chasing, the memory dependence chain of missing loads dominates the overall execution time since other computations (including those cache hits) are either overlapped with the memory access latency or only accounts for a small portion of the overall execution time. Instead of relying solely on simulation, we use performance bounds to evaluate the performance potential of memory latency hiding techniques and then we use simulation to validate the conclusions that we draw from our performance model. For a memory dependence chain containing N dependent, missing loads (which we call *a dependence chain of length N*), a lower bound of execution time (LBET) is defined as the time to resolve all these missing loads:

$$\text{LBET}_{\text{original}} = N * Miss_latency. \quad \textbf{(1)}$$

In this model, we use the same miss latency to model the penalty of all missing loads. For a memory hierarchy with multiple cache levels, the miss latency varies at each level. As the miss latency at a

lower cache level (e.g., L0 or L1) can usually be hidden successfully with out-of-order execution or aggressive instruction scheduling [23], we choose to use this memory dependence chain to model a sequence of cache misses at a higher level cache (e.g, a sequence of dependent L2 misses).

To model the performance potential of memory prefetching, we assume that if the address of a missing load is predictable (i.e., the missing load is *prefetchable*), then a prefetch can be triggered early enough so that the miss latency is hidden completely. Such an idealistic assumption favors the results of prefetching, but do not affect our conclusions. Based on this assumption, if K missing loads along the chain can be prefetched, the performance bound is then the time to resolve the remaining (N-K) load misses (the L1 cache hit time is ignored in our modeling as it is at least an order less than the L2 miss latency):

$$\text{LBET}_{\text{prefetch_K}} = (N - K) * Miss_latency. \quad \textbf{(2)}$$

In other words, prefetching K loads collapses a chain of length N into a chain of length (N-K). For example, consider the pointer-chasing code "*a->b->c->d->e*", where *a-e* are loads, and assume they result in four dependent missing loads. Prefetching any of them will reduce the length of the chain to 3, as shown in Figure 2.

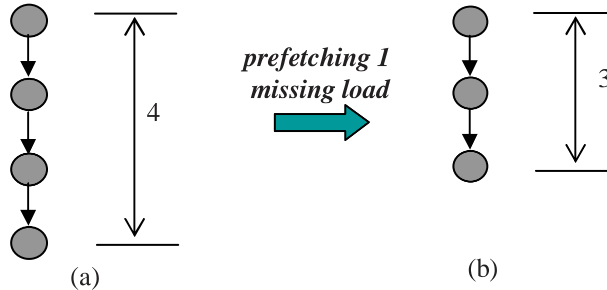


Figure 2. (a) The code 'a->b->c->d->e' resulting in a memory dependence chain of 4 missing loads; (b) Prefetching 1 missing load along the chain reduces the chain length by 1.

The model can then be extended to include the impact of load address mispredictions. If the address prediction accuracy is x% (assuming the same accuracy for all predictions for simplicity), the performance bound is the weighted sum of successful prefetching and prefetching with mispredicted

addresses. Assuming prefetching a mispredicted address has little impact on the overall performance, the performance bound can be computed as:

$$\begin{aligned}\text{LBET}_{\text{prefetch_}K\text{_}accu} &= \text{LBET}_{\text{prefetch_}K} * x\% + \text{LBET}_{\text{original}} * (1 - x\%) \\ &= (N - K * x\%) * \textit{Miss Penalty}. \end{aligned} \quad \textbf{(3)}$$

For a special case $K = 1$, i.e., prefetching one missing load, the performance bound is $N * \textit{Miss Penalty} - \textit{Miss Penalty} * x\%$.

4. Performance Modeling of Value Prediction

Predicting the value of a single missing load along a memory dependence chain, say the ith load, breaks the dependence chain into two shorter ones. The performance bound is then determined by the longer one of the resulting two shorter chains, one with the length i and the other with the length (N-i). Thus, the performance bound can be computed as:

$$\text{LBET}_{\text{prediction_1}}(i) = \max\{i*\textit{Miss_Penalty}, (N-i)*\textit{Miss_Penalty}\} \quad \textbf{(4)}$$

