Cost Effective Memory Dependence Prediction using Speculation Levels and Color Sets *

Soner Önder
Department of Computer Science
Michigan Technological University
Houghton, MI 49931-1295
{soner@mtu.edu}

Abstract

Memory dependence prediction allows out-of-order issue processors to achieve high degrees of instruction level parallelism by issuing load instructions at the earliest time without causing a significant number of memory order violations. We present a simple mechanism which incorporates multiple speculation levels within the processor and classifies the load and the store instructions at run time to the appropriate speculation level. Each speculation level is termed as a color *and the sets of load and store instructions are called* color sets*. We present how this mechanism can be incorporated into the issue logic of a conventional superscalar processor and show that this simple mechanism can provide similar performance to that of more costly schemes resulting in reduced hardware complexity and cost. The performance of the technique is evaluated with respect to the store set algorithm.*

At very small table sizes, the color set approach provides up to 21 % better performance than the store set algorithm for floating point Spec-95 benchmarks and up to 18 % better performance for integer benchmarks using harmonic means.

Keywords: *load speculation, memory dependence prediction, store sets, wide issue superscalar, speculative execution.*

1 Introduction

Future generation superscalar processors will attempt to exploit even higher degrees of instruction level parallelism (ILP) than what is deemed possible today. By employing a larger fetch, decode and issue width, these processors will expose a larger number of load and store instructions which need to be disambiguated and scheduled for execution. In such a setting, the traditional approach of delaying the scheduling of load instructions until all prior store addresses become known is not appropriate as this approach results in significant losses in attainable ILP. Although there are a number of different ways in which early scheduling of the load instructions can be achieved [12, 1], memory dependence prediction appears to be one of the most viable approaches, as demonstrated by the effectiveness of simple dependence predictors deployed in some contemporary superscalar processors [7]. Ongoing research suggests that there is more to be gained beyond what is possible with these simple predictors [2, 5, 8, 13].

Memory dependence prediction is an effective technique since it allows early issuing of the load instructions. When the addresses of the preceding store instructions in the instruction window are not yet available, brute force disambiguation of pending loads with respect to these addresses is not possible. Memory dependence prediction allows the scheduler to decide if a given load instruction is independent of the existing store instructions without a need to have the memory addresses ready. The scheduler can therefore schedule a given load instruction with high confidence that it will not collide with the existing stores. If the processor later discovers that issuing the load caused a memory order violation, recovery is initiated by using either re-execution or roll-back approaches.

It is possible to classify the existing work into three main approaches. We will refer them as the *independence prediction*, *pairing based* and *set based* approaches. With the independence prediction, a prediction is made for each load instruction that is ready to issue to see if the load instruction is independent of the existing store instructions. In other words, an attempt is made to predict whether or not there is a read-after-write dependency between the load instruction and a store instruction in the instruction window. If the predictor predicts independence, the load instruction is issued.

*Supported by DARPA, Power Aware Computing/Communications Program, Award no. F29601-00-1-0183.

Otherwise, the issuing of the load instruction is delayed until all prior store addresses become known. Note that this technique effectively creates two sets of load instructions; those that can be safely scheduled early and those that were known to collide with the unissued store instructions. As a result, only a single bit of information per load instruction is sufficient to represent the set in the hardware.

While independence prediction uses a very simple mechanism to predict if a given load instruction is independent of the unissued store instructions, it is highly conservative. The mere fact that a given load instruction has collided with an unissued store instruction in the past does not necessarily mean that it will do so in the future since the colliding store may compute a different address or may not even be present in the instruction window during a future execution of the same load instruction. As a result, later work has shown that there is great benefit in constructing dependence pairs and identifying the memory dependences among the store and the load instructions precisely [8].

More recent work involving the notion of *store sets* [2] has shown that instead of pairing, constructing sets of conflicting memory instructions may be more appropriate. As opposed to identifying precise pairs, the store set approach maintains a set per load instruction in which all store instructions that collided with that load instruction are placed. During the execution, if a load instruction's store set is not empty and any of the unissued store instructions is a member of the load instruction's store set, the issuing of the load instruction is delayed until the condition is cleared. This approach has been shown to yield performance close to that of an oracle for an 8-issue processor.

In this paper, we propose a cost effective approach to memory dependence prediction problem. The key to our approach is to employ multiple speculation levels within the processor, termed as *speculation colors*, or simply *colors*. These *colors* divide the load instructions into distinct sets, starting with the base color which corresponds to the *no violation* case. In other words, this set is the set of load instructions which have never collided with unready store instructions in the past. Each color in the spectrum represents increasing levels of aggressiveness in load speculation; a load instruction is allowed to issue only if its color is *less than or equal to* the current speculation level. If the processor later discovers that the load has collided with a store, the color assigned to the load instruction in the predictor is increased. In summary, instead of attempting to predict dependence/independence for a given load instruction, we predict the color (i.e., the speculation level) a given load instruction will successfully speculate.

The processor's speculation level is adjusted by observing the memory port utilization, as well as the presence of the store instructions which have collided with the speculative load instructions in the past. For example, if there is ample amount of base color loads at a given point in execution, (i.e., memory ports are fully utilized without scheduling the risky loads) no attempt is made to speculate the risky loads and the processor remains at the base color. On the other hand, if the memory ports are under utilized, the processor allows loads to execute speculatively from the next color level. This approach results in very effective use of the predictor space since only a few colors are sufficient to capture most of the ILP that can be obtained.

