Speculative Execution Via Address Prediction and Data Prefetching

José González and Antonio González
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
c/ Jordi Girona 1-3, 08034 Barcelona (Spain)

Email: {joseg,antonio}@ac.upc.es
Tel: + 34 3 401 6988
Fax: + 34 3 401 7055

Abstract

Data dependencies have become one of the main bottlenecks of current superscalar processors. Data speculation is gaining popularity as a mechanism to avoid the ordering imposed by data dependencies. Loads and stores are very good candidates for data speculation since their effective address has a regular behavior and then, they are highly predictable. In this paper we propose a mechanism called Address Prediction and Data Prefetching that allows load instructions to obtain their data at the decode stage. Besides, the effective address of load and store instructions is also predicted. These instructions and those dependent on them are speculatively executed. The technique has been evaluated for an out-of-order processor with a realistic configuration. The performance gain is about 19% in average and it is much higher for some benchmarks (up to 35%).

1 Introduction

Dependencies among the instructions that constitute a program are becoming one of the most important bottlenecks for current microprocessor architectures. These dependencies can be classified into three types[15]:

- *Data dependencies*: An instruction produces a result that is used by another instruction.
- *Name dependencies*: Two instructions use the same storage location, either a register or a memory position, but there is no data flow between the instructions associated with that storage location.
- *Control dependencies*: They are caused by branch instructions, which determine the instructions that must be executed later.

Name dependencies are caused by the reuse of storage locations. They can be removed by *renaming*. This technique consists of changing the name where a given value is to be stored. Register renaming [18] is a technique used by the compilers statically and by many current processors dynamically, whereas memory renaming is more difficult to implement although there are some proposals in the literature like [10].

Control dependencies have been the aim of a plethora of works. Most of the solutions proposed rely on the prediction of the behavior of conditional branches and the speculative execution of one of the two target paths. Most current processors include some mechanisms for branch prediction and speculative execution.

However, there are very few proposals dealing with mechanisms to avoid the ordering imposed by data dependencies. In the same way as control dependencies are treated predicting the branch outcome and executing instructions speculatively earlier than the branch they depend on, one could think that data dependencies can be approached by data prediction and speculative execution. All that a processor would need is a technique to predict the value required by the dependent instructions and some mechanism to recover the state in the case of a misprediction.

Predicting the source operands of all the instructions may be difficult, but we have observed that source operands of loads and stores are highly predictable. It is shown in this paper that most of the effective address of memory instructions follow an arithmetic progression, i.e. the effective address of a giving load or store is equal to the previous address of the same static instruction plus a constant stride (that can be zero). Furthermore this stride is kept constant along many consecutive executions of a static load/store.

In this paper we propose a technique based on this load/store feature. Memory instructions that are highly predictable are dynamically identified and issued speculatively. Furthermore, in the case of load instructions, the address of the next execution is predicted and the value is brought from memory into a structure called Memory Prefetching Table (MPT) in order to be available for the next execution of the same load instruction. When a load arrives at the decode stage, its effective address is predicted and the MPT is accessed in order to know if the value has been prefetched. If the value is available, the load destination register is updated and all the subsequent dependent instruction will be allowed to be executed speculatively. Otherwise, the predicted address will be used to speculatively issue this load and the instructions dependent on it. Several cycles later, memory instructions will verify their prediction. If the prediction is correct and the load obtained the value at decode stage from the MPT, it will not require any further memory access. In the case of a misprediction, all the misspeculated instructions will be re-executed.

The rest of this paper is organized as follows. Section 2 reviews the related work and points out their main differences with this paper. Section 3 presents the motivation for this work based on a study about the predictability of memory references. The mechanism proposed in this paper is described in section 4 and evaluated in section 5. Finally, the main conclusions of this work are summarized in section 6.

2 Related work

Data prefetching has been the focus of a plethora of works [1][4][5][6][7][11][14][17][22] among others. Data prefetching tries to hide the latency of memory instructions by bringing data to the highest levels of the memory hierarchy before it is required by the processor. Compiler-directed techniques [1][5][6][14][22] are based on the fact that some memory references are predictable at compile-time to cause a cache miss. By providing a non-blocking prefetch instruction it is possible to overlap useful computations with data transfer operations. Most techniques are only effective for regular memory patterns predictable at compile time and some side-effects, such as increased register pressure, can appear as a result of these optimizations.

Some hardware-based techniques keep track of data access patterns (last effective address and stride) in a Reference Prediction Table [4][11][17]. Using this information, the effective address of load/store instructions can be predicted before they are executed and the corresponding prefetch request can be issued if the referenced data is not in the cache. In [7] three variations of this design (basic, lookahead and correlated) are studied.

The mechanism proposed in this paper uses an approach to predict memory references similar to that used by hardware prefetching techniques. However, Data prefetching schemes try to hide memory latency, but do not execute instructions speculatively.

Speculative execution has been the focus of several recent proposals that deal with data dependencies. Most of them do not speculate about the value of a register or a memory location but speculate about the likelihood that a instruction is dependent on another previous instruction. We call this type of speculation *data dependency speculation* in front of *data speculation* which predicts the value of either a source or a destination operand of an instruction.