As can be seen from Equation 4, unlike prefetching, the performance bound based on value prediction is dependent on where the prediction is made along the chain. Using the example in Figure 2, we can enumerate the predictions made for all different missing loads along the dependence chain, as shown in Figure 3. Figure 3a shows the memory dependence chain. In Figure 3b, predicting the first missing load enables the second load to be executed speculatively, thereby overlapping the memory access latency of these two loads. Predicting the value of the second load breaks the chain more evenly and results in more overlapping of missing loads (i.e., more MLP), as shown in Figure 3c. Figure 3d shows that predicting the third load only enables the fourth load to be overlapped. Predicting the value of the fourth load does not reduce the chain length (i.e., *no* performance improvement) assuming that there are no missing loads dependent on it, as shown in Figure 3e.

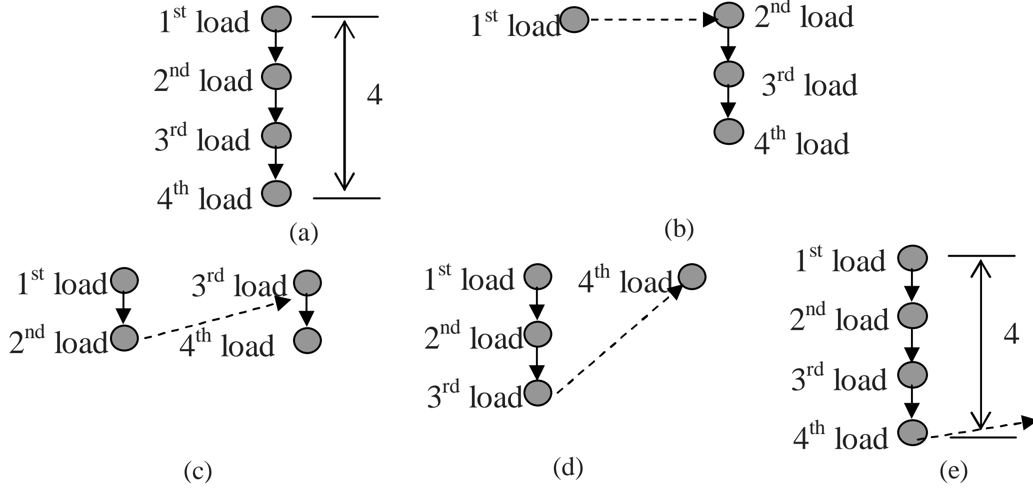


Figure 3. (a) A memory dependence chain of 4 miss loads; (b) Predicting the value of the first missing load; (c) Predicting the value of the second missing load; (d) Predicting the value of the third missing load; (e) Predicting the value of the fourth load.

As the performance bound of predicting one load along the memory dependence chain depends on which load is predicted, we can use a *probabilistic* approach to model this effect. Assuming $p(i)$ is the probability of a prediction happening at the ith missing load along the chain, the performance bound can be derived as follows.

$$\text{LBET}_{\text{prediction_1}} = \sum_{i=1}^{N} p(i) * \text{LBET}_{\text{prediction_1}}(i) \qquad \textbf{(5)}$$

For a uniform distribution of $p(i)$, (i.e., $p(i) = 1/N$), Equation 5 can be simplified into Equation 6 when N is odd:

$$\begin{aligned} \text{LBET}_{\text{prediction_1}} &= 2 * \sum_{i=1}^{\frac{N-1}{2}} [p(i) * (N-i) * Miss_Penalty] + p(i=N) * N * Miss_Penalty \\ &= \left(\frac{3}{4} * N + \frac{1}{4N}\right) * Miss_Penalty \end{aligned} \qquad \textbf{(6)}$$

Similarly when N is even, $\text{LBET}_{\text{prediction_1}} = (3/4 * N) * Miss_Penalty$.

