Control-Flow Independence Reuse via Dynamic Vectorization

Alex Pajuelo, Antonio González and Mateo Valero
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona – Spain
{mpajuelo, antonio, mateo}@ac.upc.es

Abstract

Current processors exploit out-of-order execution and branch prediction to improve instruction level parallelism. When a branch prediction is wrong, processors flush the pipeline and squash all the speculative work. However, control-flow independent instructions compute the same results when they re-enter the pipeline down the correct path. If these instructions are not squashed, branch misprediction penalty can significantly be reduced.

In this paper we present a novel mechanism that detects control-flow independent instructions, executes them before the branch is resolved, and avoids their re-execution in the case of a branch misprediction. The mechanism can detect and exploit control-flow independence even for instructions that are far away from the corresponding branch and even out of the instruction window.

Performance figures show that the proposed mechanism can exploit control-flow independence for nearly 50% of the mispredicted branches, which results in a performance improvement that ranges from 14% to 17,8% for realistic configurations of forthcoming microprocessors.

1. Introduction

Current processors' potential [13] to exploit instruction level parallelism depends on their ability to build a large instruction window. Branch instructions are the main problem [10] to build such large instruction windows for non-numeric applications. Every time a branch prediction is wrong, the pipeline is flushed, and the instruction window is built again through the correct path. However, control independent instructions, i.e., instructions that are encountered in every branch path computing the same values, could theoretically remain in the instruction window and their re-execution could be avoided.

In this paper, we present a hardware mechanism that tries to identify control-flow independent instructions to speculatively execute them ahead of time and avoid their re-execution in case of branch mispredictions. The effect of this technique is a net increase in the effective instruction window size and thus, in performance. Moreover, the mechanism can pre-execute control-flow independent instructions before they enter the pipeline, increasing, virtually, the instruction window.

We show that the proposed technique can exploit control-flow independence for about 50% of the dynamic mispredicted branches in SpecInt 2000 benchmarks, which results in an average performance improvement of 17,8%. Due to the way the mechanism precomputes values, it is not necessary to store them in the register file, so we propose a possible implementation that consists in the addition of a simple and cheap slow memory to hold those speculative values. Numbers showing that this small memory achieves the same performance as an unbounded monolithic register file are provided.

The rest of this paper is organized as follows. Section 2 describes the proposed approach. Section 3 analyzes the performance of the proposed scheme. Related work is outlined in section 4. Finally, section 5 summarizes the main conclusions of this work.

2. The Approach

2.1. Control-Flow Independent Instructions

An instruction is control-flow independent with respect to a given branch instruction if its result is the same regardless of the branch outcome. Control-flow independent instructions are common in hammock control flow structures resulting from *if-then-else* constructs. An example is shown in Figure 1.

```
I1:        MOV R1, 0
I2:        MOV R2, 0
I3:        MOV R3, 0
I4:        MOV R4, 0
I5: loop: LD  R0, a[R1]
I6:       CMP R0, 0
I7:       BE  else
I8: then: INC R2
I9:       BR  IP
I10:else: INC R3
I11:IP:   ADD R4, R4, R0
I12:      ADD R1, 4
I13:      CMP R1, 400
I14:      BLE loop
```

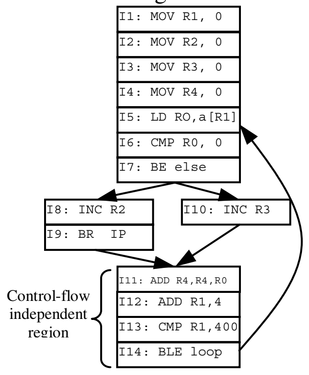

Figure 1: Sample code with a hammock.

The code in Figure 1 counts how many elements of vector *a* are equal to zero (stored in register *R3*) and how many are not (stored in register *R2*). Furthermore, the code accumulates the sum of all elements of vector *a* in register *R4*.

The branch at instruction *I7* may be hard to predict (e.g., the data of vector *a* does not follow any regular pattern). However, instructions *I11-I14* are executed and produce the same results regardless of the branch outcome.

The first sequential instruction that is common to both taken and not taken paths of a branch will be referred to as the *re-convergent point*. In Figure 1, instruction *I11* is the re-convergent point of branch *I7*. Control-flow independent instructions can be located starting from the re-convergent point onwards.

2.2. Overview of the Mechanism

The proposed mechanism works in four steps. The first two steps select the control-independent instructions when a branch misprediction is detected. The last two steps, effectively vectorize the selected instructions. The selection part and the vectorization part work separately and are communicated through just one bit in the stride predictor (as explained later in section 2.3.2 and 2.3.3). These steps are explained now following the example in Figure 1.

- First step: when a hard-to-predict conditional branch is detected (see details later in Section 2.3.1) and mispredicted, the mechanism tries to find the re-convergent point of that branch. Supposing that I7 in Figure 1 is the mispredicted branch, the first step has to find I11 as the re-convergent point.
- Second step: identification of the instructions (I11, I12 and I13) after the re-convergent point (included) that are likely to produce the same outcome after the branch misprediction recovery and can be effectively vectorized (only I11 can be vectorized). For this purpose, every instruction after the re-convergent point is analyzed and it is checked whether its source operands have been changed by an instruction after the branch and before the re-convergent point. If the source operands have not changed, the set of nearest strided loads above the branch on which the instruction depends are selected for speculative vectorization [12][22]. In the example of Figure 1, the first instruction whose operands have not changed after the branch is the re-convergent point itself (instruction *I11*). The loads above the branch on which instruction *I11* depends is just instruction *I5*.
- Third step: speculative vectorization of the selected instructions next time they are encountered. Vectorization is performed by generating multiple speculative replicas of the vectorized instruction. These speculative instructions are dispatched to the issue queue and executed but not committed until they are verified. Moreover, every time an instruction is fetched, it is checked whether any of its source operands is the outcome of a previously vectorized instruction, and if this is the case, it is also speculatively vectorized. In the example of Figure 1, instruction *I5* is the selected strided load that will be vectorized. Instructions *I6* and *I11* will also be vectorized because they are dependent on instruction *I5*. Notice that *I11* is a control-flow independent instruction.
- Fourth step: every time an instruction is fetched, it is checked whether it was previously vectorized. If so, it is checked whether the vectorization was correct, and in this case, the instruction is just marked as completed and sent to the commit stage. Otherwise, the instruction is normally executed.

