Large Virtual ROBs by Processor Checkpointing

A. Cristal, M. Valero, J. Llosa and A. Gonzalez
Universitat Politècnica de Catalunya
Departament D'Arquitectura de Computadors

Abstract

Instructions that have long latencies, such as loads that miss in the L2 cache, and have to go to main memory, considerably reduce the execution speed of programs. This phenomenon is expected to worsen in the near future. Furthermore, in-order commit of instructions results in stalls in execution due to the ROB (Reorder Buffer) getting filled with long latency instructions and subsequent instructions. Making the ROB larger is technologically possible, but the cost can be excessive.

In this work, we propose a checkpointing mechanism that allows us to construct large, virtual ROBs from physical ones with small number of entries (32 or 64). When the ROB is full and the instruction at the top has a long latency, a checkpoint is made. That is, all the information that is needed by the processor, to restart from that point, is saved. From this point onward, instructions are released from the ROB so new instructions can be placed in the ROB. Processor information needs not to be saved when branches are predicted. Checkpointing also enables to virtually commit instructions out of order. Additionally, physical registers can be early released. This property considerably reduces the pressure on the register file in comparison to the traditional mechanism of in-order commit providing an increase in performance when registers are scarce. Out of order commit though checkpointing can have a significant advantage in SMT processors where the threads compete for processor resources like the ROB and physical registers.

We evaluate the performance of the mechanism for a superscalar processor with different memory latencies and internal resources configurations. Using a ROB of 64 entries and for the most realistic configurations, we get significant speedups over the same processor with a 2K entry ROB and in-order commit.

1 Introduction

Processor speed has increased at a rate much greater than memory speed and this trend will continue in the future, making the problem of the memory wall harder and harder [40]. Many techniques have been proposed and used in the past for reducing the negative effects due to memory latency. Cache memory [35] exploits the spatial and temporal locality exhibited by most programs. Prefetching, both hardware [10][22] and software [1][38] fetches from memory data before it is requested.

In the future, the pipeline will be deeper in order to increase clock speed and therefore memory latencies may significantly increase [2][13][27]. With a main memory latency of hundreds of cycles and an issue width of several instructions per cycle, a processor could theoretically fetch and execute a few thousands of instructions during the handling of a miss. However, current architectures can barely support hundred instructions in flight.

To keep the processor active during such long latency miss handling, a ROB that can hold a large number of instructions is needed. Besides, the number of physical registers needed is directly proportional to the number of instructions in the ROB [37]. Finally, a large instruction queue is needed. We have experimentally observed that for integer programs, a 128 entry IQ can be sufficient to give almost maximum performance when we increase the size of the ROB and the number of physical registers for a typical superscalar processor. This is not the case for numerical applications where we need to scale the number of entries in the IQs in order to get better performance.

In this paper we propose a novel technique to implement a large virtual ROB from a small physical ROB (32 or 64 entries). The technique is based on checkpointing when large latency
instructions block the commit of instructions. Checkpointing allows the processors to retire younger instructions, regardless of whether they have been executed or not. After a checkpoint is created if a younger instruction causes an exception or misspeculation the state can always be restored to the point where the checkpoint was made.

Retired instructions release entries in the ROB and the register file. Virtually committing instructions out of order allows to early release a large number of physical registers. This reduces the pressure on the register files in contrast to traditional approaches where registers are released only when instructions commit in order.

The proposed mechanism is evaluated for a superscalar processor executing several SPEC2000 integer and fp benchmarks, with different memory latencies of 100, 500 and 1000 cycles and different numbers of physical registers and entries in the IQs. The results show that our checkpointing mechanism creates large virtual ROB using a small physical ROB. In addition, our ability to reduce the pressure on the physical register file gives considerable speedup over a traditional large ROB.

The paper is organized as follows. In section 2, we motivate our work. In section 3 we describe our proposal using an example. Section 4 contains related work. In section 5 we provide results, and section 6 summarizes our work and future directions.

2 Motivation

The processor-memory gap significantly degrades the IPC of the processor. Figure 1 shows the IPC of a processor whose characteristics are described in section 5, when executing some specfp2000 applications, for varying memory latencies, sizes of the ROB and IQs, and the number of physical registers.

![Figure 1. Performance for Specfp2000 programs for different configurations](image)

The left most 3 bars correspond to the baseline processor with a 64 entry ROB, 64 entry integer, fp and load/store queue, and 96 physical integer and fp registers. The other columns are obtained by increasing the sizes of the ROB, IQ and the Register File. Comparing the baseline values with those of the most extended configuration (3 right most bars), the speedup is 2.41, 5.08 and 5.33 for latencies of 100, 500 and 1000 cycles, respectively. For a 100 cycles latency, a 512 entry ROB with 544 physical registers is almost optimal. For larger latencies, we need more resources in order to obtain the same IPC as the baseline configuration. We can obtain, for latencies of 500 and 1000 cycles, the same or higher IPCs as the baseline for a latency of 100 cycles if we add sufficient resources.
Figure 2. Performance for Specint2000 programs for different configurations

In figure 2 we show the same IPC values while executing Specint2000 applications. In this case when resources are added, the increases in performance for a memory latency of 100 cycles are less than 10%. For larger memory latencies, the speedup we get by adding resources to the baseline, vary from 1.31 to 1.44 for latencies of 500 and 1000 cycles, respectively.

