Aggressive Speculative Execution for Hiding Memory Latency

Alex Pajuelo, Antonio González and Mateo Valero

Abstract-**L2 misses are one of the main causes for stalling the activity in current and future microprocessors.**

In this paper we present a mechanism to speculatively execute independent instructions of L2-miss loads, even if no entry in the reorder buffer is available. The proposed mechanism generates future instances of instructions that are expected to be independent of the delinquent load. When these dynamic instructions are later fetched, they use the previously precomputed data and directly go to the commit stage without executing.

The mechanism replicates strided loads found above the L2-miss load, that produce the data for the target independent instructions. Instructions following the L2-miss load will check if their source operands have been replicated. In this case, multiple speculative instances of them will also be generated.

This mechanism is built on top of a superscalar processor with an aggressive prefetch scheme. Compared to this baseline, the mechanism obtains 21% of performance improvement.

Keywords-**Dynamic vectorization, L2 misses, speculative execution, memory gap.**

I. Introduction

THE memory gap is a very well-known problem in current processors. This makes memory instructions to have very long latencies when going out of the processor for data. These long latencies reduce drastically the opportunities of exploiting DLP and ILP in applications, not only because dependent instructions of a L2-miss load cannot execute until the data is brought from main memory, but also because independent instructions cannot commit due to the in-order nature of the commit process. This problem becomes worse when the instruction window completely fills up stalling the processor due to the lack of entries in the ROB, preventing the fetch, decode and execution of new instructions.

To reduce this problem, one solution could be enlarging the on-chip caches. With this solution the percentage of L2 misses decreases because more memory is available in the chip. But enlarging caches is expensive and it can impact seriously the cycle time of the processor or increase the latency of memory instructions.

Prefetch mechanisms [1][2][3] have been also studied to reduce the penalty of L2 misses. These mechanisms try to predict memory addresses of loads to bring in advance the data that will be needed from main memory to the lower (and faster) levels of the memory hierarchy.

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Campus Nord, Sergi Girona 1-3, 08034 Barcelona. Email: {mpajuelo, antonio, mateo}@ac.upc.es

However, prefetch is difficult when the memory access patterns of L2-miss loads are hard to predict (i.e., pointer-based memory accesses). Wrong prefetches overload the memory bus and pollute the on-chip caches with useless data.

From another point of view, mechanisms to enlarge virtually the instruction window [4][5][6] have been proposed. These mechanisms try to solve the stall problem of the processors under L2 misses with out-of-order commit. This allows keeping executing independent instructions of the L2-miss loads before the data of this memory instruction is available. The main problem of these mechanisms is that it can be difficult to recover the state of the processor, e.g. under branch mispredictions.

We present a mechanism that emphasizes the fact that instructions independent of a L2-miss load can be executed before the data for that load is available. We propose to precompute data, speculatively, for futures instances of independent instructions of L2-miss loads that are beyond the instruction window limit. The mechanism creates these futures instances as soon as the L2-miss load is detected. When instructions enter the pipeline, they check if they have speculatively precomputed its outcome and in this case, they reuse and avoid their execution. The main benefit of this mechanism is that it enlarges virtually the instruction window without using complex checkpointing mechanisms.

To quantify its benefits, we evaluate the proposed mechanism on a superscalar processor that already implements an aggressive prefetch mechanism, to isolate the benefits of early precomputation. Our studies show that this mechanism outperforms the baseline configuration (processor with aggressive prefetch) by about 21% with a moderated amount of extra hardware.

The structure of the paper is as follows. Next section motivates this work. Section III describes the proposed mechanism. We evaluate this mechanism in section IV. Section V reviews the related work. Finally, we conclude in section VI.

II. Motivation

As memory latency increases, caches play a very important role to exploit the memory parallelism. However, when data does not exhibit spatial nor temporal locality, caches are nearly useless.

L2-miss loads and the in-order nature of the commit process provoke stalls in the processor because the instruction window completely fills (this happens about 61% of applications' execution time with the architecture described later in section IV.A). Dependent instructions of these L2-miss loads cannot execute

because their data is not available. However, independent instructions could be executed if entries in the instruction window were available. Figure 1 depicts the problem.