Based on this derivation, one very interesting observation is that predicting a single load has a similar effect to reducing the chain length from N to ¾*N. Thus, predicting K loads along the chain

would reduce the chain to $(¾)^K * N$, i.e., $\text{LBET}_{\text{prediction_k}} \approx (¾)^K * N * Miss_latency$ as one prediction usually would *not* affect the predictability of other instructions. When taking the prediction accuracy (which is the same accuracy for all predictions for simplicity purposes), $x\%$, into account, the performance bound becomes Equation 7, assuming the misprediction penalty is small compared to the load miss latencies (if we focus on using value prediction to improve MLP, the recovery from value mispredictions can be removed completely [26]).

$$\begin{aligned} \text{LBET}_{\text{prediction_k_accu}} &= \text{LBET}_{\text{prediction_k}} * x\% + \text{LBET}_{\text{original}} * (1 - x\%) \\ &= [1 - x\% + (¾)^K * x\%] * N * \textit{Miss Penalty} \end{aligned} \qquad \textbf{(7)}$$

For a special case $K = 1$, i.e., predicting one missing load, the performance bound is $N * \textit{Miss Penalty} - (¼) * N * \textit{Miss Penalty} * x\%$.

5. Comparison between Prefetching and Value Prediction in Hiding Miss Latencies

In this section, we compare the performance potential of data prefetching and value prediction. We first focus on the case of predicting a single value or address along the chain. Then, we extend our discussion to predicting K values or addresses.

As discussed in the previous sections, prefetching a single missing load (i.e., predicting the address of one missing load) reduces the chain length by 1 (from N to N-1) while predicting the value of a single load has the potential to reduce the chain length by $¼*N$. As discussed in Section 1, if the memory dependence chain is due to pointer-chasing, then the predictability of the ith load value is equivalent to the predictability of the address of the $(i+1)^{th}$ load. Thus, the performance difference between single prefetching and single value prediction is $(1 - ¼*N) * Miss_latency * x\%$. As a result, we can see that if $N < 4$, then $¼*N < 1$, which implies that prefetching outperforms value prediction for *short* memory dependence chains. When $N \geq 4$, then $¼*N \geq 1$, which shows that value prediction has better performance potential for memory dependence chains containing more than 4 dependent

missing loads. Moreover, the performance difference is proportional to the chain length, cache miss latency, and prediction accuracy, i.e., value prediction is more superior to prefetching for higher miss latencies and better prediction accuracies. This conclusion is somewhat surprising as prefetching is a widely accepted technique to overcome the memory gap while value prediction, proposed originally as an ILP optimization, has not found its application in current processor design. Here, we argue that for *memory intensive pointer-chasing workloads* the most important merits of value prediction lies in its ability to *enhance MLP instead of improving ILP*.

When K values are predicted along the memory dependence chain, the lower bound of execution time is: $\text{LBET}_{\text{prediction_k}} \approx [1 - x\% + (3/4)^K * x\%] * N * \textit{Miss Penalty}$, which is an exponential function of the parameter K as we derived in Equation 7. Compared to prefetching K loads (or K address predictions), for which the performance bound is a linear function of the parameter K, $(N - K * x\%) * \textit{Miss Penalty}$, the performance differences can be illustrated in Figure 4, in which we pick different chain length N (4, 8, 16) and different prediction accuracy $x\%$ (90%, 50%). The x-axis of the graphs in Figure 4 indicates the number of value prediction or prefetching made along the chain and the y-axis shows the resulting lower bound of execution time. From this figure, we can see that value prediction performs *consistently* better than prefetch for *long* memory dependence chains for both high and low prediction accuracies except when all the missing loads along the chain can be prefetched (i.e., $K = N$). In addition, the performance potential of value prediction is usually achieved with few predictions (i.e., $K < 1/2 * N$) while much more prefetching is required to achieve the similar performance potential. As a result, value prediction can be made more selective along the chain without sacrificing the performance. For shorter memory dependence chains (i.e., $N < 4$), better performance is achieved from prefetching. Varying prediction accuracies has a direct impact on the performance potential for both value prediction and data prefetching but it does not change the trend that we observed.

