Out-of-Order Commit Processors

Adrian Cristal[†], Daniel Ortega[⋆], Josep Llosa[†] and Mateo Valero[†]

[†]Depto. de Arquitectura de Computadores, Universidad Politécnica de Cataluña
{adrian,josepll,mateo}@ac.upc.es

[⋆]Barcelona Research Office, Hewlett Packard Labs,
daniel.ortega@hp.com

Abstract

Modern out-of-order processors tolerate long latency memory operations by supporting a large number of in-flight instructions. This is particularly useful in numerical applications where branch speculation is normally not a problem and where the cache hierarchy is not capable of delivering the data soon enough. In order to support more in-flight instructions, several resources have to be up-sized, such as the Reorder Buffer (ROB), the general purpose instructions queues, the Load/Store queue and the number of physical registers in the processor. However, scaling-up the number of entries in these resources is impractical because of area, cycle time, and power consumption constraints.

In this paper we propose to increase the capacity of future processors by augmenting the number of in-flight instructions. Instead of simply up-sizing resources, we push for new and novel microarchitectural structures that achieve the same performance benefits but with a much lower need for resources. Our main contribution is a new checkpointing mechanism that is capable of keeping thousands of in-flight instructions at a practically constant cost. We also propose a queuing mechanism that takes advantage of the differences in waiting time of the instructions in the flow.

Using these two mechanisms our processor has a performance degradation of only 10% for `SPEC2000fp` *over a conventional processor requiring more than an order of magnitude additional entries in the ROB and instruction queues, and about a 200% improvement over a current processor with a similar number of entries.*

1. Introduction

The ever increasing gap between processor speed and memory speed is steadily increasing memory latencies with each new processor generation. In order to tolerate these latencies, caches and prefetching are very useful techniques, but do not solve the problem completely. Numerical applications which work with great amounts of data are specially sensitive to these scenarios. It is common place that in order to sustain high ILP under these circumstances, a higher number of in-flight instructions must be maintained. At current memory latency trends, processors cannot keep up with this growing disparity, and as a result, long latency operations are increasingly more crucial for performance.

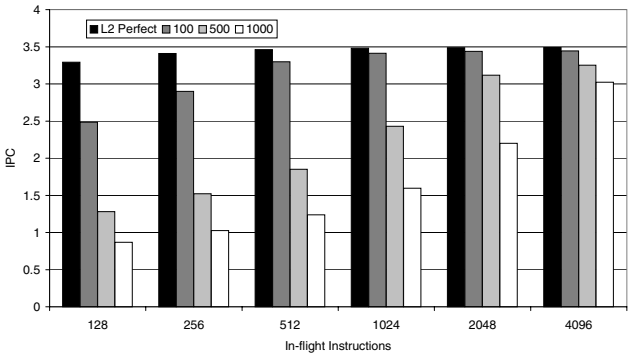

Figure 1. IPC relative to the # of in-flight insts. and the latency to memory for `SPEC2000fp` (other resources have been scaled)

In figure 1 we can see the average relation between IPC and in-flight instructions for `SPEC2000fp` applications[1]. Each number of in-flight instructions contains four bars relative to four different architectural configurations. The first one is the IPC achieved by a machine with those resources and perfect L2 cache behavior, while the other three represent different L2-main memory latencies (100 cycles, 500 cycles and 1000 cycles). From this figure we can see that increasing the number of in-flight instructions is capable of achieving nearly perfect memory behavior (this does not happen so much in integer applications due to branch speculation problems and pointer chasing references).

1 These numbers were obtained with the simulation framework explained in section 4.

Two major conclusions can be stated from this figure. First of all, future processors with 1000 cycles to main memory will severely suffer with present microarchitectural configurations. This can be noted from the difference between the first two bars of the 128 group with respect to the fourth bar, the one representing 1000 cycles to main memory. The relative difference is on the order of 3.5 times slower! The second important conclusion is that an increase in the amount of in-flight instructions allows to tolerate a higher memory latency. This is not at all new, to tolerate the 1000 cycles of a particular missing load, we must have enough instructions from which to extract ILP [13]. If the issue width of the processor is 4 instructions per cycle, this means that over 4000 instructions will be needed to execute if we want to continue full speed. All this forces the processor to have plenty of in-flight instructions in order to maintain performance.

The simple way of allowing for thousands of in-flight instructions would be to scale all the resources involved, i.e. ROB size, physical register file, general purpose instruction queues (integer and floating point ones) and load/store queue. Unfortunately, this is not at all simple, since these resources often determine the cycle time of the processor [24]. Moreover, these four resources are mingled, improving one of them will surely leave it out of the critical path, bringing another one to occupy its place as the most critical resource. Nevertheless, we believe that particular solutions to each one of them can be attacked in an orthogonal way. Several researchers do also think the same, and have focused their particular solutions on each one of these problems one at a time. For instance, in [9] we describe a technique to optimize the register file usage which can be orthogonally combined with any of the proposals of this paper.

In this paper we are going to propose solutions to two of these four important resources, the reorder buffer and the general purpose instruction queues[2]. Our proposal is based on the fact that the resources of a processor supporting a large number of in-flight instructions are underutilized, as stated in [8]. We will introduce mechanisms that effectively allow for thousands of in-flight instructions to co-exist with feasible implementations of the microarchitectural requirements of both types of resources. The other two critical resources named above, the physical register file and the load/store queue will be modeled in a pseudo-perfect way, leaving them out of the critical path in order to analyze the impact of our changes.

The first mechanism presented will focus on allowing for larger virtual ROBs in a superscalar out of order processor.

2 Load/Store queues take care of memory disambiguation by keeping instruction order, which clearly makes them totally different to normal general purpose instruction queues

The ROB itself is a critical resource as has been pointed out in [11, 23]. We call this mechanism *Out-of-Order Commit*. With the use of checkpointing we allow for the commit of instructions in an out-of-order fashion, while preserving correctness and exception preciseness by committing checkpoints in-order. The mechanism will be thouroughly explained in section 2. The second mechanism presented will explain how to implement general purpose instruction queues that allow for high ILP while being simple and not affecting cycle time. We will accomplish this by storing those instructions which are not going to be ready in a long time in a secondary buffer. This will leave important instructions the necessary resources to complete as soon as possible. Due to the insertion of this secondary buffer, we have called this mechanism *Slow Lane Instruction Queuing*. This mechanism is coupled with the previous one for presentation purposes, but we believe that independent implementations of both of them are feasible. Other mechanisms that try to achieve similar results have already been presented [18, 7]. With these two mechanisms our processor outperforms by a factor of 3 a current processor with a similar amount of hardware devoted to these tasks.

