

# CROB: Implementing a Large Instruction Window through Compression

F. Latorre, G. Magklis, J. González, P. Chaparro, A. González

Intel Barcelona Research Center, Intel Labs – UPC  
{fernando.latorre, grigorios.magklis, pepe.gonzalez, pedro.chaparro.monferrer,  
antonio.gonzalez}@intel.com

**Abstract.** Current processors require a large number of in-flight instructions in order to look for further parallelism and hide the increasing gap between memory latency and processor cycle time. These in-flight instructions are typically stored in centralized structures called reorder buffer (ROB), which is a centerpiece to handle precise exceptions and recover a safe state in the event of a branch misprediction. However, this structure is becoming so big that it is difficult to fit it in the power budget of future processors designs. In this paper we propose a novel ROB microarchitecture named CROB (Compressed ROB) that can compress ROB entries and therefore give the illusion of having a larger virtual ROB than the number of ROB entries. The performance study of CROB shows a tremendous benefit, with an average speedup of 20% and 12% for a 128-entry and 256-entry ROB respectively. For some benchmark categories such as SpecFP2000, speedup raise up to 30%.

## 1 Introduction

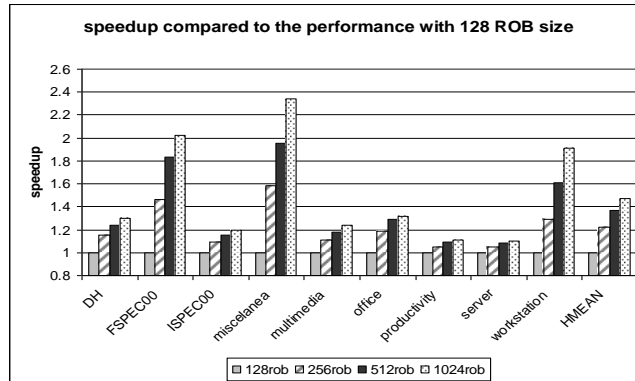
Modern out-of-order processors typically employ a reorder buffer (ROB) to retire instructions in program order [1]. In-order retirement enables precise bookkeeping of the architectural state, while making out-of-order execution transparent to the user. However, retiring instructions in program order increases the demand of some processor resources such as physical registers or the ROB entries themselves. The size of the ROB increases with every new processor generation. The reason is the continuously increasing difference between processor and memory speeds. Long-latency operations delay the instruction retirement and therefore the number of required in-flight instructions is augmented in order to keep the functional units busy. Besides, larger ROBs increase performance by exposing more instruction level parallelism.

Figure 1 shows the performance improvements obtained by increasing the ROB from 128 entries up to 1024 entries in a clustered microarchitecture. These numbers have been obtained by assuming an unbounded issue queue and physical register file to see the potential benefits of enlarging the ROB. As it can be seen, allowing just 128 in-flight instructions is an important limiting factor for performance. Nevertheless, it is not straightforward for the designers to implement larger ROBs because of power, area and cycle time constraints. As an example of the ROB size for a current processor, the Intel®Pentium® 4 supports up to 126 in-flight instructions [2].

There are a number of proposals to overcome the limitation that the ROB imposes over the number of in-flight instructions [3][4][5]. These techniques periodically

checkpoint the processor state in order to support precise exceptions allowing ROB entries to be released out of program order, or even getting rid of the ROB. The authors have demonstrated great potential by using these checkpoint mechanisms as an alternative to the conventional ROB. However, whenever either a misprediction or an exception arises a previous checkpoint must be restored and the instructions between the checkpoint and the offending instruction must be re-executed. This task and the periodic checkpoints that must be done increase the processor activity as well as the power dissipation of the processor.

In this paper we present a different approach to designing a reorder buffer that enables the processor to have many more instructions in-flight than the number of reorder buffer entries. This is achieved by introducing a level of indirection similar to the way that virtual memory pages are mapped to physical memory pages in modern operating systems. We propose that the Physical Reorder Buffer (PROB) of the processor (the traditional ROB) be divided into equally sized physical segments that are dynamically mapped to logical segments by a mapping table called the Logical Reorder Buffer (LROB). A physical segment is released (and allowed to be re-mapped) when all instructions kept in that segment are guaranteed to either atomically commit or atomically be squashed. Overall, the objective is to achieve high performance with a small reorder buffer with minimal extra activity. We refer to this novel design as *Compressed Reorder Buffer (CROB)*.



**Fig. 1.** Impact of the ROB size in a 2-clustered processor configuration with unbounded issue queue, unbounded registers and unbounded memory order buffer.

## 2 Related Work

Some hardware resources such as issue queues, physical registers, memory order buffer and reorder buffer are cumbersome to enlarge [10]. On the other hand, it has been demonstrated that the processor performance is very dependent on the size of these components. Hence, many researchers have proposed alternative designs to better utilize these components. For instance, some previous work [6][7][8][9][10] propose splitting the back-end engine into multiple processing units called clusters. Moreover, decreasing the power consumption and the complexity of the issue queue

[12][13][16][11][15][14][24] and the register file [17][18][20][19] has been widely addressed in the literature.

In [3], the authors propose a novel architecture named Cherry where ROB entries and physical registers are recycled as soon as the instructions are considered safe (all previous branches have been computed and all the previous loads have been issued), instead of waiting until the instruction commits. The main important difference between the proposal of this paper and Cherry, is that CROB does not require extra checkpointing. Another advantage of CROB is that ROB entries can be released earlier because it is not constrained by memory replay traps or branch mispredictions as opposed to Cherry. By contrast, the early register release scheme detailed in [3] is likely to be more effective than the one implemented on top of CROB. The proposals in [4] and [5] also rely on checkpointing.

Checkpointing has important overheads as described in [24] so that the number of instructions between checkpoints has to be quite large to reduce the cost of creating them. However, in the event of a branch misprediction, exception or memory replay, the instructions between the latest checkpoint and the offending instruction must be re-executed, which incurs in extra power dissipation. CROB does not require re-executions at all.

In summary, we propose a scheme that increases the number of in-flight instructions and does not require re-execution of instructions nor checkpointing. On the other hand, CROB could take advantage of some orthogonal techniques proposed in [3][4] such as the management of the memory order buffer or the early register release as well as the instruction group formation in [23]. These enhancements however are not considered in this paper.

