The Multi-State Processor: a ROB-free architecture with precise recovery

Isidro González
DAC - UPC
iglez@ac.upc.edu

Marco Galluzzi
DAC - UPC
galluzzi@ac.upc.edu

Adrián Cristal
BSC - CNS
adrian.cristal@bsc.es

Alex Pajuelo
DAC - UPC
mpajuelo@ac.upc.edu

Oliverio J. Santana
DIS - ULPGC
ojsantana@dis.ulpgc.es

Mateo Valero
BSC - CNS
mateo@ac.upc.edu

< This Technical Report was sent to Advisory Committee of MICRO-40 (June 8th, 2007) for review >

Abstract

Modern processors improve performance by taking advantage of the instruction level parallelism (ILP) by means of allowing hundreds of instructions in flight. However, they still have to face an important source of degradation coming from the increasing difference between the processor and the main memory speeds (memory wall). In order to overcome this problem, recent proposals allow even more instructions in flight by replacing a re-order buffer (ROB) with a *checkpointing* mechanism and an out-of-order retirement of the processor's resources, relaxing other desirable features like the *precise recovery*[*] of the state on mispredicted branches or exceptions, possibly re-executing correct-path instructions on a recovery.

In this paper, we propose a new processor model, called *Multi-State Processor*, that does not use a ROB either, and also replaces the checkpointing with an original mechanism to store the processor's states generated by all instructions in flight, allowing precise recovery. Furthermore, we describe the implementation model showing that is simple, modular and scalable, and it minimizes the required hardware and the access time to the involved structures. Results show that we achieve better performance and we reduce power consumption compared to the well-know checkpointing-based mechanism called CPR. Specifically, we achieve, in average, an IPC speed-up of 7.8% and a reduction in the number of executed instructions of 16.5%, running the integer benchmarks of the SPEC CPU2000 suite.

1. Introduction

Modern processors are based on mechanisms that allow having hundreds of instructions in flight, which only update the processor state when an instruction is executed and no previous instruction can invalidate its results. This way, the processor commits the state in the program order, exploiting, to a certain extent,

Tabla 1.

[*] In our context, the *precise recovery* term means a recovery restoring the processor's state at the exact point where the instruction produces an exception or branch misprediction. Checkpointing based proposals, even supporting *precise interrupts*, do not always recover the state at the exact point of the exception or branch misprediction but at the point where the previous checkpoint has been set.

the instruction level parallelism. In order to better leverage this parallelism and totally hide the latency of memory accesses, a higher number of instructions in flight would be required. However, these ROB-based mechanisms are not scalable, limiting the processor's performance [10].

Recent works propose processors with an out-of-order retirement and a checkpointing mechanism, such as *Checkpointing Processing and Recovery* or *CPR* [1] and the *Kilo-instruction Processors* [2], which allow thousands of instructions to be in flight thanks to the release of resources associated with each instruction as soon as the instruction has been successfully executed. This way, memory accesses are overlapped with the execution of a bigger amount of useful work, which may be speculative or not, better leveraging the ILP.

Checkpointing based proposals define a checkpoint as a hardware structure containing all the required information necessary to recover the processor's state, that is basically made up of the mapping of logical to physical registers. The performance of this kind of processors depends, on one hand, on the number of available checkpoints, and on the other hand, on how the control unit decides where to set them along the program execution flow. This control unit uses approximate criteria to decide where to set a new checkpoint. It normally sets a new checkpoint if the current instruction shows a high probability to interrupt, as a branch that can be mispredicted frequently or another instruction liable to produce an exception. For example, the CPR model [1] includes in this unit an estimator that computes the confidence for every branch prediction done, setting a new checkpoint if there is one available and the estimator gives a low confidence for the prediction of the current branch [6].

Each time the control unit decides to take a new checkpoint, the information of the state of the processor is stored in one free checkpoint, which is used in case of a recovery. When the instructions between the oldest checkpoint and the following one have been successfully executed, and we can guarantee that they will not be discarded due to a recovery, the oldest checkpoint can be released, allowing a large amount of instructions to be committed simultaneously.

When an exception or branch misprediction occurs, the processor sets back the state using the first checkpoint preceding the causing instruction. Since the number of checkpoints is lower than the number of instructions in flight –these models normally support a small number of checkpoints–, and the control unit that decides where to take the checkpoint is based on approximate criterions, it will occur, more or less frequently, that the state can not be recovered at the exact point of the causing instruction. This means that, after recovering the processor's state, there will be a re-execution of correct-path instructions.

This lack of precision in the recovery is an undesirable issue, since the larger is the distance between the checkpoint where the state to be recovered is stored and the instruction that produces the exception or mispredicted branch, the more useful instructions can be re-executed unnecessarily, degrading the processor's performance and increasing the power consumption.

This performance degradation could be minimized, to a certain extent, by having a high number of architectural checkpoints, but unfortunately, this solution is not feasible. The information of the states is stored into the checkpoints without taking into account any correlation between states, taking the decision to set them independently. Therefore, incrementing the number of checkpoints would increase the area, power consumption and also the cycle time since the access to these structures is made in the critical path of the processor.