The rest of this paper is organized as follows. Section 2 describes our out-of-order commit mechanism. We deal with instruction queues in section 3. In section 4 we explain our simulation framework and the different experiments run to state our results. In section 6 and 5 we discuss the related work and the links with our present work. We conclude the paper in section 7.

2. Out-of-Order Commit

After an instruction is fetched and decoded in a superscalar out-of-order processor, it is inserted in both its corresponding instruction queue and in the re-order buffer (ROB). The purposes of the ROB are multiple. First of all, it allows for precise interrupts by implementing in-order commit. In addition, it is mingled with the recovery mechanisms associated with some kinds of speculation such as branch or load speculation. The ROB controls exactly when stores may change the memory state and thus the machine state. It also correctly frees the physical registers when they are no longer in use.

Basically a ROB can be understood as the microarchitectural mechanism that keeps a history window of all in-flight instructions, allowing for the precise recovery of the program state at any of those in-flight instructions. As the latency of memory operations increases, it is needed to support a larger number of in-flight instructions to hide this access latency and achieve high performance. In this context, a centralized structure like the ROB becomes a problem, since scaling-up the number of entries in this structure is impractical, mainly due to cycle time limitations.

In this paper, we propose to replace a normal ROB structure with a mechanism oriented at making checkpoints at specific instructions of the code. Our checkpoints are very similar to the checkpoints taken by branch speculation mechanism. A checkpoint can be thought of as a snapshot of the state of the machine, which allows us to recover execution at that point.

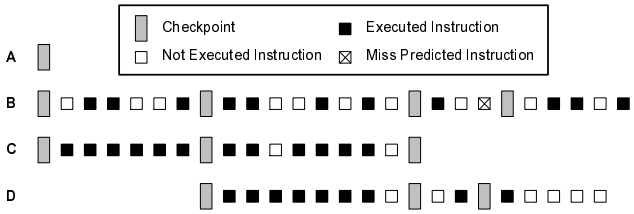

Figure 2. Checkpointing process

Figure 2 presents the checkpointing process. In timeline A we can see that there always exists at least one checkpoint in the system. The processor will bring and issue following instructions and at interesting locations it will take new checkpoints. If a particular instruction is mis-speculated (timeline B) the processor rolls back to the previous checkpoint and resumes execution from there. When all instructions between the last two checkpoints have executed (C), the last checkpoint is eliminated and its resources are freed (D).

CAM Register Mapping

Physical	1	2	3	4	5	6	7	…
Logical	3	4	2	1	8	9	7	…
Valid	1	1	1	1	0	0	0	…
Future Free	0	0	0	0	0	0	0	…
Free List	0	0	0	0	1	1	1	…

Figure 3. Extension to the CAM Register Mapping

In figure 3 we see an example of how our Register Mapping works. Our register mapping structure follows the CAM scheme such as in the Alpha 21264 [17] and the HAL Sparc [4]. This figure shows the typical CAM structure of a renaming mechanism plus another bit per entry which we call *Future Free* bit. Besides we can see in figure 3 that there also exists a *Register Free List* from where free registers are taken[3]. From this diagram we can see that at the present moment only four physical registers are mapped, i.e. the ones with the valid bit set to one.

Let us assume that in this specific moment we save a checkpoint. As in a normal branch speculation mechanism, we are forced to save the valid bits but not the logical mappings, since they are not going to change until these registers are freed and used for a new instruction. Our mechanism also needs to save the current *Future Free* bits y and reset them to to compute the new *Future Free*, which contain the information needed to free registers, as we will describe shortly. Therefore, the cost of a checkpoint in our mechanism can be computed as the number of physical register times two bits per register, which is very simple indeed.

Checkpoint

R1 = R2 + R3

Ph5 Ph3 Ph1

CAM Register Mapping

Physical	1	2	3	4	5	6	7	…
Logical	3	4	2	1	1	9	7	…
Valid	1	1	1	0	1	0	0	…
Future Free	0	0	0	1	0	0	0	…
Free List	0	0	0	0	0	1	1	…

Figure 4. State of our CAM Register Mapping after a non-checkpointed instruction is decoded

In figure 4 we see the state of this CAM register mapping after a new instruction is decoded. This new instruction (a simple add) needs a free register which is taken from the free list (physical number 5) and therefore changes the valid bit of physical 5 from 0 to 1 y the corresponding free list bit. As we have no ROB structure, someone must take care of the freeing of physical registers. We do this by setting the *Future Free* bit of entry 4 (previously mapped to logical 1) to 1. The *Future Free* bits capture which registers need to be freed when the next checkpoint is committed. When the next instruction with logical destination register number 1 is decoded, the state of our structures is the one shown in figure 5. It can be noted here that there are two registers mapped with logical 1 which will subsequently be freed at the same time.

Let us suppose that a series of instructions get decoded, and after a last instruction, we decide to take another checkpoint. This is shown in figure 6. Notice that logical register 1 which was previously mapped to physical 4 and 5 is currently mapped to physical 6, as noted by the valid bit. Log-

3 In this paper we are assuming that the free list is implemented with a bit per physical register

Checkpoint

…

R1 = R4 + R1

Ph6 Ph2 Ph5

CAM Register Mapping

Physical	1	2	3	4	5	6	7	…
Logical	3	4	2	1	1	1	7	…
Valid	1	1	1	0	0	1	0	…
Future Free	0	0	0	1	1	0	0	…
Free List	0	0	0	0	0	0	1	…

Figure 5. State of our CAM Register Mapping after decoding a second instruction

ical 4 is mapped to physical 7 which implies a change in the valid bit of two entries (physical 2, the previous mapping for logical 4, and physical 7) plus another change in the Future Free bit[4]. Then a checkpoint is taken by storing the *Valid* and the *Future Free* bits in our checkpoint table. After the checkpoint is taken, all *Future Free* bits are cleared.

Checkpoint

…

R4 = R1 + R3

Ph7 Ph6 Ph1

CAM Register Mapping

Physical	1	2	3	4	5	6	7	…
Logical	3	4	2	1	1	1	4	…
Valid	1	0	1	0	0	1	1	…
Future Free	0	1	0	1	1	0	0	…
Free List	0	0	0	0	0	0	0	…

Figure 6. State of our CAM Register Mapping just prior to another checkpoint

With this checkpoint mechanism we can effectively execute in an out-of-order fashion without a ROB like structure. When instructions arrive at decode phase, a renaming mechanism like the one explained above takes place. Every instruction is associated to the last checkpoint prior to the instruction. The instruction will carry throughout execution the index to the checkpoint table where its checkpoint lives. The checkpoint also remembers how many instructions are associated to it in a counter. When an instruction finishes it uses this index to decrease this counter. When this counter arrives to zero and this particular checkpoint has no previous checkpoints, we consider that this checkpoint has committed and modify the state of the machine to assert this point (as we will explain shortly).

If by any chance this instruction should except or needed recovery mechanism, its checkpoint allows the hardware to restore the state of the machine to the previous checkpointed instruction and resume after that. Of course, in case of an exception, the execution from the prior instruction to the excepting one should be done in a stricter sense. In this second pass, the excepting instruction should be checkpointed, leaving the processor in a precise state for which the operating system could follow.

Freeing Physical Registers A physical register is normally freed at commit phase of the following instruction that defines the same logical register as the one to which this physical register is mapped, since at this exact moment the hardware is sure that all instructions that consumed this register must have committed already. In this sense, when an instruction defines a particular register, it will have to keep the previous mapping of this register so as to free it at commit phase.

In our mechanism this is handled by the *Future Free* bits. These bits record which registers need to be freed between checkpoints. When a particular checkpoint commits, it uses these bits to free the registers associated to its history window[5]. Our mechanism increases the lifetime of these registers, since in a normal ROB mechanism would be freed beforehand, by their re-defining instructions. In our mechanism, they must wait until the following checkpoint gets committed.

Committing Store Instructions Stores must wait until commit stage to send their data to memory. This is necessary in order to allow a correct recovery of the architectural state in case of an exception or a misspeculation. In a normal microarchitecture the data is stored in the Load/Store queue until the particular store commits, when it is send to the cache. Our mechanism behaves very similarly. Data is kept in the Load/Store queue and when a checkpoint is committed, all the stores relative to the previous checkpoint are considered to be safe and are thus sent to memory. This technique has the drawback of needing a high number of entries in the Load/Store queue. We are currently working on

4 We have supposed that the intervening instructions modify other entries which are not shown in this diagrams so as to simplify the explanation

5 This mechanism does increase register lives and is used here to simplify the overall explanation. Other register managing mechanisms that allow for an earlier release of registers, thus overall decreasing register lives, can be combined with our Out-of-Order commit mechanism [11]

mechanisms for overcoming the scalability problem caused by large Load/Store queues, but this is out of the scope of this paper.

Taking Checkpoints Up to now, we have not explained when the checkpoints are taken. A particular checkpoint could be taken every n instructions ($n = 1$ would mimic the normal behavior of a ROB like microarchitecture) or after every specific type of instruction, etc. After some analysis we decided to implement a simple heuristic which works fine[6]. We have three different thresholds. The first threshold takes a checkpoint at the first branch after 64 instructions. We select branches as good places to take checkpoints so as to minimize the work done after branch mis-speculation. In case this threshold is never reached (potentially there could be hundreds of instructions with no branches) we incorporated another threshold which explicitly takes a checkpoint after 512 instructions at whatever instruction appears. Finally we have another threshold which forces a checkpoint after 64 stores. As store entries in the Load/Store queue do not get freed until the checkpoint commits, we must not associate too many stores to each threshold to prevent deadlock. Obviously, as there must always exist a checkpoint for our mechanism to work, in case of total flush of the pipeline and the arrival of a new instruction, a checkpoint will be set prior to it.

3. Slow Lane Instruction Queuing

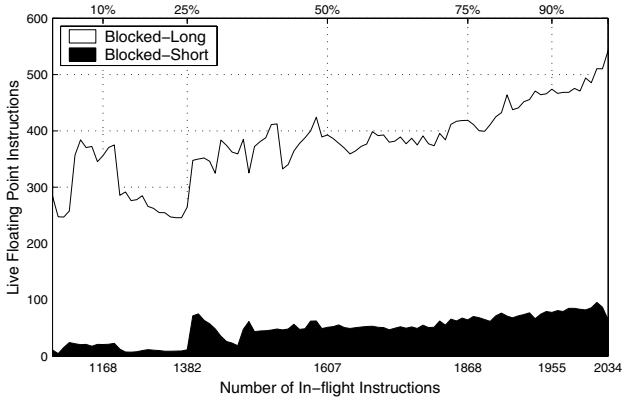

Figure 7. Distribution of live instructions with respect to the amount of in-flight instructions (assuming parameters from table 1 with 2048 ROB entries and 500 cycles to main memory)

6 In future work we expect to analyze a whole set of different strategies as to when checkpoints should be taken depending on performance or power goals

In figure 7 we can see the accumulative distribution of live floating point instructions with respect to the amount of total in-flight instructions. We consider live those instructions which are yet to be issued. In order to achieve this figure, we averaged the results from all `SPEC2000fp` benchmarks. Each cycle we compute the total amount of in-flight instructions (namely the number of instructions in the ROB) and also the amount of live instructions. We also compute the amount of cycles each of these situations happened and showed this relative distribution with percentiles. For example, in figure 7 we can see that 25% of the time the ROB had less than 1382 instructions, 50% of the time less than 1607 instructions, etc. Besides, this figure divides instructions among short latency instructions and long latency instructions, the latter formed by loads that miss in L2 and any instruction dependent on it or on its dependents. Notice that a lot of in-flight instructions (appr. 70%-75% instructions) have finished but can not commit, consuming entries in the ROB. The mechanism presented in the previous section benefits from this by releasing resources that otherwise would have been locked up for a long time.

On average, the amount of live instructions is much smaller than the amount of in-flight instructions. From this figure we can state that, using approximately 500 entries in the floating point instruction queue, we can cope with over 95% of the scenarios we are going to face. The main observation here is that, in a processor capable of supporting 2K in-flight instructions, a 512 entry instruction queue is needed, which is definitely going to affect cycle time [24]. This situation could become worse when having a larger number of in-flight instructions. Recall that our mechanisms are pushing for many more instructions. Fortunately, not all instructions behave the same way.

Some instructions take a very long time to even get issued for execution. Maintaining these instructions in the instruction queues just takes away issue slots from other instructions that will be executed more quickly. Several papers have pointed out this same point, such as [18] and [7], and have definitely profited from it. In section 6 we will discuss the main differences between our mechanism and these two previously published papers.

The mechanism we have devised consists on first of all detecting those instructions which will take a very long time to get issued for execution. Once detected, our mechanism move them to a secondary buffer where they would stay until there is any need for them to return to their respective instruction queue. This movement is what has made us denominate this mechanism *Slow Lane Instruction Queuing (SLIQ)*.

Several mechanisms have tried to speculate on the importance of instructions and either they have failed to produce accurate results or they are extremely complex and with very high costs. This is so because deciding if an in-

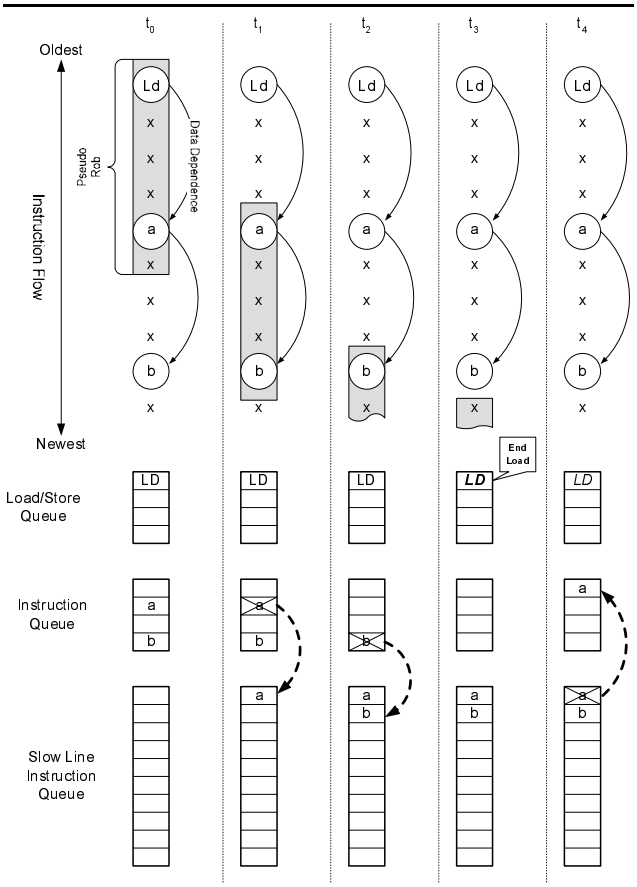

Figure 8. Time example of our SLIQ mechanism

struction is *critical* at decode stage is extremely complex. For this reason we have decided to delay the decision until later in the pipeline.

In order to do this, we use a small FIFO-like structure called pseudo-ROB[7]. At decode time, instructions are inserted both in the instruction queues (which will be very small and thus fast) and in the pseudo-ROB (cf. Figure 8). Eventually, instructions will get extracted from this pseudo-ROB, not because they commit (remember that our Out-of-Order Commit takes care of committing instructions with the use of checkpoints), but rather because they are the oldest of this structure. At the moment of their extraction we can effectively know if the instruction is a long latency instruction or not. Therefore, the main objective of the pseudo-ROB is to delay the decision of whether this instruction will be executed shortly or will consume a lot of resources for a long time.

The primary source of long latency instructions are loads that miss in L2 cache. If a particular load instruction arrives at the end of this pseudo-ROB with out having produced its output, we consider it a long latency load.

In the example of figure 8 the load in t_0 would also be considered a long latency one since it has not yet finished its execution. Any instruction that depends on it will also be considered a long latency instruction.

Once taken out of the pseudo-ROB we start the computation of dependencies on this load. This dependency computation is very simple. We keep a 32 bit register where we associate each bit to a logical register. This bit mask starts all cleared except for the destination register-bit of the long latency load. When a instruction is extracted from the pseudo-ROB, it will incorporate its destination register into this bit mask if it consumes any register from it, since this instruction is dependent on the load and so will be any instructions that consume its destination register. Registers (bits) get cleared when non-dependent instructions redefine those registers. This simple mechanism is heavily used in compiler construction[1].

Dependent instructions computed this way are removed from the general purpose instruction queue and stored in order in a secondary buffer, called *Slow Lane Instruction Queue (SLIQ)*, freeing entries from the instruction queue for short latency operations. This can be seen in figure 8 in the time moments t_1 and t_2 when instructions a and b, dependent on the load get moved into the *SLIQ*[8].

In order to simplify the wakening of these instructions, we associate the destination register of the long latency load to the corresponding entry of the *SLIQ*. This destination register microarchitecturally seems a better place to track the finishing of the instruction. When this register gets its value, a process of wakening the dependent instructions begins. This wakening is done at a pace of 4 instructions per cycle, with a starting penalty of 4 cycles. This is due to the need of re-compute the availability of their source operands before inserting again these instructions into the instruction queue.

In figure 8 at t_3 we can see that the load finishes its execution and how the instructions get inserted back into the window (in this example only one at a time). However, it could happen that a second long latency load is resolved while the instructions dependent on the first one are still being processed. Two different situations can happen. If the new load is younger than the instructions being processed, it will be found by the wakening mechanism. After that, the mechanism could continue, placing the instructions dependent on any of the two loads in the instruction queues, al-

7 This mechanism can be implemented in a normal ROB like structure but with little change. In our present research we have decided to combine it with our Out-of-Order commit ROB mechanism, which effectively has no ROB

8 Microarchitecturally we believe that it would be simpler to invalidate entries in the instruction queue and to insert into the SLIQ from the pseudo-ROB, but for clarity in the examples we are going to assume that instructions are *moved* from the instruction queue to the SLIQ

though always 4 at a time. If the new load is older that the instructions being processed, it will not be found by the wakening process, so a new wakening process should be started for the second load.

Finally, an additional advantage of the pseudo-ROB is reducing the misprediction penalty of branch instructions. Since our out-of-order commit mechanism removes the ROB, mispredicted branches force the processor to return up to the previous checkpoint, which is potentially some hundred instructions behind. The information contained in the pseudo-ROB allows to recover from branch mispredictions without needing to use a far checkpoint, whenever the branch instruction is still stored in the pseudo-ROB.

4. Experimental Results

The two mechanisms presented in the previous sections cover two of the most crucial resources when dealing with thousands of in-flight instructions. As they do not impose restrictions on one another, they are easily combined together. In this section we present the experimental framework and the results obtained for this combination of mechanisms which will probe its efficiency and power awareness.

Microprocessor Baseline	
Simulation strategy	Execution-driven
Issue policy	Out-of-order
Fetch/Commit width	4 insns/cycle
Branch predictor	16K history gshare
Branch predictor penalty	10 cycles
I-L1 size	32 KB 4-way, 32 byte line
I-L1 latency	2 cycles
D-L1 size	32 KB 4-way, 32 byte line
D-L1 latency	2 cycles
L2 size	512Kb 4-way, 64 byte line
L2 latency	10 cycles
Memory latency	1000 cycles
Memory ports	2
Physical registers	4096 entries
Load/Store Queue	4096 entries
Integer Queue	4096 entries
Floating Point Queue	4096 entries
Reorder Buffer	4096 entries
Integer General Units (lat.)	4 (lat/rep 1/1)
Integer Mult/DIV Units (lat.)	2 (lat/rep 3/1 and 20/20)
FP Functional Units (lat.)	4 (lat/rep 2/1)

Table 1. Architectural parameters

In table 1 we can see a summary of the microarchitectural configuration of our simulated baseline architecture. The benchmark suite used for all the experiments is `SPEC2000fp`, averaging over all the applications in the set. All benchmarks have been simulated 300 million representative instructions, where representativeness has been determined following [28].

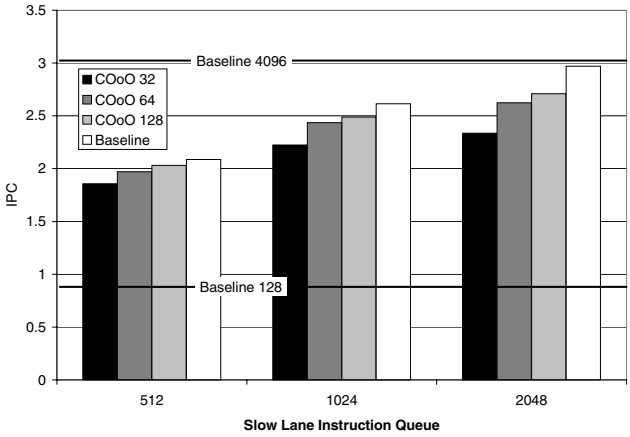

Figure 9. Main performance results

Figure 9 shows the main performance results of our Commit Out-of-Order Processor. This figure presents 3 groups of 4 bars each plus two reference lines across the figure. The reference lines correspond to our baseline processor with a reorder buffer and instruction queues having 128-entries (lower line) and 4096-entries (upper line). Each set of bars groups simulations according to the amount of *SLIQ* entries in the case of our mechanism or General Purpose Instruction Queue entries in the case of the baseline (which is clearly unrealistic to implement with current technology). The bars COoO 32, COoO 64, and COoO 128 correspond to our processor organization having a pseudo-ROB and instruction queues of 32, 64 and 128 entries respectively. In all cases the Out-of-Order commit processor has a checkpoint table with only 8 entries.

Several conclusions can be drawn from this figure. For instance, even the simplest processor having only a 8-entry checkpoint table, and 32-entry pseudo-ROB and IQ and 512 SLIQ buffer outperforms by a significant margin (about 110%) a conventional processor with 128-entry ROB and IQ (the lower reference line). If we consider a more complex processor (e.g. 8-entry checkpoint table and 128-entry pseudo-ROB and IQ and 2048 SLIQ buffer the difference in performance grows up to 204%. Notice that our mechanism always suffers a penalty with respect to the *unrealistic* baseline bar (white bar on the right of each group), which shows always a higher IPC. However, our proposal is in all cases significantly close to this *unrealistic* baseline with a fraction of the cost. We must take into consideration that having an instruction queue of 2048 entries is impossible with

present day technology. Nevertheless, having a *SLIQ* of that size is possible since this kind of secondary memory does not need wakeup logic and its selection logic is very simple (linearly from one point, contrary to normal selection logic which is totally associative).

In the third set the differences are the biggest. But it is roughly a 9% from 128 entries in our Instruction Queue to the 4096 entry queue of the baseline, which is of course a theoretical limit since we believe this structure can not be built without impacting cycle time. The main problem here is if our secondary buffer, the *SLIQ*, which is definitely simpler than an instruction window, can be built of that size.

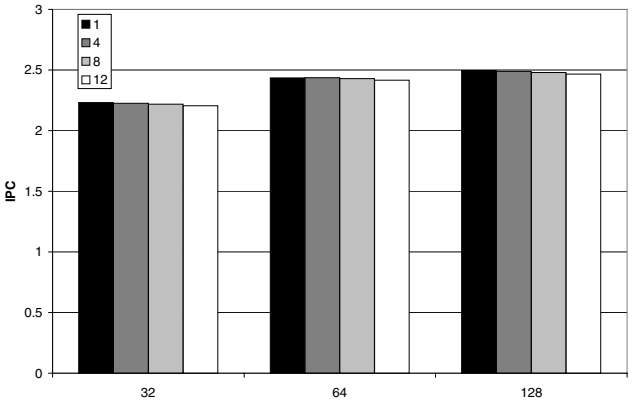

Figure 10. Sensitivity of our model to the delay of re-insertion of instructions from the SLIQ to the Instruction Queue

Fortunately for us, the data presented in figure 10 makes us believe so. This figure shows results for configurations with a 1024-entry *SLIQ* and 32, 64 and 128 pseudo-ROB and IQ entries. For each bar group we vary the number of *cycles to re-insert* instructions from the SLIQ to the instruction queue (1, 4, 8 and 12 cycles respectively), to analyze how sensitive is the mechanism respect to the latency from when the long latency instruction finishes till we start re-inserting the dependent instructions back in the window. As can be seen, we are not sensitive to this parameter. Even a 12 cycle latency only produces a negligible 1% slowdown. This roughly means that we can effectively use a very slow secondary buffer in parallel with our mechanism with hardly any penalty at all.

Figure 11 shows the average amount of in-flight instructions in both our proposal and the baseline (baseline assumes that the queues have been restricted to the size shown in the axis). The bars and reference lines correspond to the same configurations as Figure 9. The experiments in this figure show that our out-of-order commit processor is effectively allowing for a very big amount of in-flight in-

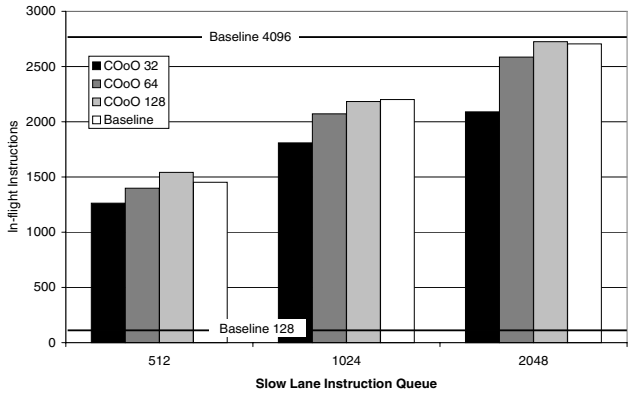

Figure 11. Average amount of in-flight instructions in our Commit Out-of-Order processor

structions. Remember that all these in-flight instructions are achieved with just 8 checkpoint entries, whose total hardware cost is minimal. This figure even shows that in certain situation, such as 128 entries in the Instruction queue with 2048 in the *SLIQ* we have even more in-flight instructions than the baseline which has a higher amount of hardware devoted to this. Although this may seem positive to our mechanism, we believe it may have a dual positive-negative effect. On one side it allows for more in-flight instructions from where to extract ILP. On the other side it is effectively increasing the number of instructions from wrong paths being executed. At the moment we are analyzing how to separate both effects to get the best out of our mechanism.

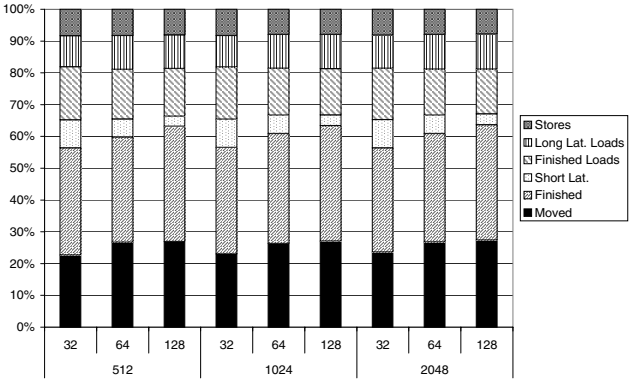

Figure 12. Breakdown of the type of instructions retired from the pseudo-ROB structure depending to the different architectural configuration parameters

In figure 12 we can see the percentual breakdown of which instructions get retired from the pseudo-ROB struc-

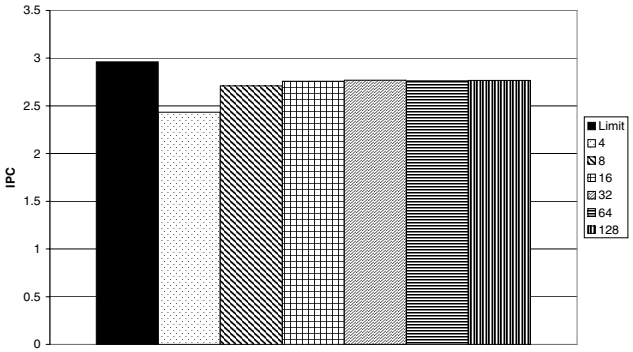

Figure 13. Sensitivity of our commit mechanism to the amount of available checkpoints (2048 entry IQ and 2048 physical registers)

ture. The bars correspond to the configurations for our proposal used along this section (i.e. *SLIQ* have 512, 1024 and 2048 entries, and pseudo-ROB and IQ have 32, 64 and 128 entries). The 6 sections on each bar correspond to the status of the instructions when are retired from the pseudo-ROB, which from bottom to top are:

Moved corresponds to the percentage of instructions that are moved from the IQ to the SLIQ. Notice that the moved represent a relatively small percentage (20 to 30 %) but they require most of the storage space (thats why the IQ varies from 32 to 128 and the SLIQ from 512 to 2048).

Finished corresponds to the instructions that have already completed their execution. Actually, this group of instructions has so small latency they they are already executed when retired.

Short Lat. corresponds to instructions that although not yet executed, they have short latency or depend on short latency instructions. Therefore this group of instructions is expected to finish in a few cycles and they remain in the IQ.

Finished Loads are the loads that have already finished or that have hit in L1 or L2.

Long Lat. Loads are loads that miss in L2. Notice that this group correspond to about 10% of the instructions which are the cause of the problem.

Stores corresponds to the store instructions.

Notice that some amount of short latency instructions get traded for moved instructions when the pseudo-ROB size increases. This effect is due to the fact that in our experiments we have always set the same size for both the pseudo-ROB and the small instruction queue. Having bigger pseudo-ROBs allows for more instructions to get executed before leaving the window.

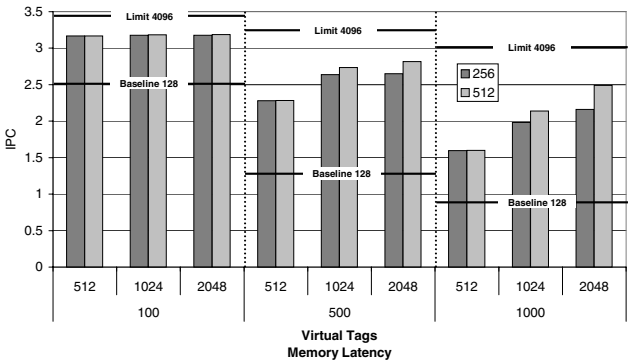

Figure 14. Combining different techniques

In figure 13 we can see the IPC obtained with different numbers of checkpoints. The limit bar shows IPC results obtained when considering ROB of 4096 entries, which is microarchitecturally unfeasible. This limit helps us show how near we are of a pseudoperfect mechanism which does not take into account complexity. Having only 4 checkpoints produces just a 20% slowdown. If we increase to 8 checkpoints this slowdown decreases to only 9% and from 32 onwards the slowdown is just a 6%. This 6% slowdown with 256 times less entries is clearly spectacular.

5. Towards affordable Kilo-Instruction Processors

In this paper we have extensively analysed two techniques that allow for future processors to have thousands of in-flight instructions, also known as *kilo-instruction processors*. These two techniques attack two problems, the ROB and the Instruction Queues using the *same* philosophical approach, locality.

Our replacement for the ROB, which is composed of the checkpointing mechanism plus the pseudo-ROB structure, is based on the locality of the instructions. Most recent instructions which a higher probability of mis-speculation are kept on the pseudo-ROB, while the checkpointing mechanism handles the instructions afterwards, once the locality effect has *gone away*. This allows for a higher number of in-flight instructions. Something similar happens in our SLIQ mechanism, where instructions are kept into two different queues with different microarchitectural needs depending on the expected delay of the instruction.

This same approach can also be followed with the other two resources involved in the problem of allowing thousands of in-flight instructions. In figure 14 we can see the combination of a modification of the two techniques presented in this paper with a register mechanism called Ephemeral registers [19, 9] which proposes an aggressive

delayed register allocation combined with early register recycling in the context of *Kilo-Instruction Processors*.

Figure 14 is divided into three zones, each of them comprising the results for different memory latencies (100, 500 and 1000). Each group is composed of three sets of two bars. Each of the sets assumes an increasing amount of Virtual Registers. The two bars of each group represent the IPC with 256 or 512 physical registers. Each zone has also two lines which represent the baseline performance when 128 ROB entries are assumed and also the Limit one, in which all the resources have been up-sized with no constraints. This Limit microarchitecture is unfeasible, but it helps us understand how well our mechanisms are behaving, since it acts as an upper bound. It can be noted that *kilo-instruction processors* are an effective way of approaching this limit in an affordable way.

6. Related Work

Several studies analyze the impact of memory latency and the future expectations for processors with thousands of in-flight instructions [15]. In [5] another study is conducted that compares prefetching and local optimizations regarding bandwidth requirements. This study concludes that even very big latencies (up to 640 cycles) can be tolerated if the processor has enough bandwidth. Similar conclusions are exposed in [25] in the context of numerical applications using stream buffers and prefetching. Other studies [32] analyze the tolerance of loads to latency and the different relation among issue width, Load/Store queue size and ROB size. In another approach, [29] the relation between branch predictors, the amount of in-flight instructions, and the cache size is analyzed. They conclude that in numerical applications having over 256 in-flight instructions still delivers performance due to the good branch speculation that these kind of applications have.

Many techniques have been proposed to create large ROB and instruction queues, such as the multiscalar processor [31], the trace processor [27], and the DMT architecture [2]. These architectures allow speculative execution of a large number of instructions. In [6], 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 [26, 12] that are also related to our work. In [26] the authors analyze the memory consistency model in the presence of loads in large ROBs. They propose a mechanism that retires instructions from the ROB to some sort of history buffer when some conditions are met. If a conflict occurs, the history buffer is used to restore the previous context.

There are many papers dealing with registers management in superscalar processors. In [22], the authors propose a mechanism that allows early recycling of registers, while at the same time providing precise exceptions. In [21], a technique that delays allocation of physical registers to instructions to the execution stage is presented. The mechanism uses tags, named *virtual registers*, to keep track of dependences between instructions and to convert these tags into real physical registers when the instruction produces its result.

In the area of instruction queues for large numbers of in-flight instructions [18] and [7] have made important contributions. [18] presents a mechanism for detecting long latency operations, namely, loads that miss in L2 as in our present research, and move dependent instructions into a Waiting Instruction Buffer (WIB) where they reside until the load completes. It also maintains a bit vector to determine the completion of the different loads on which the long latency instructions may depend. In [7], the authors propose to insert all instructions into a slow but large instruction queue first. Thereafter, when the oldest instructions in the IQ are determined not to have executed, they are moved into a smaller but faster instruction queue, for they assume that these instructions are always on the critical path. However, both approaches require a wake-up and select logic which might be on the critical path [24], thus potentially affecting cycle time.

In [8], it is shown that, in the pursuit of large numbers of in-flight instructions to tolerate long memory latencies in current out-of-order processors, resources identified as critical, such as ROB, Instruction Queues, and Physical registers, are heavily underutilized. The authors advocate that better use of these resources should be a priority for future research in processor architecture and they provide some directions. Similar observations about the use of some critical resources in processors with large numbers of in-flight instructions have also been made in [16].

A different approach is using checkpointing. In [14] the authors propose a recovery mechanism based on maintaining multiple checkpoints of the architectural register file. In [30] several mechanisms to support precise exceptions are presented.

To our knowledge, the next two independent papers are the first ones to propose to use checkpointing instructions as an efficient way to control and manage the use of critical resources inside the processor. In [11], the authors propose the use of a mechanism to checkpoint critical long latency instructions, which allows to create a very large ROB using a small physical one. This multi-checkpointing mechanism is also used to release instructions from the ROB early which essentially makes the classical ROB unnecessary. It is also used to release physical registers early, improving the mechanism previously proposed in [22], and to release

load instructions early from the load-store queues.

At the same time as the previous work, [20] proposes Cherry which is based on a single checkpointing of the ROB, where they identify the instructions that are not subject to misspeculation. In this region, Cherry is able to release registers and load-store entries early, providing quick recovery from frequent replay events using the ROB, and precise exception handling using checkpointing.

In [9] and [19], an integrated mechanism that combines the checkpointing procedure for multi-chekpointing and Cherry, respectively, with late allocation and early release of physical registers is presented. The synergy between these three techniques allows to reduce the number of physical registers required to maintain a huge number of in-flight instructions optimally.

In [10] the authors present the design and evaluation of what they coined *kilo-instructions processors* as a way to efficiently deal with future high memory latencies. They present several techniques to deal with the optimal management of the processors´s critical resources. The architecture has no ROB, organizes the Instruction Queues in a two-level implementable hierarchy and use multicheckpointing, late allocation and early release of registers to reduce at maximum the need for physical registers. The authors advocate that *kilo-instructions processors* could be a reality in a near future.

