Feedback-directed Memory Disambiguation Through Store Distance Analysis

Changpeng Fang
QLogic Corporation
2071 Stierlin Court, Ste. 200
Mountain View, CA 94043 USA
changpeng.fang@qlogic.com

Steve Carr
Soner Önder
Zhenlin Wang
Department of Computer Science
Michigan Technological University
Houghton MI 49931 USA
{carr,soner,zlwang}@mtu.edu

ABSTRACT

Feedback-directed optimization has developed into an increasingly important tool in designing optimizing compilers. Based upon profiling, memory distance analysis has shown much promise in predicting data locality and memory dependences, and has seen use in locality based optimizations and memory disambiguation. In this paper, we apply a form of memory distance, called *store distance*, to the problem of memory disambiguation in out-of-order issue processors. Store distance is defined as the number of store references between a load and the previous store accessing the same memory location. By generating a representative store distance for each load instruction, we can apply a compiler/micro-architecture cooperative scheme to direct run-time load speculation. Using store distance, the processor can, in most cases, accurately determine on which specific store instruction a load depends according to its store distance annotation. Our experiments show that the proposed store distance method performs much better than the previous distance based memory disambiguation scheme, and yields a performance very close to perfect memory disambiguation. The store distance based scheme also outperforms the *store set* technique with a relatively small predictor space and achieves performance comparable to that of a 16K-entry *store set* implementation for both floating point and integer programs.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures; D.3.4 [**Programming Languages**]: Processors—*code generation*

General Terms

Languages, Performance

Keywords

memory disambiguation, store distance

ICS06, June 28–30, Cairns, Queensland, Australia

1. INTRODUCTION

Modern superscalar processors allow instructions to execute out of program order to exploit more instruction level parallelism (ILP). Because load instructions usually appear on the program's critical execution paths, processors must schedule loads as early as possible. However, to ensure the correctness of program execution, a load should not be scheduled ahead of any preceding stores that access the same memory location as the load. *Memory disambiguation* detects these memory dependences in order to avoid memory order violations while speculating load instructions.

Previous work has employed numerous static-analysis and hardware techniques to disambiguate memory references to improve performance [5, 10, 11, 15, 16, 17]. While many hardware techniques achieve good performance, most require a large on-chip memory to record memory dependences. In contrast, static dependence analysis does not require the large on-chip memory, but can only effectively identify load/store dependences for regular array references in loop nests. In this paper, we explore a compiler/micro-architecture cooperative solution to the memory disambiguation problem for superscalar processors. We give a feedback-directed mechanism based upon memory distance analysis to assist the processor in determining the exact memory dependences at runtime. The proposed scheme does not require a large on-chip memory and is not limited to array-based codes.

Memory distance is defined as a dynamic quantifiable distance in terms of memory references between two accesses to the same memory location. Fang et al. [9] define memory distance to include *reuse distance, access distance and value distance*. The *reuse distance* of a memory reference is defined as the number of distinct memory locations accessed between two references to the same memory location. Both whole-program [7, 28] and instruction-based [8, 9, 13] reuse distances have been predicted accurately across all program inputs using a few profiling runs. Reuse distance analysis has shown much promise in locality-based optimizations. *Access distance* is the number of memory references between a load and its dependent store, while *value distance* is the access distance with the consideration that a load depends only on the first store in a sequence of stores writing to the same address with the same value. Fang et al. [9] propose a feedback-directed mechanism based upon access distance and value distance to determine whether or not to speculate a load.

In this paper, we apply a form of memory distance, called *store distance* [26], to the problem of memory disambiguation in out-of-order issue superscalar processors. Store distance is defined as the number of store instructions between a load and the previous

store accessing the same memory location. Through profiling the program with one small input, we analyze the instruction-based store distance distribution and generate a representative store distance for each static load instruction. Then, a cost effective micro-architecture mechanism is developed for the processor to determine accurately on which specific store instruction a load depends according to its store distance annotation.

The proposed store distance based mechanism shows very good results across a set of SPEC CPU2000 benchmarks. Our experimental evaluations indicate that the store distance based method performs much better than the previous access distance based memory disambiguation scheme [9], and yields performance very close to perfect memory disambiguation, which uses exact knowledge of memory dependences for each dynamic load instance. The store distance based scheme also outperforms the *store set* technique with a relatively small predictor space and achieves performance comparable to that of a 16K-entry *store set* implementation [5] for both floating point and integer programs. In both cases, store distance requires a negligible amount of chip space (16 bytes in our implementation) compared to the several thousand bytes of chip space used by store set.

We begin the rest of this paper with a review of related work and background in the field of memory distance analysis and memory disambiguation. Next, we describe our analysis techniques and algorithms for measuring and analyzing store distance. Then, we present the micro-architectural considerations of our work and our experimental evaluation of store distance based memory disambiguation. Finally, we present our conclusions and propose future work.

2. RELATED WORK AND BACKGROUND

In this section, we first introduce relevant research in memory distance analysis and dynamic memory disambiguation. Then, we explain in detail two existing memory disambiguation approaches to which we will compare our proposed scheme.

2.1 Memory Distance Analysis

Given the high cost of memory operations in modern processors, compiler analysis and optimization of the memory performance of programs has become essential in obtaining high performance. To address the limits of static analysis, much work has been done in developing feedback-directed schemes to analyze the memory behavior of programs using various forms of memory distance. Mattson et al. [14] introduce reuse distance (or LRU stack distance) for stack processing algorithms for virtual memory management. Others have developed efficient reuse distance analysis tools to estimate cache misses [1, 4, 24, 27] and to evaluate the effect of program transformations [1, 2, 6]. Ding et al. [7, 21, 28] have developed a set of tools to predict reuse distance across all program inputs accurately, making reuse distance analysis a promising approach for locality based program analysis and optimizations. They apply reuse distance prediction to estimate whole program miss rates [28], to perform data transformations [29] and to predict the locality phases of a program [21]. Beyls and D'Hollander collect reuse distance distribution for memory instructions through one profiling run to generate cache replacement hints for an Itanium processor [3]. Marin and Mellor-Crummey [13] incorporate instruction-based reuse distance analysis in their performance models to calculate cache misses. Fang et al. [8, 9] introduce the notion of memory distance to encompass reuse distance, access distance and value distance. They propose a general analysis framework to predict instruction-based memory distance, and apply memory distance analysis to optimizations that require memory or data dependence information.