Precise recoveries are only possible if we have a perfect estimator in the control unit for checkpoint setting or, in the case we are interested in, *if all states generated by instructions in flight are stored.* We know that the lower is the distance between instructions generating two different states, the higher is the correlation between the information of these states, i. e. they have more information in common. Therefore, in the extreme case, where any instruction can cause a recovery, we would need to store as much states as there are instructions in flight. If the processor stores one state per instruction, the variation of information can be, at most, equal to the number of destination registers of the instruction, which is normally *one* and in some cases *zero,* in instructions like the branches.

In this paper we propose a new processor model based on the deduction above explained, called the *Multi-State Processor* or *MSP,* which does not need a ROB and stores all the processor's states generated by the instructions in flight, without substantially incrementing the required hardware or the access time to the hardware structures, allowing:

- A substantial number of instructions in flight, still supporting precise recoveries, thus incrementing the performance and saving the power since there is no re-execution of correct-path instructions.
- New hardware mechanisms to thoroughly control the states in flight and the renaming stage, not requiring any list of free physical registers or mapping tables as those utilized in current processors [10].
- Hardware structures minimizing the control information for the states generated by the instructions in flight, storing it in a differential way using a simple, modular and scalable design.

In the rest of the paper we first introduce the operation of our proposal. We then explain what structures take part in the renaming stage, queue stage, and recovery, and how they work, following with the explanation of a hardware consideration and the introduction of our simulation environment and the results obtained. We conclude by exposing the related work and the final remarks.

2. Operation

In this section we explain how our proposal works using an example and comparing it with the operation of a checkpointing based processor.

2.1. An Example with a Checkpointing Based Processor

When an exception or a mispredicted branch occurs, a checkpointing based processor recovers the state stored in the closest checkpoint to the causing instruction –this state becoming the current state of the processor– and flushes from the pipeline all the instructions younger than this checkpoint. Recovering the state implies setting back the mappings of logical to physical registers, adding some physical registers to the list of free physical registers and releasing possible younger checkpoints. After the recovery, the instructions fetched use the information of the recovered state, continuing the execution as usual.

No.	PC	Instruction	StateId
1	@+00	store r2 → @	0
2	@+04	add r1,r2 → r2	1
3	@+08	bne r2, @+2c	1
4	@+0c	sub r2,#1 → r2	2
5	@+10	mov r2 → r1	3
6	@+14	add r1,r2 → r2	4
7	@+18	bne r3, @+3e	4
8	@+1c	add r1,r2 → r1	5

Table 1. Example of dynamic instruction flow

Table 1 shows an example of a dynamic instruction flow where a checkpoint is set at instruction 3 because a low-confident branch was estimated, although later it will result as predicted correctly. However, a branch misprediction occurs at instruction 7, where a checkpoint has not been set. The processor's state is recovered back to the state stored in the previous closest checkpoint –the one set at instruction 3– with the processor having to re-execute the correct-path instructions from 3 to 7, surrounded in Table 1 by a dashed line.

2.2. Principles of the MSP Processor

The model we propose can keep as many states as there are instructions in flight, where each state differs from the previous one, normally, in the variation of only one logical to physical register mapping –each instruction does not have to produce a new state–. The control of allocation, recovery and release of the states must be handled explicitly, so the following must be fulfilled:

1. *A new state is stored based on the information of previous states together with a variation of at least one logical register.* A new state is created when a write to a logical register occurs, which implies the allocation of a new physical register.
2. *A sequential counter tag, called StateId, is associated with every instruction entering the instruction queue.* The StateId identifies the current state and it is incremented only when a new state is generated. On a recovery, instructions with a StateId greater than the StateId of the causing instruction are discarded, while the others are kept in flight.
3. *A list of states at which each physical register belongs to is maintained, in order to avoid duplicating common information between states.* A range, called StateId Range, is associated with each physical register where the lower and upper values are defined as follows: the Lower StateId is the StateId of the instruction generating the allocation of the corresponding physical register, and the Upper StateId is the StateId of the instruction preceding the instruction that rename the corresponding logical register.

2.3. An Example with the MSP Processor

Observing again the example of Table 1; if we start from a state initialized with value 0, the different StateIds associated with each instruction are those showed in the column labelled "StateId". The table shows that only the instructions writing to a register increment the StateId. Therefore, neither stores nor branches increment it. Stores change the memory state and are issued when their states are committed, and branches only imply a change in the execution flow.

The range of states associated with each physical register is showed in Table 2. We use the notation R*x*.*y* to identify the physical register *y* associated with the logical register *x*. We can observe how on each renaming on the same logical register a new physical register is allocated, removing any false dependency. Values R1.0 and R2.0 are the values associated with logical registers R1 and R2 respectively, before the execution of the example of Table 1.

StateId Range		Associated registers	
Lower	Upper	Logical	Physical
0	0	R2	R2.0
1	1		R2.1
2	3		R2.2
4	5		R2.3 *
0	2	R1	R1.0
3	4		R1.1 *
5	5		R1.2

Table 2. StateId Range for instructions in Table 1

In the case of the MSP, the mispredicted branch at instruction 7 sets the *Recovery StateId* (see section 5) to the StateId associated with this instruction, i.e. state number 4. Therefore, instructions with StateId greater than 4 are flushed –not correct-path instructions–, and the mappings from logical to physical recovered are those associated with the range containing StateId 4, marked with the * symbol in Table 2. All physical registers whose Lower StateId is greater than 4 can be released –only the register R1.2 in our example– becoming available for the renaming of future instructions. In our case, then, we do not have to re-execute instructions from the correct path.