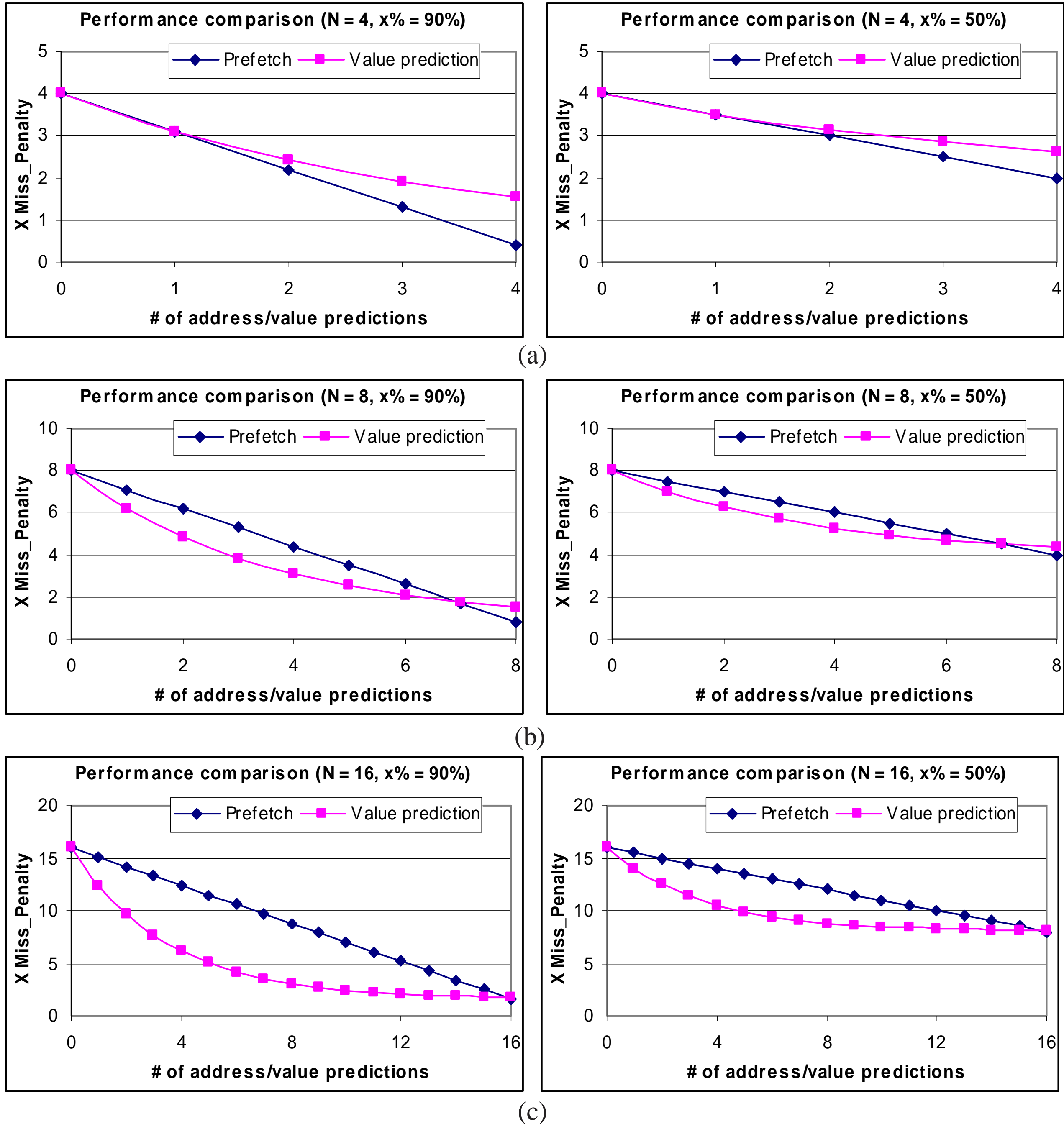


Figure 4. Performance comparison of value prediction and prefetching. (*x*-axis: number of value prediction or prefetching made along the chain; *y*-axis: the resulting execution time based on Equation3 and 7; N: chain length; x%: prediction accuracy). (a) Chain length as 4; (b) Chain length as 8; (c) Chain length as 16.

Although we establish that value prediction has greater performance potential based on the analytical model, we need to examine it more carefully to *intuitively* understand why it produces such potential. In order to do that, we take another look at the example in Figure 2 and Figure 3 and

replicate the dependence chain in Figure 5a. Assume the third missing load along the dependence chain is prefetchable, which also means that either the address of the third load is predictable or the value of the second load is predictable. The prefetching scheme uses this predicted address to execute a data fetch from memory and this fetch latency can be overlapped with outstanding misses of the first load, as shown in Figure 5b. The value prediction scheme uses the prediction of the second load (or the address of the third load) similarly to bring in the data. However, it takes this one step further by utilizing the fetched data to execute another dependent load (the fourth load in this example) so that the fourth load can be overlapped with the second load, as shown in Figure 5c.