2.2 Memory Disambiguation

In order to get high performance, load instructions must be issued as early as possible without causing memory order violations. One way to accomplish this task is to use a memory dependence predictor to guide instruction scheduling. By caching the previously observed load/store dependences, a dynamic memory dependence predictor guides the instruction scheduler so that load instructions can be initiated early, even in the presence of a large number of unissued store instructions in the instruction window. Work in this area has produced increasingly better results [11, 16, 15, 5]. The problem of memory disambiguation and communication through memory has been studied extensively by Moshovos and Sohi [15]. The dynamic memory disambiguators proposed mainly use associative structures aiming to identify the load/store pairs involved in the communication precisely. Reinman et al. [20] propose using profile analysis to mark dependences between stores and loads via tags in store and load instructions to identify opportunities to communicate values between a store and its dependent load. They do not apply their technique to memory disambiguation. Various patents [23, 11] also exist that are aimed at identifying those loads and stores that cause memory order violations and synchronizing them when they are encountered.

Chrysos and Emer [5] introduce the store set concept which uses direct mapped structures without explicitly aiming to identify the load/store pairs precisely. With sufficient resources, the store set scheme provides near oracle performance [5] for a set of SPEC95 benchmarks. We choose this scheme as one of the bases for our evaluation and describe the algorithm and implementation in detail in Section 2.3.1. Yoaz et al. [26] present a dynamic store distance based technique that uses less space than store set, but does not perform as well. Önder and Gupta [19] have shown that the restriction of issuing store instructions in-order can be removed and store instructions can be allowed to execute out-of-order if the memory order violation detection mechanism is modified appropriately. Furthermore, they have shown that memory order violation detection can be based on values, instead of addresses. Önder [17] has proposed a light-weight memory dependence predictor which uses multiple speculation levels in the hardware to direct load speculation. This scheme outperforms store set algorithm when predictor space is small.

While the above schemes are based on memory dependence predictions, Fang et al. [9] have proposed a feedback-directed memory scheme which use memory distance prediction to determine whether or not to speculate a load instruction. We examine their approach in detail in Section 2.3.2.

2.3 Background

2.3.1 Store Set

The store set algorithm relies on the premise that future memory dependences can be correctly identified from the history of memory order violations. In this respect, the *store set* of a load instruction in a running program is defined to be the set of all store instructions upon which it has ever depended. The algorithm begins with empty sets, and speculates load instructions around stores blindly. When the micro-architecture detects a memory order violation, offending store and load instructions are allocated store sets and placed into their respective sets. When the load is fetched, the processor will determine which stores in the load's store set were recently fetched but not yet issued, and create a dependence upon these stores.

Since a load may depend upon multiple stores and multiple loads may depend on a single store, an efficient implementation of the concept may be difficult. In order to use direct mapped structures,

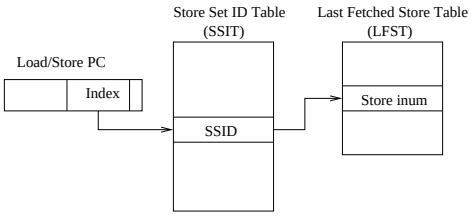


Figure 1: Store set implementation

Chrysos and Emer propose certain simplifying assumptions in their implementation which limit a store to be in at most one store set at a time and limit the total number of loads that can have their own store set. Furthermore, stores within a store set are constrained to execute in the program order. With these simplifications, they implement the store set algorithm using two direct mapped structures, as shown in Figure 1. The first is a PC indexed table called the *Store Set ID Table* (SSIT) which maintains store sets with a unique store set identifier (SSID) for each set. The second is called the *Last Fetched Store Table* (LFST) and contains dynamic identification number (store *inum*) of the most recently fetched store per store set. Recently fetched load/store instructions access the SSIT based on their PC and get their SSID. A load or store with a valid SSID will access the second table, the LFST, to get the *inum* of the store instruction that it should follow at the time of scheduling. A store instruction also updates its LFST entry with its own *inum*. When a mis-speculation is observed, the colliding load and store instructions are assigned the same SSID in their corresponding SSIT entries.

The store set algorithm provides near oracle performance when it is presented with a sufficiently large SSIT table. However, since the set information is collected dynamically there is no easy way to use the same mechanism during compile time to calculate and communicate instruction dependencies. To be effective, any compile time only mechanism should guarantee that the dependence information communicated to the hardware by the compiler will be invariant across different program inputs. The compiler can only do this if each store instruction is assigned a unique compile time identifier and the store set information is collected across all program paths. This would mean that each store set to which a load belongs would be overly conservative (a load could have been speculated successfully even though it is dependent on a store) and the sets would be prohibitively large for effective communication to the hardware by the compiler.

On the contrary, as a feedback-directed method, the proposed store distance based scheme performs off-line analysis to determine how the relevant memory dependences will manifest themselves in the store scheduling window. Doing so, the compiler does not need to assign a unique static store identifier. In other words, instead of encoding the exact dependencies among loads and stores where each is uniquely identifiable, the store distance technique encodes the distance that will be observed between a store and a dependent load in the scheduling window. Such information can be communicated to the hardware using a very small integer number. Changes that would occur with different data sets (i.e., when the program takes a different path the scheduling distance may change) can be handled using statistical analysis, as presented in the following sections.

2.3.2 Access Distance

Access distance based memory disambiguation is a feedback-directed method that identifies the load instructions that can be speculated at runtime using two profile runs of a program. For speculative execution, if a load is sufficiently far away from the previous store to the same address, the load will be a good speculation candidate. Otherwise, it will likely cause a mis-speculation and introduce penalties. The possibility of a mis-speculation depends on the distance between the store and the load as well as the instruction window size, the load/store queue size, and machine state. Taking all these factors into account, Fang et al. [9] define the *access distance* of a load as the number of memory references between the load and the previous store to the same memory address. They propose to speculate a load instruction if its access distance is greater than an empirical threshold. Having observed the change of access distance with data size, Fang et al. use two training runs with small inputs to estimate the instruction based access distance for a third input in evaluation.