In [23] on the other hand, what the authors propose is when a load miss reaches the head of the ROB, a checkpoint of the architectural state is created, and the processor start executing in a special mode from where to extract knowledge for the second non-speculative pass where this knowledge from branches and loads already pre-issued. This piece of work follows the conceptual path of [6]. At the time we were writing the final version of this paper for the conference proceedings, we received reference [3]. The latter paper presents mechanisms similar to those presented in [3], [20] and [10], and moreover a new and intelligent way of dealing with load-store queues.

7. Conclusions

In order to tolerate increasing memory latencies, a large number of in-flight instructions must be mantained. To support more in-flight instructions several resources must be up-sized. Examples of such critical resources are the ROB, the general purpose instruction queues, the Load/Store queue and the number of physical registers. However increasing the number of entries of these resources is impractical because of area, access time, and power consumption issues.

In this paper we propose new microarchitectural structures that have a much lower need of resources. In particular we propose a checkpointed based commit mechanism and a *Slow Lane Instruction Queuing* mechanism. With these two proposals it is possible to implement the functionality of a big ROB and big instruction queues, requiring a reduced number of entries.

With these two mechanisms, our processor has a performance degradation of only 10% over a conventional processor with 4096 entries ROB and instruction queues. In comparison our processor requires only a 128-entry pseudo-ROB, 128-entries instruction queues, and a low cost 2048-entry *SLIQ* (which can actually be implemented as a RAM) much cheaper than a conventional instruction queue with the same entries. At the same time, our processor has an increase in performance of 204% relative to a conventional processor with both structures having 128 entries.

8. Acknowledgements

The authors wish to thank José F. Martínez for his comments and ideas during the development of this work and to Sriram Vajapeyam and Oliver Santana for the help during the writting of this paper. The authors are also very thankful to the reviewers for their extensive comments that have helped to produce a better paper. This work has been supported by the Ministry of Science and Technology of Spain, under contract TIC-2001-0995-C02-01 and by the CEPBA.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[2] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 226–236, Dallas, Texas, Nov. 1998.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36nd annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, Dec. 2003.

[4] C. Asato, R. Montoye, J. Gmuender, E. Simmons, A. Ike, and J. Zasio. A 14 port 3.8ns 116 64b read-renaming register file. In *1995 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb. 1995.

[5] A.-H. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *Proceedings of the 15th International Conference on Supercomputing*, pages 486–500. ACM Press, June 2001.

[6] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically allocating processor resources between nearby and distant ilp. In *Proceedings of the 28th Annual International Symposium on on Computer Architecture*, pages 26–37. ACM Press, July 2001.

[7] E. Brekelbaum, J. Rupley, C. Wilkerson, and B. Black. Hierarchical scheduling windows. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 27–36. IEEE Computer Society Press, Nov. 2002.

[8] A. Cristal, J. Martínez, J. Llosa, and M. Valero. A case for resource-conscious out-of-order processors. In *Computer Architecture Letters*, volume 2, Oct. 2003. *MEDEA - Memory Performance: Dealing with Applications, Systems and Architecture*, Sept. 2003. Technical Report UPC-DAC-2003-45, July 2003.

[9] A. Cristal, J. Martínez, J. Llosa, and M. Valero. Ephemeral registers with multicheckpointing. Technical Report UPC-DAC-2003-51, UPC, Oct. 2003.

[10] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Kilo-instruction processors (invited paper). In *Proceedings of 5th International Symposium of High Performance Computing - LNCS 2858*, pages 10–25. Springer, Oct. 2003.

[11] A. Cristal, M. Valero, A. Gonzalez, and J. Llosa. Large virtual robs by processor checkpointing. Technical Report UPC-DAC-2002-39 (Submitted to Micro 35), Universidad Politécnica de Cataluña, Department of Computer Architecture, July 2002.

[12] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, Atlanta, Georgia, May 1999. IEEE Computer Society TCCA and ACM SIGARCH. *Computer Architecture News,* 27(2), May 1999.

[13] J. Hennessy and D. Patterson. *Computer Architecture. A Quantitative Approach. Second Edition.* Morgan Kaufmann Publishers, San Francisco, 1996.

[14] W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26. ACM Press, June 1987.

[15] N. P. Jouppi and P. Ranganathan. The relative importance of memory latency, bandwidth, and branch limits to performance. In *Workshop of Mixing Logic and DRAM: Chips that Compute and Remember*. ACM Press, 1997.

[16] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues, in conjunction with ISCA*, May 2002.

[17] J. Keller. The 21264: A superscalar Alpha processor with out-of-order execution. In *9th Annual Microprocessor Forum*, San Jose, California, Oct. 1996.

[18] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 59–70. IEEE Computer Society, May 2002.

[19] J. Martínez, A. Cristal, M. Valero, and J. Llosa. Ephemeral registers. Technical Report CSL-TR-2003-1035, Cornell Computer Systems Lab, June 2003.

[20] J. Martínez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 3–14. IEEE Computer Society Press, Nov. 2002.

[21] T. Monreal, A. González, M. Valero, J. González, and V. Viñals. Delaying physical register allocation through virtual-physical registers. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 186–192. IEEE Computer Society, Nov. 1999.

[22] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 202–213. IEEE Computer Society Press, Dec. 1993.

[23] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, Anaheim, California, Feb. 2003.

[24] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 206–218. ACM Press, June 1997.

[25] D. Pressel. Fundamental limitations on the use of prefetching and stream buffers for scientific applications. In *Proceedings of the 2001 ACM Symposium on Applied computing*, pages 554–559. ACM Press, Mar. 2001.

[26] P. Ranganathan, V. Pai, and S. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210. ACM Press, June 1997.

[27] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–148, Research Triangle Park, North Carolina, Dec. 1997.

[28] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, Sept. 2001.

[29] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. In *IEEE Transactions on Computers*, volume 48, pages 1260–1281. IEEE Computer Society, Nov. 1999.

[30] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. In *IEEE Transactions on Computers*, volume 37, pages 562–573. IEEE Computer Society, May 1988.

[31] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 1995. ACM SIGARCH and IEEE Computer Society TCCA. *Computer Architecture News,* 23(2), May 1994.

[32] S. Srinivasan and A. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 148–159, Dallas, Texas, Nov. 1998.