Identifying dependencies among instructions is straightforward when data flow from an instruction to another through registers, and it can be done statically or dynamically. But when data flow through memory, the compiler is limited by the existence of ambiguous references. Thus, the compiler cannot always assure that two memory references (either a load after store or store after store) access two different memory locations and it makes conservative assumptions. Run-time disambiguation can be more accurate [23], but even then, ambiguous references may occur when the effective address of a load/store is unknown because its operands are not ready since it is data dependent on some previous instruction not finished yet. *Data dependency* *speculation* tries to overcome this limitation speculating on the existence of that dependency. Examples of this technique can be found in the Address Resolution Buffer of the Multiscalar[26] or in the Address Reorder Buffer of the HP PA8000 processor [16]. In these systems when a load is ready to be issued it is predicted that its address is different from the effective address of all previous unresolved stores. Moshovos *et al* [21] use a dynamic mechanism based on history in order to detect which loads are likely to be independent of previous unresolved stores, and therefore, reduce the number of mispredictions.

We call *data speculation* to those techniques that try to overcome the constraints imposed by data dependencies in a program. Notice the difference between *data dependency speculation* and *data speculation*, the former speculates on the presence of a dependency between two instructions (normally two memory instructions) whereas the latter speculates on the data that flow between two data dependent instructions.

In the literature there are few remarkable proposals dealing with data speculation. Some of them predict the effective address of memory instructions and execute them speculatively. In [12] a Load Target Buffer is presented, which predicts effective address adding a stride to the previous address. In [2][3] a Fast Address Calculation is performed by computing load address early in the pipeline without using history information. In both proposals the memory access is overlapped with the non-speculative effective address calculation, but none of them execute speculatively the subsequent instructions that depend on the predicted load.

Other proposals dealing with data speculation execute speculatively memory instructions and also the instructions dependent on them [8][13][19][20][24].

In [8] the processor executes load instructions in parallel with instructions preceding the loads achieving a zero cycle load execution time when the speculation is correct.

The mechanism proposed in [19] speculates on the value that a load brings from memory. That paper exploited what the authors called *value locality*, which is based on the observation that many load instructions repeatedly read the same value from memory. The proposed mechanism predicts such value with a history table and the instructions that depend on predicted loads are speculatively executed. The speed-up showed for realistic configurations is about 6% for an Alpha AXP 21164 and 3% for a PowerPc620.

In [20] the concept of *value locality* is extended for all type of instructions. At the fetch stage, a Classification Table and a Value Prediction Table are accessed in order to detect which instructions are likely to have the same result as the previous execution. Predictable instructions are speculatively issued as well as those that depend on them. The speed-ups showed by some realistic configurations of a PowerPc620 and a PowerPc620+ (an enhanced PowerPC620) are 4.5% and 6.8% respectively.

In [13], a mechanism called Memory Address Prediction is presented in order to predict the effective address of a given load/store using their last effective address and a stride gathered in previous executions of that static load/store. Predicted loads and stores are issued speculatively. Instructions that depend on speculative loads are also speculatively executed. The speed-up showed for a superscalar processor with a realistic configuration is about 10%.

In [24], two hardware methods to deal with data dependencies are studied. The first method predicts the effective address of load instructions and issues them speculatively as well as subsequent instructions dependent on the loads. The second method is called data dependence collapsing and was previously studied in [28]. The objective of this technique is to identify a sequence of dependent instructions that can be collapsed into a single compound instruction of the machine.

The main differences between the mechanism proposed in this paper and previous works are the following:

- The method proposed in this paper executes loads and instructions dependent on them speculatively, like [8][13][19][20][24]. However, our method allows to obtain the value of load instructions at decode stage, which is only allowed by the value prediction mechanism proposed in [19][20]. Besides, our method also executes speculatively store instructions, which is not considered in previous proposals.
- Regarding [19][20], the difference with our method is that they base the prediction on value locality whereas the prediction of our mechanism is based on strided references, which allows a more accurate prediction than value locality as it is shown later in this paper.

3 Motivation

The maximum ILP that a processor could achieve with infinite resources and perfect control speculation would not be much higher than a few hundred IPC, and for many applications it would be about a few tens of IPC [15]. This limit is due to the restrictions imposed by the necessity to obey data dependencies (sometimes refer to as true dependencies). *Data speculation* is a family of techniques that is gaining popularity since they allow to go beyond the barrier imposed by data dependencies and therefore, they may provide a significant increase in the amount of parallelism.

The mechanism proposed in this paper is motivated by the observation that source operands of many load/store instructions are highly predictable since they follow an arithmetic progression, i.e. their effective address in successive executions is either the same or equal to the previous one plus a constant stride. We will refer to such instructions as *strided references*. The