For those load instructions with a constant access distance, the threshold is directly used to determine whether the load is *speculative* or *non-speculative*. Those load instructions whose access distance increases with data size are marked as *speculative* in favor of large inputs. All other load instructions are marked as *non-speculative* to reduce the possibility of mis-speculations. At runtime, the processor speculates the load instructions according to the speculative or non-speculative compiler annotation. The store distance based scheme that we present in this paper differs from the access distance based method in that it aims to find the specific store instruction on which a load depends, and thus is likely to be more accurate in memory dependence prediction. Additionally, the store distance based scheme requires only one training run on a very small input.

3. STORE DISTANCE ANALYSIS

In this section, we first introduce the concept of *store distance*. Then, we present our store distance analysis used for memory dependence prediction and load speculation for dynamically scheduled superscalar processors. At the end of this section, we briefly describe how to encode the store distance in a load instruction.

3.1 Store Distance

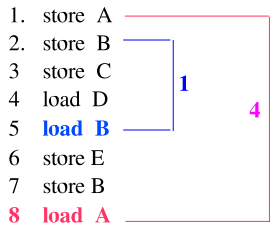


Figure 2: Store distance example

The *store distance* of a load is defined as the number of store instructions between the load and the previous store that accesses the same memory location. As shown in the code sequence in Figure 2, instruction 5 (load B) has a store distance of 1, while the store distance of instruction 8 (load A) is 4. Store distance is a program property that quantitatively specifies the dependence between a load and a store. For a load with a store distance of *n*, we can find the store (the *(n+1)*-st one) on which it depends by tracking the store execution trace in reverse program order. In modern superscalar processors, instructions are usually processed in program order in the front-end of the pipeline. If the store distance of a load instruction becomes known as it is decoded, it is not difficult for the processor to figure out the specific store instruction on which the load depends, and thus schedule the load accordingly. In this work, we use profiling with a small input to collect the store

distance of each load instruction, and then develop a novel compiler/architecture cooperative mechanism to direct the processor for efficient and accurate load speculation.

3.2 Store Distance Analysis

To simplify the compiler/micro-architecture interface, the compiler associates one store distance with each static load instruction, called the *summary store distance*. We have observed that, on average, over 82% of the load instructions in SPEC CPU2000 programs have just one store distance. For these instructions, the store distance is directly used as the summary distance. However, for the other instructions, multiple store distances for a static load instruction can result from a dependence change between different executions of a load in a single run. Many factors, such as indirect array references and conditional branches, may cause a dependence to change. For the instructions in this category, the compiler must judiciously select a summary distance to ensure correct speculations for most cases. We have developed an efficient summary store distance selection algorithm that we present later in this section. Our algorithm first collects store distance based on a profiling run on a small input. The summary distance is then chosen based on the distance distribution of each instruction.

To collect the store distance distribution for each static load instruction, we run a program using a small training input. To calculate the store distance, we keep a global store instruction counter to represent the *store cycle*. For each store, we insert its address into a hash table along with the current store cycle. Each load only needs to search the hash table and compare the current *store cycle* with the previous one for the same address to obtain its store distance. For the purpose of memory dependence prediction in superscalar processors, we only need to consider short store distances. If the store distance is greater than or equal to a threshold distance, *e.g.*, the reorder buffer size, the load instruction is not likely to depend on any stores in the current pipeline. We refer to this threshold distance as the *speculating distance*. If a load has a store distance greater than or equal to the speculating distance, we set the store distance to be the speculating distance. For such a load, the processor may always speculate it with a very low probability of incurring a mis-speculation.

To compute the store distance, we keep a counter for each distinct store distance ranging from 0 to the speculating distance. A store distance for a single load is considered as the *dominant distance* if the instances of this distance account for at least 95% of the total accesses for this instruction.

To select the summary store distance, we begin by initializing every load instruction's summary store distance to the speculating distance (rule 0). Then, according to the profiling results, if a load instruction has a dominant distance, we output this distance as its summary store distance (rule 1). Otherwise, we choose the minimum distance with a non-zero counter for its summary store distance (rule 2). Rule 0 applies to those load instructions not appearing in the training run, and we choose to speculate those load instructions always because blind speculation outperforms no speculation in most cases [5]. Rule 1 makes it likely that a dominant number of load instances will be correctly speculated. For rule 2, choosing the minimum distance reduces the number of mis-speculations. We illustrate the three rules using the following loop:

```
DO I = 1, N-1
  A(I) = B(I)
  B(I) = A(N) + A(I) + A(2)
```

No dependence exists from the store to `A(I)` to the load from `A(N)`, and `A(N)`'s summary store distance is set as the speculating distance. Additionally, the load from `A(I)` always depends on the previous store to `A(I)`, giving a store distance of 0. This indicates that `A(I)` should never bypass any preceding store instruction. Finally, the load from `A(2)` has two store distances: 0 when `I` equals 2 and a very large distance otherwise. Since the large store distance dominates in this case, our algorithm chooses to speculate `A(2)` always by using the speculating distance as its summary store distance. The store set approach cannot speculate in this case once a memory order violation occurs. However, the proposed approach correctly speculates a dominant number of load instances while incurring just a few mis-speculations. In Section 5, our evaluations show that the store distance based scheme outperforms store set in similar situations on several SPEC CPU2000 benchmarks.

Store distance based memory disambiguation must consider the case where the store distance changes across program inputs. A change in store distance implies that the summary store distance does not exactly represent the memory dependences for inputs other than the profiled one. Our method assumes that small store distances are independent of input size. Fang et al. [9] have observed that over 80% of the memory instructions in SPEC CPU2000 have constant memory distances. For short store distances, this percentage is even higher. If a dependence is loop-independent or carried by the innermost loop, or the loop bound is constant, the store distance of a load instruction is likely to be independent of the input size. To see why, consider the loops in Figure 3. The dependence between `A(I,J+3)` and `A(I,J)` in Figure 3(a) is carried by the innermost `J` loop, and the store distance of `A(I,J)` is 2 for all possible inputs. On the other hand, if the dependence is carried by an outer loop, as the case in Figure 3(b), the store distance is normally very large (on the order of `N` in the example loop). Even though the store distance may change with inputs, our scheme still considers it as constant because we can always speculate these load instructions likely without incurring mis-speculations since the store distance is greater than the speculating distance. As shown in Section 5, most loads have a store distance that can be considered constant for the purposes of memory disambiguation, whether they be constant across inputs or always larger than the speculating distance. In cases where the influence of input size upon store distances is not negligible, the memory distance analysis framework proposed by Fang et al. [9] can be applied to enhance our mechanism by predicting store distances.