In this paper we present a technique that allows us to implement a large virtual ROB using a small physical ROB. Our results show that the IPC values we obtain are very similar to, and better in many realistic configurations, than those obtained with a large physical ROB.

3 Architecture Description

In this section we discuss our approach to checkpointing for creating large virtual ROBs out of small physical ones. First, we give an informal overview of the technique introducing the main concepts. Then we give the technical details of how to implement the creation of checkpoints, the release of instructions from the ROB, the execution of instructions, and finally how registers can be released early in our approach.

3.1 General overview of processor checkpointing

The proposed architecture starts executing instructions as a normal superscalar processor does. Instructions are retired in-order from the ROB. The processor uses two mapping tables. The Renaming Mapping Table (MT), is used when instructions are decoded and renamed before they enter the ROB and the instruction queues. Information related to this table is saved by the processor at any branch prediction. The CMT Commit Mapping Table (CMT). This table also contains the mapping information between logical and physical registers at the moment instructions are in-order committed and leave the ROB. The processor uses this CMT in case of serving exceptions, as a part of the processor state. We also need this CMT information to create checkpoints.

When the ROB is full and there is a long latency instruction on the top, the checkpoint procedure starts. We need to save information about the processor state that will permit us to restart the execution from the same point. All this information is saved in what we call a checkpoint entry (CHKPT). Multiple checkpoints may be active in a given moment. They are stored in the Checkpoint Table (CT).

Each entry in this table contains the following fields: CHKPT-PC (PC of the instruction where the checkpoint is created), CHKPT-CI (instruction code), the modified register map table (which is the CMT, with minor modifications), the physical registers associated to the logical destination register of the instruction that creates the checkpoint (the current and the previous one), and a CHKPT-COI (instruction counter) whose meaning will be described later in subsection 3.2. All register mapping tables use CAM scheme register rename logic [11][36].
From now on, for an instruction \( \text{INST: } r_i := r_j \text{ op } r_k \), the current and the previous physical register associated to the logical register \( r_i \) are referred to as \( f_{i,c} \) and \( f_{i,p} \) respectively. \( f_{i,c} \) and \( f_{k,c} \) are the current physical registers associated to the logical registers \( r_j \) and \( r_k \), respectively.

After a checkpoint is created, the instruction that originated it, and the instructions following it, can be removed from the ROB allowing new instructions to be decoded and executed. These instructions are said to be released from the ROB. When we release an instruction it can be either finished or unfinished. Subsection 3.3 describes the actions to be performed when we release instructions. We have to say here that we are very conservative in our current implementation releasing instructions from the ROB. In the results we present, we stop the release of the instruction when there exists, in the top of the ROB, either a store instruction that has not computed its memory address or when there exist chains of very close dependent instructions. In the results section we will see that this considerably hurts the performance of the mechanism.

All unfinished instructions released after a checkpoint have to be identified and associated to this checkpoint until a new checkpoint is created. For this reason we need to add to any entry in the instruction queues (IQs) two additional fields. The first, IQ-V is a bit that indicates if the corresponding instruction has been released or not from the ROB. The second field, IQ-CHKPT indicates the entry number in the CT where the checkpoint entry for this instruction is located.

We also need to take care of the use of the physical registers. For this purpose, we use what we call the RCV (Release Counter Vector). The RCV is a unique vector of counters with the same number of entries as physical registers in the architecture. Each entry stores the number of reads from that register by instructions that have not finished but that have already been released from the ROB. The counters are incremented when the instructions are released from the ROB and decremented when the instructions are executed. When there is no active checkpoint, all entry counters are 0.

We call CS (Commit State) the information we need to create a checkpoint and to maintain register file state and will be made by CMT and RCV. In figure 4 we show a diagram of a processor that commits instructions in order including the new blocks used by our mechanism.

Store instructions are not allowed to send their values to memory until all checkpoints older than the store and the one to whom this instruction belongs to, have been validated. When a checkpoint is created, a control signal and the entry number in the CT are sent to the store buffer to indicate that a new control point begins. The store buffer is not allowed to send any store instruction to the memory, until it receives the corresponding unblocking signal.

Load instructions can be removed from the load/store queue when they are finished and they are released from the ROB. In case of a store instruction, the condition that it has to fulfill to be released from the ROB is that its memory address has been computed. In case it is retired without having computed the value to be stored, it adds 1 to \( \text{RCV}(f_j,c) \) and \( f_{j,c} \) is marked to propagate when the data to be stored is obtained. When this value is computed and stored in \( f_{j,c} \), it is also sent to the store buffer and we subtract 1 from \( \text{RCV}(f_j,c) \).