These steps are further detailed below.

2.3. Implementation of the Mechanism

Now, we are going to explain in detail, how those 4 steps of our mechanism work.

2.3.1. First step: Hard-to-predict Branches and Re-convergent Point Detection

First of all, in order to apply the control independence scheme to branches that are responsible for a significant number of mispredictions [8][9], a table that we refer to as the MBS table (Mispredicted Branch Status) is used. This table is indexed by the PC of branches and has a 4-bit saturated up-down counter per entry. The counter is increased by taken branches and decreased by not taken branches, if the direction is the same as the previous outcome. Otherwise, the counter is set to the value that is in the middle of its range. If the value of this counter is the maximum or minimum value, the branch is considered to be highly biased and thus assumed to be easy to predict. Otherwise, the control independence scheme is activated.

The scheme to identify re-convergent points for mispredicted branches is an extension of previous work in [5] and involves basically two hardware structures. The first one is a queue, called NRBQ (Not Retired Branch Queue), where the estimated re-convergent points of the in-flight conditional branches are stored. The second is the CRP (Current Re-convergent Point).

Identification of re-convergent points does not need to be correct. Wrongly estimated re-convergent points will affect the performance of the processor but not the correctness of the execution. Re-convergent points are estimated with the following heuristics.

If the branch is a backward branch [18][21], the re-convergent point is assumed to be the next instruction, in program order, that follows the branch (a backward branch usually corresponds to the closing branch of a loop as shown in Figure 2-a).

If the branch is a forward branch, the instruction situated one location above the target address [3] is fetched and analyzed. If the branch is predicted as taken this instruction is fetched in the next cycle, possibly together with the target instructions and succeeding ones. If the branch is predicted as not taken, this instruction is fetched just after the recovery of the misprediction. If this instruction is an unconditional forward branch (which is the common case for an *if-then-else* structure as shown in Figure 2-c), the re-convergent point is assumed to be the address pointed by this branch. Otherwise, the re-convergent point is assumed to be the destination address of the conditional branch (which is the common case for and *if-then* structure as shown in Figure 2-b).

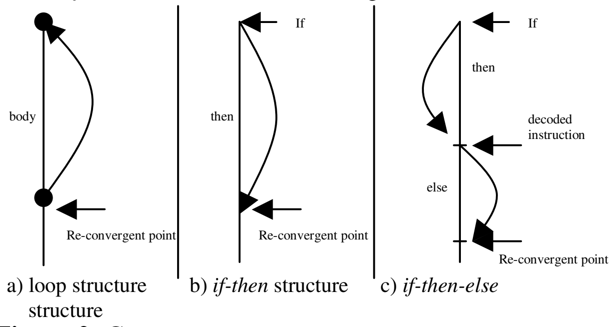

Figure 2: Common program constructs.

When a branch is executed its prediction is checked. In case of a misprediction, younger instructions are squashed and if the static branch is supposed to cause many dynamic branch mispredictions, the information regarding this static branch is introduced into the CRP register (Current Re-convergent Point). This register contains the PC of the re-convergent point and the R (Reached) flag that indicates whether the re-convergent point has been reached.

2.3.2. Second step: Control-Independent Instruction Detection and Filtering

Every time an instruction is fetched, its PC is checked with the PC stored in CRP. If they match, the R flag is set, which indicates that the re-convergent point has been reached. To identify whether an instruction after the re-convergent point does not depend on the instructions between the branch and the re-convergent point, every entry of the NRBQ is extended with a mask of bits. Each bit is associated to a logical register and indicates whether this logical register has been written after this branch and before the next branch. When a branch is found, the corresponding mask is cleared. For each new instruction, the bit corresponding to the destination register is set to one for the entry at the tail of the NRBQ. After a branch misprediction, the information of the mispredicted branch is copied into the CRP as described above. The CRP has also a mask of bits that in this case, indicates whether or not the corresponding logical register has been written since the branch was fetched and before the re-convergent point is reached (in either the wrong or the correct path). In a branch missprediction, the CRP mask is initialized by ORing all the masks in NRBQ starting from the mispredicted branch to the branch at the tail of the queue (i.e. the youngest one). Afterwards, for every new decoded instruction before the re-convergent point is found, the bit corresponding to its destination logical register is set to 1.

An instruction is considered to be control independent if it is fetched after the re-convergent point, and its source operands have their corresponding bits cleared in the mask of the CRP. These instructions will be the target of the speculative vectorization scheme. In addition, all instructions that belong to any dependence chain of the backward slice (i.e. all its predecessors) of a selected instruction are also vectorized if the chain starts with a strided load. For example, in the code of Figure 1, *I11* and *I5* are vectorized if *I5* has been observed to follow a strided pattern. But instructions I12 and I13, even if they are control independent, will not be considered for our mechanism given that they are not dependent on a strided load. In the worst case, if I5 is not an strided load, no instruction will be vectorized in the example of Figure 1.

To identify these backward chains that start with a strided load, every time a load is fetched the stride predictor [7] is checked and if the load is considered to follow a strided pattern, its PC is associated to the logical destination register. In our scheme, the stride predictor is implemented using a table that is indexed by the PC of the load instruction, and contains the PC of the instruction, the last accessed address and the last observed stride, as shown in Figure 3. A confidence field is also included, which is implemented as a two bit up-down saturating counter. The prediction is trusted when this field has a value greater than 1. The S flag indicates whether this load has been selected for speculative vectorization, as described later.

PC	Last address	Stride	Confidence	S
64	64	64	2	1

Figure 3: Stride predictor entry with the length, in bits, of every field.

To propagate the PC of a strided load down the dependence graph, every entry in the rename map table is extended with a new field called *stridedPC*., *where the PC of the strided load is stored.* Arithmetic instructions propagate the *stridedPC* of their source operands to their destination. In theory, one instruction may have many strided loads as in its backward slice. However, we have experimentally evaluated that SpecInt2000 needs on average 1,7 PCs per entry. In fact, increasing the number of PCs per entry from 2 to 4 hardly changes the performance, as shown in Figure 4 (details of the architecture are later described in section 3.1).