In the remainder of the paper, we first present some of the related work and discuss the issues involved in memory dependence prediction in Section 2. Next, in Section 3, the essential elements of our algorithm are discussed. Section 4 presents an implementation of the color sets approach. Finally, in Section 5, we present an experimental evaluation of the color sets approach with respect to the store set algorithm.

2 Memory Disambiguation Problem

The problem of memory disambiguation may be stated as deciding at any given point in time whether the execution of a given load instruction would cause a memory order violation. When the store addresses of all the preceding stores are available, a simultaneous comparison of the load address with respect to older stores yields a decisive answer about whether executing the load at this point in time would be an error. On the other hand, besides the obvious hardware complexity involved, waiting for all the prior store instructions to compute their addresses is overly conservative since in many cases there is no dependence between a load and the preceding store instructions. As a result, it makes sense to predict the dependencies and to issue the load instructions speculatively well before the store addresses become known.

The simplest form of a memory dependence predictor is a predictor that always predicts independence. In such a setting, the processor issues load instructions as soon as they compute their addresses and rolls back as collisions happen. This approach is called *blind speculation* [3]. Blind speculation almost always improves performance at the expense of increased power consumption of the processor, but comes at no hardware cost.

A better approach is to identify the *bad loads* that happen to collide with the preceding stores. In this case, a one bit predictor is used which can be implemented either as a separate predictor, or as part of the instruction cache by adding a single bit to each cache word. This is the approach taken in contemporary processors such as the Alpha 21264 processor. Upon detecting a memory order violation, the processor marks the load instruction so that when the same load instruction is encountered for the second time, it waits for all the preceding store instructions before issu-

ing [7]. Alternatively, the processor can remember the store instructions which happen to collide with the nearby load instructions and the processor does not speculatively issue load instructions until these store instructions execute. This is the approach taken in store barrier cache approach [5]. The problem with these approaches has been that they create a lot of *false dependencies*. False dependencies are a significant reason for loss of parallelism since even a single dependence can hold up the execution of the rest of the load/store instructions for long time while these instructions may be entirely independent.

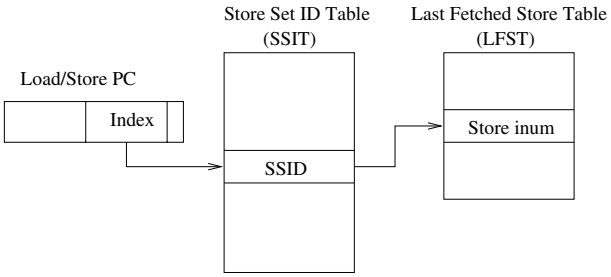


Figure 1. Store Set Implementation.

Having realized the shortcomings of these approaches, advanced techniques have been proposed. Instead of creating *barriers* which indiscriminately hold all the load instructions, these techniques record and utilize the actual dependencies among the load and store instructions. Since the actual dependence information is available, these predictors not only provide an answer to whether or not there is a dependence, they can be integrated with the issue logic and tell exactly when the condition clears by chaining dependent instructions using hardware pointers.

We pay special attention to one such algorithm, namely the store set algorithm. Since it provides near oracle performance for an 8-issue processor, we selected it as our base case and included an outline of the algorithm here.

The basic idea behind the store set algorithm is to make sure that if a load instruction and a store instruction collides (i.e., there is a dependence between them and the load instruction was mis-speculated), the algorithm remembers this fact and never schedules a colliding load before the store instruction that the load has collided with in the past. In order to coalesce many dependencies together, the algorithm constructs sets of store instructions named *store sets* per load instruction. In this respect, a *store set* is defined to be the set of store instructions a load has ever depended on.

The algorithm starts with empty sets, and blindly speculates load instructions around stores. When memory order violations are detected, the offending store and the load instructions are allocated and placed in store sets. In general, a load may depend upon multiple stores and multiple loads may depend on a single store. In order to obtain a simple implementation, a store instruction is allowed to be in at most one store set at a given time. Furthermore, stores within a store set are constrained to execute in order. With these simplifications, only the two directly mapped structures shown in Figure 1 are needed to implement the desired functionality [2].

At this point, it is important to note that while simple barrier style predictors suffer from a significant number of false dependencies, dependence based approaches typically require large tables in order to accommodate the precise dependence information. What we would like to have is the space efficiency of simpler predictors with the performance of dependence based predictors.

3 Color Sets

Even though it may appear that having the performance of dependence based predictors with the space efficiency of simple dependence predictors is not a realistic goal, a careful analysis of the memory dependence problem yields some hope:

1. The false dependencies imposed by simple predictors can be alleviated by having multiple levels of barriers which may provide similar performance to that of more precise information.
2. The dependence information provided by advanced predictors for scheduling purposes can be mimicked by having an appropriate delay in the execution of a load instruction.

The color sets approach utilizes both of these observations to yield a cost effective scheme. In order to address the false dependence problem, we start with the basic single bit independence predictor and first extend it to multiple levels. Instead of the two states, namely, *speculate* and *don't speculate*, we discriminate between loads which result in many collisions and those that occasionally collide. In other words, we employ multiple speculation levels within the processor, termed as *colors*. These colors represent a spectrum of speculation levels starting with the base color which corresponds to the no collisions case, whereas each color in the spectrum represents increasing levels of collisions in load speculation. Effectively, the scheme establishes sets of instructions and barriers for each respective set.