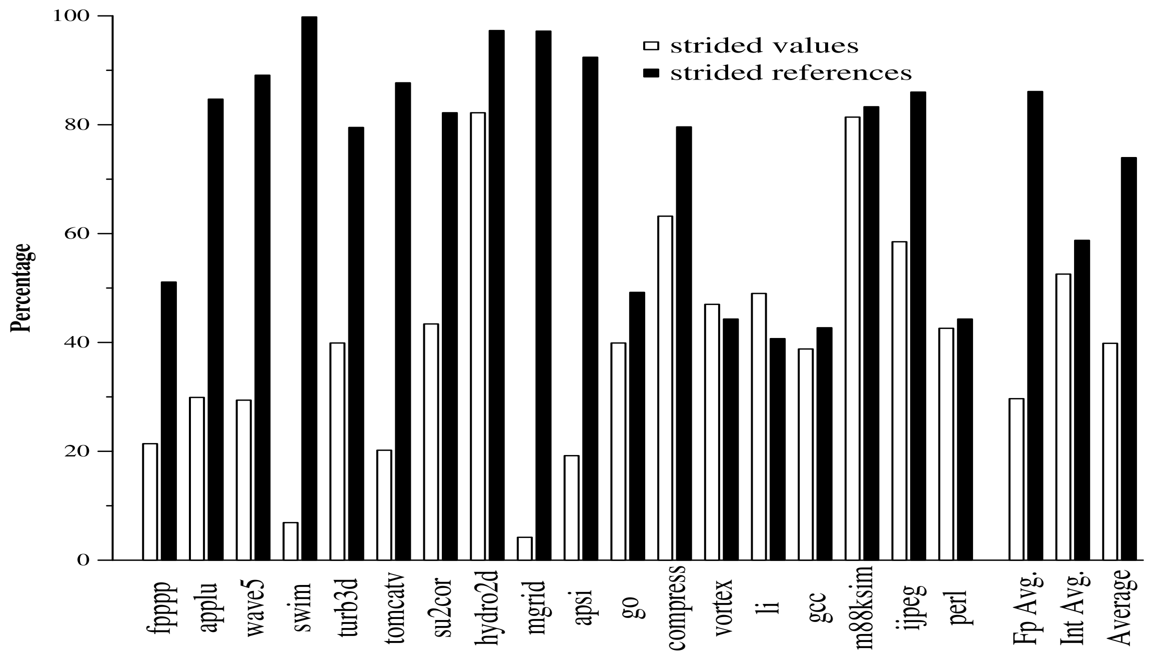

Figure 1 : Percentage of strided references compared with percentage of strided values.

constant value that is the difference between two successive effective address of the same static load/store is called the stride.This stride may be zero if the static load/store always access to the same memory address.

One can find strided references in many parts of the programs. For instance, strided references with a stride equal to zero can be generated by spill code in loops: Those values that the compiler cannot allocate to registers and are repeatedly read/written from/to the same memory position. Strided references using a stride different from zero may be found in data array operations: These arrays are usually traversed by several load/store instructions along the iterations of a loop, being each access to the same effective address plus a stride.

To validate our hypothesis on the predictability of memory references, we have run all the SPEC95 benchmark suite and measured the percentage of load/stores that perform strided references. Details about the experimental framework are provided in section 5.1. For the results in this section each benchmark has been run until the first billion of loads or until completion if it happened earlier. We have implemented a table of 2048 entries to capture strided references. This table is direct-mapped and indexed using the 11 least-significant bits of the instruction address. The table is not tagged, and therefore both constructive and destructive interferences can occur. For each entry, the table stores the last effective address and the last calculated stride. We have also compared address prediction with the value prediction scheme proposed in [20]. In that work, the authors only measured the percentage of loads that get the same value in two consecutive access, to be fair we have considered both load and stores and we have counted the percentage of strided values, that is, the percentage of load/stores that use the same value that the previous access plus an stride (which can be zero).

Figure 1 shows the percentage of loads/stores that exhibit strided references (dark bars) and strided values (light bars). As we expected, strided references are very common in the SPEC95 benchmark suite: They represent about 75% of all memory instructions. Although strided values are quite significant (about 40%), they are much less common that strided references. These results encouraged us to design a mechanism to exploit this feature in superscalar processors.

4 Address Prediction and Data Prefetching

In this paper, we propose a mechanism called Address Prediction and Data Prefetching (APDP) in order to exploit the regular behavior of the effective address of loads and stores. The mechanism identifies load and store instructions whose effective address is highly predictable at the decode stage by means of a history table similar to those tables used by some branch prediction schemes. For predictable loads and stores the APDP predicts their effective address. In addition, if the instruction is a load, the mechanism also predicts the address of the next execution of the same static instruction and issues a prefetch of the data at this address. The prefetched data will be stored in a table. Load instructions check this table in the decode stage and if the value is available, the load updates the destination register and instructions dependent on the load will be allowed to be executed speculatively. If the value is not available or the instruction is a store, the predicted address for the current instruction is used to issue it speculatively as well as instructions dependent on it.

The APDP mechanism has been implemented for an out-of-order superscalar processor with in-order retirement to support precise exceptions [25]. The processor implements a *partial disambiguation* memory model, based on the mechanism implemented in the HP PA8000 [16] and the ARB structure of the Multiscalar [26]. This mechanism allows loads to be issued even if there are earlier unresolved stores. Loads and stores are kept in the Address Reorder Buffer (ARB) until they are committed. When a load enters the ARB, it looks for some previous store to the same address in order to forward the data from the closest store to the load destination register. Otherwise the load performs a cache access as soon as a memory port is available. When a store enters the ARB, it is checked whether a subsequent load has already performed a cache access to the same address. In this case, a recovery mechanism is initiated.

The APDP mechanism is implemented by means of a table called Memory Prefetching Table (MPT) and a table called Prefetching Validation Table (PVT). The MPT is a 2048-entry direct-mapped table that is indexed with the least significant bits of the instruction address and does not contain tags. Each entry stores the following information:

- Effective Address: This is the last effective address accessed by that load/store instruction.
- Stride: This field stores the last stride observed for that load/store instruction.
- Stride History Bits (SHB): This field is used to predict the probability that the stride calculated in the last executions is kept constant for the next execution of the load/store instruction. This field is very similar to the branch history field of branch predictors. It is implemented by means of a two-bit up-down saturated counter. If the most significant bit is set then the stride will be assumed to be correct and therefore the predicted effective address will be the value in the effective address field plus the stride. Otherwise, the effective address of such load/store instruction will be assumed to be unpredictable. The counter is initialized to 0, and it is increased if the difference between the current effective address and the last effective address is equal to the stride field and decreased otherwise.
- Value: This field contains the value brought from Data cache that will likely be the value requested by the next execution of the load instruction.
- Valid value: This field will be set if the Value field contains a correct value for the load instruction.