### 3 Description of the Architecture

The evaluation of the CROB has been done on top of a state-of-the-art clustered processor. Nevertheless, the CROB can be used in any architecture as long as it has a ROB that does not store speculative register results. We believe clustered architectures are a good design for future processors because it allows building wide machines while controlling complexity. But, in order to keep wide machines busy we need to have a lot of instructions in flight, which means that bigger reorder buffers are required.

The baseline architecture is similar to the one proposed in [6]. A block diagram is shown in Figure 1. It consists of a monolithic front-end in charge of fetching, decoding and renaming instructions and a clustered back-end. This front-end fetches x86 macro-instructions and translates them into micro-operations that are stored in the trace cache. The main components in the front-end are the trace cache (TC) where micro-operations are stored, the instruction TLB (not shown in the figure), the branch predictor (BP), and the Macro Instruction Translation Engine (MITE) that translates macro-instructions into micro-operations before storing them into the TC. It also implements the instruction decoding, steering and renaming logic. The MROM is in charge of decoding complex macro-operations like string moves. A detailed description of these components can be found in [2].

Decoded instructions are steered to one of the two clusters for execution following the dependence- and workload-based algorithm described in [6]. Inter-cluster communication is performed via copy instructions that are generated on-demand by the rename logic. Every cluster includes an issue queue and three register files (integer, floating point and SSE). Once an instruction leaves the issue queue, it reads its source operands either from the register files or the bypass logic and executes in one of the functional units of the cluster. Finally, a shared memory order buffer and memory hierarchy is used to process store and load operations.

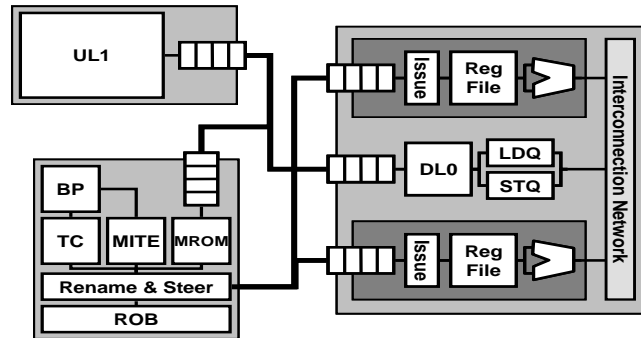


Fig. 2. Baseline architecture.

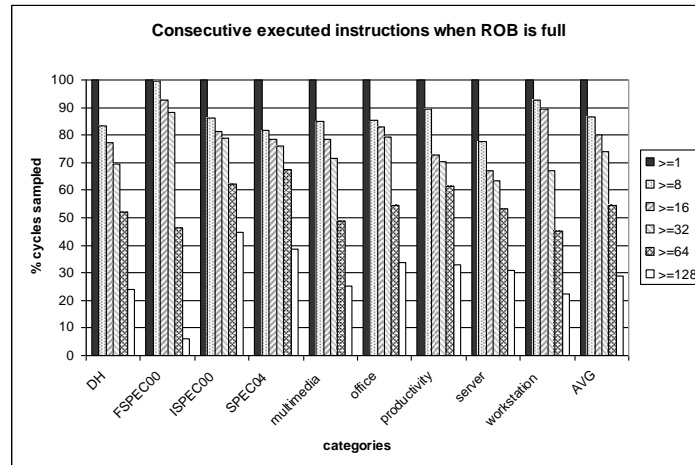
## 4 ROB Implementation

Conventional *ROB* implementations are managed like FIFO queues where the entries are allocated at renaming and released at commit. However, from the time where an instruction finishes execution until the time it either commits or is squashed, the corresponding *ROB* entry is not used at all.

The proposed *ROB* is based on the observation that it is quite common to find sequences of consecutive instructions that have been executed but cannot be committed (because of an older instruction that has not finished yet). Figure 3 shows the larger pool of consecutive executed instructions found in the event of *ROB* full for the different categories. As it can be seen in Figure 3, almost 90% of the cycles that the *ROB* is full there are chunks of more than 8 consecutive completed instructions on average. Furthermore, scenarios where chunks of more than 16 or even 32 completed instructions are found represent 80% and 73% of the cycles respectively. Besides, the probability of finding chunks of consecutive completed instructions is very dependent on the application. For instance, whereas there are chunks of more than 32 executed instructions in 90% of the cycles for *FP* workloads, chunks of this size (or larger) are only found in 65% of the cycles for *server* applications.

Consecutive executed instructions can be managed as an atomic block if all of them are free of exceptions because all the branches in between have been computed (and the prediction validated to be correct). Hence, all these instructions are either in the correct path and therefore they will eventually be committed or in a wrong path

and will all be squashed. A key observation is that the information required to either commit or squash a whole chunk of instructions can be stored in a compressed manner using a small number of bits outside the *ROB*. Hence, the entries where these instructions reside can be released.



**Fig. 3.** Percentage of cycles that we can find at least one sequence of consecutive completed instructions larger than 1, 8, 16, 32, 64 and 128 (shown by the different bars) when a ROB with 256 entries is full.

Besides, it is possible to take advantage of the atomicity of these sequences of executed instructions to implement a simple early register release mechanism. Assume a logical register ( $rX$ ) that is mapped to two physical registers by two different instructions ( $pA$  and  $pB$ , in this program order) inside a given sequence of executed instructions. In this case, the first physical register ( $pA$ ) is guaranteed not to be needed by any other instruction and can be released without waiting until the second instruction commits. In other words, all physical mappings created and re-defined inside a sequence can be released once all the instructions are completed. Therefore, conversely to other previous work, this early register release is non speculative [22][5]. Figure 4 shows an example where some physical registers can be released earlier. Assuming sequence 1 is older than sequence 2 and sequence 3, instructions from sequences 2 and 3 will not release any register until instructions from segment 1 have committed in a processor with a traditional ROB. However, if all the instructions in sequence 2 have finished and they are free of exceptions then registers shaded in the figure could be released because it is guaranteed that they won't be used any more. The same applies to sequence 3.

Pdest	Instruction	Pdest	Instruction	Pdest	Instruction
10	AX = LD(@AX)	42	AX = BX-CX	95	BX = BX-CX
20	BX = CX+AX	44	BX = LD(@AX)	1	DX = LD(@BX)
30	CX=CX+AX	47	CX=CX+AX	5	SI=SI+DI
35	DX = DX-CX	80	AX = BX-CX	9	AX = BX-CX
40	SI = LD(@AX)	85	BX = LD(@AX)	11	BX = LD(@AX)
41	DI=CX+AX	90	CX=CX+AX	14	DX=CX+AX

Sequence 1
Sequence 2
Sequence 3

**Fig. 4.** Example of physical registers that can be early released using a CROB. Pdest field shows the physical register allocated by the instruction. Instruction field shows the instruction that uses the entry.