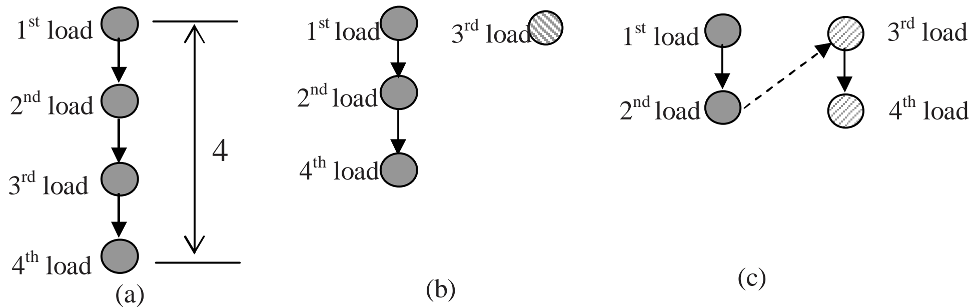


Figure 5. A memory dependence chain of 4 miss loads; (b) prefetching the third load; (c) value predicting the second load.

From this example, we can see that the *key* reason that value prediction provides more MLP than prefetching is that it *uses* the fetched data to enable more dependent missing loads to be executed. If the prediction is correct in the first place, such speculative execution *propagates the predictability* even though the dependent loads are not predictable (but it is *computable* based on the previous predictions). In this example, even if the value of the third load is not predictable, the data of the fourth load can still be fetched. This observation also explains why a recently proposed technique, called stateless, content-directed data prefetching [5] works better than traditional prefetching schemes. Content-directed data prefetching analyzes the content of the fetched data block to check whether the data could potentially be a pointer de-reference address. If so, it will attempt to fetch the data from this address as well. Similar observation was also made by Yang et al. [24] and they proposed a special

prefetch controller to dereference pointers at higher levels of memory hierarchy and push the data to the processor. Value prediction, compared to them, uses the fetched data more judiciously by following the code semantics so that it uses the resources more efficiently and has fewer chances to pollute the cache. The drawback of value prediction is that the effective instruction window should be large enough to fetch in the dependent missing loads.

6. Case Study

In this section, we use a detailed case study to validate our observations made in Section 5. We describe the experimental methodology in Section 6.1. The code segment from the benchmark *mcf* is presented in Section 6.2 as our target workload. Section 6.3 contains the experimental results and Section 6.4 briefly summaries the other workloads that we studied.

6.1. Methodology

In our experiments, a detailed timing simulator based on the Simplescalar [2] toolset is used. The underlying processor organization is modeled upon the MIPS R10000 processor [25], configured as indicated in Table 1. We also vary the instruction window size and the L2 cache miss latency to evaluate the performance impact of both value prediction and data prefetching.

Table 1. Base processor configuration

Instruction Cache	Size = 32 kB; Associativity = 2-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.
Unified L2 Cache	Size = 512 kB; Associativity = 8-way; Replacement = LRU; Line size = 128 bytes; Miss penalty = 80 cycles.
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

6.2. Workload

In this section, we choose the benchmark *mcf* as our target workload since it is pointer intensive and is well known for its infamous memory behavior. We skip its first 500M instructions and use the next

100M instructions to warm the caches. Then, the following 100M instructions are simulated to obtain the performance results. In the execution range that we simulated, we found that the loop in the function *refresh_potential*, as shown in Figure 6, is most frequently executed and causes a significant amount of L2 cache misses. We underlined the loads that tend to miss in L2 cache and labeled them as nodes 1-6 in Figure 6.

```
tmp = node = root->child;
while( node != root )
{
   while( node )
   {
      if( node->orientation == UP )
         node->potential = node->basic_arc->cost + node->pred->potential;      //nodes 1-4
      else /* == DOWN */
      {
         node->potential = node->pred->potential - node->basic_arc->cost;      //nodes 1-4
         checksum++;
      }
      tmp = node;
      node = node->child;                          //node 5, a traversing load
   }
   node = tmp;
   while( node->pred )
   {
      tmp = node->sibling;                         //node 6, a traversing load
      if( tmp )
      {
         node = tmp;
         break;
      }
      else
         node = node->pred;                        //traversing load
   }
}
```

Figure 6. A code segment from the target workload, *mcf*.