Finally, we would like to note here that the store distance profiling in this work is cost effective. Only a single, small input is required for the profiling run and the cost to calculate the store distance for each reference is just a small constant. Because we only consider short store distances, the memory requirement is limited, only on the order of the number of static load instructions, which is normally just several thousand for a SPEC CPU2000 program.

3.3 Store Distance Encoding

In order to encode the store distance in a load instruction, we must first determine the store distance range under consideration, which depends on the speculating distance. If the speculating distance is too small, loads with a store distance beyond this range may cause mis-speculations. On the other hand, if the speculating distance is too large, the encoding will require too many bits in the instruction field. As discussed previously, all loads with a store distance greater than or equal to the speculating distance may be speculated blindly with a low probability of incurring memory order violations. The reorder buffer size is an upper bound for the speculating distance. For a specific machine configuration, the speculating distance is determined by machine parameters such as the width and depth of the pipeline, the instruction window size, the

```
DO I = 1, N
  DO J = 1, N-3
    A(I,J+3) = A(I,J) + 10.0
```

(a) Innermost loop carried dependence

```
DO I = 1, N-1
  DO J = 1, N
    A(I+1,J) = A(I,J) + 10.0
```

(b) Outer loop carried dependence

Figure 3: Constant store distance

number of memory ports and the load/store queue size, as well as some program properties like the store instruction density in the code sequence. While an analytical model may be difficult to obtain, in Section 5 we use an empirically determined value for the speculating distance by observing the number of mis-speculations for a set of benchmarks using various speculating distances .

Given that the summary store distance is in the range from 0 to *s*, where *s* is the speculating distance, we can use $\lfloor \log(s) \rfloor + 1$ bits of the offset field of a load to encode the store distance into the load instruction. Even though using bits from the offset field may increase register usage and address computation, Wang [25] has observed only a negligible performance difference (usually none) by reducing the offset from 16 to 12 bits in the Alpha instruction set.

4. MICRO-ARCHITECTURE DESIGN

With the summary store distance encoded in the load instructions, the major task of the micro-architecture design is to find the appropriate store upon which a load depends using this information. For this purpose, we implement a *store table*, as shown in Figure 4. Each entry of the table contains a store instruction's ID, which can uniquely identify this store in the current pipeline. In this work, we use the reorder buffer index to identify the load and store instructions. *Cur* is a pointer that points to the most recently decoded store instruction. The store table is addressed using *cur* and an offset. When a store instruction is decoded, *cur* is advanced and the store instruction puts its ID in the *cur* position of the store table. As a load instruction is decoded, it uses its summary store distance as the offset to get the ID of the store on which it is suggested to depend and remembers it in the reorder buffer for a later speculation decision. In our experiment, we implement the store table using a small circular buffer. If the encoded summary store distance is the speculating distance, the load instruction may be speculated blindly because no dependent store will be found in the table.

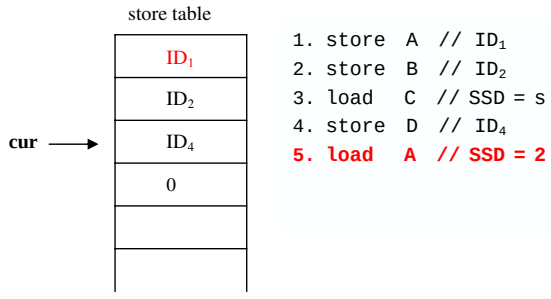


Figure 4: Store table implementation

For the example code sequence in the right-hand side of Figure 4, instruction 3 will always be speculated, and for instruction 5, because its store distance is 2, this load instruction depends on instruction 1 by using *cur* and 2 as an offset to address the store table.

As mentioned earlier, the encoded distance may not represent the store distances for all instances of a static load instruction, and mis-speculations are still possible, especially for cases where the actual store distance is shorter than the summary store distance. In this work, we use a mis-speculation handling mechanism very similar to that proposed by Önder and Gupta [19]. Detection of the memory order violations is performed by recording the load instruction's effective address, the load's ID and the producer store's ID in a table called the *speculative load table* when a load instruction is issued speculatively. If the load obtains the value from memory, its producer store's ID is set to a number out of the range of the reorder buffer index. Each load instruction is associated with an exception bit in the reorder buffer. This bit is set or reset by the store instructions. We allow the store instructions to be issued out of program order, and the checking of the speculative load table is delayed until the retire time of the store instructions. The speculative load table is probed with the address used by the store instruction. For a matching address, the producer store resets the exception bit and all other stores set this bit. This is to make sure only those stores that a load has actually bypassed can be effectively involved in memory order violation checking for this load. Once the load instruction is ready to retire, it checks its exception bit. If the bit is set, a roll-back is initiated and the fetch starts with the excepting load instruction. Otherwise, the load instruction's entry is deallocated from the speculative load table.

Once equipped with the ability to detect memory order violations and roll back appropriately, the micro-architecture becomes capable of exploiting dependences through memory. For various approaches the difference lies in how the speculating load instructions are selected. In this paper, we assume that the micro-architecture examines load and store instructions from oldest to youngest. If a store instruction is ready to issue and resources are available, it is issued. If a load instruction has completed address computation, *i.e.*, is ready to issue, it is issued unless the store in the store table on which it depends is not ready. Finally, if a load or store instruction has not completed its address computation, it is skipped.

In this paper, we use the same memory order violation handling mechanism and similar load/store issue logic for the various memory disambiguation schemes. Therefore, the major cost difference among the various schemes comes from the predictor space. The access distance based scheme uses no predictor and the proposed store distance based scheme uses an extremely small store table (16 bytes in our implementation) to resolve memory dependences. However, *store set* requires a large SSIT which usually consumes several thousand bytes of on-chip memory in order to achieve good performance.

5. EXPERIMENTAL EVALUATION

In this section, we explore the potential of using store distance as a metric to determine memory dependences in dynamically scheduled superscalar processors. We begin with a discussion of our experimental design and then examine the performance of store distance based memory disambiguation on a subset of the SPEC CPU2000 benchmark suite.

