On Reducing Register Pressure and Energy in Multiple-Banked Register Files

Jaume Abella*, Antonio González*+

* *Computer Architecture Department*
Universitat Politècnica de Catalunya
Barcelona (Spain)
jabella@ac.upc.es

+ *Intel Barcelona Research Center*
Intel Labs, Universitat Politècnica de Catalunya
Barcelona (Spain)
antonio@ac.upc.es

Abstract

The storage for speculative values in superscalar processors is one of the main sources of complexity and power dissipation. In this paper, we present a novel technique to reduce register requirements as well as their dynamic and static power dissipation that is based on delaying the dispatch of instructions while minimizing its impact on performance. The proposed technique outperforms previous schemes in both performance and power savings. With only 1.77% IPC loss, the mechanism achieves more than 13% dynamic and 15% static extra power savings in the integer rename buffers and more than 9% dynamic and 10% static extra power savings in the FP rename buffers. Significant power savings are also achieved if the processor uses a physical register file for both committed and non-committed values instead of rename buffers. Additionally the register requirements are reduced by more than 18% and 13% for integer and FP programs respectively.

1. Introduction

Power dissipation has become a critical issue for both high performance and mobile processors. Dynamic power dissipation is the dominant factor nowadays, but static power will become increasingly significant in upcoming processors. While dynamic power is directly related to the activity of the circuits, static power depends on the amount of powered-on transistors and their physical characteristics. The management of speculative register values is one of the main sources of energy dissipation in current superscalar microprocessors [6]. Additionally, this structure is one of the processor hotspots. Thus, reducing power consumption in this power hungry structure is critical not only from the energy standpoint but from the temperature standpoint.

Some banks of these structures can be turned-off if they are not used, and thus, some power is saved. Some of them may be also turned-off if they may be used without contributing significantly to improve performance.

Literature on power reduction using adaptive schemes is very extensive. Among them, we could point out some schemes to reduce power and complexity [1][9][10][12][16].

Some other authors have investigated how to reduce the power and complexity of the register files. Cruz et. al. [5] proposed a multilevel register file organization for low complexity and fast access time to the registers. Zyuban and Kogge [17] studied the complexity of a centralized register file and proposed a scheme to distribute it. Our proposal is orthogonal to these works so they can be easily combined.

Different approaches have been recently proposed in order to reduce the dynamic power of the issue queue [7][4]. Folegnani and González [7] proposed an issue queue design where energy consumption is effectively reduced using a dynamic resizing mechanism of the issue queue.

In this work, we propose an adaptive microarchitecture that achieves significant dynamic and static power savings in the register file, at the expense of a very small performance loss. Our proposal is based on observing how much time instructions spend in the reorder buffer and the issue queue, and taking resizing decisions based on these observations. Even if the instructions are ready to be dispatched, if it is expected that they would hardly contribute to improve performance, they are not dispatched. We compare this scheme with the approach in [7], and show that the proposed technique provides significant advantages.

The rest of the paper is organized as follows. Section 2 describes the baseline organization of the issue queue, rename buffers, register file and reorder buffer. Section 3 describes the proposed technique and the mechanism used for comparison purposes. Section 4 evaluates the performance of the proposed approach. Finally, section 5 summarizes the main conclusions of this work.

2. Baseline microarchitecture

In this section we describe the baseline microarchitecture, with special emphasis on the structures that are the target of this work: rename buffers, register file, issue queue and reorder buffer.

2.1. Processor

Two different organizations for the storage of speculative values have been studied. The first one is similar to that of the Alpha 21264 [6] and Pentium IV [14]. In this case, speculative and committed values are stored in a centralized register file. The second one is similar to that of the HP PA8700 [8]. In this case, committed values are stored in an

Proceedings of the 21st International Conference on Computer Design (ICCD'03)
1063-6404/03 $ 17.00 © 2003 IEEE

architectural register file, whereas speculative values are stored in rename buffers until commit. Integer and FP values are kept in separated files for both cases. There are two register files for the first organization and two sets of rename buffers for the second one. In the rest of the paper the organization based on a centralized register file will be referred to as *RegF* whereas the one based on rename buffers will be referred to as *RenB.*

2.2. Register files and rename buffers

This section describes the implementation assumed for the register file, but a similar implementation has been assumed for the rename buffers. Integer and FP register files are identical. A register file is split into banks (8 entries per bank in our experiments). In order to reduce the bank access time, the bank selection logic and the decoding of the entry to be accessed are done in parallel.

Figure 1 shows the scheme for a read operation. One entry of each bank is read, and the output logic selects the requested register among those. It can be observed that this scheme overlaps the bank selection with the decoding and reading of each bank. Figure 2 illustrates a write operation. The wordlines that select the requested register for every bank are gated by the bank selection logic. In this case, the bank selection is overlapped with just the wordline decoding because the write must be performed only in the proper bank.

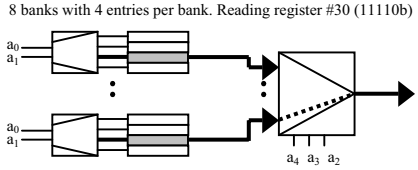

Figure 1. Scheme of a read operation

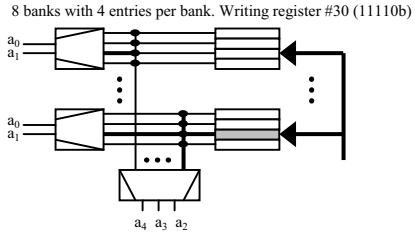

Figure 2. Scheme of a write operation

This implementation of the register file reduces their access time at the expense of increasing notably their dynamic energy consumption. If the access time of this structure is not critical, a sequential decoding scheme could be considered.

In this work, the parallel implementation of the multiple-banked register file has been assumed for all the compared mechanisms, including the baseline. This decision is justified by the estimated access time for both schemes. For this purpose, we used CACTI 3.0 [13], with a configuration of 16 banks, 8 registers per bank, 64-bit data width, 0.10 μm technology, 16 read and 8 write ports. Table 1 shows the delays obtained for each component of the register file.

Table 1. Delay and energy for the different components of a multiple-banked register file design

Component	Abbrev.	delay (ps)	energy (pJ)
Address routing	add	84	1.3
Decode (4 to 16)	4to16	232	3.8
Decode (3 to 8)	3to8	203	1.5 per deco
Wordline + bitline	wlbl	134	5.1 per bank
Data to/from bank	data	104	10.8 per bank
Out driver	out	106	27.6

Table 2 shows the delay of the critical path for both a read and a write operation in both schemes. The table shows that the parallel scheme reduces the access time by 27% for read operations and 29% for write operations with respect to the sequential scheme. #BanksOn corresponds to the number of turned on banks at the operation time.

Table 2. Delays and energy for read/write operations

Seq. scheme	Critical path	Delay (ps)	Energy (pJ)
Read	add+4to16+3to8+wlbl +data+out	863	50.1
Write	add+4to16+3to8+wlbl	653	22.5
Parallel scheme	**Critical path**	**Delay (ps)**	**Energy (pJ)**
Read	add+3to8+wlbl+data+ out	631	32.7+17.4 x #BanksOn
Write	4to16+ctrl_wordlines (=add)+wlbl	450	10.2+12.3 x #BanksOn

Turning off unused banks can save static power for both schemes and dynamic power for the parallel one. A given bank is turned on as soon as at least one of its registers is assigned to an instruction as its destination operand. A given bank is turned off when none of its registers is being used. This scheme can be easily implemented adding a bit (*BusyBit*) to every register. This bit is set when a register is assigned to an instruction and is reset when the instruction commits and frees the previous mapping of its destination register. The bank *enable/disable* signal is a NOR function of its registers' *BusyBits*.

In order to maximize the number of banks that are turned off, when a free register is requested, the one with the lowest bank identifier is chosen so that the activity in the register file is concentrated on the banks with lower identifiers.

2.3. Issue queue and reorder buffer

The assumed reorder buffer and issue queue have just one difference with respect to conventional ones: their occupancy can be limited dynamically. This feature is used to control the number of in-flight instructions and thus, to control the pressure on the register files. As previous work [7][4], no compaction mechanism for the issue queue has been assumed since compaction results in a significant amount of extra energy consumption even if it contributes to performance.

3. Adaptive schemes

This section describes the proposed mechanism and the mechanism used for comparison purposes[7].

3.1. Proposed mechanism

3.1.1. Underlying concepts

Superscalar processors try to keep full both the reorder buffer and the issue queue. In general, dispatching instructions as soon as possible is beneficial for performance, but not for power. In many cases instructions are held in the issue queue and have a destination register assigned for some cycles before they are finally issued. From the performance standpoint, it is desirable not to delay the issuing of any instruction. From the power standpoint, it is desirable that instructions remain in the issue queue for the minimum number of cycles, since reducing its occupancy allows reducing the number of registers required. Our proposal tries to achieve these objectives by means of various heuristics:

- The first heuristic tries to reduce the time that instructions spend waiting for being issued in the issue queue. If it is observed that instructions wait too long, the instruction window size (reorder buffer size) is reduced and thus, the dispatch of instructions is delayed. Reducing the number of entries in the reorder buffer reduces the number of registers in use.
- The second heuristic tries to prevent situations in which the limited instruction window size is harming performance. Even if instructions spend too much time in the issue queue, it is desirable to be less aggressive when there are few instructions in the reorder buffer.
- Finally, there are some events that require an immediate action. In particular, L2 data cache misses, which have a very long latency, stall the commit of instructions for many cycles. Thus, in case of an L2 miss it is interesting to increase the instruction window size to allow the processor to process more instructions while the miss is being serviced.

Deciding when instructions spend too long in the issue queue is one of the tricky parts of the mechanism. We are interested in finding out the minimum number of cycles that the instructions require to spend in the issue queue without losing significant IPC. In order to gain some insight, we have experimentally observed the behavior of different programs (7 benchmarks from SPEC2000) for short intervals of time. If we just consider the intervals of time with similar IPC, we can observe some trends: a) the minimum time that instructions spend in the issue queue and the time that they spend in the reorder buffer are correlated, b) this correlation is not linear: the longer the time in the reorder buffer, the longer the time in the issue queue but the ratio between the latter and the former decreases as the time spent in the reorder buffer increases.

3.1.2. Implementation of the mechanism

The first and second heuristics outlined above are based on measuring the number of cycles that instructions spend in the issue queue and in the reorder buffer, as discussed in the previous section. However, an exact computation of these parameters may be quite expensive in hardware (e.g. time stamps for each entry) and consume a non-negligible amount of energy. According to Little's law [15] for queuing systems in which a steady-state distribution exists, the following relation holds:

$$L_q = \lambda W_q$$

where L_q, λ and W_q stand for the average queue size, the average number of arrivals per time unit and the average time that a customer spends in the queue. In the issue queue and the reorder buffer, the arrival rates (λ) are exactly the same. Since we are interested in the ratio between time in the issue queue and time in the reorder buffer, instead of counting how many cycles (W_q) every committed instruction spends in the issue queue and the reorder buffer, we will count how many instructions (L_q) are in these structures every cycle. This approximation implies that all instructions that arrive to the queues but do not commit are also counted. We have observed that the effect of considering or not these instructions does not provide significant differences.

We have experimentally confirmed that this relation between queue size and waiting time holds for the 7 benchmarks mentioned in the previous section. We have observed that the average number of cycles spent in the issue queue and the average issue queue occupancy follows a near-linear relation, and the same holds for the reorder buffer. Thus, we can conclude that using occupancy ratios instead of time ratios does not result in significant differences.

In order to leverage the relation between the time spent in the issue queue and the time spent in the reorder buffer, the proposed mechanism uses the ratio between both occupancies (IQ occupancy / ROB occupancy) to take resizing decisions. If this value is higher than a given threshold, the window size is decreased by N instructions (8 instructions in our experiments), and if it is lower than another threshold, the window size is increased by N instructions.

```
(1) THRESHOLD_LOW = 1 - ROB_dynamic_size / ROB_size
(1) THRESHOLD_HIGH = THRESHOLD_LOW + 1/8

(2) FRACTION = #instr_in_IQ / #instr_in_ROB

(3) if (FRACTION > THRESHOLD_HIGH)
(3)     ROB_dyn_size = max(ROB_dyn_size-8, 32)
(3) else if (FRACTION < THRESHOLD_LOW)
(3)     ROB_dyn_size = min(ROB_dyn_size+8, ROB_size)

(4) if (L2 miss during the period)
(4)     ROB_dyn_size = min(ROB_dyn_size+8, ROB_size)

(5) if (#cycles_disp_stall > IQ_THRESHOLD_HIGH)
(5)     IQ_dyn_size = min(IQ_dyn_size+8, IQ_size)
(5) else if (#cycles_disp_stall < IQ_THRESHOLD_LOW)
(5)     IQ_dyn_size = max(IQ_dyn_size-8, 8)
```

Figure 3. Heuristic to resize the reorder buffer and the issue queue

These thresholds are dynamically adapted according to the observations made in the above section, that is, they depend on the reorder buffer size. Figure 3 details the approach for resizing the reorder buffer and the issue queue. *ROB_size* stands for the physical size of the reorder buffer (128 instructions in our evaluation), and *ROB_dyn_size* stands for the maximum number of allowed instructions in the reorder buffer at a given time (similar definition applies

to *IQ_size* and *IQ_dyn_size*). In order to avoid an extremely small reorder buffer, the following constraint is applied: $ROB\_size/4 \leq ROB\_dyn\_size \leq ROB\_size$. The thresholds are set according to (1). The fraction of time that instructions spend in the issue queue versus the time that they spend in the reorder buffer is approximated as (2). This parameter is averaged for each interval of time. At the end of each interval, resizing decisions are taken according to the criteria described in (3): the reorder buffer dynamic size is increased by 8 instructions, decreased by 8 instructions, or left unchanged depending on the value of the *FRACTION* parameter and the thresholds.

Finally the third heuristic in the above section is implemented as follows. Whenever there is an L2 cache miss, the reorder buffer size is increased, as (4) in figure 3 shows. In theory only data misses should be considered but for the sake of simplicity, we do not distinguish between instruction and data misses since the majority of L2 misses correspond to data.

Register file banks (or rename buffers) are turned off when they are not busy as explained in section 2.2. Issue queue occupancy is further controlled by a mechanism that monitors how many cycles the dispatch is stalled due to unavailable entries in the issue queue. As detailed in section (5) of figure 3, if stalls are too frequent, the issue queue size is augmented (*#cycles_disp_stall* stands for the number of cycles that the dispatch is stalled because instructions cannot be placed in the issue queue). If stalls are very rare, the issue queue size is decreased. This simple mechanism along with the adaptive mechanism to limit the reorder buffer occupancy achieves a significant register file pressure reduction with very small performance loss. In our experiments, for an interval size of 128 cycles, different values for the issue queue thresholds were evaluated (2, 4, 8, 16, 32, 64) obtaining significant power savings and small performance degradation for these pairs of values: <16,32> and <16,64>. To simplify the implementation and avoid doing some divisions and multiplications, integer arithmetic is used instead of FP one. In particular, the thresholds are scaled as follows:

```
THRESHOLD_LOW = ROB_size - ROB_dyn_size
THRESHOLD_HIGH = THRESHOLD_LOW + ROB_size/8
```

In our experiments we use a 128 entry reorder buffer, so *THRESHOLD_HIGH* corresponds to *THRESHOLD_LOW* + 16. Thresholds are compared with *FRACTION* so this parameter is also scaled as follows:

```
FRACTION = ROB_size x #instr_in_IQ / #instr_in_ROB
```

The multiplication in the above expression is trivial to implement since the reorder buffer size is a power of 2. For the division, the dividend has 11 bits and the divisor has 7 bits, assuming an interval of 128 cycles. This requires a rather small hardware. In fact, an iterative divider can be used instead of a parallel one, since delaying the resizing decisions by a few cycles does not have any practical impact.

The energy consumption of the additional hardware is negligible because only three small counters are updated every cycle and the rest of the structures work only once every interval (128 cycles in our experiments). Assuming that the divider is implemented as a radix 4 divider (2 bits of the quotient are computed each cycle), the total hardware required is one multiplexor and less than 20 units (adders, incrementers and comparators) whose inputs always have 11 bits or less. We have experimentally verified that delaying the resizing of the reorder buffer by 2 or 3 cycles to allow for an iterative divisor has negligible impact on performance.

3.2. Mechanism used for comparison

The proposed mechanism has been compared with the mechanism proposed in [7], which will be referred to as *FoGo* in the rest of the paper. The issue queue has the same structure for both the proposed mechanism and the mechanism used for comparison, but the resizing schemes are different.

FoGo reduces power consumption in the register files and rename buffers by dynamically resizing the issue queue. The mechanism monitors the performance contribution of the youngest bank of the issue queue (8 instructions in their experiments) and measures how much these entries contribute to the IPC. If the contribution is below a threshold, the issue queue size is reduced by one bank. On the other hand, the size of the queue is increased periodically. In particular, this mechanism counts the number of committed instructions that were issued from the 8 youngest entries in the issue queue. If there are less than *N* instructions issued from the youngest part during an interval of time, the issue queue size is reduced. Their experiments showed that using an interval of 1000 cycles and a threshold of 25 instructions reduces significantly the issue queue occupancy with a very small performance loss. Every 5 intervals, the issue queue size is increased by one bank.

For the comparison presented below, we have chosen the configuration with the parameters that they report as the more appropriate ones (*FoGo1000*) and the same parameters but with an interval of 128 cycles – the same as the one used by the proposed mechanism – with a corresponding threshold of 3 instructions issued from the youngest part (*FoGo128*).

4. Performance evaluation

In this section we present performance and power results for the proposed mechanism, and compare it with the technique proposed in [7].

4.1. Experimental framework

Power and performance results are obtained through Wattch [2], which is an architecture-level power and performance simulator based on SimpleScalar [3]. Some enhancements are the separation of the reorder buffer and the issue queue, and the extension to model the ports for the register files and rename buffers. The model required for multiple-banked structures has been obtained from CACTI 3.0 [13], which is a timing, power and area model for banked cache memories. The following table describes the assumed processor configuration.

Table 3. Processor configuration

Fetch, decode, issue, commit width: *8 instructions*		
Branch pred.: *Hybrid 2K Gshare, 2K bimodal, 1K selector* BTB: *2048 entries, 4-way*		
L1 Icache: *64KB, 2-way, 32 byte line (1 cycle)* L1 Dcache: *64KB, 4-way, 32 byte line, 4 R/W ports (2 cycles)* L2 unified cache: *512KB, 4-way, 64 byte line (10 cycles hit, 50 cycles miss, 2 cycles interchunk)*		
Fetch queue: *64 entries*	IQ: *80 entries*	ROB: *128 entries*
RegF microarchitecture: INT registers: *112 (14 banks x 8), 16R+8W ports* FP registers: *same as INT registers* *RenB* microarchitecture: INT rename buffers: *80 (10 banks x 8), 16R+8W ports* FP rename buffers: *same as INT rename buffers*		
INT functional units: *6 ALU, 3 mult/div* FP functional units: *4 ALU, 2 mult/div*		
Technology: *0.10 μm*		

For this study we have selected the whole Spec2000 benchmark suite [18] with the *ref* input data set. We have simulated 100 million of instructions for each benchmark after skipping the initialization part. The benchmarks were compiled with the Compaq/Alpha compiler with `-O4 -non_shared` flags.

4.2. Interval length

In order to choose a suitable interval to resize the structures, we have done some experiments. Figure 4 shows the IPC with respect to the baseline for different interval lengths using 3 benchmarks from SpecINT2000 (*gap, gzip, twolf*) and 3 from SpecFP2000 (*ammp, applu, art*). It can be seen in figure 4 that in general, longer intervals improve performance. Figure 5 shows the reorder buffer occupancy reduction for different interval lengths. It can be observed that shorter intervals achieve higher occupancy reduction. Higher occupancy reduction will translate into better opportunities to save power and reduce register pressure.

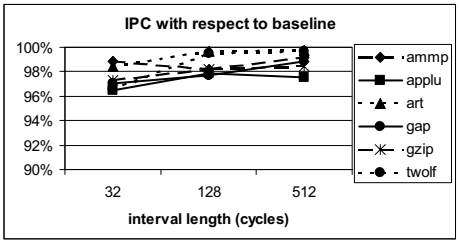

Figure 4. IPC for different interval lengths

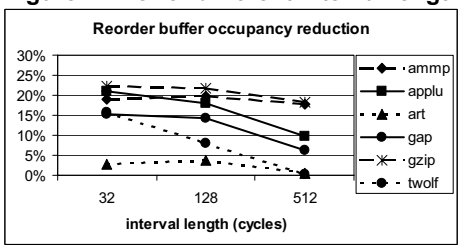

Figure 5. Reorder buffer occupancy reduction for different interval lengths

Figures 4 and 5 show that a 32-cycle interval hardly reduces the reorder buffer occupancy with respect to a 128-cycle interval whereas it results in slightly higher performance degradation. In addition, the shorter the interval, the higher the energy overhead of resizing the structures. The 512-cycle interval is slightly better in terms of performance but it is not so effective to reduce the reorder buffer occupancy. We can conclude that the 128-cycle interval achieves the best tradeoff between power and performance.

4.3. Performance and power results

The performance evaluation has been done comparing two versions of the proposed technique, two versions of *FoGo*, *FoGo128* and *FoGo1000* as described above, and a baseline with no adaptive structures. The two versions of our technique correspond to different threshold values for the *IQ_THRESHOLD_HIGH (32 or 64)*. We will refer to them as *IqRob32* and *IqRob64* respectively in the rest of the paper. The baseline architecture does not resize the issue queue nor the reorder buffer but includes the mechanisms that we have assumed for *IqRob* and *FoGo* to turn off unused register file or rename buffer banks.

4.3.1. Performance

Figure 6 shows the IPC loss for the different mechanisms. *IqRob32* and *IqRob64* have better performance than *FoGo1000* and *FoGo128* respectively for the SpecINT2000 and the whole Spec2000, and achieve similar results for the SpecFP2000. On average, *IqRob32* loses less than 2% in IPC and *IqRob64* loses less than 3.5%. *FoGo* reduces the size of the issue queue when the IPC contribution of the youngest bank is below a fixed threshold. This threshold basically determines the loss of IPC that the mechanism may cause and thus, it has a bigger impact for programs with lower IPC, such as some of the SpecINT2000.

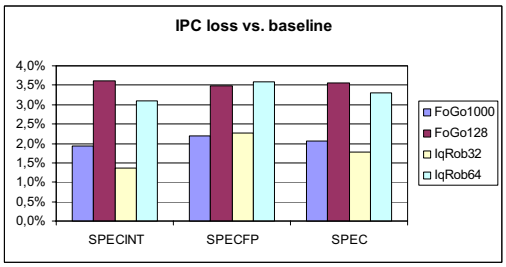

Figure 6. IPC loss for different techniques

4.3.2. Reorder buffer and issue queue

IqRob achieves lower reorder buffer and issue queue occupancies than *FoGo*. Having fewer instructions in these structures implies that fewer registers are used so more power is saved. *IqRob* is significantly more effective than *FoGo*, especially for integer applications.

Table 4 shows the effectiveness of *IqRob* for reducing the issue queue and reorder buffer sizes. On average, the

maximum reorder buffer size is set to about 70% of its total capacity. About 45% of the entries are occupied and 25% of the entries are enabled but empty. This is mainly due to sections of code where instructions spend few cycles in the issue queue. The *IqRob* mechanism tends to increase the reorder buffer size in these situations because these instructions are quite power efficient (they do not retain registers too many cycles).

Table 4. Size reduction

Reorder Buffer	SpecINT	SpecFP	Spec
IqRob32	35.4%	23.5%	29.0%
IqRob64	34.2%	22.4%	27.9%
Issue Queue	**SpecINT**	**SpecFP**	**Spec**
FoGo1000	18.7%	11.8%	15.0%
FoGo128	31.2%	20.4%	25.4%
IqRob32	28.7%	20.4%	24.2%
IqRob64	34.3%	24.1%	28.8%

4.3.3. Integer register file and rename buffers

As discussed above, reducing the number of in-flight instructions results in a lower number of registers in use. *IqRob* achieves higher reductions than *FoGo* due to its higher effectiveness at reducing the reorder buffer size. *FoGo1000* reduces the register pressure by 7%, and *FoGo128* does it by 15%, whereas *IqRob32* and *IqRob64* achieve reductions of 18% and 20% respectively. These register pressure reductions are exactly the same for both architectures (*RenB* and *RegF*) since they have been configured with exactly the same number of registers (80 rename buffers + 32 logical registers for *RenB*, and 112 registers for *RegF*). Figures 7 and 8 show that *IqRob* and *FoGo* achieve higher dynamic power savings in the register file and rename buffers than the baseline.

It can be seen that higher power savings are achieved for the *RenB* architecture. The main reason is that rename buffers with high index are freed as soon as the instruction commits and thus, used registers correspond almost always to low-index registers. In this way, high-index banks can be turned-off in most of the cases when the number of unused registers is higher than the size of a bank. For the *RegF* architecture it may happen that a register with high index is allocated to an instruction and remains allocated for a very long period of time after the instruction commits, preventing the corresponding bank to be turned off.

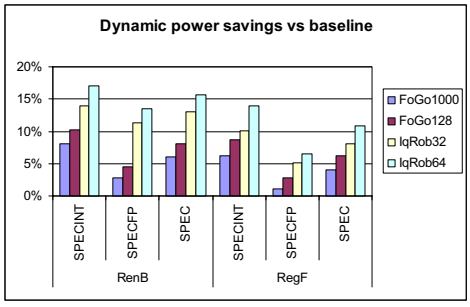

Figure 7. Dynamic power savings for the integer register file and rename buffers w.r.t. the baseline

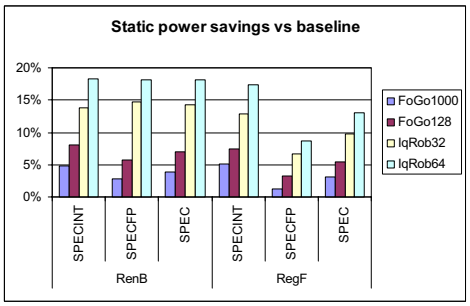

Figure 8. Static power savings for the integer register file and rename buffers with respect to the baseline

4.3.4. Floating point rename buffers

FP rename buffers are hardly used by integer programs so we report power statistics only for FP programs. Figures 9 and 10 show dynamic and static power savings. It can be seen that *IqRob* outperform *FoGo* in both dynamic and static power reduction for both architectures. Additionally the FP register requirements are reduced by more than 13% for both *IqRob* techniques and less than 10% for *FoGo* techniques.

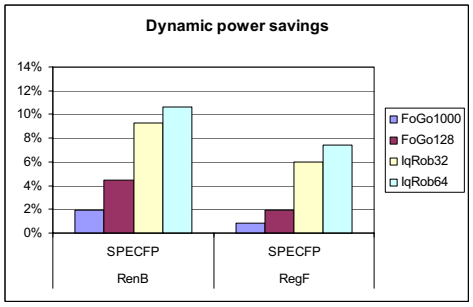

Figure 9. Dynamic power savings for the FP register file and rename buffers with respect to the baseline

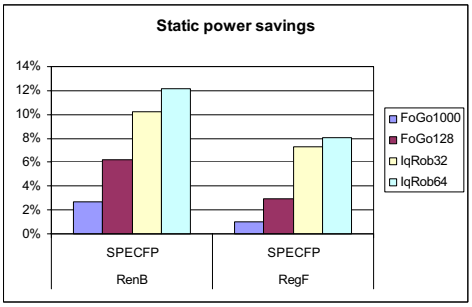

Figure 10. Static power savings for the FP register file and rename buffers with respect to the baseline

4.3.5. Summary

Table 5 summarizes the main performance metrics of the proposed *IqRob* mechanism with respect to the baseline configuration. Note that the baseline already includes a mechanism to turn off unused register file and rename buffer banks. The *FoGo* mechanism is also shown for comparison purposes.

DP, *SP*, *IQ*, *RenB* and *RegF* stand for dynamic power savings, static power savings, issue queue, rename buffers and register file respectively.