Since the precise dependence information would not be available, it is not possible to make dependent loads wait for the corresponding store instructions using hardware pointers. Instead, one has to rely on some mechanism to sufficiently delay the scheduling of the loads until such time that they won't collide with their dependent store instructions. Our approach is to use the available parallelism indicators so that the processor speculates the *risky instructions*

only when there is not enough supply of safer ones. This technique in most cases provides the required delay as most collisions occur within a window frame of just a few cycles. In case the provided delay through this mechanism is not sufficient, these loads collide again and are pushed to the next set which is polled less often. Unfortunately, this simple mechanism alone is not sufficient to provide the necessary delay. In practice, we observed that there are many cycles during which there is not enough supply of less risky load instructions. Therefore, the processor rapidly moves to the next set and starts scheduling risky load instructions. If during this time colliding store instructions are also present in the instruction window, collisions happen repeatedly.

Our solution to this problem is to make use of the colliding store information to adjust the speculation level. For this purpose, we developed two policies. In the first policy, when a store instruction enters the instruction window, it does not let the processor color move beyond its own. By assigning a lower color to colliding stores and the current processor color to safer ones, the processor speculates only those sets of loads which are not likely to cause memory order violations with the existing store instructions. We refer to this policy as *color set basic* policy. In the second policy, we specifically mark the colliding store instructions using the highest color value. In this policy, when a colliding store instruction enters the instruction window, it decrements the processor color and the processor moves to the highest color as soon as these store instructions compute their addresses. We refer to this policy as *color set aggressive*. In the evaluation section, both policies are evaluated and each are shown to have benefits depending on the available hardware resources.

Having described how the scheduling of the load and store instructions can be controlled without having precise dependence information, we now outline the basic constraints and the steps of the algorithm for the basic policy:

1. Initially, all the loads and the stores have the base color.

2. The current scheduling color is increased when the memory port utilization is low, i.e., there are not enough ready, well-behaved load instructions in the instruction window.

3. When there are preceding store instructions in the instruction window whose addresses are not yet known, the processor schedules only those load instructions which have a color less than or equal to the current scheduling color. When all the preceding store addresses are known, a load instruction issues regardless of its color.

4. Upon completion, a load instruction that is executed successfully updates its color to the current scheduling color. A colliding load instruction sets its color to a value larger than the current scheduling color. If the load is already at the maximum available color, the current color is maintained.

5. Upon completion, a store instruction sets its color entry to the current scheduling color if it did not collide with a speculated load instruction.

6. A colliding store instruction sets its color entry to one less than the color assigned to the load which collided with the store. In other words, a colliding store and a colliding load instruction are assigned to different color sets such that the load instruction gets the higher color and the store instruction gets the lower color.

7. When a store instruction enters the instruction window, the current scheduling color is set to the store instruction's color if store instruction has a lower color. Otherwise, the current scheduling color is unaffected.

The purposes of steps 1 and 2 have already been explained. Constraints 3 and 4 together make sure that load instructions speculate at the color they have been known to successfully speculate, as well as make sure that load instructions with a high color will not starve. Also, if a load instruction occasionally collides, it will first be assigned to a high color. If, during a later execution it is successfully executed at a lower color because of constraint 3, it will be placed into a lower ranking color. In this way, an occasional collision will not limit parallelism during the future executions of the same load.

Constraints 5, 6 and 7 operate in tandem; a non-colliding store instruction maintains the scheduling color where it is known to execute without a collision (and hence the scheduling of risky load instructions which do not collide with the scheduled store instruction). On the other hand, a colliding store instruction lowers the scheduling color and thus inhibits the scheduling of the loads it had collided with in the past.

For the aggressive policy, we modify the steps 2, 5, 6 and 7 as follows:

1. The current scheduling color is set to maximum when the memory port utilization is low, i.e., there are not enough ready, well-behaved load instructions in the instruction window.

2. Upon completion, a store instruction does not modify its color entry if it did not collide with a speculated load instruction.

3. A colliding store instruction sets its color entry to the maximum color. In other words, the maximum color for a store instruction means *colliding store*.

4. When a colliding store instruction enters the instruction window, the current scheduling color is decremented. Otherwise, the current scheduling color is unaffected.

Having laid out the fundamental aspects of the algorithm and the details of both policies, we present an implementation of the algorithm in the next section.

4 Color Set Implementation

Color set implementation consists of a global color register, a simple two-bit predictor shown in Figure 4 and a chain of or gates in the issue window of the processor.

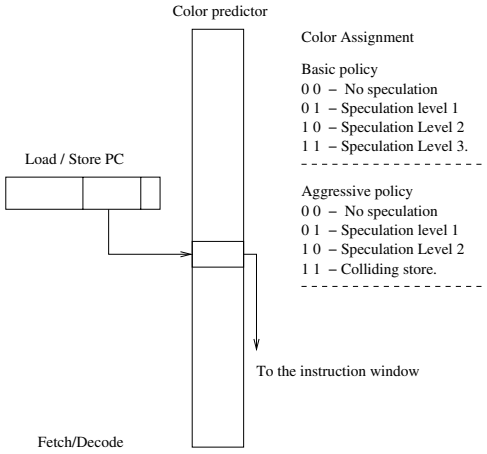


Figure 2. Color Set Predictor.