5.1 Experimental Design

To examine the performance of store distance based memory disambiguation, we use the FAST micro-architectural simulator based upon the MIPS instruction set [18]. The simulator models the superscalar pipeline and is cycle accurate. The major parameters of the baseline machine are shown in Table 1(a). We implement a 32KB directed mapped non-blocking L1 cache with a latency of 2 cycles and a 1MB 2-way set associative LRU L2 cache with an access latency of 10 cycles and a miss latency of 50 cycles. Both caches have a line size of 64 bytes.

To evaluate the relative performance of store distance, we have implemented six different memory disambiguation schemes which are listed in Table 1(b). For the access distance based scheme [9] as described in Section 2.3.2, we use the test and train input sets of SPEC CPU2000 for the two training runs. SS1K, SS4K and SS16K are three *store set* schemes with store set identifier tables of 1K, 4K and 16K entries, respectively. For store set, we implement a 256-entry last fetched store table, and apply *cyclic clearing* every one million instructions to invalidate all SSIT entries [5]. Perfect memory disambiguation never mis-speculates with the assumption that it always knows ahead the addresses accessed by a load and store operation.

For our test suite, we use a subset of the C and Fortran 77 benchmarks in the SPEC CPU2000 benchmark suite. The programs missing from SPEC CPU2000 include all Fortran 90 and C++ programs, for which we have no compiler, and 3 programs (254.gap, 255.vortex, and 200.sixtrack) which could not be compiled and run correctly with our simulator. For compilation, we use gcc-2.7.2 with the -O3 optimization flag. We use the test input sets for the training run to generate the summary store distance, and then examine the performance using the reference inputs. To reduce the simulation time, we fast forward the first one billion instructions and collect results for the next one billion instructions.

In this work, we add instrumentation to our micro-architectural simulator to collect the store distance statistics for every load instruction. However, tools like ATOM [22] and PIN [12] may be used on many architectures to reduce the profiling cost over simulation.

In this experiment, we augment the MIPS instruction set to include the summary store distance for the processor to determine which store instruction a load depends on at runtime. To encode the store distance of a load in the instruction, we use a speculating distance of 15 for our simulated micro-architecture. This is based on the observation that almost no mis-speculations are caused by those loads with a store distance greater than or equal to 15, even though they are blindly speculated. Given a speculating distance of 15, we use 4 bits from the 16-bit offset of a MIPS load instruction to encode the summary store distance.

5.2 Results

In this section, we report the results of our experiment on our benchmark suite. First, we report the store distance characteristics of SPEC CPU2000 benchmarks that we used in our experiment. Then, we report raw IPC data using a number of speculation schemes.

5.2.1 Store Distance Characteristics

As discussed in Section 3, store distance based memory disambiguation can often more effectively predict memory dependences if the store distance of a load does not change across inputs. A constant store distance allows store distance analysis to be applied to a single training run, rather than the multiple training runs used in previous memory distance analysis [7, 8, 13, 29]. An analysis of the benchmarks in our suite shows that 99.1% of the load instructions in CFP2000 programs and 94.9% of the load instructions in CINT2000 programs have constant store distances across test and reference input sets with over half of these distances being greater than or equal to the speculating distance used in our experiments. Note that we consider any distance greater than or equal to the speculating distance to be the speculating distance since anything larger than the speculating distance will allow speculation even if it grows with the data size. This result validates using a single training run to compute store distance.

Given the summary store distance from a single training run, the effectiveness of our scheme depends on how accurately the compiler generated summary store distance represents the actual store distance of the load instructions in the actual run. To determine the accuracy of store distance analysis, we compare the store distance of each instance of the load instructions for the reference input with its summary store distance, which is obtained from the training run using the test input. For a load instruction, if the actual store distance equals its summary one (EXACT), its dependent store instruction can be correctly identified at runtime and this load can be successfully speculated according to the micro-architecture design in Section 4. If the actual store distance is greater than the summary store distance (LONGER), our scheme will direct the load instruction to depend on a later store instruction, and false dependences may be introduced. Finally, when the actual store distance is less than the summary store distance (SHORTER), the load instruction may be mis-speculated. The goal of our summary store distance selection algorithm is to maximize the cases of EXACT to ensure correct speculation, while keeping SHORTER cases as low as possible in order to avoid mis-speculation.

We classify load instructions into the above three categories for our set of SPEC CPU2000 programs, as shown in Figure 5. For floating point programs, 95.4% load instructions fall into the EXACT category, and 0.2% fall into the SHORTER category, on average. 177.mesa is the only program that has more than 0.2% of the load instructions whose actual store distance is shorter than its summary store distance. This is because a number of the loads do not appear in the training run, making them speculated blindly. Some of these blind speculations, in turn, result in mis-speculations.

As shown in Figure 5(b), the integer programs have 88.3% EXACT and 1.1% SHORTER load instructions. This implies that the store distance based scheme may generate more mis-speculations and false dependences on the integer programs than on the floating point programs. 164.gzip, 253.perlbmk, 256.bzip2 and 300.twolf incur a larger percentage of loads in the SHORTER category than the other programs. For 256.bzip2 and 300.twolf, a number of loads do not appear in the training run, resulting in blind speculation. These loads then experience mis-speculations during execution on the reference input. For 164.gzip, in some cases the distribution of short and long reuse distances changes for a multiple store distance load. In these instances, a larger percentage of short reuse distances occur in the reference run due to variations in the execution paths taken. Finally, for 253.perlbmk the change in input from test to reference results in a different distribution of store distances that favor a shorter distance. Note that in each of the above cases, the misprediction of store distance only occurs for a very small percentage