As shown in the code segment, the data structure can be traversed through several possible directions (or fields of the structure node), marked as '*traversing load*' in Figure 6. After examining the dynamic execution sequence of the loop, we found that the most frequently used traversing direction is through the '*sibling*' field (i.e., node 6) and the traversing load through the field '*child*' (i.e., node 5) always fetches a value of zero although it causes many L2 cache misses. Based on the

dynamic execution sequence, we can construct a memory dependence chain as shown in Figure 7 consisting of the missing loads (i.e., nodes 1-6) in multiple iterations of the loop. Note that the memory dependence chain only contains dynamic cache misses. If a node hits in cache for some iterations, it will disappear from the memory dependence chain and the chain length could be shorten accordingly.

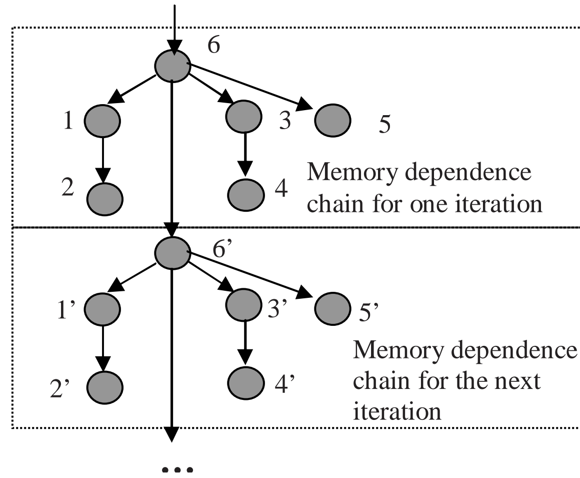


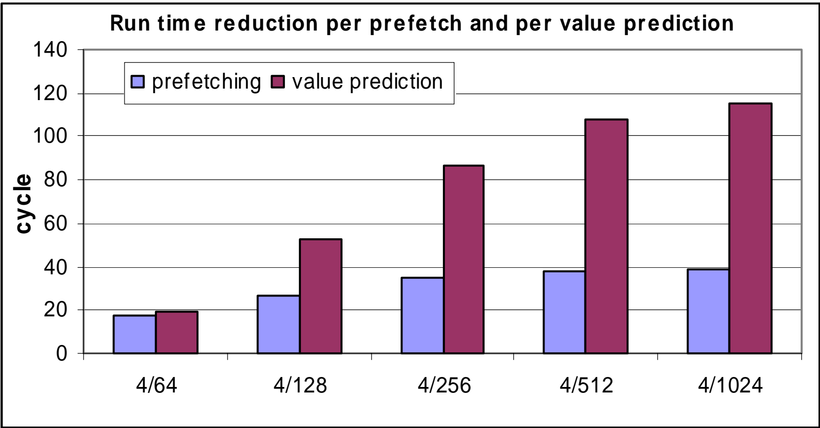
Figure 7. The memory dependence chains formed by missing loads in multiple iterations of the loop in Figure 6.

6.3. Results

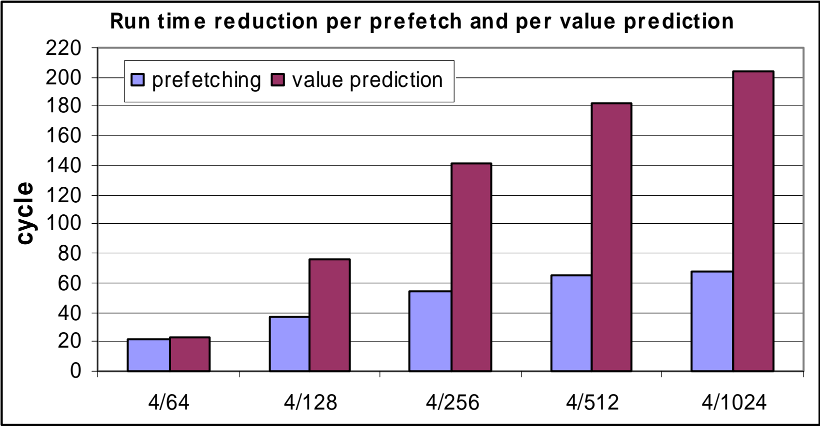
From the memory dependence chain in Figure 7, we can see that dynamic instances of the traversing load, node 6, form the critical path of this chain. Using the observations drawn from Section 5, we can expect that predicting the values of these loads will have a significant performance impact if the instruction window can hold multiple iterations of the loop body. Prefetching the same set of loads will shorten the dependence chain length as well but its performance gain is expected to be less impressive than value prediction for large instruction windows.