Two possible policies to update the stride field are the following: First, the processor changes the stride for every access to the MPT. Second, the stride is changed only when the STB field is either 0 or 1. We have adopted the second alternative since it works better for loops. Figure 2 shows the behavior of both approaches for a sample code consisting of two nested loops. The figures show the evolution of the MPT for the last three iterations of the innermost loop and the following five iterations of the next execution of the same loop. The loop contains a memory reference with a constant stride (4 in the example). Both approaches mispredict the reference in the first iteration of the innermost loop, but the first approach mispredicts also the reference in the second and third iterations. In consequence, we have adopted the second scheme.

The PVT is a 1024-entry table direct-mapped table that is indexed with the least-significant bits of the effective address of memory instructions and does not contain tags. It is used to keep coherent the MPT. Each entry stores the following information:

- Data Request (DR): This is a bit which is set when a prefetch to an address mapped to this entry is initiated and it is reset when a load that is mapped to that PVT entry is decoded. This load will speculatively use the prefetched data if it is available.
- Data Invalidation (DI): When processor retires a store, the corresponding entry in the PVT is accessed in order to verify if it contains a prefetched request (i.e., DR is equal to 1). In this case, the DI field is set. Notice that there may be more invalidations than needed

Iter	Correct Address	Stride	Predict. Address	SHB	Hit/Miss
n-3	1000	4	1000	3	Hit
n-2	1004	4	1004	3	Hit
n-1	1008	4	1008	3	Hit
0	100	908	1012	3	Miss
1	104	4	1008	2	Miss
2	108	4	Not predictable	1	Miss
3	112	4	112	2	Hit
4	116	4	116	3	Hit

a)

Iter	Correct Address	Stride	Predict. Address	SHB	Hit/Miss
n-3	1000	4	1000	3	Hit
n-2	1004	4	1004	3	Hit
n-1	1008	4	1008	3	Hit
0	100	4	1012	3	Miss
1	104	4	104	2	Hit
2	108	4	108	3	Hit
3	112	4	112	3	Hit
4	116	4	116	3	Hit

b)

Figure 2: Two alternative approaches to update the stride field: a) changing the stride each access and b) changing the stride only when SHB is 0 or 1

due to the fact that the table does not contain tags (i.e., only the 10 least-significant bits of the address are used for setting the DR and DI bits). This bit is reset when a new request is issued.

A block diagram of the main components of the APDP is shown in Figure 3. The APDP works as follows for a store: In the decode stage, the corresponding MPT entry is read, and the predicted address for the current execution is computed. If the most significant bit of the SHB is set, the prediction is assumed to be correct and the store is ready to be issued speculatively, that is, it is introduced into the ARB structure in the next cycle. Stores in the ARB will forward their value to subsequent loads to the same address. The data is written into memory in the retire stage in order to provide precise exceptions.

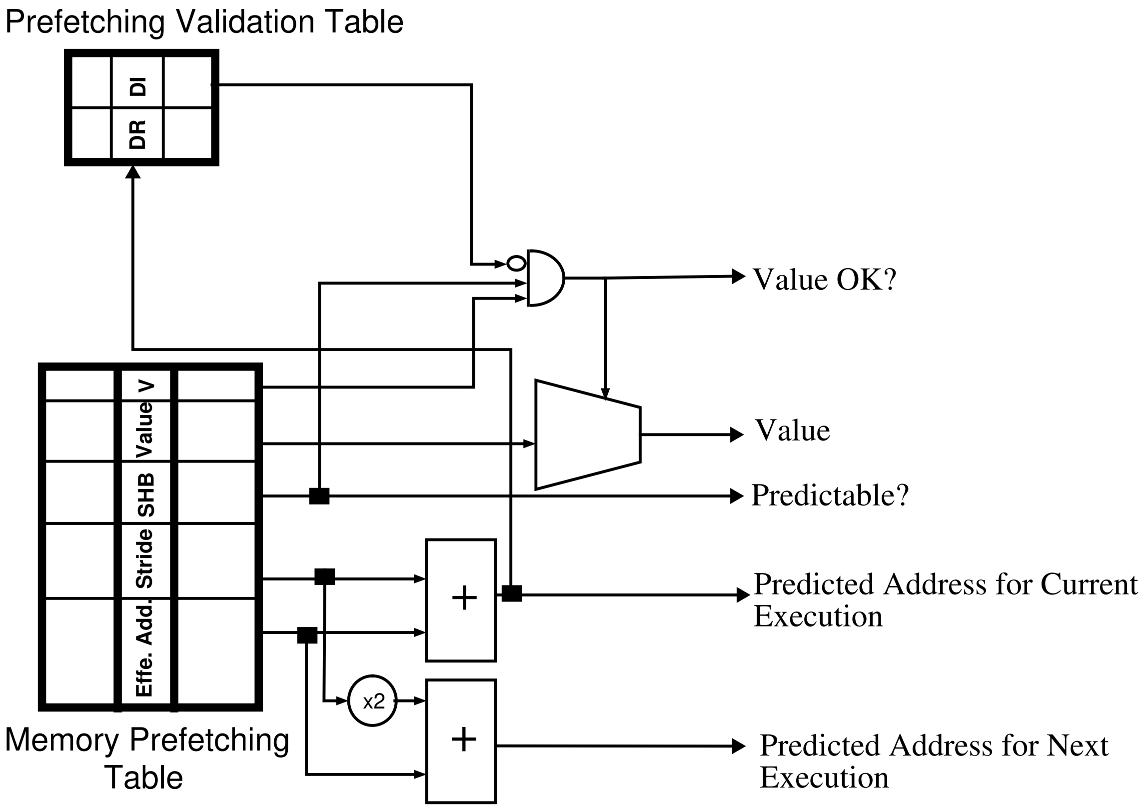

Figure 3: Block diagram of the APDP mechanism.