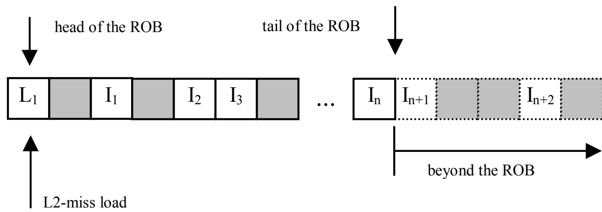
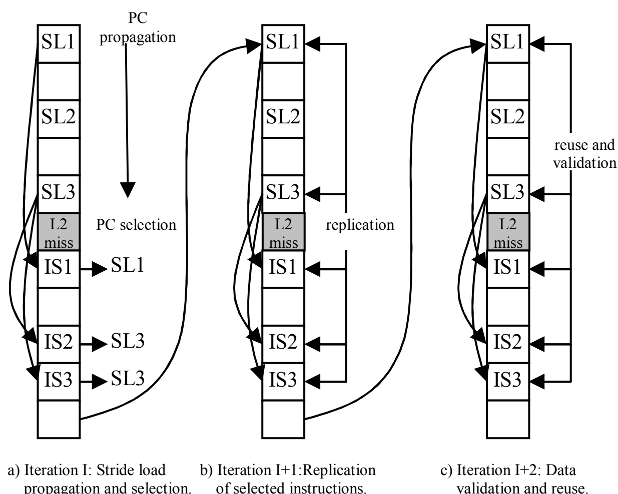


Fig. 1. Instruction window filling due to a L2-miss load.

Figure 1 shows a possible state of the ROB in a L2-miss load. *L1* is a L2-miss load. White boxes are independent instructions of *L1*. Grey boxes are dependent instructions of *L1*. From *L1* to I_n are instructions in the ROB. From I_{n+1} onwards are instructions that are going to enter the pipeline next. In this scenario, the processor can execute *I1*, *I2*… because they are independent of *L1*. However, instructions I_{n+1}, I_{n+2} … cannot be executed because the fetch and decode stages of the processor are stalled due to the full filling of the ROB. In this paper we propose the speculative execution of these instructions (I_{n+1} onwards) so that, when these instructions are encountered, they will have their data computed and they will reuse it, avoiding their execution.

III. THE APPROACH: *L2MISS*

This mechanism emphasizes the fact that futures values for independent instructions of a L2-miss load can be precomputed as soon as the instructions are encountered. In this case, the mechanism replicates sets of strided loads above the L2-miss load and dependent instructions of the stride loads after this L2-miss load. Replication is only possible if the L2-miss load is in a loop body. Figure 2 depicts how the mechanism works.



a) Iteration I: Stride load propagation and selection. b) Iteration I+1:Replication of selected instructions. c) Iteration I+2: Data validation and reuse.

Fig. 2. Steps of the L2miss mechanism.

In figure 2a, *SL1*, *SL2* and *SL3* are strided loads before the L2-miss load that begin the dependence graphs for the independent instructions *IS1*, *IS2* and *IS3* after the L2-miss load. Our first mechanism replicates *SL1* and *SL3* for later replicate *IS1*, *IS2* and *IS3* (Figure 2b). *SL2* will not be replicated because there are no instructions dependent on this strided load after the L2-miss load.

Strided loads after the L2-miss load are also considered as independent instructions because no memory address check is performed and will be also replicated.

A. Overview

This mechanism works in four steps. First of all, it detects the strided loads and propagates their PCs down the dependence graph. In Figure 2a, the PCs of *SL1* and *SL3* are propagated to instructions *IS1*, *IS2* and *IS3*. Since no L2-miss prediction hardware has been included in the processor the stride detection and the PC propagation is performed continuously, to know, when a L2 miss occurs, the PCs of those strided loads that independent instructions depend on.

When a L2-miss load is committed, the second step of the mechanism is fired to identify the instructions independent of the L2-miss load. For this purpose, it analyzes the source operands of every instruction after the load to check whether they are independent of the destination register of this L2-miss load. If the instruction is independent, the set of propagated PCs of strided loads above the L2-miss load, on which the instruction depends, are selected for replication. In Figure 2a, when instructions *IS1*, *IS2* and *IS3* commit, the loads they depend on (*SL1* and *SL3*) are selected for replication.

The third step is the replication of the selected instructions (*SL1* and *SL3*) the next time they are fetched and decoded (Figure 2b). Replication is performed by creating several copies of the selected instructions modifying their source and destination operands. These copies are dispatched and executed but they are not committed. Dependent instructions (*IS1*, *IS2* and *IS3*) of a replicated instruction are also replicated by propagating the "replicated" characteristic down the dependence graph.