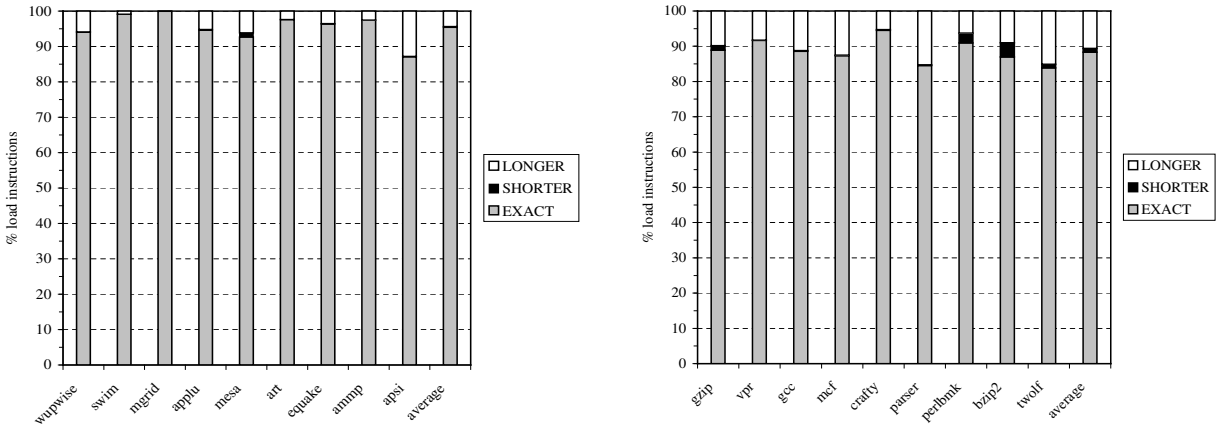
parameter	configuration
issue/fetch/retire width	8/8/8
instruction window size	128
reorder buffer size	256
load/store queue size	128
functional units	issue width symmetric
branch predictor	16K gshare
memory ports	2
data cache	L1: 32KB, D-Mapped L2: 1MB, 2-way

(a) Configuration

scheme	description
access	access distance
SD	store distance
SS1K	store set, 1K SSIT
SS4K	store set, 4K SSIT
SS16K	store set, 16K SSIT
perfect	perfect disambiguation

(b) Disambiguators

Table 1: Machine configuration and memory disambiguators



(a) CFP2000

(b) CINT2000

Figure 5: Instruction distribution around summary store distance

of loads. As shown in the next section, even with these cases store distance does an effective job of predicting memory dependences.

Whether or not a SHORTER load instruction will cause mis-speculation or a LONGER load instruction will cause a false dependence depends upon the runtime situation. For example, even if a load instruction with an actual store distance of 4 is speculated because its summary store distance is larger, the fifth previous store instruction may have already been executed when the processor issues the load instruction, resulting in no mis-speculation.

5.2.2 IPC

Given the effectiveness of store distance analysis as shown in the previous section, in this section we detail the runtime performance of store distance based memory disambiguation (SD). Overall, SD yields performance better than access distance based speculation (access), store set using a 1K SSIT (SS1K) and store set using a 4K SSIT (SS4K) for both the integer and floating-point programs in our benchmark suite. Specifically, on floating-point programs SD achieves a harmonic mean performance improvement of 9% over both access and SS1K and 4% over SS4K. On integer programs, SD obtains a harmonic mean improvement of 10% over access, 8% over SS1K and 4.5% over SS4K. When compared with store set with a 16K SSIT (SS16K) and perfect memory disambiguation (perfect), SD yields performance comparable to both. On floating-point programs, SD achieves a 1% harmonic mean improvement over SS16K and comes within 1% of the harmonic mean performance of perfect. On integer programs, SD comes within 1% of SS16K and within 2% of perfect. In the rest of this section, we focus on the relative performance of SD versus SS1K and SS16K since these latter two schemes give us a comparison with current well-known hardware techniques. We note the differences of SS4K and SS16K near the end of the section. For the performance of all disambiguation techniques, see Figures 6 and 7. In addition, Tables 2 and 3 detail the effectiveness of each disambiguation method in terms of the number of mis-speculations, the total number of speculations and the number of false dependences incurred.

We have found two main factors that contribute to a performance advantage for SD over both store set schemes. First, SD has an advantage when aliasing occurs in the SSIT. Since the SSIT is accessed using the PC as an index into the direct-mapped structure, multiple PCs map to the same entry in the table. This may cause a load and store that never incur a memory order violation to be mapped to the same SSID, preventing load speculation. Increasing the SSIT size and clearing the table periodically reduce the problem, but do not eliminate aliasing completely. As described in Section 3, the second factor giving SD a performance advantage is that dependences between a load and store may only occur occasionally. In this situation, SS1K and SS16K will cease load speculation upon the first occurrence of a memory order violation, even if the violation only occurs rarely thereafter. In contrast, SD recognizes

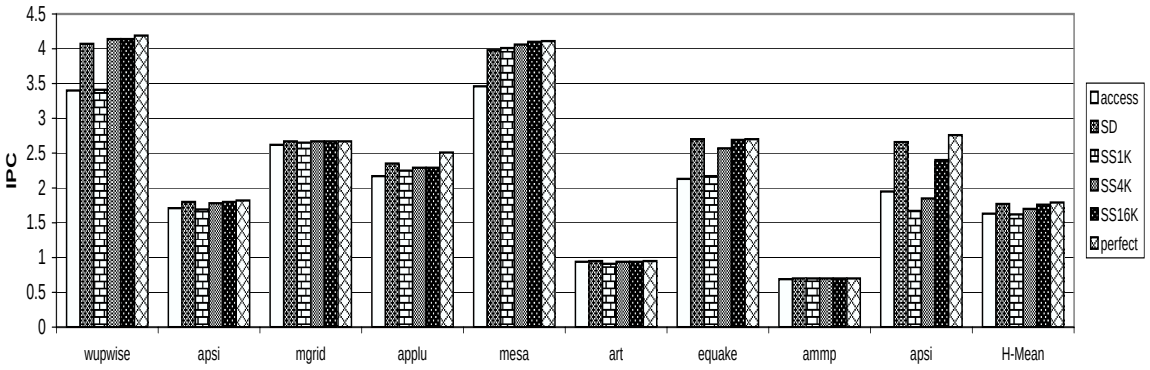


Figure 6: CFP2K IPC

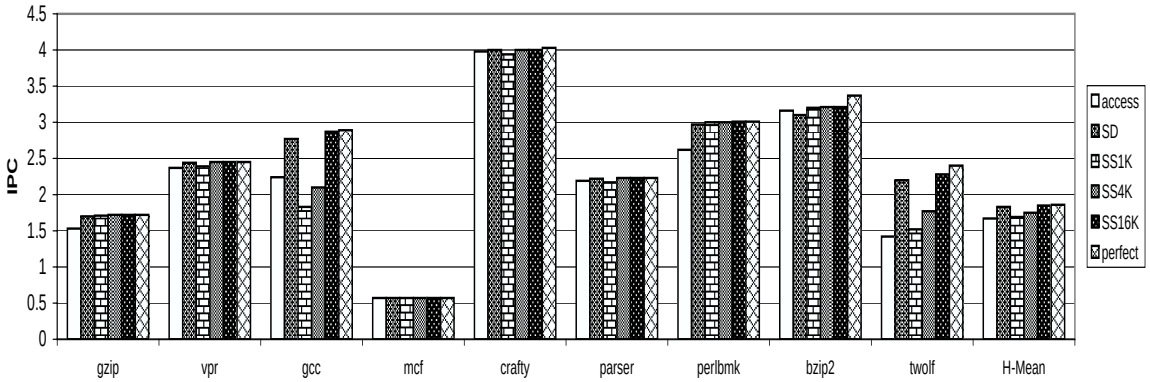


Figure 7: CINT2K IPC

the rarity of the dependence and sacrifices a few mis-speculations for increased performance in the majority of cases.