The color set predictor is similar to the predictors used for branch prediction. When a load or store instruction is fetched and decoded, its PC is used to access the predictor. The instruction copies the color value obtained and carries the value as part of the control information associated with the instructions. In a reservation station, in addition to the usual data dependencies, a load instruction has to watch for the value in the global scheduling register versus its own color. A load is ready to execute if it has satisfied its data dependencies and either all prior store addresses are known or the global scheduling color is greater than or equal to the load's own color.

The necessary changes that need to be incorporated into the instruction window logic is shown in Figure 3. For the purpose of clarity, we show the additional logic for a single entry of the issue logic in the figure. Instructions enter the window from the bottom, but any instruction in any position may be issued at any time. It is also assumed that the window is collapsed after performing a dispatch, much like the Alpha 21264 processor. When a slot is occupied by a store instruction, the inhibit flag is set as long as the store instruction has not yet computed its address. This signal is propagated to younger cells in a chain of *or* gates. Each cell therefore performs a color compare and performs a logical or of the result and the negation of preceding store instructions' inhibit signal. As a result, a load instruction is ready to issue when all the preceding store instructions have computed their addresses, or it has a color less than or equal to the global scheduling color.

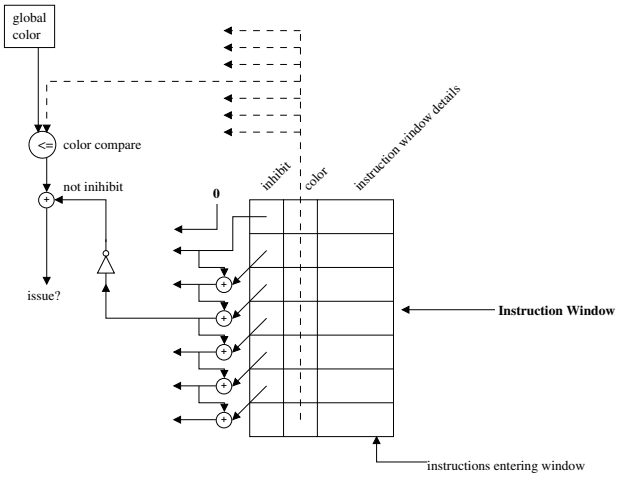


Figure 3. Instruction window extensions.

As described before, the handling of the store instructions is different for the two policies. For the basic policy, a store instruction upon entering the reservation station sets the global scheduling color if its color is less than the global scheduling color. With the aggressive policy, only a colliding store instruction affects the scheduling color. In both policies however, the store instruction waits in the reservation station for any data dependencies normally and issues when it becomes ready. No specific ordering of the store instructions is maintained since we also employ the out-of-order store issuing mechanism [10, 11]. For completeness, we include a brief summary of the technique here:

1. The checking for exceptions in case of memory references is delayed until the store retires.

2. Each load that is issued speculatively makes an entry in a table called the *speculative loads table* where the address and the value the load instruction obtained are stored.

3. In the reorder buffer, each load instruction is associated with an exception bit. As a store instruction retires, its address is compared to those in the speculative loads table as well the value it has stored.

   If the addresses match and values differ, it sets the exception bit associated with the load.

If the addresses match and values match, it resets the exception bit associated with the load.

If the addresses do not match, no action is taken.

4. Once the load 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 loads table.

As it can be easily seen, the exception bit may be set by a number of store instructions if they reference the same memory location, but store a different value. However, if there is no memory order violation, the last store to the memory location will have both an address and a value match and will reset the exception bit. In other words, this mechanism precisely identifies the provider store instruction.

For an implementation using a two bit predictor as outlined above, the set of color states for the processor and the load instructions for the basic policy are shown in Figure 4. In the figure, the set of load instructions eligible for scheduling when the processor color is 01 are shown in the shaded region.

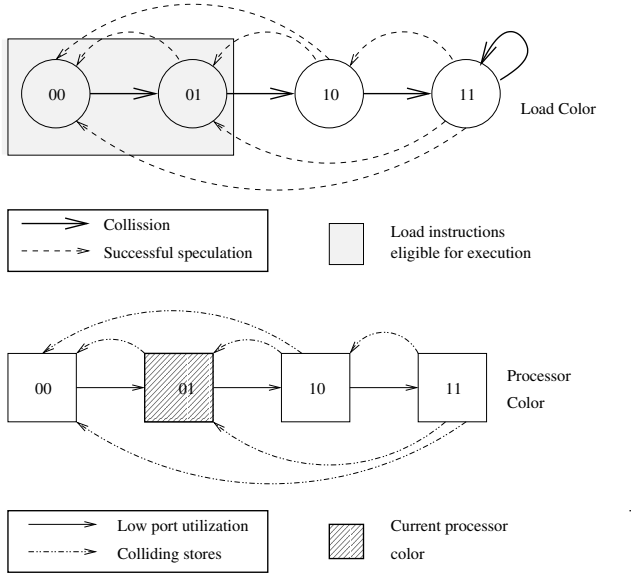


Figure 4. Load instructions and Processor Color States.