If the instruction is a load, the MPT may contain the value read by this load. More precisely, if the Valid Value field (V in Figure 3) is set, and the DI field of the corresponding entry in the PVT is zero (i.e., the data value has not been invalidated by previous stores), and the current address is supposed to be predictable (this is indicated by the most-significant bit of SHB) then the load gets the value to be written in the destination register from the Value field of the MPT. The MPT also predicts the effective address of the current execution of the load. This address is used to access the PVT as explained above. In addition, if the load cannot obtain the value the predicted address is used to issue the load to the ARB in the next cycle.

Finally, the effective address of the next execution of the same static load is also predicted if it is supposed to be predictable (again this is indicated by the most-significant bit of SHB). This address is used to issue a prefetch of the data required by the next execution and to set the DR bit of the corresponding PVT entry. Notice that this prefetch is only issued if there is an available port. Otherwise it is discarded. This is a conservative approach in order not to overload the memory ports. A potential improvement that we plan to investigate is the addition of a prefetch request buffer that will keep prefetch request until a memory port is available.

If a load/store instruction cannot be predicted, it is dispatched to the instruction window and executed as any other instruction.

Speculatively issued load/store instructions must be verified. This is done issuing them to the address computation units in order to calculate their real effective address when they are free of dependencies. Once it is computed, the effective address is compared against the predicted one. In the case of a misprediction, a recovery action is initiated.

The recovery mechanisms for loads and stores are different. In the case of load instructions, a mechanism similar to [20] is used: Each speculative load has associated a tag which is propagated to all the subsequent instructions that depend on it. When the load is verified all dependent instructions are either ready to be committed (if the prediction was correct) or re-executed (if there was a misprediction). This mechanism cannot be used for stores since the destination of a store is not a register but it is a memory location. This implies that dependent instructions are not known until the correct effective address is computed. For stores, it has been implemented the same recovery mechanism which is used to recover from a mispredicted branch: Once a mispredicted store is detected, all the subsequent instructions are flushed from the pipeline and the fetch is restarted to the instruction after the mispredicted store.

Because of store misprediction penalty is higher than that of loads, store effective address is predicted only if its source operands are not ready at decode stage, whereas load address prediction is performed even if its source operands are ready. There are two reasons for doing that: The first one is that load instruction may have its value ready at de MPT and therefore the latency of that operation is reduced in several cycles even if its source operands are ready. The second reason is that if its value is not available in the MPT but its effective address is predictable, it can be issued (speculatively) earlier.

5 Performance evaluation

In this section the effectiveness of the APDP mechanism is studied for a superscalar out-of-order execution processor with a partial disambiguation memory scheme.

5.1 Experimental Framework

We have developed a simulator of a superscalar processor with out-of-order execution. Figure 4 shows a block diagram of the processor architecture. The execution of an instruction goes through the following stages:

- Fetch: Up to four instructions are read from the Instruction cache. Branch prediction is performed by means of a 2048-entry Branch History Table. An infinite Icache is assumed.

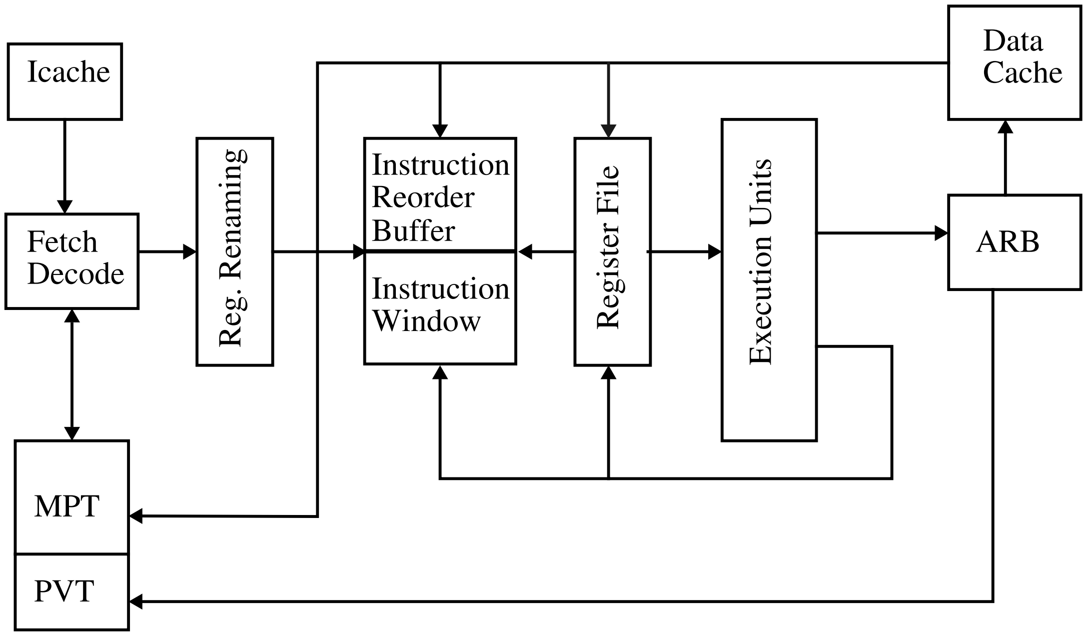

Figure 4: Processor architecture.