When an instruction after the re-convergent point is selected for vectorization, the strided loads on which it depends are also selected for vectorization, by setting to 1 the flag S in the stride predictor. When the selected load reenters the pipeline, it checks whether the stride keeps on being the same, and in this case, this load is vectorized. Every time an instruction is fetched, if any of its source operands is vectorized, the instruction is also vectorized.

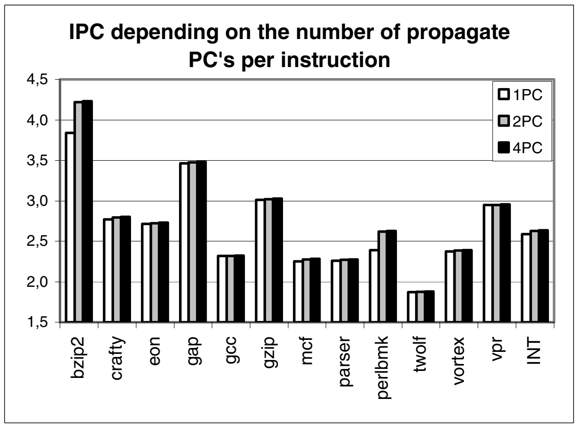

Figure 4: IPC depending on the number of propagated PCs per instruction.

Figure 5 shows the percentage of mispredicted branches for which the mechanism does not find any control independent instruction (white portion), selects at least one control independent instruction (gray portion), and selects control independent instructions and successfully reuse precomputed instances (through speculative vectorization) of them (black portion) for SpecInt2000. Control independent instructions are selected for about 70% of the mispredicted branches (black and gray portions). For 49% of the mispredicted branches (black portion), at least one control-independent instruction is correctly vectorized. The remaining 21% of the mispredicted branches (gray portion) where vectorization is not successful are basically due to the fact that they do not depend on strided loads.

It is important to remark that speculative vectorized instructions perform work that is beyond the current instruction window. These speculative instructions can be at any distance from the mispredicted branch of which they are control independent.

2.3.3. Third step: Instruction Replication

Once a load with the corresponding S flag set is fetched, multiple instances of it are speculatively dispatched to the issue queue. Each dynamic instance will read a different memory address, which is computed by adding to the last effective address the stride multiplied by the order rank of the dynamic instance (*current_address+(stride*n)*). Depending on where to store the precomputed data, replicas will use, as an outcome, a different scalar register (monolithic register file) or a different position of the provided small memory (see details later is Section 2.4.6) The processor keeps on fetching instructions in the conventional sequential approach. When a replicated instruction is fetched again, the first speculative instance is validated and if the validation is correct, the instruction is just marked as completed. Following replicas will be validated in the same way. When the last replica is validated, another set of multiple speculative instances of the instruction are dispatched again.

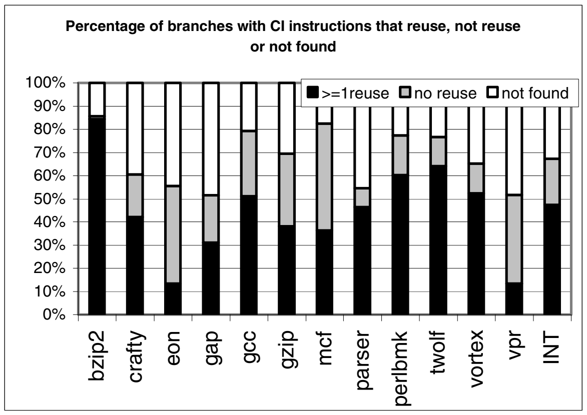

Figure 5: Percentage of mispredicted branches for which no control independent instruction is selected, (white portion), at least one control independent instruction is selected (gray portion), at least one control independent instruction is correctly vectorized (black portion).

These multiple instances are managed by means of an additional table, the SRSMT table (Scalar Register Set Map Table). This table is indexed by the PC and stores the PCs of the replicated instructions and several other fields, as shown in Figure 6 (the purpose of the last two fields will be explained later in Section 2.5).

The *Set of registers* field holds the identifiers of the physical registers (for monolithic register files or positions for implementations following the memory implementation detailed in Section 2.4.6) that will be used as destination registers for the replica instructions and the field *Nregs* stores the number of registers that have been allocated (which equals to the number of replicas). Note that, in the case that not enough free registers are available for the desired number of replicas, a lower number of replicas or none at all are created.

PC	Set of registers	Nregs	Decode	Commit	Issue	Seq1	Seq2	DAEC	Range
64	4x8	2	2	2	2	64	64	2	2x64

Figure 6: Entry in the SRSMT assuming 4 replicas per instruction and 256 available registers. Field length is in bits.

The next two fields, the *decode* and the *commit* fields, track the state of the set of replicas. The *decode* indicates which is the next replica to be validated. This field is incremented when a new dynamic instance of the instruction enters into the decode stage. The *commit* field indicates the last replica that has been committed. This field is incremented when a dynamic instance of an instruction is successfully validated and commits. When a recovery action is needed, (e.g. in case of a branch misprediction) the state of the table can be easily recovered by copying the content of the *commit* field into the *decode* field for every entry of the table. When the *decode* and *commit* fields are equal, the entry in the SSRMT is deallocated. Note that this does not imply the deallocation of physical registers.

The *issue* field holds the number of replicas that are being executed (i.e., have been issued but their execution has not finished). The purpose of this field is later discussed in this section.

The next two fields, *seq1* and *seq2,* identify the instructions that compute the source operands if they have been vectorized, or the value of the scalar operand otherwise (not all source operands must be vector operands). The identifier of a vectorized instruction is its PC (also called *Seq or Sequence*).

The rename table is extended to include for each logical register whether the latest instruction that writes to it has been vectorized and if this is the case, it contains the PC of that instruction. Figure 7 depicts and entry of the rename table.

Physical reg.	V/S	Seq	stridedPC
8	1	64	64

Figure 7: Entry of the rename map table. Lengths of fields are in bits.