As it can be seen from the figure, with the basic policy, the processor increases its color only when the memory port utilization is low, but may jump from any state to any state upon encountering a store instruction that demands a lower speculation level. Similarly, the color assigned to a given load instruction will be increased upon a collision, but may be set to any lower value upon successful execution at a lower color.

5 Experimental Evaluation

In order to evaluate the performance of the color sets approach, three processor descriptions written in the ADL language [9] have been developed and the simulators have been automatically generated from these descriptions using the FAST simulation system. Simulators model the basic superscalar pipeline shown in Figure 5 and are cycle accurate.

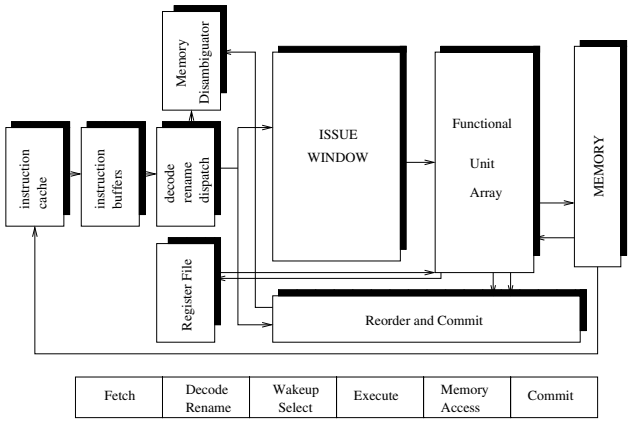


Figure 5. Machine Model.

We kept the machines with the store set predictor and the color set implementations identical in all aspects except the memory disambiguator. Modeled superscalar processors employ an aggressive multi-block instruction fetcher that delivers up to 8 instructions every cycle. Similarly, the issue window is a large central window implementation which can schedule instructions as soon as the data dependencies for an instruction are satisfied.

Issue width	8 instructions
Fetch width	8 Instructions
Retire width	16 Instructions
Instruction Window	64 Instructions
Functional Units	Issue width Symmetric Functional units.
Instruction fetch	Multiblock Gshare
Dcache	Perfect
Memory ports	2

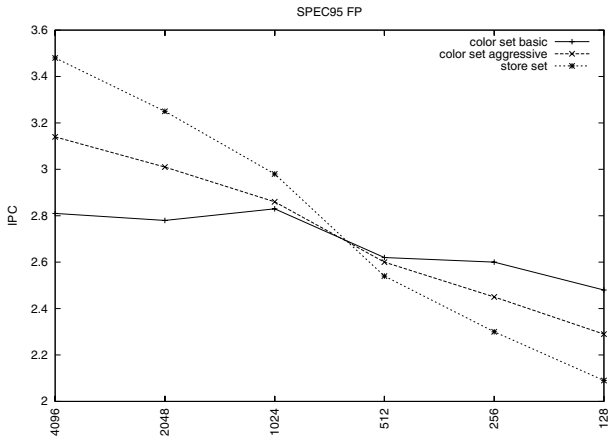
(a) Machine parameters

Functional Unit	Latency (cycles)
Load	2
Integer division	8
Integer multiply	4
Other integer	1
Float multiply	4
Float addition	3
Float division	8
Other float	2

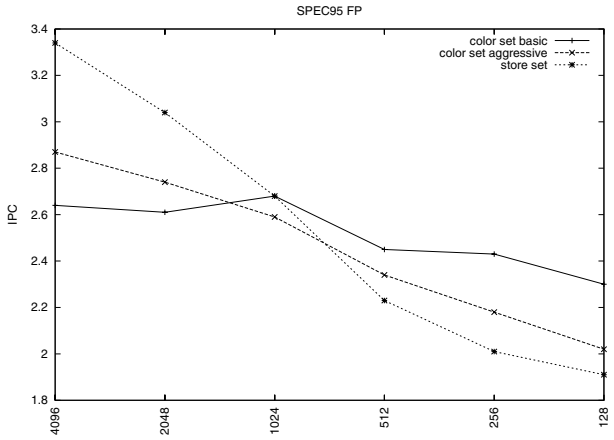
(b) Functional Unit latencies

Figure 6. Machine Configurations.

In order to show the effects of the predictor table size on the performance, we considered table sizes ranging from

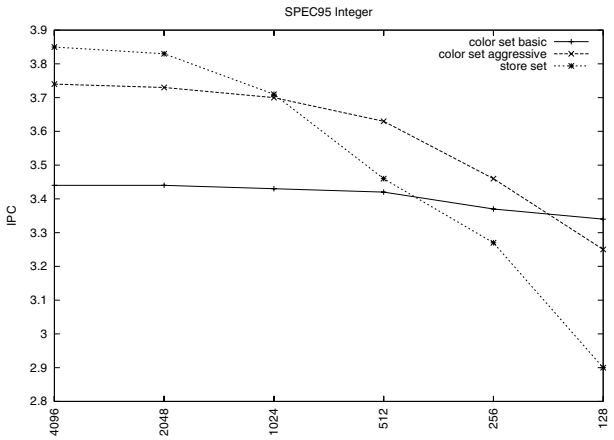


(a) Mean IPC

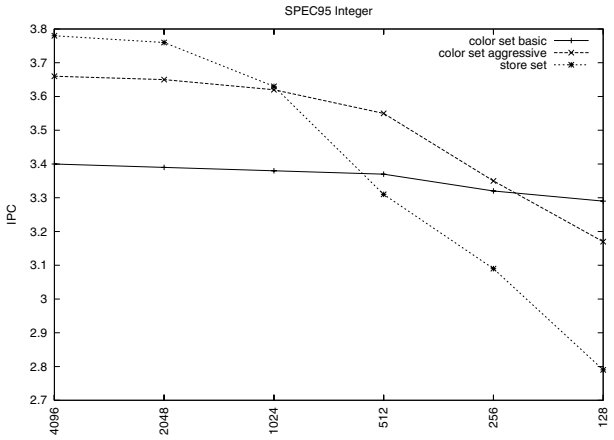


(b) Harmonic Mean IPC

Figure 7. Spec-95 Float.