3. Renaming Stage

We have designed the renaming stage in order to minimize the cycle time required, because the critical control of our model is located in this stage. In order to keep our hardware implementation simple, modular and scalable, and, at the same time, be able to store the states generated by all instructions in flight, we have the following restrictions:

1. Each logical register is renamed to a fixed subset of physical registers

2. Physical registers are released in order

These two restrictions make our model need more physical registers than other proposals, such as the *CPR*, to achieve a similar or better performance. Furthermore, the maximum number of states in flight matches the number of physical registers, which limits the number of instructions in flight. Assuming that 30% of the instructions do not generate a new state, we can have at most 1.3 x *M* instructions in flight, where *M* is the total number of physical registers. However, the renaming control of such a greater amount of physical registers is simpler since:

1. The allocation of new physical registers is done using the FIFO policy

2. Neither a list of free physical registers nor mapping tables are needed

3. The register file is arranged in separate banks, each one associated with a logical register

Below, we explain the structures involved in the renaming stage.

3.1. Hardware Structures

We have divided this section according to the scope of control of the hardware structures: *local scope,* control for each logical register, and *global scope*, control interacting with the main control of the processor.

1) Local Scope Control

In this scope we have the *State Control Table* or *SCT*, showed in Figure 1. Each entry in the table is associated with a physical register and contains the following fields:

- *StateId:* stores the value of the *Lower StateId,* initialized with the StateId of the instruction having a destination operand and associated with the corresponding physical register. Since the StateId contains values incremented in program order, the *Upper StateId* is set implicitly, matching the value of the next entry minus one or, in the case of the most recent entry, having a null value.
- *Validation Bit* or *Vb:* specifies whether the entry is active or not. There is always at least one active entry, which is the last renaming of the logical register. This bit can be used to enable or disable each entry of the table via hardware, reducing the power consumption [4].

Each entry, in addition, is associated with the following control hardware:

- *Range StateId Comparator:* compares the StateId ranges of the processor, and it is used in:
  - The release of physical registers, identifying those registers that can be potentially released, because their state has been committed and they fulfill *StateId[i]* < *LCS,* where *LCS* is the *Last Committed StateId* (see next subsection).
  - The recovery process, identifying the active mapping for the recovering StateId, stored in the *Recovery StateId* register (see section 5).

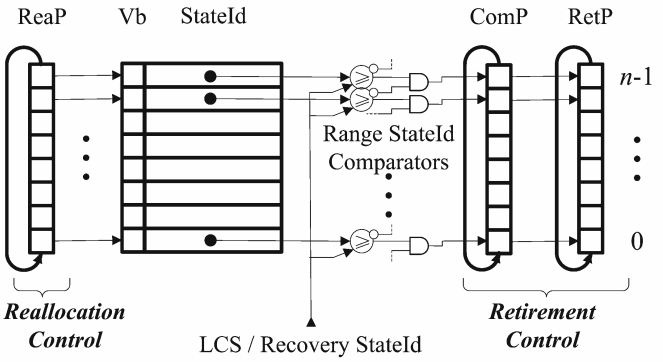

Figure 1. The State Control Table or SCT

We have three hardware structures of the type *one-hot rotating pointer* – one bit is 1 and the rest are 0s – to identify three different entries, or StateId, of the SCT:

- *Reallocation Pointer* or *ReaP:* points at the last StateId mapped to the logical register in the renaming. On each new renaming, the pointer points at the next spatially adjacent –in a cyclic way– entry. This way, the physical register allocation policy is defined as a FIFO policy. Moreover, the mapping from logical to physical register of every operand of each renamed instruction is identified by the pair logical register identifier (*LogRegId,* as defined in section 3.2) and ReaP index.
- *Commit Pointer* or *ComP:* points at the potentially committable StateId whose instructions associated with older StateIds have been correctly executed and take part in the computation of the *Last Committed State* or *LCS* (see next subsection).
- *Retirement Pointer* or *RetP:* points at the oldest StateId whose associated physical register value has not been consumed. Once the value is consumed, the pointer points at the next entry, defining the physical register release policy also as FIFO. In order to avoid front-end stalls of the processor during the renaming due to this pointer, RetP must advance at least the number of instructions renamed on every cycle, which can not be always possible.

From these definitions we can deduce that:

- The range of states defined from RetP to ReaP is the set of active states for each logical register, where RetP points at the oldest StateId and ReaP at the most recent one.
- The range defined from ComP to ReaP is a subset of the previous one, that is, RetP never goes ahead of ComP and ComP never goes ahead of ReaP.
- All the three pointers point at the same entry when reallocations do not occur during a considerable number of cycles.
- On a new renaming, if ReaP points at the previous entry to this pointed by RetP, a front-end stall occurs because there are no more available physical registers for the current logical one.