- Decode: Logical registers are renamed and mapped onto physical registers. Address Prediction and Data Prefetching is performed in this stage. Decoded instructions are dispatched to a global Instruction Window and to an Instruction Reorder Buffer (IRB). Instructions are kept in the Instruction Window until they are issued to a functional unit, whereas they are kept in the IRB until they are retired.
- Issue: Every cycle, the control logic searches the Instruction Window for instructions that are free of dependencies. If a ready-to-issue instruction is found and there is some free functional until, that instruction is issued and removed from the Instruction Window
- Execute: Instructions are executed in their corresponding functional unit. In this stage, load/store instructions compute their effective address, and then, they are introduced in the ARB. Memory instructions in the ARB send their request to the Data cache when there is some available memory port.
- Write-back: Instructions are completed and results are written into the register file.
- Retire (o commit): Instructions are retired in-order, so a precise state can be recovered at any time. In this stage, store instructions send their request to the Data cache.

Table 1 shows the number of different functional units and their latency. The size of the reorder buffer is 32 entries. There are two separate physical register files (FP and Integer), having each one 64 physical registers. The processor has a loockup-free data cache [9] that allows 16 outstanding misses to different cache lines. The cache size is 8Kb and it is direct-mapped with 32-byte line size. The hit time of the cache is two cycles and the miss penalty is 18 cycles. A infinite L2 cache is assumed.

Functional Unit	Latency	Repeat rate
Simple Integer	1	1
Complex Integer	9 multiply 67 divide	9 67
Int. Effective Address	2	1
FP Effective Address	3	1
Simple FP	2	1
FP Multiplication	2	1
FP Divide and SQR	21 divide 35 SQR	21 35

Table 1: Functional units and instruction latency.

Our experimentation methodology is trace-driven simulation. Programs are compiled with full optimizations for a DEC AlphaStation600 5/266 with an Alpha AXP 21164 processor. Then they are instrumented using the Atom tool [27]. Programs are run and their trace feeds the superscalar simulator. An accurate cycle-by-cycle simulation is performed in order to obtain precise results. Because of the detail at which simulation is carried out, the simulator is slow, so we have collected results for 50 million of instructions after skipping the first 100 million of instructions for each benchmark.

Performance figures have been obtained for eight Spec95 benchmarks: Four floating point (104.hydro2d, 107.mgrid, 110.applu, 146.wave5) and four integer (124.m88ksim, 129.compress, 130.li, 134.perl).

5.2 Results

Table 2 shows the IPC (instructions committed per cycle) achieved by different schemes. The first column shows the performance of a processor using a *total disambiguation* scheme. This scheme is used as a baseline for comparison since this is the scheme implemented by most current microprocessors. Total disambiguation does not allow load instructions to be issued until all previous stores have computed their effective address. Besides, store instructions are not issued until the effective addresses of all previous memory instructions are known. The second column shows the performance of a partial disambiguation scheme as described in section 4. Finally, the third column shows the performance when the APDP is implemented in a processor with partial disambiguation.

	No APDP		**APDP**
	Total disambiguation	**Partial disambiguation**	**Partial disambiguation**
104.hydro2d	1.14	1.21 (6.1%)	1,28 (12.28%)
107.mgrid	1.68	1.68 (0%)	1.96 (16.62%)
110.applu	1.20	1.20 (0%)	1.25 (4.17%)
146.wave5	0.95	0.99 (4.2%)	1.21 (26.37%)
124.m88ksim	0.91	0.91 (0%)	1.23 (35.16%)
129.compress	1.15	1.28 (11.3%)	1.32 (14.78%)
130.li	0.92	1.02 (10.9%)	1.11(20.65%)
134.perl	0.91	1.03 (13.2%)	1.11 (21.98%)
Avg. improvement		5.7%	19.00%

Table 2: Instruction completion rates. In brackets it is shown the percentage improvement over the total disambiguation scheme

Notice the significant improvement (19%) achieved by combining partial disambiguation and the APDP mechanism in front of the improvement achieved by the partial disambiguation alone (5.7%).

The improvement of the APDP relies on the percentage of strided references and on the amount of loads that obtain the value (previously prefetched) from the MPT. There are also other issues that have an influence on the results, such as the accuracy of the memory address predictor and the percentage of memory instructions that are free of dependencies at decode time. The more instructions are not free of dependencies, the more potential improvement has the APDP mechanism.

For instance, 110.applu has a high percentage of strided references (83%) as it is shown in Figure 1 but its improvement is the lowest among that of the benchmarks presented in this work. It is due to its high percentage of memory instructions that are free of dependencies when they are decoded, as it is shown in Figure 6. Besides, 100.applu has also the lowest percentage of prefetched loads that get their value from the MPT in the decode stage (see Figure 7) and a percentage of mispredictions for strided references quite high (as shown in Figure 5).