When instructions are executed, they are removed from the IQs. If the instruction was not released from the ROB, there is still a copy of it in the ROB. If the instruction was released from the ROB we will lose any copy of it and we need to check whether this instruction is the last one belonging to a checkpoint that needs to be executed. If this instruction is the last one, the entry associated to its checkpoint can be removed from the CT. Moreover, if this checkpoint is the oldest one in the CT, all the instructions belonging to it will be committed.
When an exception associated to any instruction belonging to a checkpoint or a branch misprediction occurs, the processor is forced to restart executing instructions from the checkpoint instruction. This is because in our current implementation, we do not save any processor state associated to branch prediction. Currently, we are working on it, to reduce the additional penalty associated to the way we do checkpoint. When an exception occurs, we have to flush all instructions located in the IQs belonging to this checkpoint and to all checkpoints following it. This action can require some extra overhead. We propose to do this flush of the instructions in parallel with the re-execution of new instructions as described in subsection 3.4. The proposed mechanism supports precise exceptions.

### 3.2 Creating Checkpoints

In order to explain how the system works, we use the example presented in figure 5. At the right of each instruction, we put the event and the action that will be performed. We assume that the processor has 6 logical registers, 16 physical registers, a ROB of 6 entries and an integer IQ and Load/store queue with 6 entries each. We also suppose that the CT has 3 entries.

#### Instruction Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>Ld r1,@r2</td>
<td>L2 miss Create checkpoint</td>
</tr>
<tr>
<td>I2</td>
<td>r3:=r1+r2</td>
<td></td>
</tr>
<tr>
<td>I3</td>
<td>r1:=r4+r5</td>
<td></td>
</tr>
<tr>
<td>I4</td>
<td>r1:=r1+1</td>
<td></td>
</tr>
<tr>
<td>I5</td>
<td>br r3==0,L1</td>
<td>branch misprediction Release unfinished</td>
</tr>
<tr>
<td>I6</td>
<td>Ld r3,@r1</td>
<td>L2 miss Create checkpoint</td>
</tr>
<tr>
<td>I7</td>
<td>br r3==0,L2</td>
<td>correct prediction Release unfinished</td>
</tr>
<tr>
<td>I8</td>
<td>Ld r4,@r5</td>
<td>L1 hit Release finished</td>
</tr>
<tr>
<td>I9</td>
<td>r6:=r4+1</td>
<td></td>
</tr>
<tr>
<td>I10</td>
<td>r5:=r3+8</td>
<td></td>
</tr>
<tr>
<td>I11</td>
<td>r4:=r2*2</td>
<td></td>
</tr>
<tr>
<td>I12</td>
<td>r4:=r5+r4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. Example we use to explain the mechanism
A checkpoint will be created when the ROB is full and there is an instruction with high latency on top. In our example, instruction I1 misses in L2 and the ROB is full. In figure 6, we show the content of the different structures when the processor creates the checkpoint associated to instruction I1.

Since there are free entries in the CT, a checkpoint is created. Figure 7 shows the information stored in the tables after the checkpoint has been created. The processor does the following actions:

- The CMT is copied to the field CHKPT-CMT. After that we set the entry CHKPT-CMT(fi_c) to 1, where fi_c is the physical register number assigned to the logical destination register ri of the instruction that creates the checkpoint. In our case, the logical register r1 of the load instruction has associated the physical registers 1 and 11 in CHKPT-CMT. We modify this table in this way with the objective of avoiding registers to be released during the time the checkpoint is active.

- The PC of the instruction is copied to CHKPT-PC.

- The instruction code is copied to the field CHKPT-CI, I1 in our case.

- The numbers of the current and previous physical registers of the logical destination register ri are copied to the fields CHKPT-f_c and CHKPT-f_p, respectively. In our case, the numbers 1 and 11 will be copied.

- The field CHKPT-COI is initialized to the value 1. This field counts the number of instructions belonging to a checkpoint that have been retired/released from the ROB and have not finished yet.

- Update the Commit State (CS). The RCV and CMT are updated as described later in section 3.3. In our case, RCV(12) is set to 1, CMT(1) is set to 1 and CMT(11) is set to 0.
- Send a signal to the buffer store queue in order to avoid the issue of stores to the memory protected by this checkpoint.

- Set bit IQ-V to 1 for the entry in the IQ that contains the instruction that creates the checkpoint (entry number 1 of the load/store queue). This instruction has now been released from the ROB. The index of this checkpoint entry in the CT is set in the IQ-CHKPT field of the instruction (in our case value 1 since this is the entry used in the CT).

- In a traditional processor with in-order commit, when a instruction commits, the busy flag associated with the previous physical register (f_p) is set to 0 (e.g. busy_flag (11) is set to 0). In our case, this does not mean that this register can be assigned to another logical register. This register is blocked due to CHKPT-CMT(f_p) field is set to 1 (explained later).