## 5 Compressed ROB (CROB)

In this section we describe the main hardware components involved in the implementation of this technique as well as the way these structures behave in typical situations: instruction allocation, retirement, exceptions and branch mispredictions.

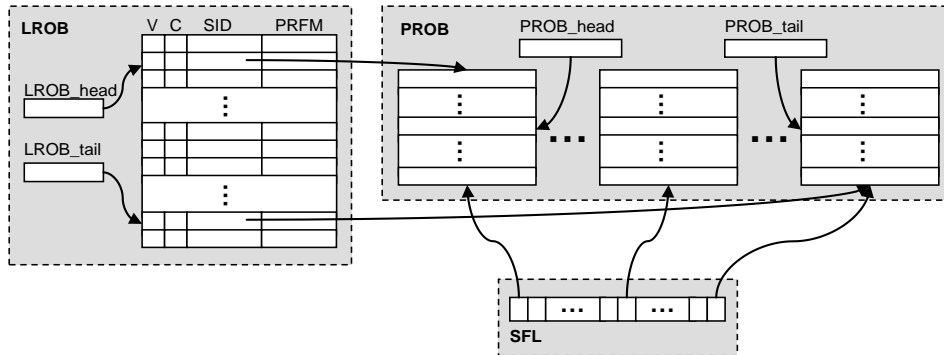
### 5.1 Hardware Components

The implementation of the compressed reorder buffer is described below and a block diagram is shown in Figure 5.

#### Physical Reorder Buffer (PROB)

This structure is an extension of a typical *Reorder Buffer (ROB)*—a FIFO structure implemented as a circular buffer with a head and a tail pointer. The *PROB* is divided into  $N$  equal segments ( $N > 1$ ), each one having  $W$  entries. Moreover, the *PROB* has a head and tail pointer that point to the oldest and youngest instruction in-flight respectively.

Conversely to what happen with other processor out-of-order designs like the processors based on the Alpha ISA, the x86 architectures like the Intel processors deal with complex CISC operations (macro-operations) that are difficult to manipulate by an out-of-order engine. Therefore, as commented in section 3, the front-end cracks these macro-operations into as many micro-operations as needed in order to keep the semantics of the original CISC operation. These micro-operations (aka uops) may comprise one destination operand and two sources as conventional RISC instruction set architectures. This translation of macro-operations into uops has significant implications in the design of common *ROBs* and therefore in the design of the *PROB*. As an example, *PROB* entries are allocated per uop so that a macro-operation may comprise multiple *PROB* entries. Moreover, since the processor must provide precise exceptions at the macro-operation level it is not possible to commit any *ROB* entry belonging to a macro-operation until all uops comprising the x86 instruction have executed and they are free of exceptions. Once all uops of the macro-operation have



**Fig. 5.** CROB structure: Logical Reorder Buffer, Physical Reorder Buffer and Segment Free List.

correctly completed execution the *PROB* entries of the macro-operation can be retired in order in as many cycles as needed. Since we cannot reclaim any *PROB* entry until the whole macro-operation has completed we could only execute macro-operations with a number of uops lower or equal than the number of *PROB* entries. However, the definition of the x86 instruction set architecture solves this problem by allowing very complex macro-operations like string moves that may imply hundreds of uops to be partially committed. Therefore, interruptions that occur while these macro-operations are executed could be served by using an architectural state as if the macro-operation would have been half executed.

Every uop stores a *PROB* entry at the renaming stage including the following fields:

- Physical source register names (*PSRC1*, *PSRC2*) are the identifiers of the physical registers where the two source operands reside.
- Physical destination-register name (*PDST*) is the identifier of the physical register where the value produced by the uop (if any) will be stored.
- Logical destination-register name (*LDST*).
- Previous mapping of logical destination-register (*LASTMAP*) shows the physical register identifier where the previous instance of the same logical destination register in program order resides.
- Exception and control information. This information involves among other control bits: a bitmask describing the exception that the instruction should fire at commit time if any; bits to identify the beginning and the end of the macro-operation in order to perform the commit as commented above; and the outcome of the branch predictor in case of branches.

### Logical Reorder Buffer (LROB)

The *LROB* must have a number of entries greater than the number of *PROB* segments. The goal of this structure is to keep the order in which the *PROB* segments were allocated. Thus, a new *LROB* entry is allocated every time a new *PROB* segment is allocated or reused. Note that the number of entries in this structure defines the maximum number of instructions the processor can have in-flight. This maximum

number of in-flight instructions is equal to the number of *LROB* entries multiplied by the *PROB* segment size.

The *LROB* is a FIFO structure managed as a circular buffer. Apart from the *LROB* head and tail pointers (*LROB\_head* and *LROB\_tail*), the *LROB* manages the *PROB* head and tail pointers (*PROB\_head* and *PROB\_tail*), as will be described later. Each *LROB* entry requires the following fields:

- *Valid bit (V)* is set when the entry is in use.
- *Compressed bit (C)* indicates if the segment pointed by the *LROB* entry has been compressed.
- *PROB segment identifier (SID)* points to a *PROB* segment in the *PROB*. This field binds each *LROB* entry to a *PROB* segment.
- *Physical register free mask (PRFM)*. The *PRFM* is a list of physical register identifiers and logical register identifiers. It indicates which registers should be de-allocated when the *PROB* segment pointed to by *SID* is committed. For those register identifiers that were not allocated by instructions in the segment the logical register they were mapping is also stored. This information is needed to rollback in the event of an exception as explained in the exception handling subsection.

### Segment Free List (SFL)

The *SFL* indicates which *PROB* segments are currently unused. It is used every time the processor needs to allocate a new *PROB* segment and every time a *PROB* segment is released (de-allocated). It is implemented as a bit mask with as many bits as *PROB* segments (the *i*-th bit is set when the *i*-th *PROB* segment is available).

## 5.2 Hardware Behavior

The *CROB* is based on a segmented *ROB* structure as commented before where segments are allocated and released on demand. In this section we present the main actions that take place in the *CROB* during the execution of an application. It works very similar to a conventional *ROB*.