(a) Mean IPC



(b) Harmonic Mean IPC

Figure 8. Spec95 Integer.

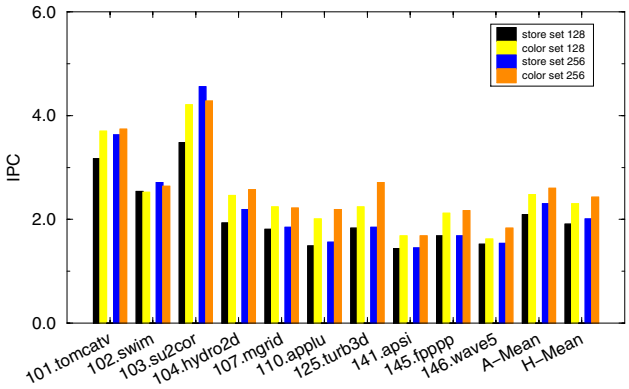
four kilobytes down to 128 bytes. Other machine parameters used in the simulations are shown in Figure 6. We used the `SPEC-95` benchmark suite with their test inputs.

Instructions per cycle. For the measurement of the instructions per cycle figures, we compare both the harmonic means of the IPC values as well as the arithmetic means. The performance of the algorithm as a function of the predictor sizes is shown in Figure 7 and Figure 8.

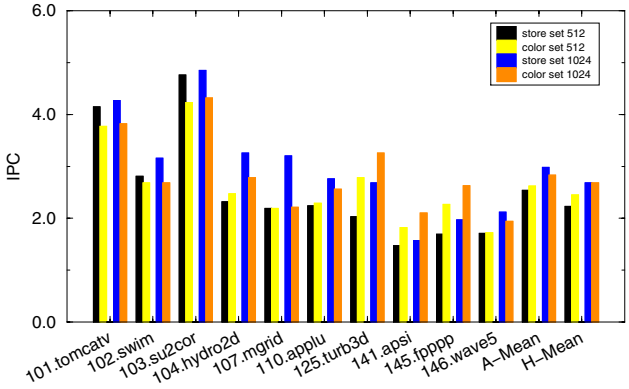
As it can be seen from both graphs, the basic policy is superior to both the aggressive policy and the store set at small predictor sizes. Specifically, when equipped with a predictor of 1 KB or less, basic policy loses very little performance as the predictor size is decreased. On the other hand, this policy cannot take advantage of larger predictor sizes. In fact, the performance of the algorithm does not change much when the predictor size is reduced from 4KB to 512 bytes.

For larger tables, the aggressive policy closely matches the performance of the store set algorithm. For both policies, harmonic means and arithmetic means follow similarly shaped curves and is indicative of uniform performance across the benchmark suite.

The performance of the algorithm using the basic policy for individual benchmarks is shown in Figure 9 and Figure 10 and the performance of the algorithm using the aggressive policy for individual benchmarks is shown in Figure 11 and Figure 12.

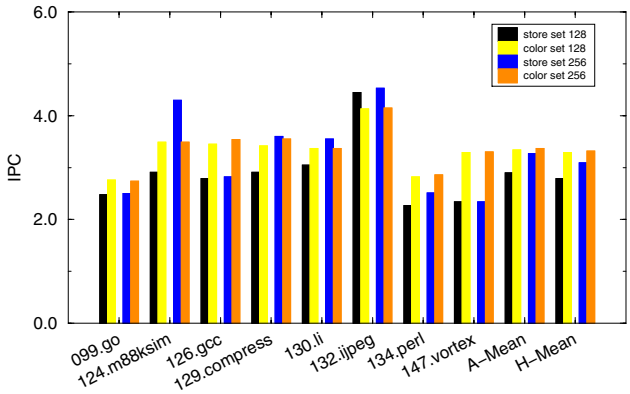


(a) 128 and 256 bytes

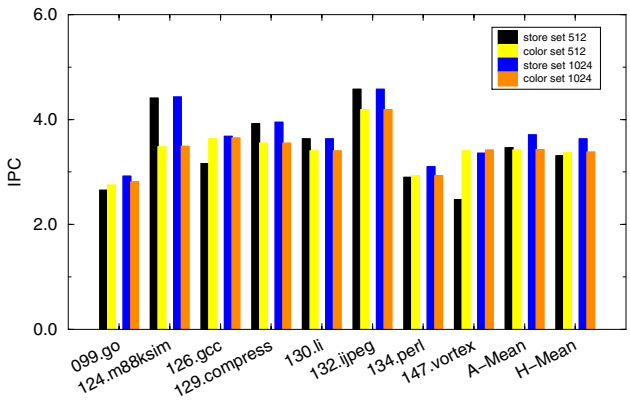


(b) 512 and 1024 bytes

Figure 9. Spec-95 Float: Basic policy.