- The instruction is released from the ROB.

<table>
<thead>
<tr>
<th>Commit State</th>
<th>RCV</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMT</td>
<td>0</td>
</tr>
<tr>
<td>RCV</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checkpoint table</th>
<th>PC</th>
<th>F_p</th>
<th>F_c</th>
<th>Instruction counter (COI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>Entry 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 1 1 1 1 1</td>
</tr>
<tr>
<td>Entry 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 1 1 1</td>
</tr>
</tbody>
</table>

Figure 7 CS and CT information after I1 creates a checkpoint

### 3.3 Release of instructions

As discussed above, the creation of a checkpoint allows the processor to remove instructions that follow the checkpointed instruction, from the ROB. These instructions are said to be “released” from the ROB.

When any instruction is retired/released from the ROB (e.g., when a checkpoint is created), the Commit State (CS) information will be updated as follows:

- Update of the CMT:
  - CMT(f_p) ← 0
  - CMT(f_c) ← 1

- Update of RCV: This is only done if the instruction do not finish:
  - RCV(f_j_c) ← RCV(f_j_c) + 1
  - RCV(f_k_c) ← RCV(f_k_c) + 1

There are three kinds of release of instructions in addition to the ordinary instruction retirement.

The first case corresponds to the creation of a checkpoint. This case has been described previously.

The second case corresponds to the situation in which the instruction that we are going to release from the ROB has not finished its execution. This will be the case of instruction I2 that cannot be executed due to its dependency on I1. The processor performs the followings actions:

- Update the CS as described previously. (e.g. RCV(1) is updated to 1, and RCV(12) is updated to 2, CMT(2) is set to 1 and CMT(13) is set to 0).
- The CHKPT-COI field of the later created CT entry is increased by 1 (i.e., the field has the value 2).

- Mark the instruction in the instruction queue (in the example entry 1 of the integer instruction queue). The corresponding field IQ-V is set to 1 to indicate that this instruction has been released and the IQ-CHKPT field is set to the new entry number in the CT (value 2 in our case).

- As in a traditional processor with in-order commit, the busy flag associated with the previous physical register (fi_p) is set to 0.

The third case corresponds to the situation in which the instruction that we are going to release from the ROB has finished its execution. This is the case for instructions I3 and I4, which depend neither on I1 nor on I2. Suppose that the logical registers of the destination and the source operands are ri, rj and rk respectively (i.e., r1, r4 and r5, respectively, for instruction I3) to which were assigned most recently the values fi_c, fj_c and fk_c (physical registers 3, 14 and 15, respectively), along with that the physical register assigned previously to ri has been the fi_p (register 1 in our case). For this case the processor will do the following actions:

- Update the CS. (e.g. for instruction I3, CMT(3) is set to 1 and CMT(1) is set to 0. Instruction I4: CMT(4) is set to 1 and CMT(3) set to 0)
- Update the busy flag (busy_flag(1) is set to 0 and busy_flag(3) is set to 0).

The fourth case is the normal case in which the oldest instruction in the ROB is retired and there is no checkpoint. This is the normal commit of a processor that does in-order commit of instructions. The actions that we will have to do are as follows:

- Update the CS as described previously.
- Update the busy flag.

Figure 8 depicts the state of the processor after instruction I2 is released without having finished, instructions I3 and I4 that already have finished and instructions I7 and I8 having been decoded.

Figure 8. Intermediate state of the processor
### 3.4 Execution of Instructions

Instructions can finish their execution while they are in the ROB or after they have been released from it. If an instruction has been released from the ROB, we must consider 3 cases.

The first corresponds to the case that the instruction has finished in a normal way and, in addition, is not the last one for its associated checkpoint. The actions to take by the processor are as follows:

- Subtract by 1 the CHKPT-COI counter.
- Subtract by 1 the RCV(fi) and RCV(fk_c) counters.
- Retire the instruction from IQ.

The second case is when the instruction finishes normally and is the last one associated to its checkpoint (IQ-CHKPT). In this case, the actions to carry out by the processor are as follows:

- Subtract by 1 the RCV(fi) and RCV(fk_c) counters
- Retire the instruction from IQ.
- Clear the entry (IQ-CHKPT) of the checkpoint in the CT
- Send a signal to the store buffer to inform the end of the checkpoint IQ-CHKPT. If the instruction belongs to an older checkpoint, it means that the blocked stores of this checkpoint can be sent to memory.

The third case is when the instruction ends with an exception or is a branch instruction that is mispredicted. It is necessary to distinguish 2 cases depending on whether the instruction that has generated the checkpoint has finished or not.