### Segment Allocation

When new instructions are allocated<sup>1</sup> new entries in the *PROB* are needed. If the current *PROB* segment holding the youngest instructions (pointed by *LROB\_tail*) has room for the new instructions, the allocation procedure involves simply incrementing the *PROB\_tail* pointer. If the current *PROB* segment is full, a free *PROB* segment is obtained from the *SFL*. If there are no *PROB* segments available the allocation stalls; otherwise a *LROB* entry is allocated by increasing *LROB\_tail*. The *SID* field of the new *LROB* entry points to the new *PROB* segment returned by *SFL* and the *PRFM* field is reset. The *PROB* segment is then removed from the *SFL*. When the *PROB\_tail*

---

<sup>1</sup> We use the term *instruction allocation* to refer to the action of dispatching a decoded and renamed instruction to the *ROB* and the issue queue.

reaches the end of a segment, the *LROB* is checked to find the next *PROB* segment in logical sequence and *PROB\_tail* is updated to point to the first entry of that segment.

Every time a new instruction is allocated, the physical register free mask (*PRFM*) of the associated *LROB* entry is updated with the new instruction's *LASTMAP PROB* field. Moreover, if *LASTMAP* was not previously allocated by an instruction in the segment, the logical register is also written in the *PRFM*.

### Segment Release

When a *PROB* segment is full and all the instructions of the *PROB* segment have finished execution without any exception the segment is released and the *C* bit is set to true. Segment release implies adding the *PROB* segment identifier to the *SFL* so that it becomes available for re-use.

### Segment Retirement

Instructions from the “head” *PROB* segment (pointed to by *LROB\_head*) are committed in the conventional way if the segment has not been released: each instruction frees the register designated by *LASTMAP* and *PROB\_head* is incremented. When *PROB\_head* reaches the end of the current *PROB* segment the following actions take place: (a) the *LROB\_head* is updated to point to the next *PROB* segment in *logical sequence* (i.e. the *PROB* segment holding the oldest instructions), and the *LROB* entry is freed, (b) the *PROB\_head* is updated to point to the first entry of the new *PROB* segment “head”, and (c) the released *PROB* segment is added to the *SFL*. If the whole *PROB* segment can be committed, the following actions must be taken: (a) the *PRFM* is walked in order to de-allocate the physical registers released by the instructions in the segment, (b) the *LROB\_head* is updated as above, (c) the *PROB\_head* is updated as above, (d) the *PROB* segment is added to the *SFL*.

### Branch Misprediction Recovery

During allocation of a conditional branch instruction, a copy of the *PRFM* entry of the current segment is saved along with a copy of the processor's Free List. If the branch is mispredicted the Free List and the *PRFM* are copied back in order to restore the correct logical to physical register mappings.

### Exceptions handling

Exceptions rarely occur so slow mechanisms can be implemented with no impact on performance. Hence, *CROB* relies on traversing the *ROB* to handle exceptions. *ROB* can be traversed either backwards (rollback) or forward (having an architectural *RAT*). We chose rollback because an architectural *RAT* would have to be updated by every committed instruction; on the other hand, rollback only requires updating the speculative *RAT* at the event of an exception. When an exception occurs, rollback is performed as in a conventional *ROB* for those segments that have not been compressed. For the rest of segments *LROB* is accessed. *LROB* stores the previous mappings of the logical registers overwritten in the segment (*PRFM*). Note that the compression algorithm guarantees that exceptions never occur inside a compressed segment. Therefore, when a compressed segment is reached by the rollback mechanism, this segment is atomically traversed. Thus, *LROB* must only store the

mappings that registers overwritten in the segment had before entering the segment. Thus, the number of entries in that structure is the minimum between the number of logical registers and the size of the *PROB* segments. In our case we need between 4 and 16 entries.

### ROB wrapping

Traditional reorder buffers are implemented using a circular buffer structure. However, in the face of segment compression it is impossible to maintain correct segment order using the *LROB* structure if we allow each segment to be used as a circular buffer. The reason is that in such a case we could have two uncompressed logical segments mapped to the same physical segment. Therefore, in our proposed implementation a *PROB* segment cannot be used as a circular buffer. Assume entries in a *PROB* segment are numbered from 0 to  $W-1$ . Our design requires that  $PROB\_head \geq PROB\_tail$  if they both point inside the same *PROB* segment. This means that in some cases we may not be able to allocate instructions although there are free entries in the *CROB*. Assume the case where all  $N$  *PROB* segments are in use (with logical order of segments same as physical). If  $PROB\_head$  is in the last entry ( $W-1$ ) of last segment ( $N-1$ ), it cannot wrap around to *PROB* segment 0, even if it has free entries in it (since  $PROB\_tail$  points to this segment). Nevertheless, the number of entries unused by this reason is minimal.

## 6 Early Register Release

Early register release can be easily included into the segment release mechanism. It is an optional feature that enhances performance but it is not necessary for the correct operation of the *Compressed Reorder Buffer*. Note that our early register release proposal only releases registers that are guaranteed not to be needed any more. Therefore, neither an extra recovery mechanism is required nor costly (in terms of power and complexity) extra speculation is incurred. We present two different alternatives to implement the early register release mechanism.

### Option A. Early Release via the *PROB*

In this implementation the *PROB* has an extra one-bit field (*EARLY*) for each entry that indicates whether the physical register identified by *LASTMAP* should be released when the *PROB* segment is released (*i.e.*, early) or should wait until the segment commits. Moreover, all renaming table entries are augmented with a single-bit field (*WRITTEN*) that indicates whether the current mapping has been established by an instruction in the current segment (the *WRITTEN* fields of all renaming table entries are set to zero whenever we allocate a new segment). When a new instruction re-defines a logical register, the corresponding *WRITTEN* bit (before processing the instruction) is copied to the *EARLY* bit of the *PROB* entry of this instruction. At segment release time, the *PROB* segment is scanned and all entries with the *EARLY* bit set will update the processor's Free List to release the corresponding registers.

### Option B. Early Release via the *LROB*

This implementation uses the same structure of renaming table as the above scheme but the early release information is kept in the *LROB* instead. Each *LROB* entry is extended with an *Early Release Mask* (ERM) field (with as many entries as number of slots in the segment). The *ERM* works similarly to the *PRFM*, but specifying which physical registers are to be released early (at segment release as opposed to segment commit time). As above, an instruction gets the early release information from the renaming table. If the register can be released early (*WRITTEN* is set), then the corresponding bit in *ERM* is set. If the register cannot be released early, then the corresponding bit in *PRFM* is set. At segment release time (a) the *ERM* is walked and all indicated registers are released, and (b) the *ERM* is reset. At commit time the *Free List* is updated by walking the *PRFM* and the *ERM*.