When an instruction is vectorized, an entry is allocated in the SRSMT table. A free entry is chosen but if none is available, an entry is tried to be deallocated. An entry can be deallocated when the fields *decode* and *commit* have the same value, and the field *issue* is set to 0. If several entries are candidates to be deallocated (depending on the indexing function) the LRU is chosen. When an entry is deallocated, the resources allocated by it are released. If no entry can be deallocated, the instruction is not vectorized.

The identifiers of the source operands for a newly vectorized instruction are obtained from the rename table. If an operand is scalar, its value is read from the register file. If the value is not ready, the instruction and following ones are stalled.

2.3.4. Fourth Step: Speculation Validation

Every time an instruction is fetched, its PC is looked up in the SRSMT and if found, it means that the instruction has been vectorized and must be validated. Validation of arithmetic instructions consists in checking whether the producer's identifiers currently found in the rename table for its source operands are equal to those of the SRSMT validates. For a load, the stride must keep on being the same. If these checks are successful, the instruction is not executed and it is sent to the commit stage where it will finalize its validation. In the commit stage it will wait until the fields *decode* and *commit* of its source operands in the SRSMT table are equal. When it commits, the *commit* field of its entry in the SRSMT is increased. Notice that every dynamic instance of a replicated instruction sets the bit V/S to 1 and the field *sequence* is set to the "sequence" of the instruction in the rename map table.

If the speculation is not correct, the corresponding entry in the SSRMT and the scalar registers associated to the replicas of this instruction are deallocated, and new replicas are created with the new operands.

2.4. Other Microarchitecture Considerations

2.4.1. Issue Logic

Speculative vectorized instructions are given less priority than the rest. In case of a branch misprediction they are not squashed (they are supposed to be control independent). These speculative instructions are not committed (their activity is committed by the corresponding scalar instruction that does the validation). After execution, they are retired in the write-back stage.

2.4.2. Releasing Physical Registers

One of the critical resources of the processor is the register file. Scalar instructions and replicas use the register file to store the computed data. Furthermore, since speculative values are computed much earlier than in a conventional microarchitecture, their lifetimes are stretched and the register pressure is increased. Several implementations [1][6] have been discussed previously at literature to alleviate the problem. For out study we have chosen the inclusion of a small and cheap memory as described later in Section 2.4.6.

Physical registers are released after their value is used, using the same approach as a conventional processor does (i.e. when an instruction commits, the physical register allocated by the previous instruction with the same logical destination is released). However, there may be values that are never used due to wrong speculation. In that case, the associated registers are not released until the entry of the SRSMT is deallocated and given to another vectorized instruction. To release earlier these registers, a new field in the SRSMT, called DAEC (Dead Association Elimination Counter) is included. This field is incremented for those entries whose *decode* and *commit* fields have the same value (otherwise the counter is set to 0) when a branch misprediction recovery action is performed. When this field reaches the value of 2, the registers allocated by the corresponding vectorized instruction that have not been validated are deallocated. This avoids long lifetimes of values because the scheme supposes that if an entry is not used during several branch mispredictions, the speculative work done for this entry is wrong.

For SpecInt2000, in unified register file implementations, the average number of physical registers in use when their number is unbounded is 812 without this scheme (see details in section 3.1) and 304 if this scheme is in place. Notice that this counter prevents the utilization of large register files that can impact seriously in the cycle time.

2.4.3. Memory Coherence

When a strided load is detected, the mechanism creates speculative replicas that fetch data into scalar registers. When a store instruction commits and modifies the data of the L1 data cache, it checks whether this data has been speculatively loaded in any scalar register.

For this purpose, a conservative but simple mechanism is proposed. One extra field is included in each entry of the SRSMT to hold the initial and final memory addresses of each vectorized load instruction (this field is meaningless for other instructions). When a store commits, it checks if its address is inside the range of addresses of any entry and if this is the case, the entry and the resources assigned to the replicated instruction are deallocated, and the instructions in the conventional instruction window (i.e. all except those generated by the speculative vectorization scheme) that follow the store are squashed. To account for this extra activity, an additional cycle has been assumed for committing a store instruction and only up to 2 stores can be committed in the same cycle. Fortunately, less than 3% of the stores write into an address whose data has been previously read by a speculative load.

2.4.4. Branch Mispredictions

In a branch misprediction, the content of the *commit* field of each entry of the SRSMT table is copied into the corresponding *decode* field. No speculative vectorized instruction is squashed and no resource assigned to replicated instructions is deallocated. This gives the processor the opportunity of exploiting control-flow independence.

2.4.5. Wide Bus

Vectorizing loads opens the possibility of better exploiting spatial locality since most loads have a unit stride. For this purpose, buses of the data cache are assumed to be wide. A wide bus [11][14][23] can read a whole cache line for every access and multiple outstanding loads can use these data. To limit the number of additional ports to the register file, only up to 4 loads can be served in one of these wide accesses.

Figure 8 shows the number of cache accesses for the superscalar processor with scalar ports (*scalxp*), with wide buses (*wbxp*), and the mechanism of control independence (*cixp*), with 1 port (*x*=1) or 2 wide ports (*x*=2). The wide bus significantly reduces the number of cache access but the control independence mechanism reduces it further in spite of executing more speculative loads.

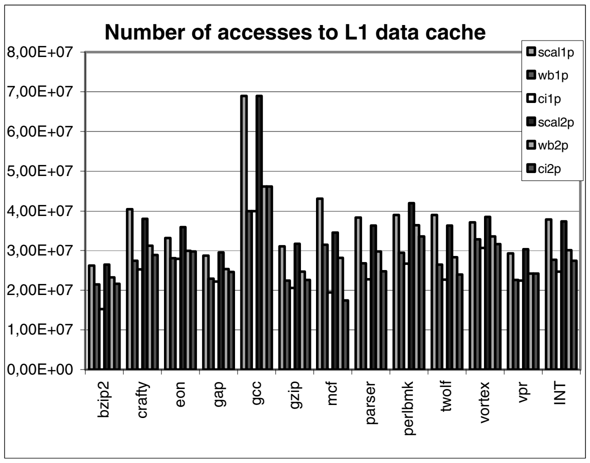

Figure 8: Number of accesses to L1 data cache for the baseline (*scalxp*), the baseline with a wide bus (*wbxp*) and the control independence mechanism (*cixp*) with one (x=1) and two wide ports (x=2).