First, if the instruction that generated the checkpoint has finished, it must carry out the following actions:

- Invalidate all ROB entries.
- Restart fetching from the next instruction to the one that creates the checkpoint.
- The normal MT and the CMT are updated with the modified value of the CHKPT-CMT, so that MT(fi_p) = CMT(fi_p) = 0.
- Clear in the CT all entries created after this checkpoint, including itself.
- Clear from the store buffer all stores subsequent to the creation of the checkpoint.
- It is necessary to eliminate in the IQ´s all the instructions subsequent to the instruction that created the IQ-CHKPT checkpoint. In addition, the problem is that we have to reflect the influence that these instructions have on the values of the RCV. This process could be very slow if we do it in a sequential way. Moreover, the processor could be blocked. We solved this problem as follows. We mark all the instructions that belong to IQ-CHKPT checkpoint and younger checkpoints. This process is very fast. Once they are marked, the processor continues executing other instructions. When the mechanism of selection of instructions has free slots (there is no sufficient ready instructions in the program) the processor selects marked instructions to be executed like a non-operation. The write back stage will correctly update the values of the RCV without overhead.

Second, if the instruction that generates the checkpoint has not finished, the actions to be done are the same as in the previous case, except that we keep the IQ-CHKPT checkpoint entry and the copy of the instruction in the IQ.

Figure 9 shows the information of the tables after finishing instructions I3 and I4 and releasing instruction I5, after creating a new checkpoint for instruction I6 and after finishing instruction I8. At this point, it is not possible to continue decoding instructions since the IQ is full.
Figure 9. Intermediate state of the processor

Figure 10 depicts the content of the tables when the second checkpoint has been eliminated, since instructions I1, I2, I6 and I7 have finished and the branch was correctly predicted. Instructions I7 and I8 are released from the ROB.

Figure 10. State of the processor after the elimination of the second checkpoint
In figure 11 the state of the processor can be seen when the branch instruction I5 has been solved and was mispredicted. In this case, we need to restore the state of the processor using the checkpoint table entry 1, so we start fetching instruction I2 again.

### ROB

<table>
<thead>
<tr>
<th>Inst</th>
<th>f_p</th>
<th>f_c</th>
<th>s1</th>
<th>s2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r3:=r1+r2</td>
<td>13</td>
<td>10</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>r1:=4+r5</td>
<td>1</td>
<td>11</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>r1:=r1+1</td>
<td>11</td>
<td>2</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>br r3==0,L1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>br r3&gt;0,L3</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Integer Queue (IQ)

<table>
<thead>
<tr>
<th>Inst</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3:=r1+r2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>R1:=4+r5</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>R1:=r1+1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>br r3==0,L1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>br r3&gt;0,L3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Ld/St Queue (LSQ)

<table>
<thead>
<tr>
<th>Inst</th>
</tr>
</thead>
</table>

### Physical Registers

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td># Logical Reg.</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MT</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Busy Flag</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Commit State

<table>
<thead>
<tr>
<th></th>
<th>MCF</th>
<th>RCV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Checkpoint table

<table>
<thead>
<tr>
<th>Entry</th>
<th>PC</th>
<th>f_p</th>
<th>f_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entry 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction counter (COI)

- WAW hazards
- busyn flag(f) = 0 means that the register is free for the "normal" processor.
- CMT(f)=0 implies that this register is not needed for checkpoint creation.
- RCV(f)=0 implies there is no released instruction that needs to read this register.
- \( \forall CHKP T \rightarrow C M T \in C T : C H K P T \rightarrow C M T(f) = 0 \) implies that the register is not needed by any checkpoint to restore the processor state

**Figure 11 Final state**

### 3.5 Early release of the registers

In a normal processor, registers are released when instructions are retired from the ROB. In our case, we cannot release registers when the processor releases instructions from the ROB. However, checkpoints allow us to early release some registers. Consider the code in figure 12. In instruction I3, physical register f7 is assigned to the logical destination register R1. Physical register f5 was the previous register assigned to logical register R1. After executing instructions I3 and I5, the processor can release physical register f7, since there is no instruction that needs its value. Also instruction I9 can finish. However, the processor cannot release physical register f8 that was assigned to logical register R1 in instruction I5 for two reasons. Instruction I8 must read the register f8. After finishing the execution of instruction I8, it cannot be released since the value of physical register f8 belongs to the checkpoint that was created in instruction I6. For that reason, physical registers f8 and f6 cannot be released until instruction I6 corresponding to the second checkpoint is finished. When this happens, and checkpoint 2 has been validated, it is possible to release register f8, but not register f6 since this register belongs to checkpoint 1. In this example, 2 physical registers are release, the registers f7 and f8.