## 7 Experimental Results

This section first describes our simulation methodology and then presents the evaluation of the CROB and the early register release. Since CROB can be implemented with or without the early register release proposal we evaluate both separately. We present results for both an ideal configuration to show the potential and a realistic configuration.

Table 1. Benchmarks.

Category	Description/Examples
<b>DH</b>	Digital Home algorithms
<b>FSPEC00</b>	Floating Point benchmarks from SPEC2K
<b>ISPEC00</b>	Integer benchmarks from SPEC2K
<b>Multimedia</b>	Mpeg, speech recognition
<b>Office</b>	Power Point, Excel
<b>Productivity</b>	Sysmarks2K
<b>Server</b>	TPC traces
<b>Workstation</b>	CAD, rendering
<b>Miscellanea</b>	Games and matrix algorithms

### 7.1 Simulation Methodology

The experiments have been conducted by using an in-house simulator that models the microarchitecture described in Section 3. The simulator is trace-driven but traces

hold enough information to faithfully simulate wrong path execution. In a nutshell, the trace contains the state of the memory and the registers before the captured sequence of code started. It also includes a dictionary with the static instructions needed to execute the trace and the number of dynamic instructions retired before we found an interrupt (DMA, etc) in the captured trace. Then, the simulator uses the memory state to execute the instruction pointed by the IP pointer, executes it and updates the memory and register state accordingly. Instructions in wrong path are executed as any other instruction but the register and memory state are eventually recovered. Finally, interrupts are triggered as soon as the number of retired instructions matches with the one specified in the trace file. Our pool of benchmarks comprises 72 traces classified in 9 main categories (8 traces per category) based on their characteristics as shown in Table 1. The processor baseline configuration is described in Table 2.

## 7.2 Potential Benefit of CROB

The size of a segment is a trade-off between performance improvement and hardware overhead. Enlarging the segment size reduces the CROB cost because the number of ROB slices is smaller. By contrast, implementing small ROB segments increases the chances of compressing a segment and therefore, achieves higher performance.

In this section, the potential of compressing the ROB is first studied. The goal is to evaluate the impact of the ROB compression for different number of segments and ROB sizes. For this initial potential study, an unbounded register file, issue queue and memory order buffer are assumed, to isolate the effect of ROB compression. Figure 6 shows the potential performance benefits in an architecture limited by the ROB size. Configurations with 128, 256, 512 and 1024 ROB entries are considered. *No CROB* shows the speedup obtained by just enlarging a traditional ROB. *CROB 4 entries*, *8 entries* and *16 entries* show the speedups of a CROB with segments of 4, 8 and 16 entries respectively. All speedups are relative to a conventional ROB with 128 entries.

All the workloads benefit from enlarging the ROB as it can be seen in Figure 6. However, two main groups of categories can be differentiated. The first group comprises *DH*, *ISPEC00*, *multimedia*, *office*, *productivity* and *server*. The performance curve for these groups flattens out at around 1024 ROB entries. There are two main reasons why these applications do not benefit from larger ROB: on the one hand, some programs such as ISPEC contain a significant number of branch mispredictions; on the other hand, some benchmarks such as server are extremely memory bounded. These applications have many chains of dependent instructions that consume values from loads that miss in cache. *CROB* performance is always better than the traditional ROB for all the configurations and categories as shown in Figure 6. *CROB* with 4 entries per segment typically performs about the same as a conventional ROB with twice its size. *CROB* improves performance by 20% on average for 128 ROB entries, 12% for 256 entries and 9% for 512 entries. For *FSPEC00*, the average speedup is 30% for 128 ROB entries, and 23% for 256 ROB entries. Nevertheless, implementing 4-entry segments is challenging because it may require allocating instructions in more than one segment for those cycles where more

than 4 instructions are renamed. For the remainder of this paper, we assume that instructions renamed and dispatched in a given cycle must be in the same segment; otherwise an additional cycle is needed. We have experimentally observed that the effect of this constraint on performance is negligible for 8-entry and 16-entry segments.

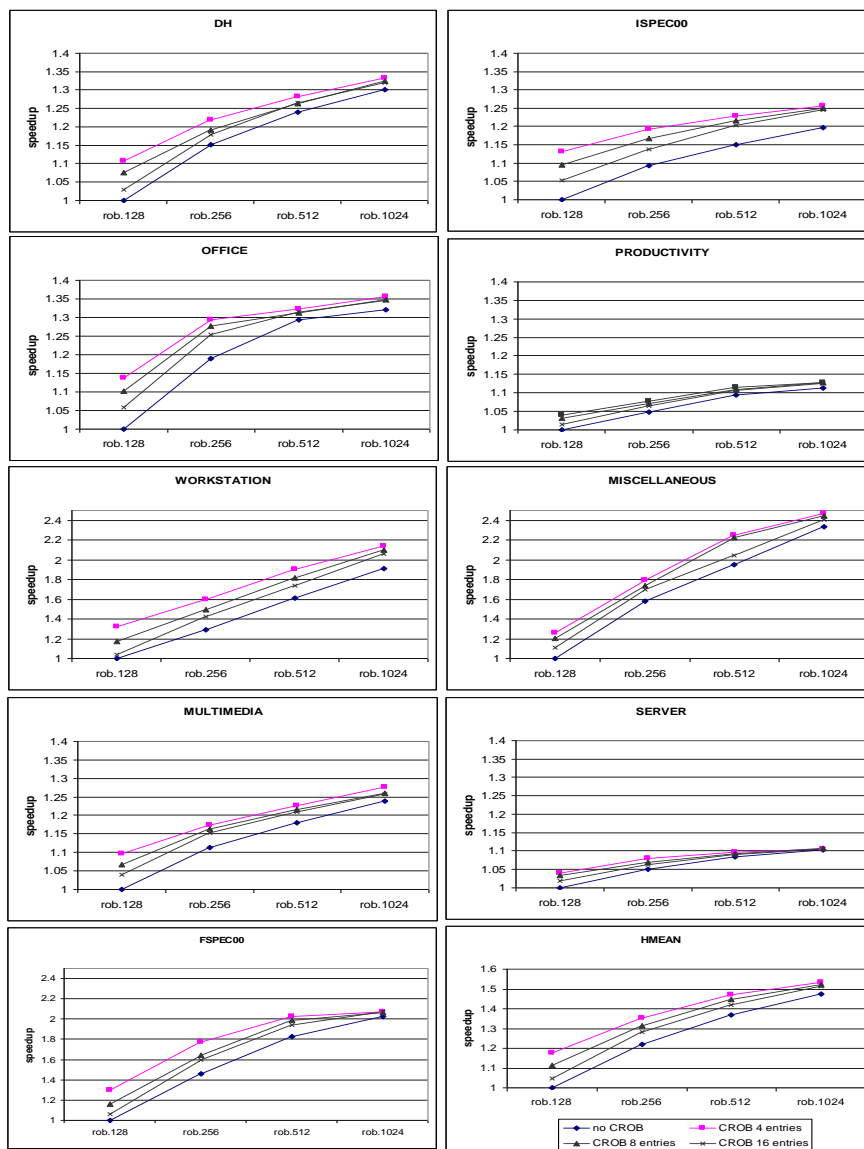
**Table 2.** Baseline processor configuration.

Parameter	Value	Parameter	Value
Fetch width	6	Commit width	6
Misprediction pipeline length	14	ROB size	128-1024
Indirect branch predictor entries	4096	Gshare entries	32K
ITLB entries	1024	ITLB assoc.	8
Trace Cache size	32K micro-ops	Issue rate per cluster	4 + 2 for ld/st
Issue queue size per cluster	32	MOB	128
Int. registers per cluster	64-256	FP registers per cluster	64-256
SSE registers per cluster	64-256	DTLB entries	1024
DTLB assoc	8	L1 ports	1 read/ 1 write
L1 assoc	2	L1 size	32KBytes
L1 hit latency	1 cycle	L2 assoc	8
L2 size	4MB	L2 hit latency	12 cycles
Point to Point Links	2	Point to Point latency	1 cycle
Data buses (between L1 and L2)	2	Memory Latency	275 cycles

It can be seen that the larger the segments the more difficult to find candidates to be compressed, and the lower the benefit. This is explained by several reasons. On the one hand, by enlarging the segments, their number is reduced, as well as the

probability of having a segment with all its instructions executed. Moreover, free entries of the oldest segment (the one pointed by `LROB_head`) cannot be reused until the whole segment is free. In spite of this, the benefits obtained by CROB with larger segment sizes are still very important. The average speedup for 8- and 16-entry segments is 10% and 5% respectively for a ROB with 128 entries. The average speedup is 8.5% and 6.5% respectively for a 256-entry ROB. Categories such as *miscellanea* get very high speedups even with large segments. These benchmarks achieve 20% and 10% average speedup for 8 entries and 16 entries per segment respectively and a ROB of 128 entries (18% and 14% respectively for 256 ROB size).

Finally, it can be seen that the potential of the *CROB* is very limited for configurations with ROB size greater than 1024 in all the categories but *workstation* because the ROB is not a bottleneck any more.



**Fig. 6.** CROB speedups for 4, 8 and 16 entries per segment and ROB size from 128 to 1024. Note that the Y-scale differs among charts.

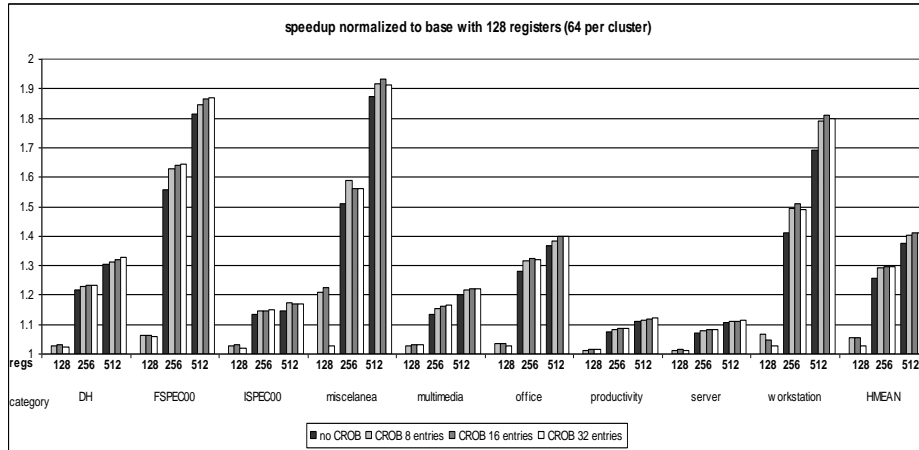


Fig. 7. Potential study of early register release based on CROB.

### 7.3 Early Register Release

Another advantage of the *CROB* implementation is that it enables techniques to perform non-speculative early register release, as described in Section 4.

In this section the potential performance improvements of early register release based on *CROB* is evaluated. For this potential study, a processor with unbounded issue queue, memory order buffer and ROB is assumed to isolate the impact of this register release by using different segment sizes. Note that for the experiments of this section, only registers are released early, as opposed to the whole ROB segment. Figure 7 shows the average performance for the different categories normalized to the performance obtained without early register release for a processor with 128 registers. No *CROB* stands for the baseline processor whereas *CROB 8*, *16* and *32* entries refer to the early register release with segments of *8*, *16* and *32* entries respectively. Configurations with *128*, *256* and *512* registers are considered.

We can observe in Figure 7 that the configurations with either 8 or 16 entries per segment have the highest potential. For 128 registers, early register release improves performance by around 5% for all categories but *miscelanea*, for which the average speedup is 20%. On average, the performance benefits are 5% for 128 registers, 4% for 256 and 2% for 512 registers.

The moderate benefits of early register release are due to the fact that it needs large segments in order to increase the number of physical registers that can be early released. However, the larger the segment size, the lower the probability of finding a segment with all its instructions executed. Therefore, the segment size is a trade-off between the probability of finding a fully executed segment and the probability of finding registers eligible to be released inside the segment.

Finally, the number of available physical registers is an important factor in the effectiveness of this technique. As it can be seen in Figure 7, the average performance improvement for 128 registers in *Workstation with 16 entries* is 5% whereas for 256 it grows up to 7%, and goes down to 1% for 512 registers. The reason is that limiting

the number of physical registers limits the number of in-flight instructions which in turn reduces the number of segments full of executed instructions. On the other hand, when the number of physical registers is very large, the register file is not a limiting factor any more and the performance benefits are very small. Intermediate design points where registers are neither scarce nor abundant are the scenario where this technique performs the best.

#### 7.4 Putting It All Together

The previous section showed an analysis of the potential performance of CROB. The study assumed that physical registers, issue queues and memory order buffer were large enough to satisfy the applications requirements. In this section, we evaluate the behavior of CROB under a realistic processor configuration where memory order buffer, physical registers and issue queues are also constraining performance. We assume a clustered processor with the configuration shown in Table 2 with 256 registers (128 per cluster).

Figure 8 shows the performance improvement for a realistic configuration with 128, 256 and 512 ROB entries (both compressed and non-compressed). Each bar shows the speedup with respect to a baseline with a non-compressed ROB of 128 entries. For each configuration, results are presented for the baseline, non-compressed ROB (*no CROB*) and for the proposed compressed ROB with different segments sizes (*CROB.16*, *CROB.8* and *CROB.4* stand for CROB with 16, 8 and 4 entries per segment respectively).

We can first observe that increasing the ROB size from 256 to 512 entries results in very small performance improvement on average (2.5%). However, there are some applications whose performance is more sensitive to the ROB size, such *FSPEC00*. For these applications, increasing the ROB size from 256 to 512 entries provides a performance increase of 7%.

When CROB is enabled, the performance of a 256-entry CROB with 8 entries per segment is the same as that of a 512-entry non-compressed ROB for all the categories (including *FSPEC00*). For 128-entry ROB, CROB clearly outperforms the conventional one, providing a speedup of 15% and 10% with 4 entries and 8 entries per segment respectively. Besides, we can see average performance improvements of up to 28% and 16% in categories with high ROB demand such as *FSPEC00*. Benefits are reduced to 5% improvement with 16 entries per segment, although for some benchmarks such as *workstation* they are still significant (12% average speedup).

We have also evaluated the benefits of the proposed early register release scheme when implemented on top of ROB compression. We found that the gain across all benchmarks was negligible for a 256 ROB size, and it achieved a 1.5% average improvement for *FSPEC00 benchmark suite*, where a very high register pressure is observed. The reason is that 256 registers are usually enough to allocate the destination registers of all in-flight instructions (up to 256). Note that the maximum number of registers required at any point in time would be given by the ROB size, plus the number of architectural registers, plus some extra registers due to replicated values in clusters.

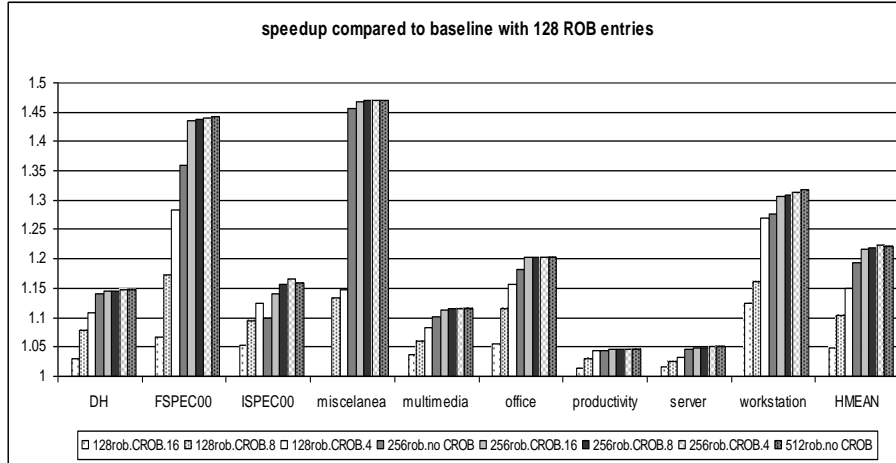


Fig. 8. Speedup of CROB with 16, 8 and 4 entries per segment.

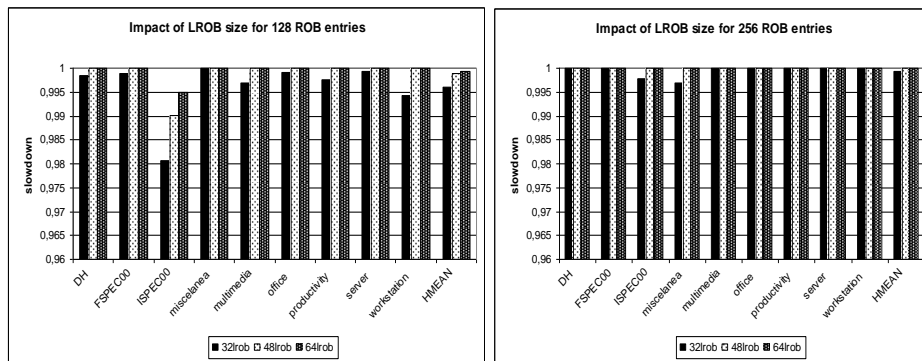
The number of *LROB* entries required to reach similar performance to an unbounded *LROB* has also been explored. For this evaluation we have chosen a configuration with 128 ROB entries and another with 256 ROB entries. Both configurations have been evaluated assuming 16 segments (8 entries per segment for the configuration of 128 and 16 entries per segment for the one with 256).

Figure 9 shows the performance obtained when using 32, 48 and 64 *LROB* entries normalized to the performance when the *LROB* is unbounded. As it can be seen, *ISPEC* is the most affected category when the *LROB* size is limited. However, this category is only affected when the ROB has 128 entries. In general, 48 *LROB* entries are enough to obtain the maximum benefit from the *CROB* in both configurations. However, this *LROB* size may be lowered to 32 entries without getting important performance drops (less than 0.5% on average).

Finally, the ROB structure represents a significant part of the area and power for microprocessors with large instructions windows. Parts of the ROB required multi-ported random access – not FIFO (e.g. completion flag, exception flags, target address of branches, instruction IP, etc.). As an example of the benefits of the *CROB* compared to a conventional ROB, we compare in Table 3 the area, access latency and energy per access of a 512-entry ROB and a *CROB* with 256 entries in the *PROB* and 48 entries in the *LROB*. Both configurations have about the same performance and are especially useful for applications that require large ROBs like the *FSPEC00* category. The estimates have been computed using *CACTI* v3.0 [21] with some modifications to allow the modeling of such structures. As it can be seen, *CROB* has a faster access time (which may not be useful if the ROB access is not in the critical path). However, an area reduction of 27% constitutes a very significant decrease in the area devoted to the ROB. Furthermore, the energy per ROB access is decreased by 19%.