2) Global Scope Control

The control hardware structures on this scope are:

- *StateId Counter* or *SC:* it is the current processor's StateId. It will be incremented on each renaming generating a new state.
- *Last Committed StateId* or *LCS:* it stores the StateId from the last committed state. This last committing occurs when all the instructions associated with all previous states have been successfully executed. The LCS is involved in the release of the processor's resources:
  - In the local control of each logical register, allowing RetP and ComP to advance, and hence, releasing the associated physical registers.
  - In the store queue, allowing stores with a StateId smaller than the LCS to be sent to memory.

The LCS is computed as the minimum of the states pointed by the ComPs of all SCTs. The hardware needed for this computation is a binary, ternary or quaternary tree, having a complexity in area and response time of logarithmic order that only depends on the number of logical registers –not on the number of physical registers–, both for integer and floating point values. This tree is composed of units that compute the local minimum of the involved StateIds, propagating the smallest value –the desired LCS– to the root. Despite the fact that propagation time is of logarithmic order, it could take more than one cycle and, in this case, the computation would be split between stages in intermediate points of the tree *(pipelined execution)*. However, delaying the computation of the LCS is not a critical issue, since our studies give us less than 1% of decreasing in performance even for a propagation delay of 4 cycles.

In order to control a possible case of deadlock, the StateId of the SCT whose ComP reaches the ReaP is not used in the computation of the LCS.

3.2. Renaming of Multiple Instructions

The renaming stage can be a critical stage if we consider renaming multiple instructions per cycle. However, the hardware structures proposed in this paper have been focused to allow the scalability of this stage.

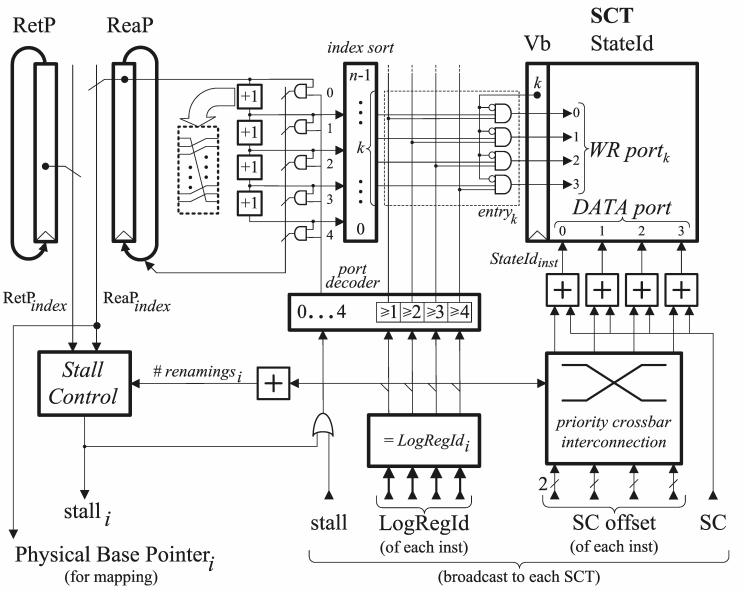

Figure 2. Hardware structures involved in a renaming width of 4

In Figure 2, we show a broad outline of the hardware control structures involved in the renaming of four instructions, per cycle and per SCT, that could have a destination register. First, the SCTs to update are activated using the logical register identifiers, *LogRegId.* We can have at most four active SCTs, the other ones being disabled to help saving power. Second, the activated SCTs entries to write will be pointed by the ReaP. Third, the port decoder identifies the write ports to use. Finally, the data to write to the selected entries are the StateIds of the new instructions, which are computed by adding the current SC to the *SC offset* of the instructions –depending whether they generate a new state or not–. The *priority crossbar interconnection* is responsible for carrying the SC offsets to the corresponding consecutive positions.

In order to compute the physical address of the source and destination operands of the instructions being renamed, the bank is specified using the LogRegId and the position inside the bank is specified using two values: the *Physical Base Pointer,* given by the *ReaP_index,* and an *offset* given by the number of renamings done in the cycle on the specific logical register.

A stall of the front-end, due to the renaming, occurs when the number of the renamings is greater than the available physical registers associated with the specific logical register. When the stall is detected on a SCT, it will be broadcast to the rest of stall detection units and to the global control unit of the renaming, avoiding the possible advancing of ReaP in other SCTs. Such a stall control could work more precisely by allowing a partial stall, that is, controlling which instructions can be renamed in the current cycle.

4. Queue Stage

The dependency matrix [3], located in the Queue stage, allows to release the physical registers speeding up the advance of pointers RetP and ComP on each cycle. In Figure 3, we show how this matrix interacts in parallel with the instruction queue and the SCTs, associating each column with an instruction and each row with a physical register or state. Each element of the matrix is made of two bits: *b1* specifies if the physical register is an operand of the instruction, and *b2* specifies the state to which the instruction belongs. Furthermore, each row is associated with a *ready bit* or *Rb* that specifies if the value of the associated physical register is available. The control of the matrix is spatially divided into:

Control by column, the control of the wake-up of instructions with a wired-OR, on each column, of the ANDs of bits b1 and Rb.