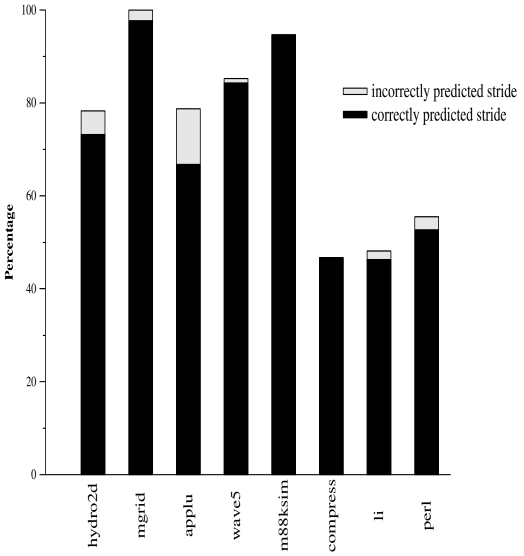

Figure 5: Percentage of loads that are predicted to be strided references. It is also shown the percentage of loads with a correctly predicted stride

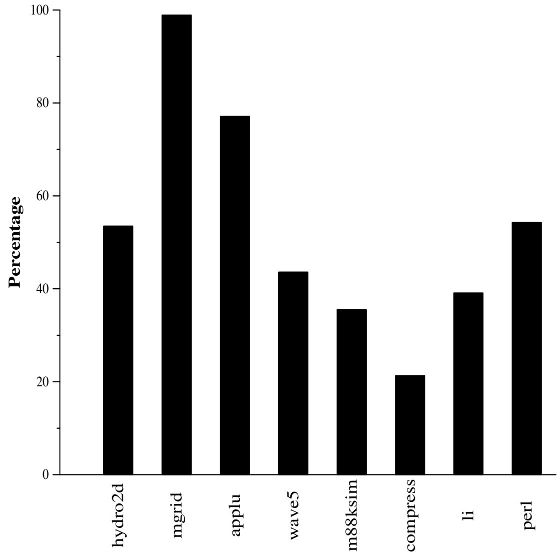

Figure 6: Percentage of loads/stores that are free of dependencies at decode stage.

On the other hand, 124.m88ksim experiments the highest improvement because it has a high percentage of loads that get their value from the MPT (Figure 7) and it has many loads that are not free of dependencies at decode stage (Figure 6). Besides, the address predictor works very well as it can be observed in Figure 5.

An interesting issue that deserves further analysis is the percentage of loads that cannot get their value from the MPT in the decode stage despite having correctly predicted the next address in the previous execution. This occurs when there is not enough distance between two consecutive executions of a static load. Figure 7 shows this percentage in light shaded bars. It can be seen that there are some benchmarks (104.hydro2d, 129.compress,130.li) that if the interval between two consecutive executions of a static load were longer they would benefit more from APDP mechanism. This fact suggests that adding some features to the compiler and/or the hardware in order to increase the gap between some consecutive executions of predictable loads may improve the performance of APDP technique. This is something that we are currently investigating.

It is important to note that the benefit from APDP comes from different sources. First, in average 21% of loads gain their value in the decode stage, allowing subsequent dependent instructions to be issued speculatively. Second, an important percentage of loads (70% in average) predict correctly their address in the decode stage, which enables the Dcache access to be performed before computing their effective address and also allows dependent instructions to be executed speculatively. Another important source of improvement comes from predicting the

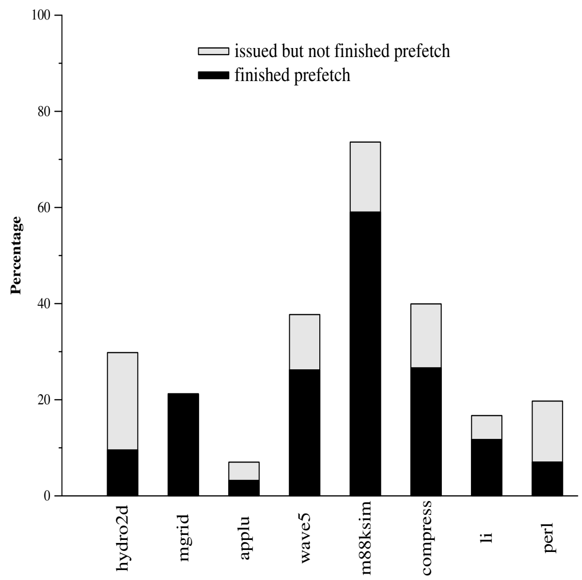

Figure 7: Percentage of correctly prefetched loads split into those that have the value ready at the decode stage and those that do not.

store effective address. In a partial disambiguation scheme, predicting the effective address of stores results in a lower number of misspeculated loads because the predicted address of previously unresolved stores can be used to perform a more accurate disambiguation.

The hardware cost of the APDP is not negligible mainly because of the large MPT, but with the high growth rate in transistor count on a chip, it may be affordable for next generation microprocessors.

6 Conclusions

Whereas control speculation has been the aim of a plethora of works, and it seems to be approaching its limits in performance improvement, data speculation may be regarded as a future approach to increase processor performance by eliminating some barriers imposed by data dependencies.

In this paper a technique called Address Prediction for Data Prefetching has been presented. This mechanism is based on the observation that most load/store instructions have a predictable effective address by keeping track of their history. This fact allows the processor not only to predict the current effective address of either a load or store instruction but also the next effective address of loads and prefetch the data at this address. This data is brought from memory to a table (MPT) that allows load instructions to obtain its value at decode stage. Predicting the effective address of loads/stores and bringing load values from memory removes the data dependency between such memory instruction and previous instructions that they depend on. Memory instructions and those that are dependent on them are speculatively executed.

This mechanism has been evaluated for an out-of-order superscalar processor with a partial disambiguation memory scheme. We have observed significant performance gains (19% over a baseline machine) and, for some programs it is much higher (up to 35%). Some issues for further improvement, such as the short distance between consecutive executions of some static loads, will be the focus of our future research.