(a) 128 and 256 bytes



(b) 512 and 1024 bytes

Figure 10. Spec95 Integer: Basic policy.

Analysis. The superior performance of the algorithm with the basic policy at smaller predictor sizes is based on two factors:

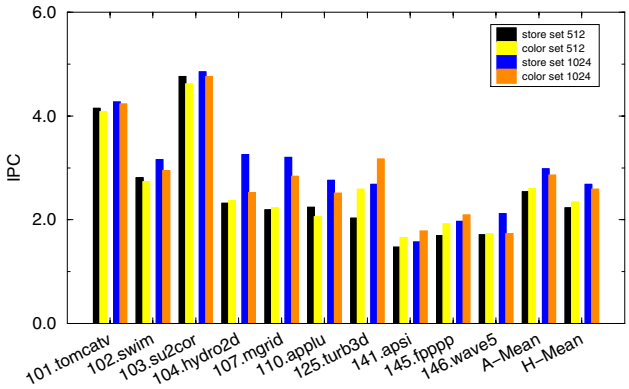
1. The space requirement per entry is much smaller than the dependence based approaches. Therefore, the algorithm can use more entries for a given hardware budget.

2. The algorithm takes advantage of the positive interference that is occurring and minimizes the effects of negative interference.

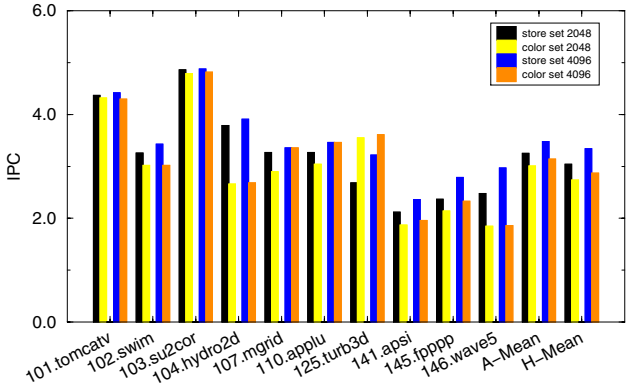
While the first point is easy to see, the interference aspect deserves some closer look. As opposed to dependence based predictors which need to periodically clear the tables and restart fresh, in case of color set approach, entries are *self healing*. For example, a successful speculation of a load instruction will result in the entry being refreshed with a lower color value, whereas an unsuccessful attempt will heal the entry with a higher color value. Unless the collisions on the predictor entries are too closely spaced, occasional overwriting of the entry by some other instruction will be repaired quickly.

The most important source of negative interference from the performance perspective is the loss of colliding store information. Such an entry may be overwritten by another load instruction, or a non-colliding store instruction. This may result in a mis-speculation if the global scheduling color is set to high and a dependent load instruction is ready and accounts for a significant percentage of the performance loss.

Another source of negative interference occurs when an independent load instruction's entry is overwritten by a dependent load instruction. In this case, the independent load instruction is forced to carry a higher color and will not be scheduled at the earliest time. However, upon observing that the machine does not have enough supply of ready loads at the given scheduling color, the processor will move rather quickly to higher colors and make those load instructions eligible again. In the experiments, we observed that both types of negative interference contribute to the perfor-

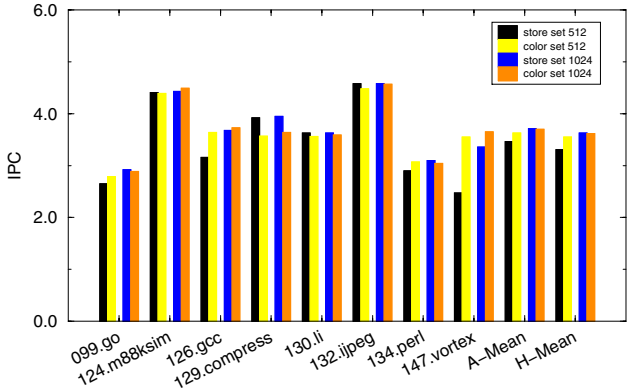


(a) 512 and 1024 bytes

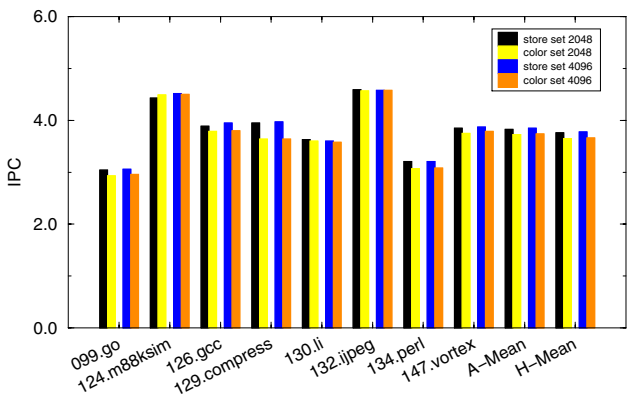


(b) 2048 and 4096 bytes

Figure 11. Spec-95 Float : Aggressive policy.



(a) 512 and 1024 bytes



(b) 2048 and 4096 bytes

Figure 12. Spec95 Integer : Aggressive policy.

mance loss, especially at very small predictor sizes.

In summary, the basic policy makes very effective use of limited information, but cannot make use of additional information. On the other hand, the aggressive policy makes too many mistakes when the information at hand is imprecise because of significant degrees of interference, but works perfectly when the interference is minimal. This suggests a two tiered approach to memory dependence prediction which is discussed further in the conclusion section.

Load Misspeculations. When we analyze the misspeculation data, we observe that the store set approach consistently out-performs the color set approach. In fact, the number of misspeculations with the store set approach does not change as the predictor size is decreased. Due to the space limitations, we do not include the store set misspeculation data here and the reader is referred to the original publication [2]. On the other hand, at small predictor sizes the store set merging process creates too many false dependencies and the algorithm simply does not speculate when it is indeed safe to do so. On the contrary, the color set yields more misspeculations but also yields better performance at these small predictor sizes.

The misspeculation data for the color set basic policy is given in Table 1 and Table 2. As it can be seen from these tables, some floating point benchmarks such as 101.tomcatv, 102.swim and 146.wave5 actually show somewhat significant number of misspeculations. Among these benchmarks, the worst benchmark, 102.swim at a predictor size of 128 bytes results in 9.42 misspeculations per thousand load instructions executed, yielding a figure slightly less than 1 percent. Even though this is a small figure in terms of percentages, these figures indicate that there is still room for improvement for these benchmarks. One important observation is the variation with some of the benchmarks where a smaller table actually yields a smaller number of misspeculations. This is due to the aliasing and illustrates that there is more to be gained by applying well known branch prediction table access techniques in this context.

Spec-95 Integer	Predictor Size in Bytes					
	128	256	512	1024	2048	4096
099.go	0.42	0.09	0.22	0.08	0.03	0.00
124.m88ksim	0.07	0.02	0.03	0.02	0.01	0.01
126.gcc	1.01	0.21	0.46	0.04	0.02	0.01
129.compress	0.01	0.00	0.01	0.00	0.00	0.00
130.li	0.63	0.00	0.62	0.00	0.00	0.00
132.ijpeg	0.04	0.01	0.02	0.01	0.01	0.01
134.perl	2.90	0.06	1.57	0.07	0.07	0.04
147.vortex	1.39	0.49	0.84	0.09	0.02	0.02

Table 1. Misspeculations per 1000 Loads

Spec-95 Float	Predictor Size in Bytes					
	128	256	512	1024	2048	4096
101.tomcatv	5.95	2.26	4.64	1.86	1.66	0.93
102.swim	9.42	0.00	1.54	0.00	0.00	0.00
103.su2cor	1.10	0.55	0.55	0.55	0.55	0.00
104.hydro2d	1.57	0.51	0.97	0.39	0.34	0.18
107.mgrid	0.01	0.01	0.01	0.00	0.00	0.00
110.applu	0.00	0.00	0.00	0.00	0.00	0.00
125.turb3d	0.76	0.19	0.30	0.08	0.07	0.04
141.apsi	0.74	0.22	0.41	0.10	0.07	0.01
145.fpppp	3.94	0.87	1.72	0.50	0.05	0.04
146.wave5	8.45	2.09	2.63	1.86	0.00	0.00

Table 2. Misspeculations per 1000 Loads

6 Conclusion and Future Work

We have presented a cost-effective mechanism for load dependence prediction. The proposed algorithm requires very small hardware budgets for dependence prediction, making the budget available for use elsewhere.

The design space of the color predictor has not yet been explored fully. In this regard, we will be studying the effects of the number of colors versus the number of entries tradeoff, as well as various techniques that can be deployed for the minimization of negative interference. Because of the similarity of the color predictor to the well known branch prediction techniques, we believe the established base of branch prediction mechanisms will enable us to improve the algorithm further. In its current form, the algorithm is particularly suitable for the application of various forms of confidence mechanisms [6, 4].

We will be exploring the design space from the power consumption perspective as well. It should be remembered that the two policies we have developed for the color set approach are close enough to each other such that both can be reasonably implemented within the same processor. Such a processor can choose the appropriate policy depending on the available power and performance requirements. Specifically, it is possible to equip the processor with a large predictor but disable a significant portion of the predictor when the available power is limited. In this case, the processor may employ the basic policy and still provide competitive performance. Similarly, when better performance is desired, the rest of the predictor can be enabled and the processor may switch to the aggressive policy.

References

[1] B. Calder and G. Reinman. A comparative survey of load speculation architectures. *Journal of Instruction Level Parallelism*, May 2000.

[2] 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.

[3] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the International Conference on Parallel Processing*, pages 245–257, August 1991.

[4] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th International Conference on Computer Architecture*, pages 122–131, 1998.

[5] 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.

[6] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 142–152, December 1996.

[7] R. Kessler, E. McLellan, and D. Webb. The alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, December 1998.

[8] 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.

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

[10] S. Önder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *32nd Annual IEEE-ACM International Symposium on Microarchitecture*, November 1999.

[11] S. Önder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. *Journal of Instruction Level Parallelism*, 2002 (to appear).

[12] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *The 31st Annual IEEE-ACM International Symposium on Microarchitecture*, pages 127–137, December 1998.

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