Control by row, there are two kinds of control interacting with the SCT, each one with a wired-OR:

- Row of b1 and Rb bits, specifying if the value of the associated physical register has been produced and consumed. The result of the wired-OR associated with an SCT is the *Retirement Vector* or *RetV,* which controls the advancing of pointer RetP. On a recovery, we just disable the bits of the column when an instruction is discarded. This way, we do not need other aggressive techniques based on counters [9] in order to know if the physical register has been consumed, also avoiding updating counters on a recovery.
- Row of b2 bits, specifying if there is at least one instruction, associated with this state, that is still in the instruction queue. The result of the wired-OR associated with an SCT is the *Commit Vector* or *ComV,* which controls the advancing of pointer ComP together with the LCS.

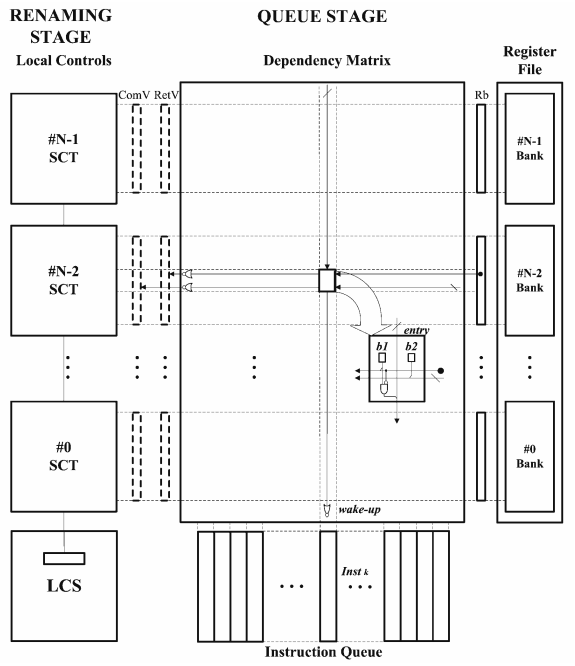

Figure 3. Dependency Matrix

5. Recovery Mechanism

The potential state to which the processor can recover is stored in the *Recovery StateId* register. On a recovery of the state, the mappings of logical to physical register whose StateId Range contains the Recovery StateId value will be restored.

On the detection of a mispredicted branch, a recovery to the StateId associated with the branch instruction is done.

On the detection of a masked exception there is a front-end stall and only instructions with a StateId smaller than the StateId of the excepting instruction are issued for execution, reducing the number of total executed instructions. Once unmasked, the Recovery StateId is the StateId associated with the excepting instruction or the previous one if this instruction produced a new state.

After a recovery, the *Recovery StateId* is disabled and the SC is set to the last valid state, proceeding further with the normal ordering.

6. A Hardware Consideration

The StateId is a value used in almost all control structures, and it is generated by the SC. Ideally, the StateId is an unlimited integer value which is always incremented, in order to control the chronological

order of states during the execution. However, on an actual implementation, this value must have a finite number of bits and the case of overflow must be managed.

The number of physical registers is equivalent to the maximum number of states in flight. In the case of having *M* physical registers, i.e. a maximum of *M* states, the StateId is coded in *log2(M) = m* bits.

When the SC reaches the maximum value, a trivial solution would be stalling the front-end until all instructions already in the pipeline have been executed, and then resetting the SC and restarting the execution. Nevertheless, this solution would reduce the processor's performance significantly. Therefore, we use a different solution, that includes adding one more bit, the *saturation bit* or *Sb,* as the most significant bit of the StateId, and controlling the case of overflow as follows.

The SC, which now has *m+1* bits and can code up to 2x*M* states, is initialized with zeros and incremented on each state generated by the instructions being renamed, until reaching the maximum value, where all bits are 1. Since we still have a maximum of *M* states in flight, all current states must now have the Sb set to 1. In order to go on with the sequencing of states, all Sb bits of the stored StateIds are reset to 0 and the SC is set to value *M+1,* that is, the Sb to 1 and the rest of the bits to 0. The process is repeated on every saturation of the SC.

7. Simulation Environment

The results showed in this paper have been obtained by modifying the execution-driven version of the SMTsim simulator [17]. We have modelled, and later compared in the evaluation section, the following architectures:

- *Baseline*. Aggressive superscalar single-thread processor.
- *CPR*. Architecture without ROB and including mechanisms for selective checkpoints, hierarchical store queue and aggressive release of physical registers.
- *n-SP*. The Multi-State Processor architecture we are proposing, where the parameter *n* refers to the number of physical registers associated with each logical register. It has the same hierarchical store queue as the CPR model.
- *MSP ideal*. Idealized architecture of a Multi-State Processor having an infinite hierarchical store queue and an ideal register file. This model shows the upper bound for the performance.

The configuration of the significant hardware parameters of the four models are showed in Table 3, which is based on that used in the paper explaining the CPR model [1]. A notable difference with the

configuration used in that paper is that we use a better branch predictor, the *perceptron,* and therefore we loose some benefits in the performance results when comparing with the CPR model, since the higher is the branch hit rate the better performance obtains the CPR model.

In the experiments we use the integer benchmarks of the SPEC CPU2000 suite (*spec2k-INT*) [14]. The SMTsim emulate standard binaries of the Alpha architecture, therefore the benchmarks have been compiled with the Compaq C V5.8-015 compiler running on a Compaq UNIX V4.0 with the optimization option –O3. In order to reduce the simulation time, we have simulated 300 millions of representative instructions of each benchmark, using the input reference set. Representative segment of instructions have been identified analyzing the distribution of basic blocks as described in [12].

Processor core	**Baseline**	**CPR**	***n*-SP**	**MSP ideal**
Reorder buffer size	128	-	-	-
Instruction queue size	48	128	128	128
Rename | Issue | Retire width	3 | 5 | 3	3 | 5 | -	3 | 5 | -	3 | 5 | -
Int | Fp register file size	96 | 96	192 | 192	*n* | *n* (each LogReg)	∞ | ∞ (each LogReg)
Ld | L1St | L2St buffer size	48 | 24 | -	48 | 48 | 256	48 | 48 | 256	48 | ∞ | ∞
Confidence branch estimator	-	64 KB | 4 bits	-	-
LCS propagation delay	-	-	1 cycle	0 cycle
Int | Fp | LdSt units	4 | 4 | 2			
Branch predictor	Perceptron			
Memory Subsystem				
I-cache size	64 KB, 4-way, 1 cycle hit			
D-cache size	64 KB, 4-way, 4 cycle hit			
L2-cache size	1 MB, 8-way, 16 cycle hit			
Caches line size	64 bytes			
Main memory latency	380 cycles			

Table 3. Configuration for the hardware parameters of each processor model

8. Results

Next, we show the evaluation results for performance and power consumption of the described models.

8.1. Evaluation of performance

In Figure 4, we show the IPC results achieved for the described models. The n-SP model has been evaluated using different configurations, where *n* is equal to 8, 16 and 32, in order to see the number of physical registers required to obtain a significant performance.

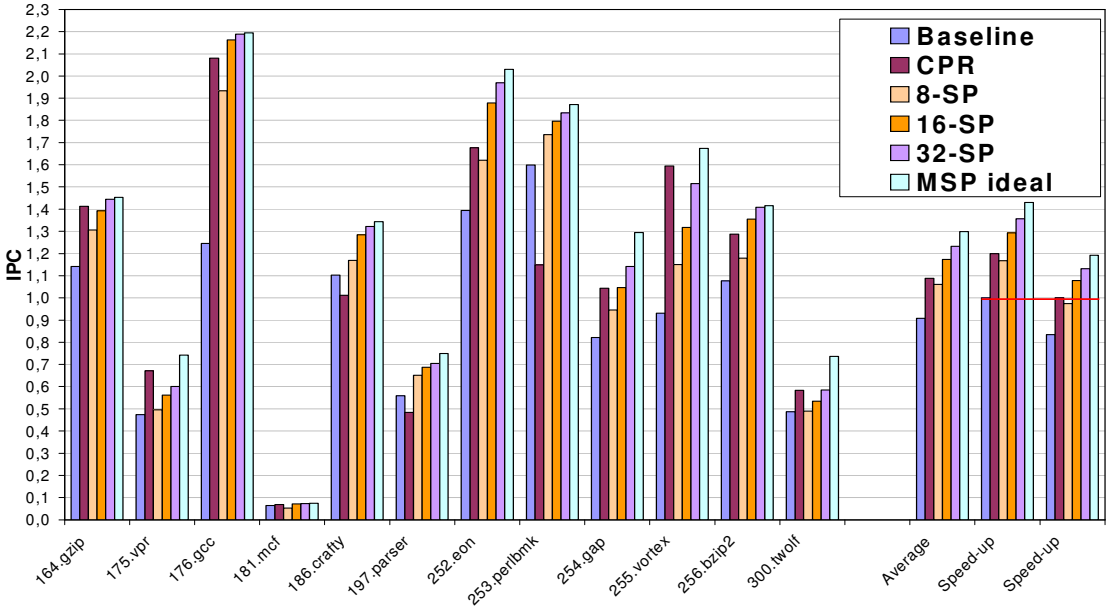

Figure 4. IPC obtained for the n-SP and the other reference models running the spec2k-INT

We can observe that the performance achieved by the 8-SP is lower than that achieved by the CPR, because it suffers from constant stalls of the front-end due to the insufficient number of physical registers for renaming. We can see that the 16-SP already achieves an important performance improvement compared to the CPR. We also see that the 32-SP does not achieve a significant performance improvement considering the huge amount of physical registers it would require. Furthermore, we have verified that going beyond the 32-SP practically does not improve performance since simulations with a 64-SP give us a speed-up of only 1.8% against the 32-SP.

Now, comparing the 16-SP with the other models, we obtain a speed-up of 29.3% against the Baseline, 7.8% against the CPR and a degradation of 10% against the MSP ideal, showing us that we are not far from an optimum performance.

The *perlbmk* benchmark shows a high degradation using the CPR due to the great amount of re-executed correct-path instructions as observed later in Table 4.

It must be noticed that the results obtained with the 16-SP can be improved with compilation techniques taking into account the renaming restrictions. These techniques would require obtaining a binary code where the use of logical registers must be as uniform as possible, because the extensive use of a small number of registers would produce a greater amount of stalls due to the lack of available physical registers for renaming these logical registers.

8.2. Evaluation of Power Consumption

One of the advantages of the MSP model we propose is the precise recovery, avoiding the re-execution of any correct-path instruction. We have indirectly evaluated the reduction of power consumption due to not re-executing correct-path instructions by showing in Table 4 the percentage of instructions re-executed in the CPR model. It shows a significant average of 9.5%, having, in some benchmarks, more than 25%.

We also give the number of total executed instructions by both models in Figure 5. The 16-SP achieves a reduction of 16.5% compared with the CPR model.

spec2k-INT	% Re-executed correct-path instructions **CPR**
164.gzip	1.58
175.vpr	7.01
176.gcc	1.55
181.mcf	9.17
186.crafty	14.00
197.parser	0.67
252.eon	10.72
253.perlbmk	27.02
254.gap	26.84
255.vortex	1.03
256.bzip2	9.89
300.twolf	4.41
Average	**9.49**

Table 4. Re-execution of correct-path instructions by the CPR model running the *spec2k-INT*

A criticism that can be done to the MSP model, in relation to the power consumption, is the big quantity of physical registers required. However, recent studies about multi-banked register files [11, 7] show that, for a large number of banks –e.g. 32 banks–, the number of ports per bank can be reduced obtaining a very low degradation due to access conflicts. With such a reduction in the number of ports per bank, quite an important reduction of power consumption can be achieved, and also a reduction of area and access time to the register file.

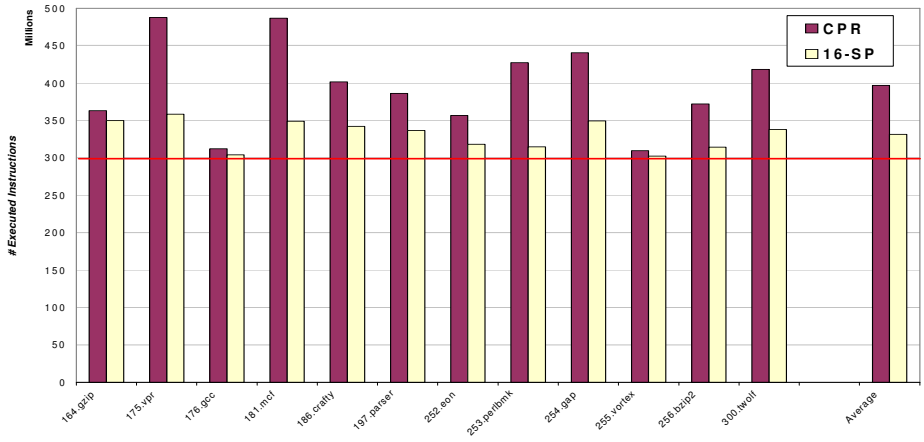

Figure 5. Total number of executed instructions running the *spec2k-INT*

As showed in Figure 6, we have verified with our model that, using only one read port and one write port –labeled "16-SP (1R/1W)"–, we obtain a degradation of only 0.36% against an a register file with all the required ports –labeled "16-SP"–. In order to achieve these results we have applied two simple mechanisms to reduce the number of conflicts per bank: *conservative bypass-skip* and *read sharing* [16]. However, the conflict detection control added to the wakeup-select logic can increase the cycle time and the hardware complexity. Therefore, we have moved the conflict detection control to a new stage located after the issue stage, called *arbitrate.* In Figure 6, we see the results adding the new stage –labeled "16-SP (1R/1W) + Arbitrate"– where the degradation is still low, only 1.66%. The obtained results, then, show not only the significant savings in power, area and access time to the register file that we can achieve, but also show that our renaming policy satisfy the same expectations as those policies used in [16].

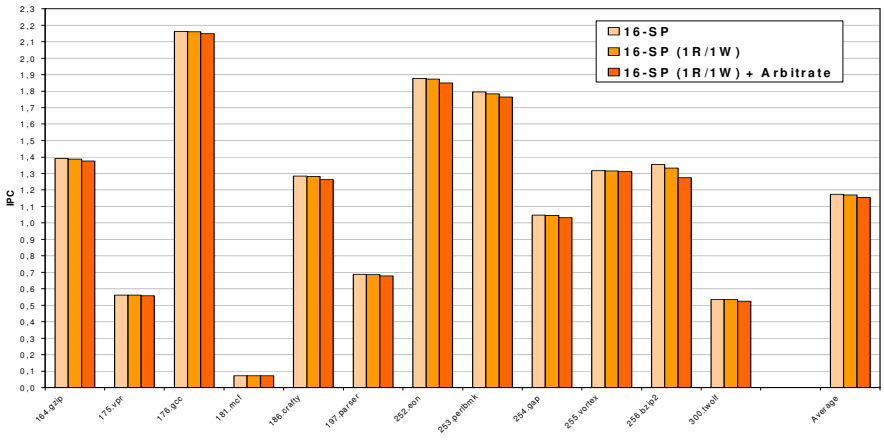

Figure 6. IPC degradation for the 16-SP model reducing the register file ports and running the *spec2k-INT*