2.4.6. Speculative Data Memory

In scenarios where the register file is critical, dynamic vectorization of control-independent instructions can produce slowdowns due to the lack of scalar registers. Although our mechanism does not need a large amount of scalar registers (on average 304 as shown in Section 2.4.2), it is always recommendable to alleviate the pressure in the register file and to avoid large register files that can impact seriously in the processor's cycle time. For this, we propose, as a possible implementation, the use of a small and cheap slow memory, similar to a hierarchical register file [6] to hold the speculative data created by replicas. Speculative instructions are assigned positions from this small memory as destination registers. Data cannot be read directly from this memory as input operands in the functional units because this memory is not connected to the functional units but the register file. Data movement between this memory and the register file is performed through a copy instruction inserted in the issue queue when a validation instruction enters the decode stage. This copy instruction allocates one register from the register file to store the value moved from this small memory. Dependent instructions of the validation instruction become dependent on the copy instruction to read the values from the register file. Validation instructions, as previously commented, pass from the decode stage to the commit stage to validate the speculative data without waiting for the execution of the copy instruction. Both the register of the register file and the position in this memory are deallocated when the following scalar instruction with the same destination logical register as the validation instruction commits.

This small memory only has two write ports from the functional units and two read ports to the register file. It is also twice slower than the register file.

3. Performance Evaluation

3.1. Experimental Framework

For the performance evaluation we have extended the SimpleScalar v3.0c [2] to include the microarchitectural extensions described above.

The baseline microarchitecture is an 8-way issue superscalar processor. To evaluate the register file impact, we use, for a clear evaluation, monolithic register file configurations of 128, 256, 512, 768 and infinite registers. Afterwards, we will provide numbers of the inclusion of the small memory to hold speculative data.

To evaluate the impact of the aggressiveness of the speculation, dynamic vectorization schemes that generate 1, 2, 4 or 8 replicas per scalar instruction are explored. Other parameters of the microarchitecture are shown in Table 1.

For the experiments we use the whole SpecInt2000 benchmark [20] suite because they include programs with significant number of branch mispredictions. Programs were compiled with the Compaq/Alpha compiler using –O5 –ifo –non_shared optimisation flags. Each program was simulated for 100 million instructions after skipping the initialization part.

From the description of the implementation of the mechanism, one could derive that the proposed structures are large and complex. But in fact they are not. Since control logic hardware requirements are difficult to compute without the layout of the processor, we only give numbers of the hardware needed for the structures. For the evaluated configuration, the size of the additional hardware required by the control-flow independence scheme is:

- The SRSMT occupies 11520 bytes (4 ways * 64 elements per way * 45 bytes per element).
- The stride predictor occupies 24576 bytes (4 ways * 256 elements per way * 24 bytes per element).
- The MBS occupies 2048 bytes (4 ways * 64 elements per way * 8 bytes per element).
- The NRBQ occupies 128 bytes (16 entries * 8 bytes per entry).
- THE CRP occupies 16 bytes (8 bytes from the PC and 8 bytes from the mask of bits).
- The extension of the rename map table occupies 1024 bytes (16 bytes per entry * 64 entries).

Parameter	Value
Fetch Width	8 instructions (up to 1 taken branch)
I-Cache	64Kb, 2-way set associative, 64 byte lines, 1 cycle hit, 6 cycle miss time
Branch Predictor	Gshare with 64K entries
Inst. Window size	256 entries
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); 1 load/store
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, 1 cycle hit time, write-back, 6 cycle miss time, up to 16 outstanding misses
L2 cache	256Kb, 4-way set associative, 32 byte lines, 6 cycle hit time, 18 cycle miss time
L3 cache	2Mb, 4-way set associative, 64 byte lines, 18 cycle hit time, 100 cycle miss time (main memory access time)
Commit width	8 instructions
Stride predictor	4-way set associative with 256 sets
SRSMT	4-way set associative with 64 sets
MBS	4-way set associative with 64 sets

Table 1: Processor configuration.

This results in a total of 39 Kbytes of extra storage easily affordable in current designs. On the other hand, using this amount of extra hardware in i.e., the L1 data cache only increases about 5% the performance of the processor, while we will report performance improvements of about 17%.

3.2. Performance Evaluation

Figure 9 shows the IPC obtained with the proposed mechanism (*cixp*) compared with a superscalar processor (*scalxp*) and a superscalar processor with wide buses (*wbxp*), for a varying number of L1 data cache ports (1 port *x*=1, 2 ports *x*=2) and a varying number of physical registers (128, 256, 512, 768 and infinite). In this first set of experiments, a single-level register file is considered. Harmonic means are used to average IPC across the whole benchmark suite. Vectorization creates 4 replicas per vectorized instruction.

Several conclusions can be drawn from Figure 9. First, we can see that wide buses provide a significant benefit for a superscalar processor. This is due to the fact that a wide bus exploits the spatial of memory accesses. As the number of ports increases, this performance benefit decreases since multiple ports can also exploit spatial locality although with a much higher implementation cost.

For the baseline configuration with and without wide buses, performance is significantly improved when the number of registers increases from 128 to 256, except for 1 non-wide port, which is mainly limited by memory bandwidth. For configurations with more than 256 registers the reorder buffer has been increased to the size of the number of registers; otherwise, many registers would be useless due to the lack of instructions in-flight. However, this hardly improves performance due to the branch mispredictions and the limited ILP of these applications.

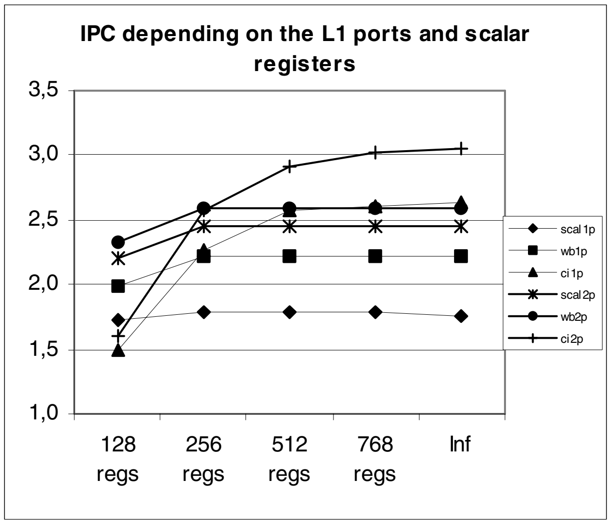

Figure 9: IPCs for a superscalar processor (*scalxp*), a superscalar processor with wide buses (*wbxp*) and a superscalar processor with wide buses and the proposed mechanism (*cixp*), for 1 L1 data cache port (*x*=1) and 2 ports (*x*=2), and for 256, 512, 768 and infinite registers.

When the mechanism for control independence is included, and enough registers are available, the performance increases more than 17,8% for both configurations (1 port and 2 ports) over the superscalar processor with wide buses. This is basically due to the exploitation of control independence, which allows the processor to execute instructions ahead of the resolution of the branches on which they depend, regardless of the correctness of the branch prediction, and even if they are far away of the current instruction window. The vectorization scheme also favors the exploitation of spatial locality, since the speculative instances of a vectorized load instruction are unit strided most of the times.

The control independence scheme increases the pressure in the register file due to longer lifetimes and wrongly speculated instructions. Because of that, when the number of registers is too low (128 registers for a 256-entry reorder buffer), the control independence scheme results in some performance degradation. For configurations with 256 registers, the control independence mechanism hardly affects performance when compared with the superscalar configuration with wide buses. This is due to the fact that a large number of scalar registers are used to store the values created by the speculative instructions, slowing down the execution of the code that has not been vectorized because less registers are available for it. However, when the number of physical registers keeps on increasing, the superscalar processor performance flattens out whereas the control independence scheme provides significant performance gains.

As discussed above, one of the main benefits of the proposed scheme is its potential to pre-execute instructions that are outside the instruction window (and potentially far away). To quantify this effect, we have simulated a hypothetical scheme where control independence is exploited only for the control-independent instructions that have entered the instruction window before the branch misprediction is detected and the recovery action is initiated (this scheme is sometimes referred to as *squash reuse*). Figure 10 shows the performance of that scheme (*ci-iw*) compared with a superscalar processor (*scal*), a superscalar processor with a wide bus (*wb*) and the proposed control independence scheme (*ci*).

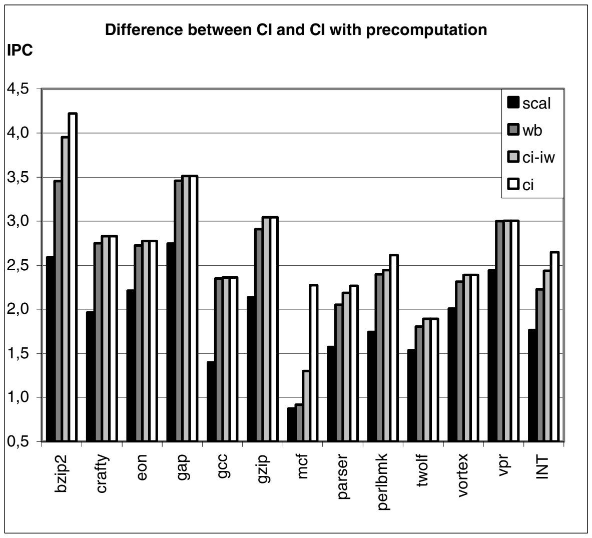

Figure 10: Performance of exploiting control independence only inside the instruction window (*ci-iw*). The other bars correspond to a superscalar processor with one L1 data cache port, either single (*scal*) or wide (*wb*), and the proposed control independence scheme (*ci*).

Exploiting control independence by reusing only instructions of the wrong path that have entered the instruction window provides an improvement of 9,1% whereas the proposed control independence scheme results in a 17,8% performance gain.

An important parameter of the proposed scheme is the number of speculative instances that are generated for every vectorized instruction. A higher number of speculative instances implies a higher potential to exploit control independence but also a higher pressure on the register file (i.e., more mispeculations and longer lifetimes). Figure 11 shows the effect when this parameter is varied from 1 to 8 instructions. From this experiment we can conclude that either 2 or 4 replicas per vectorized instruction seem the most convenient approach. Generating only 1 speculative version looses a lot of opportunities to exploit control independence. On the other hand, generating 8 replicas only improves performance by very little when the number of registers is very high.

Figure 12 shows the number of: a) committed instructions that do not reuse a precomputed value (dark portion), b) committed instructions that reuse a precomputed value (dark gray), c) fetched instructions that do not commit due to a branch missprediction (light gray), and d) speculative instructions generated by the control independence scheme (white) for 2 (left bars per spec) and 4 (right bars) replicas per vectorized instruction.

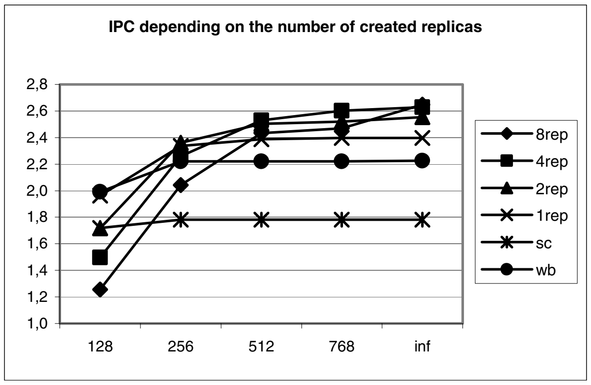

Figure 11: IPC depending on the number of replicas per vectorized instruction and the number of available registers.

Figure 12 shows that increasing the number of replicas from 2 to 4 increases the percentage of committed instructions that benefit from reuse increases from 12,3% to 14%. On the other hand, this extra reuse comes at the expense of a non-negligible increase in number of speculative instructions generated by the control independence scheme. We can also observe in Figure 12 that the amount of speculative activity generated by the control independence scheme is comparable to the activity generated by wrongly speculated instructions due to branch misspredictions.

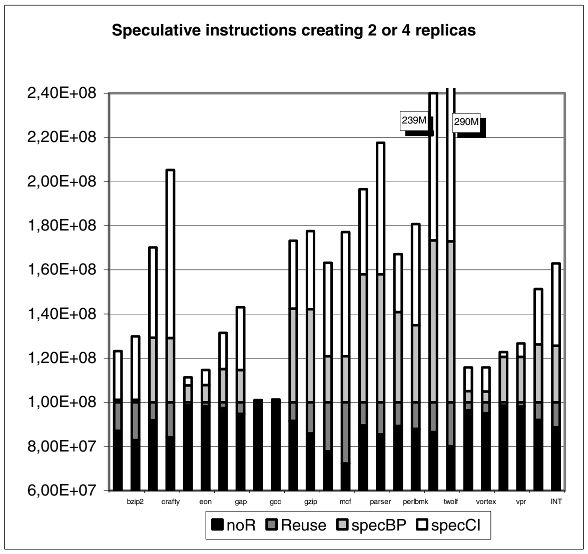

Figure 12: Number of committed instructions that do not reuse (dark), committed instruction that reuse (dark gray), fetched instructions that do not commit due to a branch missprediction (light gray), and instructions generated by the control independence scheme (white) for 2 replicas (left bars) and 4 replicas (right bars) per vectorized instruction.

Although the mechanism does not need a large amount of registers, we have evaluated the benefits of including a memory to hold speculative data created by replicas, as shown in Figure 13, for both alleviating the pressure in the register file and not to increasing the cycle time due to a large register file.

For our simulations we have restricted the number of ports to this memory (2 read ports and 2 write ports) to easily accommodate the access time in 2 cycles. Since this memory is out of the critical path and movements of values from this memory to the register file are not critical, longer latencies are allowed without degrading significantly the performance (a latency of 5 cycles only slowdowns about 3% in configurations with 256 registers in the register file and 768 positions in the proposed small memory). Several configurations of this memory have been simulated as shown in Figure 13.

Figure 13 shows that a register file of 256 registers and a memory holding 768 speculative values has about the same performance as a monolithic, single-latency register file with an unbounded number or registers.

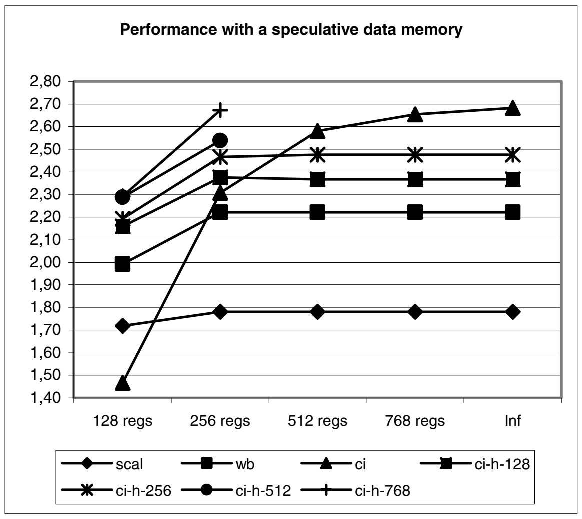

Figure 13: Performance of the control-flow independence mechanism with a memory able to hold 128 (ci-h-128), 256 (ci-h-256), 512 (ci-h-512) and 768 speculative values (ci-h-768). The number of registers in the register file is given by the X-axis. The graph also shows a superscalar processor (scal), a superscalar processor with a wide bus (wb) and a superscalar processor with a wide bus and the control-flow independence mechanism with a monolithic register file (ci), for a varying number of registers.

4. Related Work

Chou *et al.* [5] present a mechanism for exploiting control independence that is based on a structure called DCI that stores copies of decoded instructions. After a recovery from a branch misprediction, new fetched instructions are looked up in the DCI to locate the beginning of a control independent region. Furthermore, an out-of-order fetch mechanism is presented to start fetching from the control independent region after a branch prediction is performed.

Rotenberg *et al.* present a mechanism to exploit control flow independence in superscalar [15] and trace [16][17] processors. Their approach is based on identifying re-convergent points dynamically, and a hardware organization of the instruction window that allows the processor to insert the instructions after a branch misprediction in between instructions previously dispatched, i.e., after the mispredicted branch and before the control independent point.

Sodani *et al.* [19] present a mechanism for global reuse based on keeping history of previous executions of instructions. When an instruction is decoded a structure, called Reuse Buffer, is checked to see if there is a previous execution of this instruction and if the result for that execution can be reused. In that paper, several implementations of the Reuse Buffer that differ in the way the reuse test is performed are proposed and evaluated.

Cher *et al.* present Skipper [4], a mechanism to overlap the latency of hard-to-predict branches with the execution of control-flow independent instructions following the re-convergent point of those branches. To achieve this, when a repeatedly mispredicted branch is detected, the fetch is redirected to the re-convergent point, creating a gap in the reorder buffer (large enough to hold the skipped instructions), and the processor proceeds to execute the instructions after the re-convergent point. When the branch is resolved, the skipped instructions are executed. Dependences among skipped instructions and the instructions after the re-convergent point are checked to ensure the correctness of the execution, performing recovery actions when needed.

Pajuelo *et al.* [12] present a mechanism that dynamically creates speculative vector instructions to exploit SIMD parallelism. Vectorization is triggered by strided loads, which create speculative instances of the load instruction. Dependent instructions are also vectorized. The exploitation of SIMD parallelism, data prefetching and control independence instructions are reported to be the three sources of performance improvement.

The scheme proposed in this paper is a much more cost-effective approach to exploit control independence than the one in [12] because it is tailored to exploit control independence instead of full-blown vectorization of every strided load and all their successors in the dependence graph, including instructions in wrongly predicted paths. Besides, it requires neither vector units nor vector registers since vectorization is performed by replication of scalar instructions.

Figure 14 shows the performance comparison between the control independence scheme proposed in this paper (*cix2p*) and the vectorization approach proposed in [12] (*vect2p*) for 2 wide L1 data cache ports and a varying number of registers (for comparison purposes, a hierarchical register file is not included).

We can conclude from Figure 14 that the scheme proposed in this work has better performance than the dynamic vectorization scheme of [12], except for a huge number of registers. For an unbounded number of registers, the dynamic vectorization scheme outperforms the control independence mechanism by a very small difference (4%). The control independence scheme, in addition to require a simpler hardware (no vector resources are needed, which simplifies the dispatch, issue and execution stages because the processor only deals with scalar instructions), generates much less useless activity: wrongly speculated instructions represent 29,62% of the total executed instructions in the control independence scheme, whereas they represent 48,45% for the dynamic vectorization approach. The control independence generates much less speculative activity and this activity is much more accurate. Overall, both schemes reuse pre-computed values for a similar number of committed instructions (14% the control independence scheme vs. 17% the dynamic vectorization mechanism). Besides, the few additional instructions (3%) reused by the dynamic vectorization scheme are in general less critical (i.e., have less effect on performance) than the control independent ones.

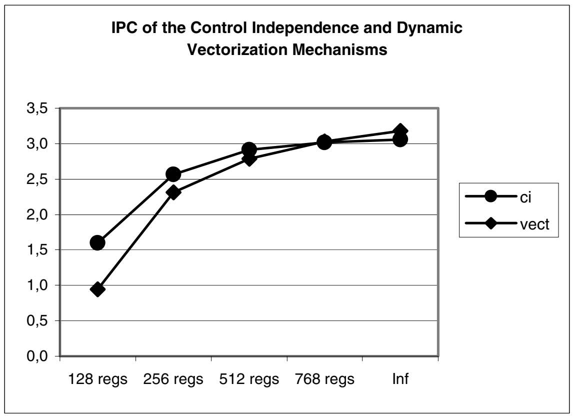

Figure 14: Comparison of the control independence mechanism with the dynamic vectorization scheme of [12].

5. Conclusions

In this paper we have proposed a mechanism to exploit control-flow independence. The mechanism is based on a simple heuristic to detect re-convergent points, a technique to identify the strided loads on which the control independent instructions depend and a scheme to dynamically generate speculative instances of these strided loads and their successors in the data dependence graph. The proposed scheme can pre-execute control independent instructions before a branch is resolved, no matter how far these instructions are from the branch.

The performance evaluation experiments reveal an average speedup of 17,8% over a superscalar processor, about the same performance as a previously proposed full-blown vectorization approach but with a much lower hardware cost, and better performance than the full-blown vectorization when a moderate number of registers is considered (less than 700).

Furthermore, we have studied the benefits of including a small memory to hold the speculative data. We have shown that this memory can be easily implemented in the processor and alleviates the additional register pressure generated by replicas. A configuration with a register file of 256 registers and memory able to hold 768 speculative values has practically the same performance as an unbounded single-level register file.

Acknowledgments

This work has been partially supported by the Ministry of Education and Science under grants TIN2004-07739-C02-01 and TIN2004-03072, the CICYT project TIC2001-0995-C02-01, Feder Funds, Intel Corporation and the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). We would like to thank the anonymous reviewers by their comments.

References

[1] R. Balasubramonian, S. Dwarkadas and D. Albonesi, "Reducing the Complexity of the Register File in Dynamic Superscalar Processors", *in Proceedings of the International Conference on Microarchitecture*, Dec. 2001.

[2] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0", *Technical Report No. CS-TR-97-1342,* University of Wisconsin-Madison, June 1997.

[3] P. Chang, E. Hao and Y. Patt, "Target Prediction for Indirect Jumps", 24th *International Symposium on Computer Architecture*, June 1997.

[4] C. Cher and T. N. Vijaykumar, "Skipper: A Microarchitecture For Exploiting Control-flow Independence", *in Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[5] Y. Chou, J. Fung and J. P. Shen, "Reducing Branch Misprediction Penalties Via Dynamic Control Independence Detection", *in Proceedings of the 13th International Conference on Supercomputing*, June 1999.

[6] J. L. Cruz, A. González, M. Valero and N. Topham, "Multiple-Banked Register File Architectures", *in Proceedings of 27th International Symposium on Computer Architecure*, June 2000.

[7] J. González and A. González, "Memory Address Prediction for Data Speculation", *in Proceedings of Europar 97*, August 1997.

[8] D. Grunwald, A. Klauser, S. Manne and A. Pleszkun, "Confidence Estimation for Speculation Control", *in Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[9] E. Jacobsen, E. Rotenberg and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions", *in Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[10] M. Lam and R. Wilson, "Limits of Control-Flow on Parallelism", *in Proceedings of 19th International Symposium on Computer Architecture*, June 1992.

[11] D. López, J. Llosa, M.Valero and E. Ayguadé, "Widening Resources: A Cost-Effective Technique for Aggressive ILP Architectures", *in Proceedings of the 31st International Symposium on Microarchitecture*, Dec. 1998.

[12] A. Pajuelo, A. Gonzalez and M. Valero, "Speculative Dynamic Vectorization", *in Proceedings of the 29th International Symposium on Computer Architecture*, May, 2002.

[13] S. Palacharla, N. P. Jouppi and J. Smith, "Complexity-Effective Superscalar Processors", *in Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[14] J.A. Rivers, G. S. Tyson, E. S. Davidson and T. M. Austin, "On High-Bandwidth Data Cache Design for Multi-Issue Processors", *in Proceedings of the 30th Symposium on Microarchitecture*, 1997.

[15] E. Rotenberg, Q. Jacobson and J. Smith, "A Study of Control Independence in Superscalar Processors", *5th International Symposium on High Performance Computer Architecture*, Jan 1999.

[16] E. Rotenberg, Q. Jacobson, Y. Sazeides and J. Smith, "Trace Processors", *in 30th International Symposium on Microarchitecture*, Dec 1997.

[17] E. Rotenberg and J. Smith, "Control Independence in Trace Processors", *32nd International Symposium on Microarchitecture*, Jan 1999.

[18] T. Sherwood and B. Calder, "Loop Termination Prediction", *3rd International Symposium on High Performance Computing*, October 2000.

[19] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse", *24th International Symposium on Computer Architecture*, May 1997.

[20] SPEC 2000. http://www.specbench.org/osg/cpu2000/

[21] J. Tubella and A. González, "Control Speculation in Multithread Processors through Dynamic Loop Detection", *in Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[22] S. Vajapeyam, J. P. Joseph and T. Mitra, "Dynamic Vectorization: A Mechanism for Exploiting Far-Flung ILP in Ordinary Programs", *in Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[23] K. M. Wilson and K. Olukotun, "High Bandwidth On-Chip Cache Design", *IEEE Transactions on Computers, vol. 50, No. 4*, April 2001.