**Table 3.** CROB (PROB + LROB) vs banked conventional ROB

<b>Latency</b>	30% faster
<b>Area</b>	27% less area
<b>Energy per access</b>	16% energy reduction

**Fig. 9.** Impact of the LROB size in the processor performance. Both ROB configurations (128 and 256 ROB size) implement 16 segments.

## 8 Conclusions

Enlarging the ROB enables the exploitation of further ILP by allowing a larger number of in-flight instructions. However, this structure does not scale with ease and, increasing its size may negatively affect the processor power dissipation, area and cycle time. In this paper, we propose a new ROB design named CROB where each ROB segment is released as soon as all the instructions stored in it behave as an atomic unit. The proposed scheme does not need any type of speculation nor replay mechanism. Conversely to other previous proposals, CROB does not increase the number of instructions to be re-executed in the event of either branch mispredictions or exceptions, and thus it does not incur in this extra energy cost.

The potential study of CROB has shown an important speedup of 20% on average for a ROB of 128 entries and 12% for a ROB size of 256. For some benchmark categories such as *FSPEC00*, the speedup reaches up to 30% and 23% respectively. These statistics show what could be achieved by using smart techniques to implement very large register files, memory order buffers and issue queues, as can be found elsewhere in the literature. For a realistic configuration with a conventional register file, memory order buffer and issue queue, the benefits are somewhat lower but still very significant (15% on average and 28% for *FSPEC00*). Finally, the early register

release has shown an average potential speedup of 5% and up to 20% for *miscellanea* workloads.

## 9 Acknowledgments

This work has been partially supported by the Spanish Ministry of Education and Science under grants TIN2004-03702 and TIN2007-61763 and Feder Funds.

## 10 References

1. Smith J., Pleszkun A.R. Implementing precise interrupts in pipelined processors. IEEE Transactions on Computers, 37(5):562-573 May 1988.
2. Hinton G., Sager D., Upton M., Boggs D., Carmean D., Kyker A., Roussel P. The Microarchitecture of the Pentium® 4 Processor. Intel Technology Journal, February 2001.
3. Martinez J.F., Renau J., Huang M.C., Prvulovic M., Torrellas J. Cherry: Checkpointed Early Recycling in Out-of-order Microprocessors. In Proceedings of International Symposium on Microarchitecture. November, 2002.
4. Akkary H., Rajwar R., Srinivasan S.T. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In Proceedings of International Symposium on Microarchitecture. 423-434. December, 2003
5. Cristal A., Santana O., Valero M. Toward Kilo-instruction Processors. ACM Transactions on Architecture and Code Optimization. Vol. 1 N° 4, 389-417 December 2004.
6. Canal R., Parcerisa J.M., González A. Dynamic Cluster Assignment Mechanisms. In Proceedings of International Symposium on High Performance Computer Architectures, 2000.
7. Balasubramonian R., Dwarkadas S., Albonesi D.. Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors. In proceedings of the Annual International Symposium on Computer Architecture. June 2003.
8. Baniasadi A., Moshovos A. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Schedule, Superscalar Processors. In proceedings of International Symposium on Microarchitecture. December 2000.
9. Aggarwal A., Franklin M. An Empirical Study of the Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors. In proceedings of ISPASS, 2001.
10. Palacharla S., Jouppi N.P., Smith J.E.. Complexity-effective Superscalar Processors. In proceedings of the Annual International Symposium on Computer Architecture, 210-218. June, 1997
11. Brown M. D., Stark J., Patt Y.N. Select-free instruction scheduling logic. In proceedings of International Symposium on Microarchitecture, 204-213, December, 2001.
12. Buyuktosunoglu A., Bose P., Cook P.W., Schuster S.E. Tradeoffs in Power-Efficient Issue Queue Design. In proceedings of International Conference on Parallel Architectures and Compilation Techniques. November, 2000.
13. Folegnani D., Gonzalez A. Energy-Effective Issue Logic. In Proceedings ACM/IEEE 27th Intl. Symposium Computer Architecture. pages 230–239, June 2001.
14. Fields B., Rubin S., Bodik R. Focusing Processor Policies via Critical-Path Prediction. In Proceedings 28th annual Intl. Symposium on Computer Architecture, pages 74–85, 2001.
15. Lebeck R., Li T., Rotenberg E., Koppanalil J., Patwardhan J. A Large, Fast Instruction Window for Tolerating Cache Misses. In Proceedings ACM/IEEE 29th Intl. Symposium on Computer Architecture, pages 59–70, June 2002.

16. Ponomarev D., Kucuk G., Ghose K. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In Proceedings 34th ACM/IEEE International Symposium on Microarchitecture, pages 90–101, 2001.
17. Capitanio A., Dutt N., Nicolau A. Partitioned Register Files for VLIWs: A Preliminary Analysis of Trade-offs. In Proceedings of the International Symposium on Microarchitecture. 292-300. December, 1992.
18. Wallace S., Bagherzadeh N. A Scalable Register File Architecture for Dynamically Scheduled Processors, in Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pp. 179-184, 1996.
19. Gonzalez A., Gonzalez J., Valero M. Virtual-Physical Registers. In Proceedings of International Symposium on High-Performance Computer Architectures, pages 175–184, February 1998.
20. Cruz J.-L., Gonzalez A., Valero M., Topham N. Multiple-Banked Register File Architectures. In Proceedings of International Symposium on Computer Architecture, pages 316–325, June 2000.
21. P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Research Report 2001/2, Aug. 2001.
22. Ergin O., Balkan D., Ponomarev D., Ghose K. Increasing Processor Performance Through Early Register Release. In proceedings of 22nd International Conference on Computer Design, pages 480-487, October 2004.
23. <http://www-03.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html>
24. Raasch S. E., Binkert N. L., and Reinhardt S. K. A Scalable Instruction Queue Design Using Dependence Chains. In Proceedings of 29th Annual Int'l Symp. on Computer Architecture, pp. 318-329, May 2002
25. Moshovos A. Checkpointing Alternatives for High Performance, Power-Aware Processors. In Proceedings of the IEEE Intl' Symposium on Low Power Electronic Devices and Design, Aug. 2003.