The condition for a physical register to be freed is as follows:

\( (\text{f is ready}) \land (\text{busyn flag}(f) = 0) \land (\text{MCT}(f) = 0) \land (\text{RCV}(f) = 0) \land (\forall \text{CHKPT} : \text{CMT} \in \text{CT} : \text{CHKPT} \rightarrow \text{CMT}(f) = 0) \)

- where “f is ready” means that the instruction that writes register f has finished. This avoids WAW hazards
- busyn flag(f) = 0 means that the register is free for the “normal” processor.
- CMT(f)=0 implies that this register is not needed for checkpoint creation.
- RCV(f)=0 implies there is no released instruction that needs to read this register.
- \( \forall \text{CHKPT} : \text{CMT} \in \text{CT} : \text{CHKPT} \rightarrow \text{CMT}(f) = 0 \) implies that the register is not needed by any checkpoint to restore the processor state.
Our approach registers can be released early, relieving register pressure. We discuss a few realistic sizes of IQ and register file but using a small ROB of 64 entries. Next we discuss how in our approach registers can be released early, relieving register pressure. We discuss a few

### 3.6 Load/Store Queues

Our mechanism works because instructions belonging to a checkpoint are not validated until they have finished all correctly. An exception raised by anyone of them or the misprediction of any branch, causes all instructions to be invalidated and they are restarted to execute from the next instruction, which created the checkpoint. Store instructions do not have to send their values to memory until all checkpoints older than the store have been validated.

When a checkpoint is created, a control signal and the entry number in the CT are send to the store buffer to indicate that a new control point begins. The store buffer does not have to send any store instruction to the memory, until it receives the corresponding unblocking signal.

Load instructions can be removed from the load/store queue when they are finished and they were released from the ROB. In case of a store instruction, the condition that it has to fulfill to be released from the ROB is that its memory address has been computed. In case it retires without having the value \((f_j)\) to be stored, it adds 1 to RCV\((f_j)\) and \(f_j\) is marked to propagate when the data to be stored is obtained. When this value is computed and stored in \(f_j\), it is also send to the storebuffer and subtract 1 to RCV\((f_j)\).

### 4 Performance Evaluation

In this section we evaluate the proposed approach. First, we discuss our experimental setup. Then we compare the performance obtained by the extended baseline configuration and our approach. Note that the extended baseline configuration is a very aggressive configuration with a large ROB and we show that our approach is capable of obtaining better performance for realistic sizes of IQ and register file but using a small ROB of 64 entries. Next we discuss how in our approach registers can be released early, relieving register pressure. We discuss a few
shortcomings of our current implementation and possible solutions for them. Finally, we discuss the size of the virtual ROB that is created by the checkpoint mechanism.

### 4.1 Experimental Setup

For this study, we developed a proprietary simulator that is a highly modified version of SimpleScalar (version 3.0b). Our SPEC2000 benchmarks are pre-compiled binaries, obtained from the simplescalar developers. The SPEC benchmarks operate on their reference data sets. Four SpecFP programs (applu, art, apsi and lucas) and four SpecInt programs (gap, twolf, vpr and vortex), have been selected. They represent applications that produce a sufficient number of misses in L2 cache and moreover exhibit different branch prediction behavior. For the benchmarks, we skip the first 2000 million instructions, then we execute the next 50 million instructions warming up the instruction and data caches, and after that, we take statistics for 200 millions of instructions.

The characteristics of the architecture are described in figure 13.

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Baseline Extended</th>
<th>Release out of order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reorder Buffer</td>
<td>64</td>
<td>2048</td>
<td>64</td>
</tr>
<tr>
<td>Load/store queue</td>
<td>64</td>
<td>128,256,512,1024</td>
<td></td>
</tr>
<tr>
<td>Integer queue</td>
<td>64</td>
<td>Equal LSQ entries</td>
<td></td>
</tr>
<tr>
<td>Floating point queue</td>
<td>64</td>
<td>Equal LSQ entries</td>
<td></td>
</tr>
<tr>
<td>Decode with</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Instruction fetch queue</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Functional units</td>
<td>4 integer ALUs (1/1 cycle), used to calculate address too, 2 integer mult/div (3/1,20/20 cycles), 4 fp address (2/1 cycles), 1 mult/div/sqrt (4/1,12/12,24/24 cycles), 1 memory port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch predictor</td>
<td>Bimodal 2048 entries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>miss predict recovery</td>
<td>3 cycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data L1</td>
<td>32 kb 4 ways 32 bytes line, 1 cycle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions L1</td>
<td>32 kb 4 ways 32 bytes line, 1 cycle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 unified</td>
<td>256kb 4 ways 64 bytes line, 10 cycle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLB</td>
<td>64 entries, 4 way, 8kb page size, 30 cycles penalty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>100, 500 and 1000 cycles first chunk + 4 next chunk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer register file</td>
<td>96</td>
<td>Equal LSQ entries + 32</td>
<td></td>
</tr>
<tr>
<td>Fp register file</td>
<td>96</td>
<td>Equal LSQ entries + 32</td>
<td></td>
</tr>
<tr>
<td>Checkpoints table</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checkpoint creation</td>
<td>4 cycles</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 13. Processor Configuration

Checkpoints are created at two moments. First, when an instruction remains for 4 cycles in the top of the ROB having its source operands ready, and the ROB is full. Second, by the first branch instruction after 100 instructions from the last checkpoint that is going to be released and has not been executed. This is done because we have to avoid the negative effect for our mechanism of having large sequences of dependent instructions that end on a mispredicted branch (e.g. list traversal). We did simulations using different number of entries in the CT (Checkpoint Table). We found that 8 entries is a good choice. When the CT is full, we do not stop the release of instructions and all these new instructions belong to the last checkpoint.