Next, we detail other considerations we can take into account, which contribute to a reduction of the power consumption of the proposed design:

- The free entries of the instruction queue directly disable their corresponding column in the dependency matrix. In the case of busy entries, we can leverage the fact that the renaming of the instruction defines the active b1 bits in the column according to the source physical registers, being able to disable most of the column and keeping active only those groups associated with each SCT – one for each source operand–. We can also leverage the fact that only one b2 bit is active, corresponding to the StateId of the instruction, being able to let only the group associated with its SCT to be active, disabling most of the column also for these bits.
- We can use a simple mechanism to disable the physical registers, as described by the proposal Dead Register Desactivation Control [4], together with the entries of the SCT, using the Vb bit. In order to enable them again, we use the write signal for the physical registers and again the Vb bit for the SCT entries.

9. Related Work

Smith and Pleszkun [13] studied different alternatives to support precise interrupts, such as the *history buffers*, organized in a manner very similar to ROB, and the *future file* that works together with a ROB to modestly improve scalability. However, none of these approaches can support a large amount of instructions in flight, since they use monolithic structures and our proposal make a distributed control.

Hwu and Patt [5] proposed the use of checkpoints to implement precise interrupts but discarding useful work on recoveries, thus not allowing what we have defined as precise recovery. *Cherry* [8] allows more instructions in flight by still using a ROB in combination with one checkpoint to release resources earlier when it can be guaranteed that all branches have been completed and all memory instructions have been issued.

The *Kilo-instruction Processor* [2] is a checkpointing based architecture, allowing even more instructions in flight, and uses a pseudo-ROB for younger instructions to minimize the amount of correct-path instructions re-executed. Another similar proposal is the *CPR* [1] which uses checkpointing and does not use ROB, having also re-execution of useful instructions, and includes other mechanisms like the hierarchical store queue and an aggressive release of physical registers. The *Continual Flow Pipeline* architecture or *CFP* [15] improves the CPR model, by incorporating a bi-level instruction queue, adding

the Slice Data Buffer where the instructions depending on a L2 cache miss are stored. This proposal does not improve performance significantly compared with the CPR proposal.

10. Concluding Remarks

Our original proposal is part of the set of ROB-free architectures but, in addition, it allows precise recovery on mispredicted branches and exceptions, showing satisfactory performance results since we obtain a speed-up of 7.8% with a 16-SP configuration against the reference model CPR.

Other implicit characteristics of our design, such as simplicity and modularity, allow reducing the power consumption and not increasing the cycle time of the processor, without degrading the resulting IPC.

In future works we want to include the evaluation of other benchmarks, as the floating-point ones of the SPEC CPU2000 suite, and employ suitable compiler optimization techniques for our architecture. It would be also worth carrying out a study of power consumption to compare a monolithic register file versus the multi-banked register file used in our proposal.

References

[1] Akkary, H., Rajwar, H.R. and Srinivasan, S.T. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. MICRO-36, 2003.

[2] Cristal, A. et al. Out-of-Order Commit Processors. HPCA-10, 2004.

[3] Goshima, M. et al. A high-speed dynamic instruction scheduling scheme for superscalar processors. MICRO-34, 2001.

[4] Heo, S. et al. Dynamic Fine-Grain Leakage Reduction Using Leakage-Biased Bitlines. ISCA-29, 2002.

[5] Hwu, W. and Patt, Y. Checkpoint repair for out-of-order execution machines. ISCA-14, 1987.

[6] Jacobsen, E., Rotenberg, E. and Smith, J.E. Assigning confidence to conditional branch predictions. MICRO-29, 1996.

[7] Kim, N.S. and Mudge, T. Reducing Register File Ports Using Delayed Write-Back Queues and Operand Pre-Fetch. Proc. ICS-17, 2003.

[8] Martínez, J.F. et al. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. MICRO-35, 2002.

[9] Moudgill, M., Pingali, K. and Vassiliadis, S. Register renaming and dynamic speculation: an alternative approach. MICRO-26, 1993.

[10] Palacharla, S. Complexity-Effective Superscalar Processors. Ph.D Thesis, 1998

[11] Park, I., Powell, M.D. and Vijaykumar, T.N. Reducing Register Ports for Higher Speed and Lower Energy. MICRO-35, 2002.

[12] Sherwood, T., Perelman, E. and Calder, B. Basic block distribution analysis to find periodic behaviour and simulation points in applications. PACT-10, 2001.

[13] Smith, J.E. and Pleszkun, A.R. Implementation of precise interrupts in pipelined processors. ISCA-12, 1985.

[14] SPEC. Standard performance evaluation corporation (spec) 2000 benchmark suite. http://www.spec.org

[15] Srinivasan, T. et al. Continual Flow Pipelines. ASPLOS-XI, 2004.

[16] Tseng, J.H. and Asanovic, K. A Speculative Control Scheme for an Energy-Efficient Banked Register File. IEEE Transactions on Computers, 2005.

[17] Tullsen, D.M. Simulation and modelling of a simultaneous multithreading processor. Int'l Ann. Computer Measurement Group Conference, 1996.