To validate our expectations, we first identified those instances of node 6 (i.e., the same load in multiple iterations) that will miss in L1 and L2 caches. We do so by probing the cache when the load is decoded. Since the actual load address can be dependent on previous missing loads, we use the information from functional simulator to probe the cache. Such a probe only checks whether the address is present in cache or not and will not change any cache state. After we identify those loads,

we perform either oracle value prediction (see Section 6.4 about our experiments of using real value predictors) or ideal prefetch. The ideal prefetch is realized by turning a L2 cache miss into a L1 cache hit, emulating the prefetch is issued early enough to hide the miss latency completely. Oracle value prediction is used correspondingly to examine the performance potential of value prediction. Since the prefetching and the value prediction are upon the same set of missing loads, we can then evaluate the performance impact of either of them by calculating how much run time can be saved from one prefetching or one value prediction in average. The results are shown in Figure 8 for different instruction window sizes (64, 128, 256, 512, and 1024 entries) and memory access latencies (80 and 160 cycles).



(a)



(b)

Figure 8. The run-time reduction from prefetching and value prediction. (a) memory access latency: 80 cycles (b) memory access latency: 160 cycles.

Several important observations can be made from Figure 8. First, the effectiveness of both prefetching and value prediction improves when the instruction window (ROB) size increases. For value prediction, a larger instruction window means a longer dependence chain can be broken down, thereby leading to better performance improvement, as discussed in Section 5. Prefetching, on the other hand, also favors a larger window size since more dependent missing loads can be fetched into the instruction window to utilize the prefetched value. For example, when the pipeline is stalled due to resolving cache misses due to nodes 1 and 2 (see Figure 7) and the instruction window is full, a successful prefetching of node 6 is not very useful when its dependent missing loads (i.e., nodes 1-6 of the next iteration) are not present in the instruction window. Secondly, the results show that value prediction performs consistently better than prefetch for window size ranging from 64 to 1024 and for different memory latencies. The performance differences between value prediction and prefetching scale with window size and memory access latency, validating the analytical results from Section 5. Thirdly, when memory access latency is 80 cycles, there shows a sign of diminishing returns when the window size is increased from 512 to 1024 since the performance pressure starts shifting to control transfer prediction. For a longer memory access latency of 160 cycles, such diminishing returns are not so evident. Overall, we can see that the effectiveness of value prediction scales much better with both window sizes and memory latencies, confirming our conclusion that value prediction is a very powerful technique to hide memory latencies for high performance microprocessors.

In the memory dependence chain shown in Figure 7, the dynamic instances of the traversing load, node 6, forms the critical path. For other missing loads in the dependence chain, the dependence chain is short and generally not on the critical path. For example, node 5 depends on node 6 and there is no missing loads following node 5. Thus, predicting the value of node 5 has almost minimal performance impact (7.4 cycle run time reduction per prediction for a 4/64 instruction window). Prefetching the

same node, on the other hand, shows some performance improvement (23.6 cycles per prefetch in average) for small instruction windows (a 4/64 window) and less impact on larger windows (15.6 cycles per prefetch for a 4/512 window). The reason is that the window size becomes a resource constraint for small windows when serving a cache miss. Once this constraint is removed, the missing loads on the critical path will overlap the non-critical misses. As a result, removing those non-critical misses through prefetching will not shorten the overall run time significantly.