We compare the results obtained with our proposal against the results obtained by the baseline and the extended baseline. The baseline architecture has a ROB and IQs with 64 entries and 96 physical registers of each type.

For the extended baseline we have at least 2 options. One is to consider a ROB that is sufficiently large not to be the limiting resource. On the other hand, we can vary the size of the ROB depending on the sizes for the IQ and on the number of physical registers. As a matter of fact, a ROB with a size twice the number of physical registers behaves as an unbounded ROB and produces a speedup of around 10% over the same configuration with a ROB of size equal
to the number of physical registers. We decided to use a ROB of 2048 entries for all the extended baseline configurations because we are interested in comparing this baseline with our proposal of using a small ROB to create a larger virtual ROB. For our proposal, we always consider a ROB of size 64.

### 4.2 Performance Measurements

![Performance Measurement Diagram](image)

**Figure 14** Performance for the mix of Spec2000 programs

In figures 14, 15 and 16, we show the average IPC for different machine configurations executing the mix of spec 2000 applications, specfp 2000 and specint 2000, respectively. In all figures performance is normalized with respect to the baseline architecture for latency of 100 cycles (see figures 1 and 2).

The 3 bars (columns) on the left represent the normalized values of the IPC of the baseline architecture for latencies of 100, 500 and 1000 cycles, respectively.

Each group of 4 bars represents speedups of the extended baseline (Ext. Bas.) and our proposed architecture (Proposed), for latencies of 500 and 1000 cycles. For each group of bars the sizes of the instruction queues and the physical registers are varied.

We see, in figure 14, that the configurations of extended baseline and our proposal, for the same values for the instruction queues and registers of the baseline architecture, have a speedup of 1.06 for latency 500 cycles over the baseline. As we increase the hardware resources of the extended baseline architecture and our proposal, the speedup increases considerably.

![Performance Measurement Diagram](image)

**Figure 15.** Performance for the Specfp2000 programs
For many intermediate values of resources, we observe that our architecture reaches better benefits than the extended architecture. Only for the most aggressive configurations the extended baseline architecture gets better results than our proposal. Below we discuss how these results are related to the way in which both architectures do the release of the registers, as well as the mechanism that we have implemented to retire instructions from the ROB.

Our proposed architecture can get speedups of 1.26 compared to the baseline for a latency of 500 cycles a queue size of 96 entries and 128 physical registers for both configurations (the speedup for IQ of 512 entries and 544 physical registers is 1.77). For latencies of 1000 cycles, our mechanism reaches a speedup of 1.5 for IQ of 256 entries and 288 registers, and a speedup of 2.2 for IQ of 1024 entries and 1056 registers.

It is worthwhile to remark that for the 2 right most groups of 4 bars, both, the extended baseline and our proposed architecture, for memory latencies of 500 and 1000 cycles, obtain the same or better IPC than the baseline architecture for 100 cycles latency. These results encourage further research in this field.

For the SpecFP applications in figure 15 we observe that the gains obtained when we increase the resources in the extended baseline architecture and in our proposal are much higher than those found in figure 14 for the full set of the applications. For latencies of 500 and 1000 cycles we reach speedups of 4 and 3 respectively. For our proposal, we obtain speedups close to the maximum, for an IQ of 96 entries and 128 physical registers, and speedups near 3 for a IQ of 512 entries and 544 registers.

![Figure 16. Performance for Specint2000 programs](image)

Concerning the non-numerical applications, for a latency of 500 cycles, the maximum gains, higher than 1.25, are obtained for an IQ of 512 entries and 544 registers. For a latency of 1000 cycles, a speedup of 1.38 is obtained. With our technique, we get speedups of 1.19 with 96 entries in the IQ and with 128 physical registers. For these integer programs and latencies of 500 and 1000 cycles, our technique is better than the extended baseline architecture in all the configurations except for the most aggressive right bars.

### 4.3 Early Register Release

Our mechanism is able to release early many registers, which makes it more efficient than the extended baseline architecture with the same number of entries in the IQ and with the same number of registers in the case the number of registers is the factor that blocks the entry of new instructions in the ROB. In figures 17 and 18 we show how registers are released for the integer and floating point applications, respectively.
Figure 17. Release of registers for Specfp2000 applications

In figure 18 we show that for integer applications the extended baseline releases on average a 50% of the registers in the in-order commit stage (denoted “normal”) and other 50% when there are branch mispredictions (“rollbackrob”). Moreover, the percentage of released registers in commit is much higher for less aggressive speculation. In floating point applications, see figure 17, the extended baseline releases almost all the registers in the in-order commit stage, because branches are well predictable for these applications.