Finally, fourth step (Figure 2c), every time an instruction is fetched, it is checked whether it was previously replicated to validate the speculative precomputed data. If the speculation is correct, the instruction is not executed but passed to the commit stage, reusing the precomputed data. Otherwise, a recovery action is performed from the mispredicted instruction discarding the speculative precomputed result.

This mechanism focuses on the fact that only replication will be performed for independent instructions of loads with high probability of missing the L2 cache. Note that the first time a L2-miss load is detected, independent instructions may not reuse data because no data has been precomputed.

In Figure 1 it is supposed that the L2-miss load has a high ratio of L2 misses. Independent instructions will not have precomputed data the first time the L2-miss load is detected. But for next iterations (in Figure 2c, Iteration I+2 onwards), as independent instructions have been replicated, precomputed data would be available for reusing.

These four steps are further detailed below.

B. First step: strided load propagation

Every time an instruction is fetched, its PC is checked on the stride predictor to see whether it is a strided load or not. If the instruction is a strided load, its PC is put in the new field *StridedPC* of the entry corresponding to its destination logical operand in the extended rename map table.

If the fetched instruction is a load but not a strided load, the *StridedPC* field is set to 0. Arithmetic instructions copy the content of the field *StridedPC* of their source operands to the ones of their destination operands. In parallel, a copy of the fields *StridedPC* of the source operands is stored in the ROB. As the number of *StridedPCs* is limited (2 per entry), only the first *StridedPC* of the source operands is propagated to the destination operand.

This strided load detection and propagation is performed continuously during instruction execution to ensure that independent instructions after the L2-miss load know these PCs when the selection of instructions is fired.

C. Second step: strided load selection

As soon as a L2-miss load is committed this second step is fired. In this step, the target strided loads for replication are selected. As said before, only independent instructions of the L2-miss load will be replicated. The independence check for instructions following a L2-miss load is performed when the instructions commit to reduce the amount of extra hardware.

To implement this step, few modifications in the commit stage are needed.

Stride load selection involves two main structures: a table called delinquent load table (*DLT*) where the PCs of the L2-miss load are stored and a mask of bits (one per logical register) called *DM* (Dependence Mask) used to know whether an instruction is dependent on the L2-miss load. An extension of the stride predictor (every entry is extended with a bit called *S*) is also needed.

When this step is fired, the mechanism clears every position of the *DM* except the bit corresponding to the destination operand of the L2-miss load. Following instructions set the bit corresponding to their destination operand with the logical OR of the bits corresponding to their source operands. Independent instructions are those instructions with all the bits of their source operands cleared.

Once independent instructions are detected, they select the strided loads from which they are dependent. To perform this, the bit *S* of the stride preditctor is set for every strided load an independent instruction depends on and which PC is stored in the entry of the ROB for this instruction.

The last structure, the *DLT*, holds up to 8 PCs of the L2-miss loads for which the mechanism has previously selected instructions. If no entry is available in this table, the LRU load is replaced. Every committed non-L2-miss load checks its PC in this table. If the PC is found, following independent instructions of this load clear the bit S for the strided loads they depend on. This prevents previously selected strided loads from keeping being selected, and replicated, when a load, that has previously missed the L2, hits the L1 or the L2.

D. Third step: instruction replication

Once a strided load with the corresponding *S* flag set is fetched and decoded, multiple replicas of it are created and dispatched to the issue queue. Every replica writes its computation in a different scalar register and they will read the data from a different memory address created with the last effective address and the stride detected with the stride predictor. Arithmetic instructions are replicated when they are dependent on a replicated instruction (instructions *IS1*, *IS2* and *IS3* in Figure 2b).

Every replicated instruction must know which physical scalar registers hold the speculative data created by its replicas. For this purpose, a new table called *RM* (Replication Maps) is implemented. This table is indexed by the PC and stores the PCs of the replicated instructions and a set of fields for data validation purposes.

Replication of selected instructions begins by allocating a set of physical scalar registers whose identifiers are stored in the field *REGS_ID* and the number of them is stored in the fields *NREGS* and *issue*. The *decode* and *commit* fields are cleared and the *PC1* and *PC2* fields are initialized with the PCs of the producers of the source operands. These PCs are obtained from the field *PPC* (Producer PC) of the extended rename map table (figure 3). This field, *PPC*, holds the PC of the replicated instruction that produces the value for the logical register of the corresponding entry in the rename map table.

The *AC* and *Range* fields will be explained in sections III.G and III.H.

The fields *decode* and *commit* track the state of the set of registers. When a new instance of a replicated instruction enters into the decode stage, it increases the field *decode* to make it point to the next replica that will be validated. The *commit* field indicates which replicas have been validated (subsection III.E).

The *issue* field is used to know how many replicas have not still executed. When the result of a replica is written back, the field *issue* of the corresponding entry of the *RM* is decreased. To avoid wrong register allocation, an entry of the *RM* cannot be deallocated until the field *issue* is 0. Moreover, when deallocating an entry, the mechanism must ensure that no speculative data is in use. To ensure this, only entries with the fields *decode* and *commit* with the same value and the *issue* field equal to zero, can be deallocated.

Finally, the *SR* (Source Register) field is used when an instruction is replicated with one non-replicated source operand. The value of this operand is stored in this field for validation purposes.

When replicating an instruction, the mechanism looks for an empty or a deallocatable entry in the *RM*. If several entries are candidates, the LRU is chosen. If no entry is available, the instruction is not replicated.

If an instruction is successfully replicated, the mechanism sets the bit *R* and puts the PC of the instruction in the *PPC* field of the entry of the extended rename map table corresponding to the destination

logical operand. The bit *R* is used to know which arithmetic instructions are dependent on replicated instructions, and thus, they must be replicated. The *PPC* field is used for validation purposes (see validation details on next section).

E. Fourth step: data validation

To use the precomputed data in a non-speculative way, the mechanism must check if the speculation is correct.

Every time an instruction is fetched, its PC is looked up in the *RM* table to know if the instruction has been previously replicated and therefore data must be validated. Checking whether the producer identifiers (*PPC* in the rename map table) of the source operands are equals to those stored in the *RM* (fields *PC1* and *PC2*) validates an arithmetic instruction. For arithmetic instructions with one non-replicated source operand, the field *SR* is checked to see if the value of the scalar register used when replicating the instructions remains unchanged. Strided loads must check if the stride keeps being the same. Replicated instructions pass to the commit stage to check the correctness of the speculation and to mark the data as validated, and it does not execute. When the instruction is committed, the field commit of the corresponding entry in the *RM* is increased and if this value is equal to the one of the *NREGS* field, a new set of replicas is created.

Replicated instructions store their PC in the field *PPC* of the rename map table and set the field *R* to propagate the replication characteristic.

When the speculation is wrong a recovery action is performed from the mispredicted instruction onwards reexecuting normally to ensure that dependent instructions have the correct data. Furthermore, resources allocated (the entry in the *RM* and the set of registers) for the replicated instructions are deallocated, and if possible, new replicas are created with the new source operands or detected stride.

F. Other microarchitecture considerations

Due to the nature of replicas, two main modifications have been implemented. Since replicas are created and executed out of the critical path, and the precomputed values will not be used as soon as created, a hierarchical register file has been implemented to store the speculative data.

Replicas also exploit the data locality. To enhance the L1 data cache access, a wide bus is provided.

Some other architectural considerations are also discussed.

G. Branch mispredictions

When a branch misprediction is detected, the *RM* table for the mechanisms can be in an inconsistent state (the *decode* and *commit* fields are not equal, meaning that there are values that are going to be validated). To solve this easily, on branch mispredictions, the *commit* field of the *RM* is copied in the *decode* field squashing the validation instructions after the mispredicted branch. Furthermore, the field *AC* is increased, and when equal to *MAX_AC*, the mechanism deallocates the resources (scalar registers) and the entry in the *RM* of the instruction. This field *AC* is used to detect dead associations (i.e. replicated instructions will not reenter the pipeline) among replicated instructions and allocated set of registers. High values of *MAX_AC* avoid the early release of set of registers, but force the mechanism to use more registers to allocate data for replicas because registers and replicated instructions are associated for too long. Low values of *MAX_AC* are useful for early detection of dead associations but provoke that most of the speculative data will be discarded before it is used increasing the speculative traffic. Our studies show that 2 is the optimal value for *MAX_AC*.

H. Memory consistency

As the objective of replicas is to precompute values for instructions beyond the instruction window, scalar registers of strided load replicas hold memory values that can be modified by stores, provoking memory inconsistency. A simple approach to solve this problem is to deallocate entries (and associated registers) of the *RM* of the replicated loads, whose field *RANGE* includes the memory address of a committing store. Even if the *RM* table is small, this may cause the deallocation ofseveral entries of the *RM*. To simplify the hardware, up to 2 stores can be committed per cycle, and the commit latency for stores has been increased by 1 cycle.

I. Hierarchical register file

The extra speculative traffic increases the pressure on the register file. To alleviate this, a two-level hierarchical register file has been considered. The low level (the fastest one) has been implemented similar to a monolithic register file. The upper level (the slowest one) is used to store the precomputed values of replicas. With this organization, the mechanism exploits the non-criticality of replicas since values created by replicas will normally not be used until some time after they are created. When a validation instruction enters the decode stage, it allocates a register in the low level and a copy instruction is inserted in the issue queue. When executed, this instruction performs a movement of the value from the upper level to the lower one. Dependent instructions will use the register of the lower level as source operand. Validation instructions pass from decode to the commit stage to validate the speculative data without waiting for the execution of the copy instruction. Both registers, (from the lower and the upper levels) are deallocated when the following instruction with the same logical destination register commits. For our simulations, the copy instructions have a latency of 2 cycles, and up to 4 values can be moved between levels.

J. Wide bus

Instruction replication exploits the spatial locality of data since most of the strided loads have a unit stride. For this purpose, buses of the data cache are assumed to be wide. A wide bus can read a whole cache line for every access and multiple outstanding loads can use these data. For complexity considerations (reducing the number of ports to the register file) up to four outstanding loads can be served during a wide access.

IV. PERFORMANCE EVALUATION

A. Experimental framework

For the perfomance evaluation we have extended the *SimpleScalar v3.0c* to include the microarchitectural extensions described before.

The baseline microarchitecture is an 8-way issue superscalar processor. Replication always creates up to 4 replicas per instruction. Other parameters of the microarchitecture and the mechanism are shown in table 1.

TABLE I

PROCESSOR CONFIGURATION

Parameter	*Value*
Fetch width	8 instructions (up to 1 taken branch)
I-Cache	64Kb, 2-way set associative, 64 bytes lines, 2 cycle hit, 10 cycle miss time
Branch predictor	Gshare with 64K entries
Inst. window size	256 entries (128 entries in the IQ)
Scalar functional units (latency in brackets)	6 simple int (1); 3 int mult/div (2 for mult and 12 for div); 4 simple FP(2); 2 FP mult/div (4 for mult and 14 for div)
Load/store queue	64 entries with store-load forwarding
Issue mechanism	8-way out-of-order issue; loads may execute when prior store addresses are known
D-Cache	64Kb, 2-way set associative, 32 byte lines, 2 cycle hit, write-back, 10 cycle miss time, up to 16 outstanding misses
L2 cache	256Kb, 4-way set associative, 64 byte lines, 10 cycle hit time, 400 cycle miss time
Commit width	8 instructions
Stride predictor	4-way set associative with 512 sets
Prefetch mechanism	Prefetch up to 4 elements per L2-miss load. Prefetch begins when L2 miss is detected and stride is confirmed. Prefetch data to L2 data cache

For the experiments we use the whole SPEC2000 benchmark suite, which includes programs with a wide range of percentages of L2 misses. Each program was simulated for 100 million instructions skipping the initialization part.

The *RM* table is 4-way set associative with a total of 64 sets, given a total of 10,5Kbytes of extra hardware. The upper level of the hierarchical register file has 768 registers.

The total amount of extra hardware, without counting the control logic, for the *L2miss* mechanism is 11048 bytes of storage (512 bytes of extended rename map table, 32 bytes from *DLT* and 8 bytes from *DM*).

B. Performance evaluation

To evaluate the mechanism we will assume a baseline configuration with a wide bus to the L1 data cache. Furthermore, the baseline configuration includes an aggressive prefetch (wb+pref in Figures). This is done to emphasize the benefits of the proposed mechanism beyond those coming from just prefetching. Statistics of the proposed mechanism with the aggressive prefetch will be also provided (miss+pref in Figures).

Figure 3 shows the IPC of the proposed mechanism compared with the baseline (wb) and the baseline with aggressive prefetch (wb+pref).

As shown in Figure 3, the aggressive prefetch mechanism improves the baseline about by 11,2% on average, by successfully reducing the number of L2 load misses by about 40%.

Our mechanism outperforms the baseline by about 29%. However, when compared against the baseline with aggressive prefetch, this speedup is reduced to 16%. Figure 3 also shows that there is room for improvement.

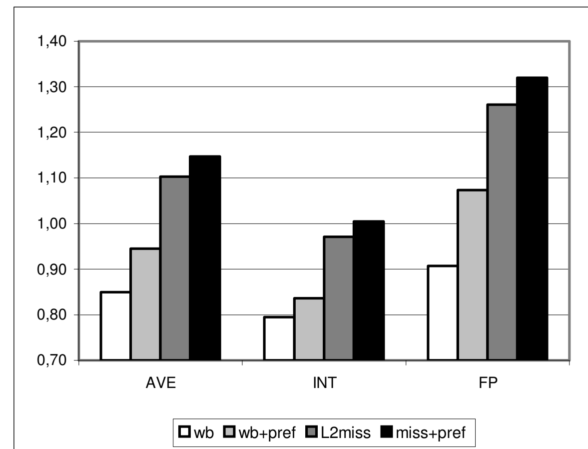


Fig. 3. Performance comparison between mechanisms.

When implementing the proposed mechanism on top of the configuration that already includes a prefetcher, there is a noticeable performance improvement (21%). The reason is that the prefetch schemes augment the potential of the proposed mechanism. Prefetch increases the performance by removing strided-based L2 misses. Our proposals are basically fired on L2-miss loads. When these mechanisms are combined, prefetch prevents our mechanism to be fired on easy-to-predict L2 misses, focusing in those L2-miss loads without a clear access pattern. So, there is a net performance increment due to prefetch easy-to-predict L2-miss loads, and preexecution of independent instructions of not prefetcheable loads.

There are three main sources of performance improvement. The first one is the memory subsystem. The ability of going beyond the instruction window allows our mechanism to avoid future L2 misses even before the prefetch detects them. Furthermore, the ability of preexecuting instructions allows prefetching loads without a clear access pattern. This L2 miss reduction is show in Figure 4.

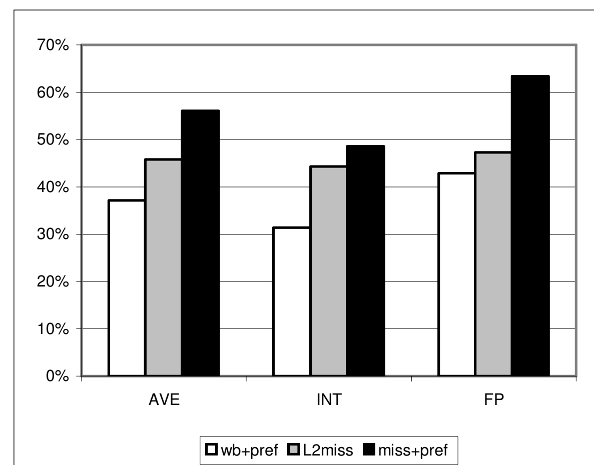


Fig.4. L2 miss reduction for the proposed mechanisms.

Furthermore, our mechanism is able to manage more efficiently the wide bus due to the high percentage of unit strides and the exploitation of data locality

explicitly denoted by replicas, reducing the total amount of accesses. Figure 5 shows the percentage of accesses in which 1, 2, 3 or 4 elements can be served from the same L1 access, for the baseline configuration compared with the proposed mechanism. Although this exploitation is not important in the baseline configuration, compared with a superscalar processor with 2 scalar ports, the wide bus suffers a slowdown of less that 2%. Note that the control logic of the wide bus is simpler than that of 2 scalar ports.

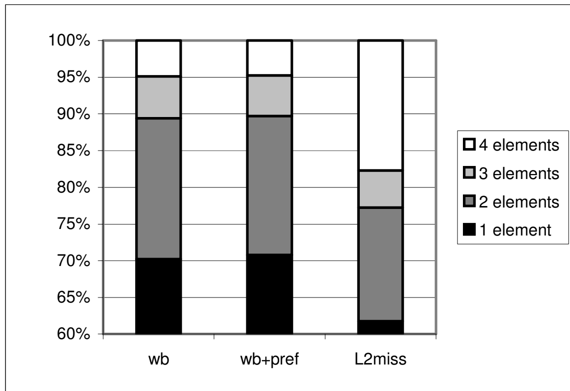


Fig. 5. Percentage of number of elements bypassed per wide bus access.

The second source of performance improvement is the reuse of speculative precomputed data. Figure 6 shows the number of commit instructions that cannot reuse data (black portion, commit), the commit instructions that reuse data (dark grey, reuse), the speculative instructions created by branch mispredictions (light grey, specbp) and the number of speculative traffic created by our mechanism (white, specL2).

Finally, the third source of improvement is the virtual enlargement of the instruction window, due to the ability to create data for instructions that have not enter into the pipeline. With smaller instruction windows, thus, reducing processor's complexity, we can achieve nearly the same IPCs (in configurations with 64 entries in the ROB and an issue queue of 32 instructions the *L2miss* mechanisms only loses 0,5% of IPC compared to configurations with 256 entries in the ROB and an issue queue of 128 instructions).

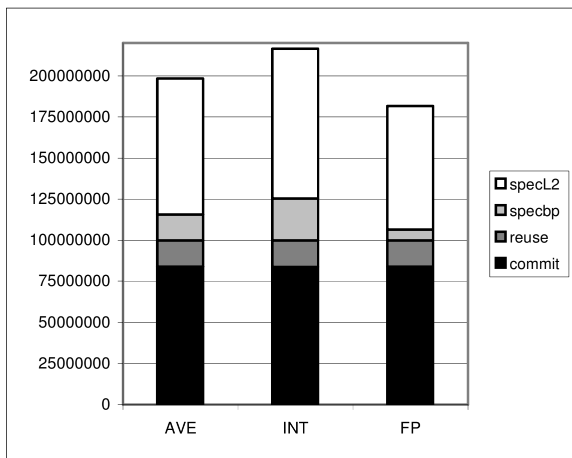


Fig. 6. Distribution of instructions splitted in commit without reuse, commit with reuse, speculative instructions due to branch mispredictions and total extra traffic created by the proposed mechanism.

V. Related Work

Prefetch mechanisms have been widely studied in literature. Software approaches [3] rely on the compiler to detect and evict cache misses at compile time.

Hardware prefetching schemes [1] study the access patterns of memory instructions to predict the next memory address to prefetch data.

Prefetch in multithread scenarios [2] uses secondary threads to prefetch data for a primary thread.

Dundas in [4] proposed run-ahead execution. This mechanism creates special values for L2-miss loads that are propagated to dependent instructions. Independent instructions continue execution until the instruction window is full. When the data of the L2-miss load is available, a recovery action is performed to re-execute correctly the instructions following that load.

Several mechanisms to enlarge virtually the instruction window [4][6] have been proposed in literature. These mechanisms perform out-or-order retirement of instructions to avoid stall cycles due to the lack of entries in the ROB. Complex advanced checkpointing mechanisms are used to recover the state of the processor e.g. in branch mispredictions.

Lebeck in [5], proposed a scheme where instructions dependent on a long-latency operation are moved from a small (and faster) scheduling window to a large (and slower) waiting buffer until the operation is completed.

VI. Conclusions

In this paper we have proposed a novel mechanism to preexecute instructions independent of a L2-miss load, preexecuting instructions as soon as they are found, starting with the strided loads above the L2-miss load. As shown in the paper, a moderate amount of extra hardware is needed to implement this mechanism (10.5 Kbytes) obtaining about 21% of speed-up.

VII. Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01 and by a research grant from Intel Corporation.

VIII. References

[1] J. Baer and T. Chen, "An Effective On-chip Preloading Scheme to Reduce Data Access Penalty", *in Proceedings of Supercomputing '91*, 1991.
[2] R. Balasubramonian, S. Dwarkadas and D. Albonesi, "Dynamically Allocating Processor Resources Between Nearby and Distant Parallelism", *in Proceedings of the 28th Symposium on Computer Architecture*, 2001.
[3] D. Callahan, K. Kennedy and A. Porterfield, "Software Prefetching", *in Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.
[4] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss", *in Proceedings of the ICS*, 1997.
[5] R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses", *in Proceedings of the 29th ISCA*, 2002.
[6] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors", *in Proceedings of the 35th International Symposium on Microarchitecture*, 2002.