6.4. Other workloads

In addition to the benchmark *mcf*, we studied other memory intensive workloads including those from SPEC 2000 INT and FP benchmark suite [12] as well as the Olden benchmark suite [3]. Similar to what we did with the benchmark *mcf*, we first identified the critical memory dependence relationship in the most frequently executed path(s) of the program. Then, we examine the performance impact of predicting and prefetching the missing loads in the memory dependence chain to validate the observations made in Section 5. The results based on the selected pointer-chasing SPEC 2000 INT and Olden benchmarks are similar and the benchmark *mcf* is most representative among them. SPEC 2000 FP benchmarks feature with large array accesses and the memory dependence chains are usually short. As a result, data prefetching combined with larger instruction windows provides an effective way to hide the memory access latencies for these workloads.

We also performed experiments with stride and context value/address predictors using recovery-free value prediction [26]. There shows little change in the results since we reported the performance improvement as how much run time reduction can be achieved for each prediction/prefetching in average. The major difference is the number of value predictions that can be made by a real value predictor. The cache pollution impact of mispredictions is limited when a standard confidence mechanism is utilized.

7. Summary

An analytical model is developed in this paper to model the performance potential of data prefetching and value prediction for memory intensive workloads. It is established that for pointer-chasing codes with long dependence chains (e.g., chasing a link-list), value prediction results in better MLP utilization and outperforms data prefetching given the same predictability model. The key reason for such a success is that the fetched data is not only placed in the cache but also used to drive the dependent loads, thereby propagating the predictability through speculative execution. The performance model also shows that the performance difference between value prediction and address prediction based prefetching scales with the chain length, the memory access latency, and prediction accuracy, thus making it highly compatible with trends in current microprocessor design.

Based on these important observations, the following interesting directions are worth careful exploration to hide memory access latencies more effectively.

- Designing more powerful value/address prediction techniques to break memory dependence chains more aggressively.
- Combining both prefetching and value prediction can potentially provide better results, since prefetching works well for short chains and value prediction is better for longer chains.
- Using profile information. The analytical model in this paper is based on several assumptions: prediction happens along a chain in a uniform distribution and all predictions have the same accuracy. Profile-based analysis can refine the model and guide the compiler to use the prediction and/or prefetch techniques more effectively.

8. References

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

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

[3] M. Carlisle, “Olden: parallelizing programs with dynamic data structures on distributed-memory machines”, In *Ph.D. thesis, Princeton University Computer Science Department*, 1996

[4] T. F. Chen and J. L. Baer, “Reducing memory latency via non-blocking and prefetching caches”, In *Proc. of the 5th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems*, 1992.

[5] J. Collins, S. Sair, B. Calder, and D. Tullsen, “Pointer Cache Assisted Prefetching”, *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*, 2002.

[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”, *Proceedings 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”, *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.

[8] K Farkas, P. Chow, N. Jouppi, and Z. Vranesic, “Memory-system design considerations for dynamically-scheduled processors”, *Proceedings of the 24th International Symposium on Computer Architecture (ISCA-24)*, 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] F. Gabbay and A. Mendelson, “Using Value Prediction to Increase the Power of Speculative Execution Hardware”, *ACM Transactions on Computer Systems*, August 1998.

[11] J. Gonzalez and A. Gonzalez, “Speculative execution via address prediction and data prefetching”, *Proceedings of the 1997 International Conference on Supercomputing (ICS-97)*, 1997.

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

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

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

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

[16] M. H. Lipasti and J. P. Shen, “Exceeding the dataflow limit via value prediction,” *Proceedings 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,” *Proceedings 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”, *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-28)*, 2001.

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

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

[21] T. Sherwood, S. Sair, and B. Calder, “Predictor-directed stream buffers”, *Proceedings of the 33rd International Symposium on Microarchitecture (MICRO-33)*, 2000.

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

[23] 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”, *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, 2002

[24] C. Yang and A. Lebeck, “Push vs. Pull: data movement for linked data structures”, *Proceedings of the 2000 International Conference on Supercomputing (ICS-2000)*, 2000.

[25] K. C. Yeager, “The MIPS R10000 superscalar microprocessor”, *IEEE Micro*, 1996.

[26] H Zhou and T. Conte, “Enhancing Memory Level Parallelism via Recovery-Free Value Prediction”, *Proceedings of the 2003 International Conference on Supercomputing (ICS-2003)*, 2003.

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