Several floating-point programs incur problems with aliasing that reduce performance. For SS1K, 168.wupwise, 171.swim, 179.art, 183.equake and 301.apsi exhibit a performance degradation with respect to SD due to aliasing. Table 2 shows that SS1K exhibits significantly more false dependences than SD. These dependences result from aliasing in the SSIT. For SS16K, only 301.apsi incurs a significant amount aliasing in the SSIT as shown by the increase in the false dependences when compared with SD.

173.applu exhibits the second phenomenon mentioned above. As reported in the original store set paper [5], some load instructions in 173.applu only depend on neighboring stores occasionally. Store distance analysis recognizes the rare occurrence of a dependences and allows speculation. Both SS1K and SS16K limit the number of speculations because of this frequently false dependence as illustrated in Table 2.

Considering integer programs, 176.gcc and 300.twolf exhibit significant aliasing for SS1K. This is due to a large number of static load instructions in both programs. However, SS16K is able to overcome this problem. Examining the number of false dependences per one thousand loads found in Table 3, SS1K incurs significantly more false dependences than both SD and SS16K with SD incurring more false dependences than SS16K.

In examining the cases where the store set technique performs better than SD, we have found three major reasons giving store set an advantage. First, SD must summarize the store distance with a single value, choosing a minimum store distance if a dominant distance does not exist. In some cases, the store distance of a load correlates to the execution path in the program. Consider the example in Figure 8. If execution follows path 1, the load at instruction 9 exhibits a store distance of 5. However, if execution follows path 2, the load exhibits a store distance of 1. If neither distance is dominant, SD selects 1 as the store distance. Thus, every time execution follows path 1, a false dependence may occur. The store set algorithm puts both instructions 1 and 7 into the store set for instruction 9. When execution follows path 1 and instruction 1 executes, issuing of instruction 9 is possible. In other words, store set makes a load wait only as long as necessary.

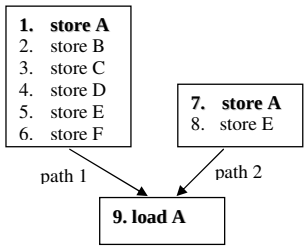


Figure 8: Path-correlated store distance

The second reason causing SD to perform worse than store set occurs when load instructions do not appear in the training run. In this case, SD may incur more mis-speculations than store set due to blind speculation, reducing performance. Finally, the third phenomenon giving store set an advantage occurs when the influence of input size change causes SD to incur an increase in false dependences. This occurs due to a failure to detect that the store distance has increased with the data size. This last phenomenon can be overcome using the memory distance prediction developed by Fang et al. [9].

Benchmark	mis-speculation				total speculation				false dependences			
	SD	SS1K	SS4K	SS16K	SD	SS1K	SS4K	SS16K	SD	SS1K	SS4K	SS16K
168.wupwise	2.44	1.96	0.13	0.54	874	771	873	874	30.62	91.2	3.83	1.88
171.swim	0.02	0.27	0.07	0.04	964	930	959	964	0.32	30.54	12.16	2.63
172.mgrid	0.12	0.05	0.01	0.01	816	813	816	816	0.03	1.41	0.29	0.16
173.applu	2.62	0.97	0.01	0.01	851	832	815	815	28.88	54.42	55.5	58.56
177.mesa	2.64	1.32	0.91	0.05	713	710	715	718	6.68	13.28	2.42	0.25
179.art	0.02	0.04	0.04	0.04	724	697	724	724	0.30	30.25	0.01	0.01
183.equake	0.00	14.44	2.24	0.07	550	478	522	551	0.00	18.43	5.62	0.10
188.ammp	1.09	0.25	0.35	0.11	149	134	149	152	0.16	7.43	1.64	0.03
301.apsi	2.64	18.16	12.68	0.08	806	498	582	758	37.25	117.34	107	53.73
Average	1.29	4.16	1.83	0.11	716	651	684	708	11.58	40.48	20.94	13.04

Table 2: Mis-speculations, total speculations and false dependences per thousand loads for CFP2K

Benchmark	mis-speculation				total speculation				false dependences			
	SD	SS1K	SS4K	SS16K	SD	SS1K	SS4K	SS16K	SD	SS1K	SS4K	SS16K
164.gzip	2.03	0.79	0.04	0.04	265	273	277	277	39.84	11.55	11.48	11.45
175.vpr	0.16	1.31	0.66	0.65	430	425	437	437	5.28	16.15	2.98	0.65
176.gcc	5.50	7.98	7.08	0.24	587	298	421	600	5.78	150.96	104.01	2.49
181.mcf	0.00	0.01	0.01	0.01	608	608	608	608	28.18	27.89	27.75	27.71
186.crafty	0.10	1.44	0.08	0.08	242	235	243	243	0.88	6.10	0.32	0.17
197.parser	0.66	2.13	0.12	0.06	330	326	343	345	8.42	19.03	1.04	0.71
253.perlbmk	0.06	0.62	0.17	0.17	385	384	389	390	6.67	11.26	2.71	1.26
256.bzip2	8.53	0.05	0.04	0.04	380	382	383	383	14.30	41.56	39.33	39.33
300.twolf	5.47	6.11	4.26	0.89	631	300	466	649	35.90	100.68	63.1	14.21
Average	2.50	2.27	1.38	0.24	429	359	396	437	16.14	42.83	28.08	10.89

Table 3: Mis-speculations, total speculations and false dependences per thousand loads for CINT2K

When comparing SD and SS16K, SD suffers from issues related to path-correlated store distance on 168.wupwise, 176.gcc and 300.twolf. However, the negative effects of multiple store distances counter balance the effect of aliasing in SS1K giving SD an advantage. SD performs worse than both store set schemes on 256.bzip2. This occurs since 11% of the load instructions appearing in the reference run do not appear in the test run. In these cases, SD's use of blind speculation yields a performance degradation. Finally, SD performs worse on 177.mesa because the store distance calculation is sensitive to the change in data size between the test and reference input sets.

As mentioned previously, Tables 2 and 3 summarize the number of mis-speculations, total speculations and false dependences for SD, SS1K and SS16K. In general, SD incurs fewer mis-speculations and false dependences and more total speculations than SS1K on floating-point programs with the same trend on integer programs except that SD incurs slightly more mis-speculations than SS1K. Compared with the performance of SS16K on floating-point programs, SD incurs fewer false dependences and more total speculations, but incurs a higher number of mis-speculations. On integer programs, SD incurs a higher number of mis-speculations and false dependences and fewer total speculations than SS16K. The reason that SD does better on floating-point programs than on integer programs can be traced to Figure 5 in Section 5.2.1. As shown in this figure, the predicted store distance falls into the EXACT category more often for floating-point programs than integer programs. Thus, store distance analysis is more accurate for floating-point codes and can yield better performance results.

SS4K and SS16K yield similar results except on four programs: 176.gcc, 300.twolf, 183.equake and 301.apsi. On each of these four programs SS4K exhibits significantly more mis-speculations and false dependences than SS16K. In these cases, the reduction in table size yields an increase in aliasing, resulting in significantly lower performance.

SD outperforms access by over 8% on both integer and floating-point programs. Access yields better performance than SD on only one program – 256.gzip. In 256.bzip2, most load instructions appear in only one of the runs using the test or train inputs. Because access uses both the test and train input sets, access is able to use one or the other input set measurements rather than use blind speculation as SD does. SD could regain the performance loss if we chose to use multiple training runs to compute store distance. In all programs other than 256.bzip2, the additional hardware and dynamic dependence checking performs better than the single bit used to denote speculative loads in access.

Finally, we note that using fewer than 4 bits to encode the store distance yields worse results for SD. Using 3 bits for the store distance decreases the performance of SD between 8 and 10% over using 4 bits. Additionally, using 5 bits to encode the store distance yields no appreciable improvement.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have developed a novel compiler and micro-architecture cooperative mechanism for dynamic memory disambiguation in superscalar processors. Based upon store distance analysis, the compiler generates a representative store distance for each load instruction and passes this information through an instruction annotation to the micro-architecture. Guided by the store distance annotation, the processor can accurately identify the specific store instruction on which a load depends and make speculation decisions accordingly.

The store distance based mechanism shows very promising results across a set of SPEC CPU2000 benchmarks. Our experimental evaluations indicate that the store distance based method performs much better than the access distance based memory disambiguation scheme and yields a performance very close to perfect

memory disambiguation. The store distance based scheme also outperforms the store set technique with a small predictor space and achieves a performance comparable to a 16K-entry store set implementation for both floating point and integer programs.

We are currently adding path information to our computation of memory distance to help disambiguate store distances on multiple paths. In the future, we plan to incorporate this analysis into our speculation scheme to enhance store distance based memory disambiguation.

Acknowledgments

This work was partially supported by NSF grant CCF-0312892.

7. REFERENCES

[1] G. Almasi, C. Cascaval, and D. Padua. Calculating stack distance efficiently. In *Proceedings of the first ACM Workshop on Memory System Performance*, Berlin, Germany, 2002.

[2] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2001.

[3] K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Proccedings of the 8th International Euro-Par Conference*, August 2002.

[4] C. Cascaval and D. Padua. Estimating cache misses and locality using stack distance. In *Proceedings of the 17th International Conference on Supercomputing*, pages 150–159, San Francisco, CA, June 2003.

[5] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Conference on Computer Architecture*, pages 142–153, June 1998.

[6] C. Ding. *Improving effective bandwidth through compiler enhancement of global and dynamic reuse*. PhD thesis, Rice University, 2000.

[7] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, California, June 2003.

[8] C. Fang, S. Carr, S. Önder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of the Second ACM Workshop on Memory System Performance*, pages 60–68, Washington, D.C., June 2004.

[9] C. Fang, S. Carr, S. Önder, and Z. Wang. Instruction based memory distance analysis and its application to optimization. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, September 2005.

[10] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Ontario, June 1991.

[11] J. Hesson, J. LeBlanc, and S. Ciavaglia. Apparatus to dynamically control the Out-Of-Order execution of Load-Store instructions. *US. Patent* 5,615,350, Filed Dec. 1995, Issued Mar. 1997.

[12] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, June 2005.

[13] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, New York, NY, June 2004.

[14] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[15] A. I. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin - Madison, 1998.

[16] A. I. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Conference on Computer Architecture*, pages 181–193, June 1997.

[17] S. Önder. Cost effective memory dependence prediction using speculation levels and color sets. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 232–241, Charlottesville, Virginia, September 2002.

[18] S. Önder and R. Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, IL, May 1998.

[19] S. Önder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. *Journal of Instruction Level Parallelism*, Volume 4, June 2002. (www.microarch.org/vol4).

[20] G. Reinman, B. Calder, D. Tullsen, G. Tyson, and T. Austin. Classifying load and store instructions for memory renaming. In *Proceedings of the 13th Annual ACM International Conference on Supercomputing*, pages 399–407, Rhodes, Greece, June 1999.

[21] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, Boston, MA, Oct. 2004.

[22] A. Srivastava and E. A. Eustace. Atom: A system for building customized program analysis tools. In *Proceeding of ACM SIGPLAN Conference on Programming Language Design and Inplementation*, June 1994.

[23] S. Steely, D. Sager, and D. Fite. Memory reference tagging. *US. Patent* 5,619,662, Filed Aug. 1994, Issued Apr. 1997.

[24] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 24–35, Santa Clara, CA, May 1993.

[25] Z. Wang. *Cooperative hardware/software caching for next-generation memory systems*. PhD thesis, University of Massachusetts, Amherst, 2004.

[26] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 42–53, Atlanta, GA, May 1999.

[27] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Language, Compilers, and Runtime Systems for Scalable Compilers*, Washington, DC, 2002.

[28] Y. Zhong, S. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 91–101, New Orleans, LA, September 2003.

[29] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Washington, D.C., June 2004.