Table 5. Summary of results

	FoGo 1000	*FoGo 128*	*IqRob 32*	*IqRob 64*
IPC Loss	2.1%	3.6%	1.8%	3.3%
INT Reg Pressure	7.2%	15.3%	18.1%	19.8%
FP Reg Pressure	5.8%	9.9%	13.0%	15.5%
INT RegF DP	4.1%	6.2%	8.1%	10.9%
INT RegF SP	3.2%	5.4%	9.8%	13.0%
FP RegF DP	0.8%	2.0%	6.0%	7.4%
FP RegF SP	1.0%	2.9%	7.3%	8.0%
INT RenB DP	6.0%	8.0%	13.0%	15.7%
INT RenB SP	3.9%	7.0%	14.2%	18.2%
FP RenB DP	2.0%	4.5%	9.3%	10.6%
FP RenB SP	2.7%	6.2%	10.2%	12.1%

Table 5 shows that *IqRob32* and *IqRob64* outperform *FoGo1000* and *FoGo128* in all metrics respectively. Additionally *IqRob* techniques achieve higher power savings in register files and rename buffers, and reduce register pressure more effectively than *FoGo* techniques.

5. Conclusions

We have presented a novel scheme that dynamically limits the number of in-flight instructions in order to save dynamic and static power in register files or rename buffers, and reduces register pressure. The proposed mechanism is based on monitoring how much time instructions spend in both the issue queue and the reorder buffer and limit their occupancy based on these statistics.

The proposed mechanism has been evaluated in terms of performance, and dynamic and static power savings for the whole Spec2000.

Results have been compared with a state-of-the-art issue queue resizing technique, and it has been shown that the proposed technique outperforms previous work in terms of performance and power savings. The proposed technique achieves more than 13% dynamic and 15% static extra power savings in the integer rename buffers (45% dynamic and 54% static total power savings with respect to not turning off banks) and more than 9% dynamic and 10% static extra power savings in the FP rename buffers (23% dynamic and 30% static total power savings with respect to not turning off banks). Significant power savings are also achieved for the register files if they are used instead of rename buffers.

Additionally the register requirements are reduced by more than 18% for the integer registers and more than 13% for the FP ones.

Acknowledgements

This work has been supported by CICYT project TIC2001-0995-C02-01, the Ministry of Education, Culture and Sports of Spain, and Intel Corporation. We would like to thank the anonymous reviewers by their comments.

References

[1] R.I. Bahar, S. Manne. Power and Energy Reduction Via Pipelining Balancing. In ISCA 2001.

[2] D. Brooks, V. Tiwari, M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In ISCA 2000.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 3.0. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1999.

[4] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose and P. Cook. A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors. In GLSVLSI 2001.

[5] J.L. Cruz, A. González, M. Valero, N. Topham. Multiple-Banked Register File Architectures. In ISCA 2000.

[6] J. Emer. EV8: The post-ultimate alpha. Keynote at PACT 2001.

[7] D. Folegnani and A. González. Energy-Effective Issue Logic. In ISCA 2001.

[8] D. Halperin. PA-RISC 8x00 Family of Microprocessors with Focus on PA-8700. Hot Chip Conference, 2000.

[9] T. Karkhanis, J.E. Smith, P. Bose. Saving Energy with Just in Time Instruction Delivery. In ISLPED 2002.

[10] S. Manne, A. Klauser, D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In ISCA 1998.

[11] S. Palacharla, N.P. Jouppi, J.E. Smith. Complexity-Effective Superscalar Processors. In ISCA 1997.

[12] D. Ponomarev, G. Kucuk, K. Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In MICRO 2001

[13] P. Shivakumar and N.P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Research report 2001/2, WRL, Palo Alto, CA (USA), 2001.

[14] E. Sprangle, D. Carmean. Increasing Processor Performance by Implementing Deeper Pipelines. In ISCA 2002.

[15] W.L. Winston. Operations Research Applications and Algorithms. Ed. Duxbury Press. Second edition, 1991.

[16] S.H. Yang, M.D. Powell, B. Falsafi, T.N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. In HPCA 2002.

[17] V. Zyuban, P. Kogge. The Energy Complexity of Register Files. In ISLPED 1998.

[18] SPEC2000. www.specbench.org/osg/cpu2000/