Overall, the promising results together with the possibility to improve them by compiler/hardware-aided mechanisms and the few known techniques to exploit data speculation encourage us to extend this work trying to reduce the hardware cost and studying data speculation for other kinds of instructions.

7 References

[1] S.G. Abraham,R.A. Sugumar, B.R. Rau, R. Gupta
Predictability of Load/Store Instruction Latencies
Proc. of Int. Symp. on Microarchitecture, pp139-152, 1993.

[2] T.M. Austin, D.N.Pnevmastikatos G.S. Sohi
"Streamling Data Cache Access with Fast Address Calculation"
Proc of the Int. Symp. Computer Architecture, pp. 369-380, 1995.

[3] T.M. Austin, G.S. Sohi
"Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency"
Proc. of Int. Symp. on Microarchitecture, pp 82-92, 1995.

[4] J-L. Baer and T-F. Chen
"An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty"
Proc of Supercomputing'91 Conference, pp. 176-186, 1991.

[5] D.Bernstein, D. Cohen, A. Freund
"Compiler Techniques for Data Prefetching on the PowerPc"
Proc of PACT, pp 19-26, 1995.

[6] D. Callahan
"Software Prefetching"
Proc. of the ACM Conf. on Architectural Support for Programming Languages and Operating Systems, pp 40-52, 1991.

[7] T-F. Chen and J-L. Baer
"A Performance Study of Software and Hardware Data Prefetching Schemes"
Proc of the Int. Symp. Computer Architecture, pp. 223-232, 1994.

[8] R.J. Eickemeyer and S. Vassiliadis
"A Load Instruction Unit for Pipelined Processors"
IBM Journal of Research and Developement, 37(4), pp. 547-564, July 1993

[9] K.I. Farkas and N.P. Jouppi
"Complexity/Performance Tradeoffs with Non-Blocking Loads"
Proc. of the Int. Symp. on Computer Architecture, pp. 211-222, 1994.

[10] M. Franklin and G.S. Sohi
"ARB: A Hardware Mechanims for Dynamic Reordering of Memory References"
IEEE Transactions on Computers, 45(6), pp. 552-571, May 1996.

[11] J.W. Fu, J.H. Patel and B.L. Janssens
"Stride Directed Prefetching in Scalar Processors"
in *Proc of the 25th. Int. Symp. on Microarchitecture (MICRO-25)*, pp. 102-110, 1992.

[12] M.Golden and T.N. Mudge
"Hardware Support fot Hiding Cache Latency"
Technical report # CSE-TR-152-93. University of Michigan, 1993.

[13] J. Gonzalez and A. Gonzalez
"Memory Address Prediction for Data Speculation"
Technical report # UPC-DAC-1996-50, Universitat Politecnica de Catalunya, 1996.

[14] E.H. Gornish, E.D. Granston A.V. Veidenbaum.
"Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies"
Proc. of the Int. Symp. on Computer Architecture, 1990.

[15] J.L Hennessy and D.A. Patterson
Computer Architecture. A Quantitative Approach. Second Edition. Morgan Kaufmann Publishers, San Francisco 1996.

[16] D. Hunt
"Advanced Performance Features of the 64-bit PA-8000"
Proc. of the CompCon'95, pp. 123-128, 1995.

[17] Y. Jegou and O. Temam
"Speculative Prefetching"
Proc. of the 1993 Int. Conf. on Supercomputing, pp. 57-66, 1993.

[18] M. Johnson
Superscalar Microprocessor Design, Prentice-Hall, 1991.

[19] M.H. Lipasti and J.P. Shen
"Exceeeding the Dataflow Limit via Value Prediction"
Proc. of Int. Symp. on Microarchitecture, 1996.

[20] M.H. Lipasti, C.B. Wilkerson and J.P. Shen
"Value Locality and Load Value Prediction"
Proc. of the ACM Conf. on Architectural Support for Programming Languages and Operating Systems, Oct. 1996.

[21] A.I. Moshovos, S.E. Breach, T.N. Vijaykumar and G.S. Sohi
"A Dynamic Approach to Improve the Accuracy of Data Speculation"
Technical report # CS-TR-96-1316, University of Wisconsin-Madison, March 1996.

[22] T.C. Mowry, M.S. Lam and A. Gupta
"Design and Evaluation of a Compiler Algorithm for Prefetching"
Proc. of the ACM Conf. on Architectural Support for Programming Languages and Operating Systems, pp 62-73, 1992.

[23] A. Nicolau
"Run-Time Disambiguation: Copying with Statically Unpredictable Dependences"
IEEE Transactions on Computers, 38(5), pp. 663-678, May 1989.

[24] Y. Sazeides, S. Vassiliadis and J.E. Smith
"The Performance Potential of Data Dependence Speculation & Collapsing"
Proc. of Int. Symp. on Microarchitecture, December 1996.

[25] J.E. Smith and A.R. Pleszkun
"Implementing Precise Interrupts in Pipelined Processors"
IEEE Tranactions on Computers, 37(5), pp. 562-573, May 1988.

[26] G.S. Sohi, S.E. Breach and T.N. Vijaykumar
"Multiscalar Processors"
Proc. of the Int. Symp. on Computer Architecture, pp. 414-425, 1995.

[27] A. Srivastava and A. Eustace
"ATOM: A system for building customized program analysis tools"
Proc of the 1994 Conf. on Programming Languages Design and Implementation, 1994.

[28] S. Vassiliadis, B. Blaner and R.J. Eickemeyer
"SCISM: A Scalable Compound Instruction Set Machine Architecture"
IBM Journal of Research and Development, 38(1), pp. 59-78, Jan. 1994.