Figure 18. Release of registers for integer applications

Our architecture can release registers in 4 additional situations. In our case the “normal” situation is when the registers are released in order because there is not any active checkpoint. “rollbackrob” again denotes the situation that registers are released due to branch misprediction and the instruction is in the ROB. We say that the registers are released “Unblock-first” when they are released at the time the oldest checkpoint is retired. “normal check” situation when registers are released at the time we release from the ROB an instruction already executed. We call an “out of order” situation when a register is released in any of the three following situations: first, if the instruction was released from the ROB and finished later releasing any register either associated to the destination or sources registers; second, when the register is released due to a checkpoint but not the oldest one finished, and finally, when the register is blocked by a store instruction and the value is computed and the address and value are sent to the store queue. Finally, we denote by “rollbackchk” the situation when a branch that was
released from the ROB was mispredicted. In this case, we know we have to return to the checkpoint to which this branch belongs and we release registers.

For our architecture and SpecFP applications, around 35% of the registers are released before they would have been using an in-order commit of the instructions. As the architecture becomes more aggressive this percentage increases. In the integer set of applications, the percentage of registers that our architecture early released is approximately 17%, and is also greater for more aggressive configurations.

5 Simultaneous Multithreading

Simultaneous multithreading (SMT) is a technique where several tasks are executed simultaneously in parallel sharing some of the resources of the processor. The proposed mechanism has many advantages for SMT architectures. In SMT processors, each thread requires an independent ROB. With our approach each ROB can be very small while highly effective. Moreover, all the simultaneous threads can share a single checkpoint table, making it very cost effective.

Another problem that arises in many SMTs is that the physical register file is shared among all threads. If the register file is small there is high contention between threads severely limiting performance. In order to effectively exploit parallelism between threads a big register file is required. With our approach, registers are released earlier, reducing the register pressure between threads.

6 Related Work

The results presented to motivate our research are similar to those reported in [25] and [3]. There is much work done oriented to reduce the negative effect of the memory wall problem [40]. Among them we can cite the papers proposing and evaluating hardware prefetching methods [10][22], software prefetching [1][38] or a combination of both [5]. In some recent papers, authors propose the use of helping threads that are executed with the objective of preexecute those instructions that access the locations that miss in the L2 cache. Some of these techniques are Assisted Execution [41], Simultaneous Subordinate Microthreading (SSMT) [33], and some variations [17][29]. Other authors propose to generate and execute slices of instructions that allow also to preexecute offending loads and branches that are difficult to predict [4][8]. Other work in a similar direction is the Slipstream processor [26].

Many techniques have been proposed to create large ROB and IQs, such as the multiscalar processor [15], the trace processor [12] and the DMT architecture [16]. These architectures allow speculative execution of a large number of instructions. In [32], a method to execute a large number of instructions with a small ROB and a reduced number of physical register is presented. There are other papers dealing with the performance optimization of memory consistency models for shared memory multiprocessor systems [7][31] that are also related to our work. Other papers propose technique to reduce the complexity, the access time and the power of the IQs [3][6][9][19][24][30][36].

Concerning to the problem of the register file, there are some proposals targeted to reduce the number of physical registers, using the fact that physical registers are not needed until results are produced [14] or by exploiting temporal locality [34]. There are other proposals oriented to release registers as soon as possible either at run time [28] or with the help of the compiler [18]. Hierarchical register files have been proposed as a way to organize large register files without increasing the access time [20][21].

Checkpointing has been proposed by Hwo and Patt [39] to support precise exceptions in out of order processors. In parallel to our proposal Martinez et al. [23] proposed Cherry to early release registers and load/store queue entries. In their work they use a large reorder buffer and a single checkpoint to recover the state if an exception occurs after the non return point.
In contrast our proposal uses multiple checkpoints while allows out of order release of instructions requiring a small ROB.

7 Conclusions and Future Work

Processor speed has increased at a rate much greater than memory. In the future, pipelines will be deeper in order to increase clock speed and therefore memory latencies will be even greater. To reduce the negative impact of technology on performance, large ROBs, large IQs and many physical registers are needed.

In this paper, we have proposed and evaluated a mechanism that permits to create a large virtual ROB from a physical one with a small number of entries, such as 32 or 64. The main idea is to create a checkpoint when the ROB is full and there is a long latency instruction on top of it. All the information the processor needs to restart the execution from this point is saved. This allows the processor to remove instructions from the ROB. We do not need to save processor information when branches are predicted. Checkpointing also enables to virtually commit instructions out of order.

Additionally, checkpointing instructions provide a protection mechanisms that allows early release of many physical registers reducing the pressure on the register file. This is an important difference with a normal processor that has in-order commit of instructions and that releases registers at the same time as the commit.

The proposed mechanism has been evaluated for several SPEC2000 applications, processor configurations and memory latencies. We showed that a large virtual ROB can be effectively built to mitigate the negative effect on performance of future memories with long latencies. On the other hand, the early release of registers makes our proposal more efficient than current processors with a larger physical ROB for many combinations of sizes of the IQs and the number of registers.

References:


