

Low-Complexity Distributed Issue Queue

Jaume Abella* and Antonio González*⁺

* Computer Architecture Department
Universitat Politècnica de Catalunya
Barcelona (Spain)

+ Intel Barcelona Research Center
Intel Labs, Universitat Politècnica de Catalunya
Barcelona (Spain)

jabella@ac.upc.es, antonio@ac.upc.es

Abstract

As technology evolves, power density significantly increases and cooling systems become more complex and expensive. The issue logic is one of the processor hotspots and, at the same time, its latency is crucial for the processor performance.

This paper presents a low-complexity FP issue logic (MB_distr) that achieves high performance with small energy requirements. The MB_distr scheme is based on classifying instructions and dispatching them into a set of queues depending on their data dependences. These instructions are selected for issuing based on an estimation of when their operands will be available, so the conventional wakeup activity is not required. Additionally, the functional units are distributed across the different queues.

The energy required by the proposed scheme is substantially lower than that required by a conventional issue design, even if the latter has the ability of waking-up only unready operands. MB_distr scheme reduces the energy-delay² product by 35% and the energy-delay product by 18% with respect to a state-of-the-art approach.

1. Introduction

Technology and microarchitecture evolution is driving microprocessors towards higher clock frequencies and higher integration scale. These two factors translate into higher power density, which calls for more sophisticated and expensive cooling systems. Reduction of power dissipation in the hottest spots of the processor can be very beneficial not only in terms of energy reduction, but also for reducing cooling costs or increasing performance for a given thermal solution.

The issue logic of superscalar processors is one of the main hotspots. Besides, this logic has a significant delay and is difficult to pipeline [6][19]. Overall, the design of low latency and low power dissipation issue logic is an important challenge for continuing scaling up the performance of superscalar processors.

The issue logic is typically implemented using fully associative schemes for the wake-up process [19][8]. This kind of schemes requires a mechanism for checking which operands become ready for all the instructions in the issue queue every time that an instruction is selected for execution. Those instructions whose all operands are ready are considered in the selection process that follows the wake-up. Even though this approach results in high IPC rates, its latency may not be compatible with high clock rates. Additionally, its complexity grows drastically if the issue width or the issue queue size are increased. Large instruction windows are required for augmenting the opportunities to extract more ILP, which in turn requires wider pipelines.

This paper proposes a new issue logic organization that achieves high performance and reduces its power and complexity. The proposed approach is based on having multiple instruction queues, placing the instructions in queues based on their dependences and estimating their issue time.

The rest of the paper is organized as follows. Section 2 reviews some related work. Section 3 presents the proposed scheme. Section 4 evaluates its performance. Section 5 summarizes the main conclusions of this work.

2. Related Work

Instructions are dispatched into the issue queue after they are fetched and decoded, where they stay until they are issued to the functional units. This process includes several key aspects that determine the issue logic performance, complexity and power: where the instructions are placed, how it is known when they are ready to be issued, which instructions are selected for issue, and which functional unit is used by each selected instruction.

2.1. CAM Based Issue Queue Schemes

Conventional issue logic schemes are based on CAM/RAM structures [19], which achieve high IPC but at the expense of dissipating significant power. Figure 1

shows the structure of an issue queue entry based on the CAM/RAM approach. Folegnani and González [14] propose an issue queue design where energy consumption is reduced by using a dynamic resizing mechanism of the issue queue. They also propose to disable the wakeup for empty entries and ready operands. Abella and González [2] propose a similar scheme for resizing the issue queue and the register file based on observing the relation between the issue queue and the reorder buffer occupancies.

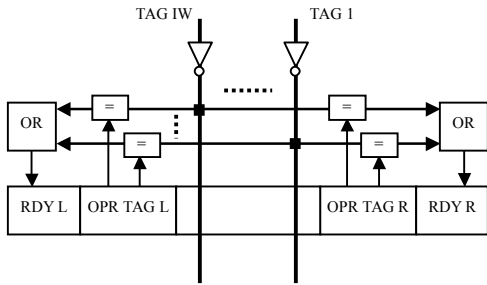


Figure 1. Issue queue entry

Buyuktosunoglu *et al.* [9] propose a detailed issue queue implementation for a similar scheme. This implementation is based on subbanking the issue queue (both CAM and RAM arrays). Based on this design, different approaches for turning off banks in order to reduce power dissipation can be implemented. In [9], the authors propose a simple mechanism to dynamically adapt the issue queue size.

Further research has proposed different approaches to achieve high performance having small CAM/RAM arrays or putting them away from the critical path as described below.

Brown *et al.* [6] propose a mechanism to pipeline the issue logic and still allowing dependent instructions to be issued in consecutive cycles.

Ernst and Austin [13] propose an issue queue organization which has three kinds of entries: those without CAM cells, for instructions that are ready at dispatch time, those with CAM cells for only one operand, for instructions that have just one pending operand at dispatch time, and those with CAM cells for both operands as in a conventional issue queue. This reduces the power and delay of the issue logic.

Cotofana *et al.* [12] propose a counter-based resource conflict check that significantly reduces the complexity and delay of the selection logic as well as its area. This approach allows the processor to wakeup and select instructions faster than with a conventional issue scheme.

Kucuk *et al.* [16] propose power efficient comparators, zero byte encoding and bitline segmentation for the issue queue to make this structure less power hungry.

Lebeck *et al.* [17] propose a waiting instruction buffer to place those instructions dependent on a load cache

miss. Those instructions are moved from the buffer to the issue queue when the load cache miss is serviced. This mechanism reduces the issue queue requirements for those applications with significant load miss rates.

Brekelbaum *et al.* [3] propose an issue queue design based on hierarchical scheduling windows. Critical instructions are placed in a small CAM/RAM issue queue whereas latency tolerant instructions are placed in a buffer that does not use power hungry CAM logic but it requires a longer latency for the wakeup operation.

Canal and González [10][11] propose different schemes to issue instructions. One of them is based on computing the issue cycle of each instruction at dispatch time and issuing them in-order using the dynamically pre-computed scheduling. Given that it is hard to compute precisely this information at dispatch time, a small CAM/RAM array is required to place those instructions that were not issued at the predicted cycle. These instructions stay in this conventional issue queue until they are finally issued.

Michaud and Sez nec [18] propose a two-level issue queue such that the small first level works as a conventional CAM/RAM issue queue and the second level stores instructions but has no wakeup capability. The instructions are prescheduled into the second level taking into account their dependences and latencies. They are promoted to the first level issue queue if it is expected that they are ready to be issued and there are free entries in the first level issue queue. This mechanism is shown to work better than dependence based ones but introduces some more complexity because it requires hardware similar to the dependence check logic of the register renaming, but using adders instead of comparators. This problem is handled by using some simplifications that make the prescheduling process less precise.

Raasch *et al.* [20] propose a scalable issue logic design based on subbanking the issue queue and promoting instructions from bank to bank based on the expected number of cycles that instructions require for being ready. Even though the banks are small, all of them are implemented using the CAM/RAM structure so this mechanism can work at high clock rates but it still dissipates significant power, as the authors outline.

Some of these mechanisms [10][11][18][20] require the computation of the issue cycle of all the instructions dispatched simultaneously in just one cycle, which may be too optimistic and may require complex hardware.

2.2. Other Issue Queue Schemes

Since CAM-based schemes dissipate significant power, alternative issue logic designs based on structures other than CAM ones have been proposed. Based on the observation that most of the instructions have very few consumers, Canal and González [10][11] propose a

mechanism consisting on waking-up only N dependent instruction for each producer. It is implemented using a RAM array (N -use table). Additionally, there are two more buffers: one of them contains ready instructions coming from the dispatch stage and one buffer to place those instructions that are not in the first N uses of one of their producers. This second buffer has two different implementations: in-order or a CAM-based out-of-order. A similar scheme is proposed by Huang *et al.* [15].

Palacharla *et al.* [19] propose an issue queue design based on a small number of *first in first out* (FIFO) queues. Only the instructions at the head of each FIFO are considered for issue. Since our proposal is partially based on these FIFO queues, we describe below Palacharla’s approach in more detail. Instructions are dispatched to the FIFOs with the following heuristics:

- If there is a queue whose tail instruction produces the first operand of the instruction being dispatched, the instruction is placed in this queue. If the queue is full and the instruction has only one source operand then dispatch is stalled.
- If there is a queue whose tail instruction produces the second operand of the instruction being dispatched, the instruction is placed in this queue. If the queue is full then dispatch is stalled.
- Otherwise the instruction is placed in an empty FIFO. If there are not empty FIFOs then dispatch is stalled.

These heuristics guarantee that instructions in a given FIFO must be executed sequentially. This mechanism only requires a table to store for each register which queue (if any) has its producer at the tail of the queue. This table can be implemented in two different ways: storing the mentioned information for each physical register or for each architectural register. If the former is chosen, the table has not to be modified under a branch missprediction. If the latter is chosen, the table stores wrong information under a branch missprediction so it has to be regenerated or cleared. We have experimentally observed that clearing the table does not have significant impact in performance and simplifies the hardware.

A FIFO-based organization does not require the wakeup logic. Instructions at FIFO heads check if their operands are ready every cycle in a small table. This table stores just one bit per physical register indicating whether it is available.

A FIFO-based issue queue organization works well for integer applications since, in general, this kind of programs has narrow dependence graphs that fit in a small number of FIFO buffers. Additionally, integer operations have short dependence chains with short latencies, so after being allocated to one dependence chain, a FIFO usually becomes empty in short, which allows another dependence chain to be placed in it. Since FP programs have wide dependence graphs and long latency operations, they require a large number of FIFOs.

Detailed evaluation of this observation is provided in the evaluation section.

For further details in issue queue designs, we refer the reader to the survey by Abella, Canal and González [1].

This paper presents a new issue logic organization designed to achieve high performance with both integer and FP programs. Our proposal does not require CAM arrays like most of the previous approaches and fully distributes the issue logic: queues, selection logic and crossbars to send instructions to the functional units, with small impact on performance. Our approach differs from previous ones in the fact that it is the first one, to our knowledge, that combines the benefits of considering dependence chains and expected issue cycles reducing complexity and power.

3. Proposed Issue Logic Design

As outlined in the previous section, we have observed that the FIFO based issue queue works quite well for integer applications but not for FP ones. This can be observed in the following experiments. We will refer to this organization as *IssueFIFO_AxB_CxD* where A and C correspond to the number of integer and FP queues respectively, and B and D correspond to the size of the integer and FP queues respectively. Details of the processor configuration can be found in Table 1.

Table 1. Processor configuration

Parameter	Configuration
Fetch, decode and commit width	8 instructions
Issue width	8 integer + 8 FP instructions
Branch predictor	Hybrid with 2K entry Gshare, 2K entry bimodal and 1K entry selector
BTB	2048 entries, 4-way set associative
L1 Icache	64K, 2-way, 32 byte/line, 1 cycle
L1 Dcache	32K, 4-way, 32 byte/line, 2 cycle, 4 R/W ports
L2 unified cache	512K, 4-way, 64 byte/line, 10 cycle
Main memory	64 byte bandwidth, 100 cycles for first chunk, 2 cycles inter-chunk
Fetch queue	64 entries
Reorder buffer	256 entries
Registers	160 INT + 160 FP
INT functional units	8 ALU (1 cycle), 4 mult/div (3-cycle mult, 20-cycle div)
FP functional units	4 ALU (2 cycles), 4 mult/div (4-cycle mult, 12-cycle div)
Technology	0.10 μm

In this section, the issue queue of the baseline processor has the same size as the reorder buffer (256 entries). This corresponds to an unbounded issue queue, since the dispatch process is never stalled due to lack of entries in the issue queue. Smaller issue queues may be more cost-effective and are considered for the proposed scheme.

The benchmarks used for this study are the whole Spec2000 benchmark suite [24] with the *ref* input data set. This suite consists of 12 integer and 14 FP programs. 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.

Figure 2 and Figure 3 show the IPC loss of the *IssueFIFO* scheme with respect to the baseline for the integer and FP programs respectively. Different configurations varying the number of queues and their sizes have been evaluated. For SpecINT2000 the integer queues are varied whereas different configurations of the FP queues are explored for SpecFP2000 benchmarks.

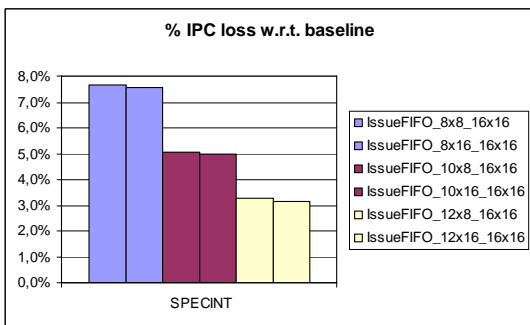


Figure 2. IPC loss of *IssueFIFO* technique w.r.t. unbounded conventional issue queue (specINT)

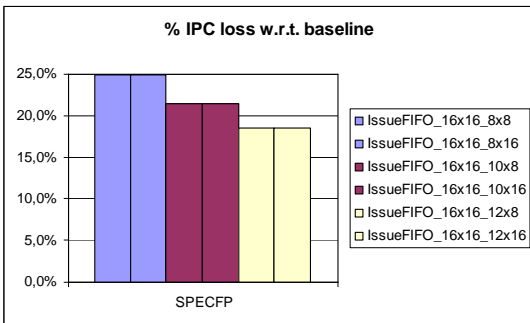


Figure 3. IPC loss of *IssueFIFO* technique w.r.t. unbounded conventional issue queue (specFP).

It can be observed that the IPC loss of FIFO queues is relatively small for integer benchmarks whereas the complexity is reduced significantly. Increasing the number of FIFO queues achieves higher performance since the dispatch stage is stalled less frequently. On the other hand, large queues do not provide significant benefits. Our experiments show that increasing the number of queue entries from 8 to 16 improves performance by 0.1% for 8, 10 and 12 queues.

FP benchmarks show similar trends regarding the number and size of the queues, but it can be seen that these applications lose much more performance than

integer ones. FP benchmarks have wider DDG's (Data dependence graphs) than integer ones, so more queues are required. Increasing the number of queues does not come for free since:

- The logic to dispatch instructions to the queues becomes more complex.
- There are more candidate instructions to be issued at a given cycle, so more instructions must check if their operands are ready.
- The hardware for issuing the instructions to the functional units is more complex.

We can thus conclude that the *IssueFIFO* organization is suitable for integer DDG's but not for FP ones. Below, we propose more appropriate approaches for FP codes.

3.1. Latency Based Organization

The study of the *IssueFIFO* organization for FP benchmarks revealed that the dispatch process is stalled very often, but most of the queues store a very small number of instructions. Given that most FP operations have long latencies, interleaving different dependence chains in a single queue could be an interesting approach to reduce complexity with minimal impact in performance. However, it is crucial to interleave these dependence chains in an appropriate way, since instructions of the same queue are issued in the same order as they are placed. Ideally, one would like to place instructions in a given queue in such a way that a new instruction can be issued every cycle. This is what the scheme proposed in this section tries to achieve. For this purpose, it is necessary to estimate the issue time of each instruction, which depends on its dependences and the latencies of the operations, among other factors. We will refer to this organization as *LatFIFO*. This organization is exactly the same as *IssueFIFO* for integer codes. However, for FP ones, instructions are placed in FIFO queues considering the expected time when they will be ready to be issued. The expected issue time is computed at dispatch stage as follows:

```

IssueCycle = MAX(current_cycle + 1,
                  OpLeftCycle, OpRightCycle)
If (inst is load)
    IssueCycle = MAX(IssueCycle,
                    AllStoreAddr)
else if (inst is store)
    AllStoreAddr = MAX(AllStoreAddr,
                      IssueCycle of its address +
                      AddressLatency)
If (inst has destination register)
    DestCycle = IssueCycle +
                InstructionLatency

```

where *OpLeftCycle* and *OpRightCycle* stand for the cycle when its left operand (if any) and its right operand (if any) will be available respectively. *AllStoreAddr* stands for the first cycle when the address of all previous store instructions will be known. It should be noted that load

and store instructions are split into two operations: one for computing the memory address and another for accessing memory. The memory access requires knowing that no conflict with previous stores exists, but the address computation does not. Store instructions update the cycle when the addresses of all store instructions will be known. *AddressLatency* stands for the number of cycles required to compute the address of a load or store instruction. *DestCycle* stands for the cycle when the destination operand (if any) will be available. *InstructionLatency* corresponds to the latency of the corresponding operation. L1 Dcache hit latency is assumed for loads. We experimentally checked that knowing the exact number of cycles for each memory access has no significant effect on the proposed schemes. We assume that the above computations can be performed in a single cycle, which may be an optimistic assumption.

Each instruction is placed in that queue that is not full and whose last instruction has an estimated issue time at least one cycle earlier than the instruction being dispatched. If there is more than one queue that meets these conditions, the one whose last instruction is expected to be issued later is selected. If no queue meets these conditions, an empty queue (if any) is selected. If no queue can be selected the dispatch is stalled. Choosing the queue in this way leaves more opportunities for younger instructions to be dispatched without any stall.

The performance of the *LatFIFO* scheme for the FP benchmarks is shown in Figure 4. It can be observed that the performance loss is much smaller than the one of the *IssueFIFO* scheme, but it is still significantly high. On average, the performance of *LatFIFO* is about 10% better than that of *IssueFIFO*. It can also be observed in Figure 4 that increasing the size of the queues hardly improves performance.

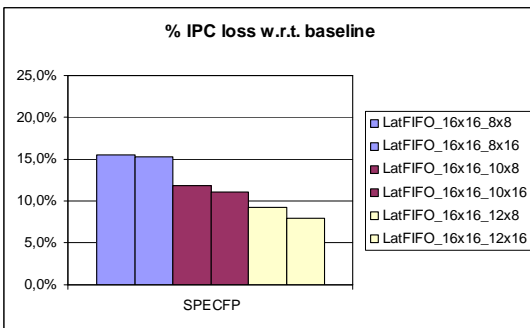


Figure 4. IPC loss of *LatFIFO* technique w.r.t. unbounded conventional issue queue

3.2. Mixed Approach

The main reason for the loss of performance of the *LatFIFO* scheme is that instructions in a given queue must be issued in the same order as they are dispatched. A

new dispatched instruction has to be always placed at the tail of a queue, even if it is expected to be issued in between two instructions placed consecutively in a given queue. An alternative could be using conventional CAM/RAM issue queues but they dissipate significantly much power and are much slower. In order to avoid the use of CAM cells but still having the flexibility of this kind of queues, the issue queue organization that we propose is based on a RAM structure similar to a register file. The main features of the proposed organization are the following:

- Instructions do not have to be placed in order in this buffer.
- Only one instruction from each queue can be selected for issuing, so the selection logic is quite simple.
- Instructions do not need to know whether their operands are ready before they are selected, so the wakeup process is not necessary.
- Dependent instructions are placed in the same queue as *IssueFIFO* scheme does. Given that each cycle only one instruction is selected per queue, having dependent instructions in the same queue reduces the probability of having more than one ready instruction in the same queue. Different independent dependence chains of instructions can share the same queue. These dependence chains will be referred to as *chains* in the rest of this paper.
- Instructions that are considered for issue for the first time have priority over those that were not issued the first time that they were supposed to be ready. This heuristic avoids selecting instructions that depend on either loads that missed in cache or unfinished instructions of other queues, instead of those instructions whose issue has not been delayed.
- Latencies are considered in order to know when the instructions will be ready for issuing, but it is done locally at each queue so no complex hardware is required.

3.2.1. Implementation. There is a table that maps logical registers to queues. This table is similar to the one used by the *IssueFIFO* scheme, but in this case it stores the queue identifier and the *chain* identifier since each queue can contain different *chains*. Thus, each queue has its own set of *chains*, and each entry in the table contains some bits identifying the queue where the operand is mapped and some bits identifying the *chain* of that queue which last instruction produces the operand. The use of the *chain* number is justified later. At dispatch time each instruction accesses this table to know the mapping of its source operands and the queue where it will be placed is determined in a similar way that *IssueFIFO* scheme does. The only difference is that an instruction is placed in the same queue as its predecessor only if it is the last instruction of the *chain* instead of the last instruction of

the queue. If the preferred queue is full or it is not found an appropriate queue, then a free *chain* identifier is assigned to the instruction. There are as many *chains* as the product of the number of queues and the number of *chains* per queue. In order to balance the number of busy *chains* per queue, the lowest free *chain* identifier is assigned. For instance, if there are 2 queues and 3 *chains* per queue, then the *chains* will be assigned with the following priority order: *chain* 0 from queue 0, *chain* 0 from queue 1, *chain* 1 from queue 0, *chain* 1 from queue 1, *chain* 2 from queue 0, and *chain* 2 from queue 1. When an instruction is dispatched to a queue, the mapping table is updated with both the queue and the *chain* number.

Each queue has an associated selection logic that every cycle picks up just one instruction from those in the queue, and a small table for the *chain* latencies. This table stores for each *chain* how many cycles will take the last issued instruction of this *chain* to finish. This table is very small since it has as many entries as the number of *chains* per queue, and the number of bits required to encode the largest functional unit latency. Every cycle the entire table is read and written. It is written to decrease by one all the entries using saturated counters, except that entry corresponding to the *chain* of the instruction being issued (if any), that is updated with the instruction's latency. All the entries are read and their information is compressed and broadcast to all the entries in the queue. An instruction in the queue needs to know if its predecessor in the *chain* has finished, or it is going to finish next cycle, or it will take 2 or more cycles to finish. Thus, each entry of the latency table is encoded into 2 bits: 00 if the instruction is going to finish next cycle, 01 if it has finished, and 11 if it will take 2 or more cycles to finish. Each entry in the queue selects its corresponding pair of bits and concatenates its *age identifier* to this pair of bits. The *age identifier* is a field that indicates the older/younger relationship among instructions in-flight. It can be implemented by using the reorder buffer position plus one extra bit concatenated on the left that is reset every time that the first position of the reorder buffer is assigned. Combining the bits in this way allows the selection logic to select the oldest instruction among those with higher priority according to the criteria described above using the same type of hardware as the one used by the baseline scheme. This mechanism is illustrated with an example in Figure 5.

Even if the selection hardware is not trivial, it is much simpler than the one required by a conventional issue queue since it has to select one instruction in each small issue queue instead of the N oldest ready instructions among the whole issue queue.

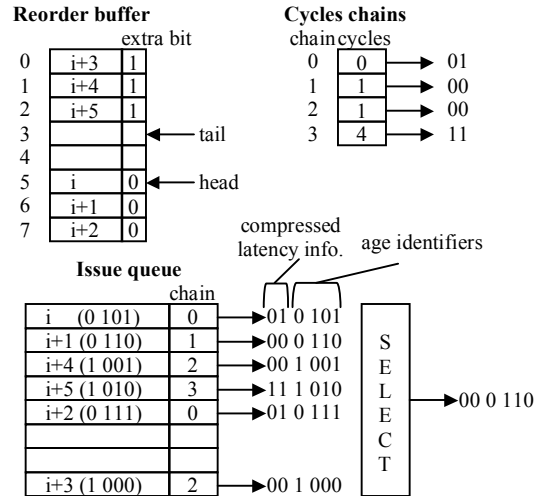


Figure 5. Example of selection

In this example instruction $i+1$ is selected for issuing the next cycle since it is the oldest one from those with higher priority (those belonging to *chains* 1 and 2). The selection logic just picks up the instructions with smaller identifier. This example shows that instructions belonging to the same *chain* have the same most significant pair of bits, so the oldest one is the one with higher priority.

This scheme will be referred to as *MixBUFF* in the rest of this paper. This scheme uses buffers instead of FIFO structures for the FP queues and both dependence and latency criteria are considered. Its performance has been evaluated assuming that unbounded *chains* per queue are allowed. As Figure 6 shows, the performance of this scheme with only 8 queues of 16 entries each is just around 5% lower than that of an unbounded (256 entries) conventional issue queue. *MixBUFF*'s performance is much better than that of *IssueFIFO* and *LatFIFO* schemes. For instance, with 8 FP queues of 16 entries each, the performance loss of *MixBUFF* is 5.2% whereas *IssueFIFO* and *LatFIFO* lose 24.8% and 15.2% respectively for the same configuration.

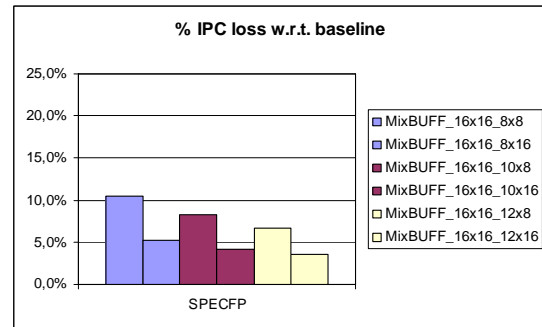


Figure 6. IPC loss of *MixBUFF* technique w.r.t. unbounded conventional issue queue

It can be also observed that for *MixBUFF*, increasing the size of the buffers results in more benefits than increasing the number of buffers. Since this mechanism distributes quite effectively the instructions that are ready at a given cycle across all the queues, increasing the number of queues does not provide significant improvements. On the other hand, placing multiple chains in a given queue increases its occupancy, so larger queues reduce the number of dispatch stalls.

3.3. Other Observations

Another source of complexity of conventional issue schemes lies on the interconnects that are required to issue an instruction from the issue queue to any functional unit. The above schemes have multiple issue queues, so functional units can be distributed across these FIFO queues or buffers with small impact on performance, and reducing significantly the complexity.

In order to take advantage of this, the proposed *MixBUFF* scheme has the functional units distributed across the different queues. The same distribution scheme is assumed for *IssueFIFO*, which is evaluated for comparison purposes. For both schemes the following configuration has been assumed:

- 8 integer FIFO queues of 8 entries each.
- 8 FP queues (buffers for *MixBUFF* and FIFO queues for *IssueFIFO*) of 16 entries each. For *MixBUFF* a maximum of 8 *chains* per queue has been assumed.
- 1 integer ALU per integer queue.
- 1 integer mult/div unit per pair of integer queues.
- 1 FP add and 1 FP mult/div per pair of FP queues.

Higher performance could be achieved from increasing the number of queues (both integer and FP), but doing so would increase both power dissipation and complexity.

4. Evaluation

In this section we present performance and power results for the proposed *MixBUFF* scheme, and it is compared with the *IssueFIFO* scheme.

4.1. Experimental Framework

Power and performance results are derived from CACTI 3.0 [21], which is a timing, power and area model for cache memories, and an enhanced version of Wattch [5], which is an architecture-level power and performance simulator based on SimpleScalar [7]. The main enhancements are the separation of the reorder buffer and the issue queue, and modeling ports for the register files.

The processor configuration and evaluated benchmarks are those described in section 3. The only difference is the issue queue configuration assumed for the baseline. In order to do a reasonable comparison, the size of the issue

queue assumed for the baseline scheme is not unbounded. The baseline configuration has two issue queues: one for integer instructions and another for FP ones. They store instructions out-of-order, like in P6 family (Pentium Pro, Pentium II and Pentium III) and Pentium IV [22], and any instruction in the queue can be issued if its operands are ready and the required resources are available.

4.2. Configurations Evaluated

Based on the study presented in section 3, the configurations that have been chosen are *MixBUFF_8x8_8x16* and *IssueFIFO_8x8_8x16*, both with distributed functional units. For the sake of readability they are referred to as *MB_distr* and *IF_distr* respectively. They have been compared in terms of power and performance with a baseline with 64 entries for the integer issue queue and 64 entries for the FP issue queue. It is referred to as *IQ_64_64* in the rest of this paper. A baseline with the same number of issue queue entries as *MB_distr* and *IF_distr* (64 and 128 entries for integer and FP queues respectively) has not been considered because it implies higher power dissipation and more complexity than the chosen baseline, and it achieves only 1.0% extra IPC with respect to the chosen baseline.

It has been assumed that the baseline consumes energy for waking-up only those CAM cells corresponding to unready operands, as proposed in [14] in order to make it more power efficient. A multiple-banked implementation of the issue queue has also been assumed: each queue consists of 8 banks with 8 entries each. Additionally, the selection logic does not dissipate power if the queue is empty for both the *IQ_64_64* and *MB_distr* schemes (*IF_distr* does not have selection logic).

4.3. Performance

Figure 7 shows the performance for integer applications. As expected, both *MB_distr* and *IF_distr* schemes achieve the same performance, except for *eon* since it has a significant number of FP instructions. These schemes lose on average 7.7% IPC w.r.t. the baseline, which is a reasonable loss since the complexity of both schemes is quite low for integer queues.

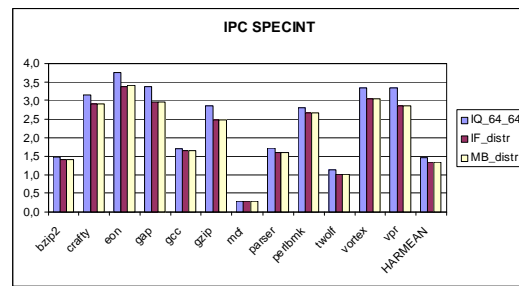


Figure 7. Performance for the integer benchmarks

It can be observed in Figure 8 that *IF_distr* loses significant performance (26.0%) for FP applications, whereas *MB_distr* only loses 7.6% IPC w.r.t. the baseline. *MB_distr* allows several *chains* to share the same queue in an efficient way, so dispatch stage is stalled fewer times than it is when the *IF_distr* scheme is used. It can be also seen that *MB_distr* outperforms *IF_distr* for all FP benchmarks.

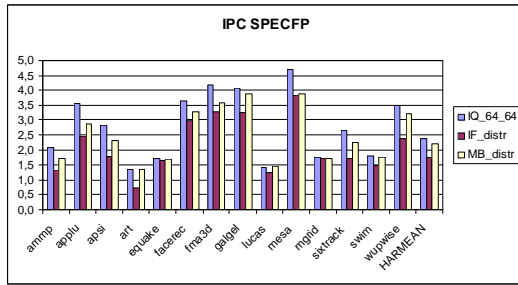


Figure 8. Performance for the FP benchmarks

4.4. Energy Consumption

This section analyzes where the energy is consumed for each scheme. Figure 9 shows the energy breakdown for the baseline (*IQ_64_64*). Even though only unready operands are woken up, and a multiple-banked implementation is assumed, the wakeup dissipates most of the power. Reading and writing instructions from/into the issue queue (*buff*) as well as the selection logic (*select*) dissipate significant power. The logic to drive instructions to the functional units is significant only for integer ALUs (*MuxIntALU*).

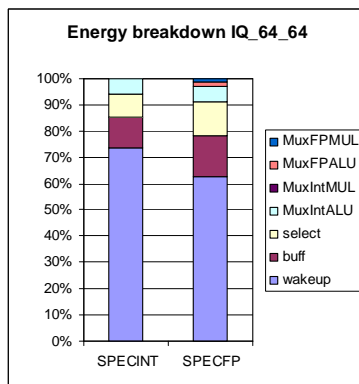


Figure 9. Energy breakdown for *IQ_64_64* scheme

Figure 10 and Figure 11 show the energy breakdown for *IF_distr* and *MB_distr* schemes. It can be observed that integer applications consume 25-30% of the energy in the table that stores the corresponding queue for each logical register (*Qrename*). Reading and writing instructions from/into the FIFO queues requires around 35% of the energy (*fifo*). Similar percentage is consumed

for reading and writing the information regarding which registers are ready (*regs_ready*). Since the functional units have been distributed across the queues, the logic to drive the instructions to the functional units dissipates negligible power.

FP benchmarks show similar trends than integer ones for *IF_distr*, whereas *MB_distr* has other sources of power dissipation. *MB_distr* scheme places FP instructions into buffers (*buff*) instead of FIFO queues (*fifo*). Some energy is spent selecting instructions (*select*) and managing the information concerning chains' latencies (*chains*). Finally, the energy required to drive instructions to the functional units and to save the last selected instruction (*reg*) is negligible.

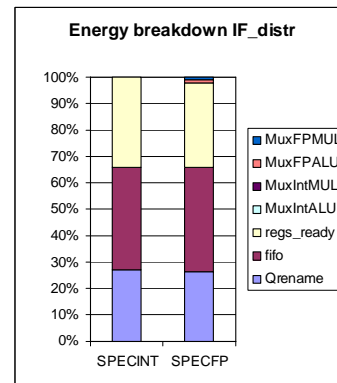


Figure 10. Energy breakdown for *IF_distr* scheme

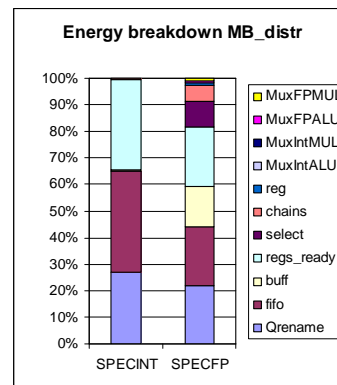


Figure 11. Energy breakdown for *MB_distr* scheme

4.5. Power Efficiency

Different metrics related with power-efficiency have been proposed to compare different schemes [4]. Depending on what the constraints are, different metrics should be used. **Power** is adequate when the heat is the main constraint, whereas **energy** is used for comparing schemes where the battery lifetime is the strongest constraint. Other metrics like **energy-delay** and **energy-**

delay^2 are more appropriate when execution time is also important, as it usually is.

Different systems based on a superscalar processor can have different limitations. For instance, laptops have limitations in their cooling systems, so heat is a constraint (strongly related to power). Additionally, battery lifetime (energy) is also a constraint that must be considered in laptop’s processor design. If the processor is to be used in a desktop or mainframe, then execution time is a significant factor (energy-delay). For the highest performance server-class machines, it may be appropriate to weight the delay part even more (energy-delay²). Thus, we have decided to compare the different schemes using all the metrics described above. The comparison has been normalized to the baseline configuration.

Figure 12 and Figure 13 show the comparison in terms of power and energy respectively for the issue queue. It can be observed that both *MB_distr* and *IF_distr* dissipate much less power and consume much less energy than *IQ_64_64*. Their power and energy requirements are the same for integer applications, but for FP ones *MB_distr* spends a bit more energy.

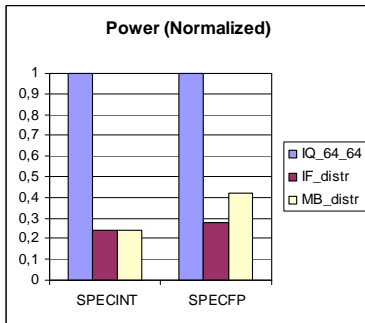


Figure 12. Normalized power dissipation

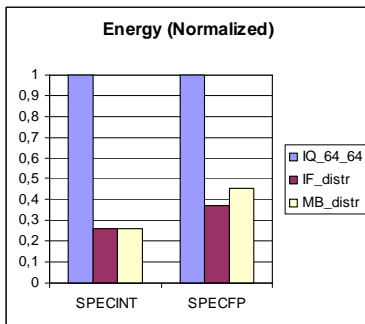


Figure 13. Normalized energy consumption

Figure 14 and Figure 15 compare the different schemes using the energy-delay and energy-delay² metrics for the whole processor considering that the issue queue contribution to the total chip power is 23% [23]. It can be observed in Figure 14 that *MB_distr* outperforms both *IF_distr* and the baseline in energy-delay product for FP applications. The poor performance of *IF_distr* is

basically due to its significant loss in IPC. Figure 15 shows that *MB_distr* significantly outperforms *IF_distr* and achieves practically the same performance as the baseline in terms of energy-delay².

It must be taken into account that the reduced complexity of the issue queue for both *MB_distr* and *IF_distr* schemes may enable a reduction of the cycle time, which may significantly reduce the execution time for these two schemes and thus, significantly improve their energy-delay and energy-delay² metrics with respect to the baseline. Measuring the effect of a shorter cycle time requires a detailed circuit analysis of the whole processor, which is out of the scope of this paper.

We conclude that *MB_distr* results in the best tradeoff among performance, energy and power.

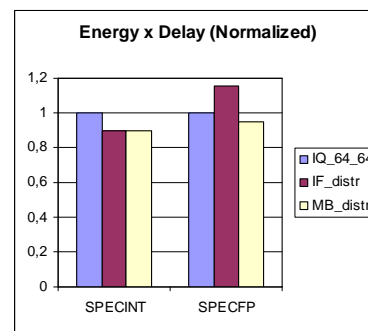


Figure 14. Normalized energy-delay product

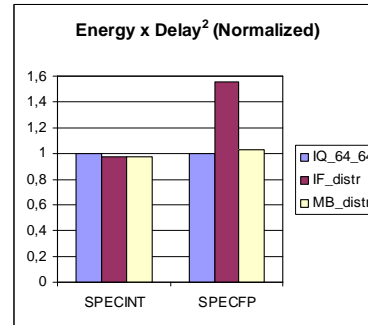


Figure 15. Normalized energy-delay² product

5. Conclusions

This paper presents a low-complexity FP issue queue organization (*MB_distr*) that achieves high performance with small energy requirements. The *MB_distr* scheme is based on dispatching instructions into a set of multiple queues depending on their data dependences at dispatch time. Selection logic is based on estimating the availability time of each operand, instead of the complex and power-hungry conventional wakeup logic. Additionally, the proposed issue logic organization distributes the functional units across the different queues. Thus, the complexity of the crossbar from the issue queue to the functional units is significantly reduced.

The energy required by the proposed *MB_distr* scheme is substantially lower than the energy required by a baseline conventional issue queue, even if the baseline has the capability of spending energy for waking-up only unready operands. This baseline scheme has a much longer delay than *MB_distr*.

It has been shown that *MB_distr* achieves similar energy-delay² product than the baseline and reduces the product by 35% with respect to the *IF_distr* scheme. If the energy-delay metric is used, then the reductions are 5% with respect to the baseline and 18% with respect to the *IF_distr* scheme. The IPC loss of *MB_distr* scheme for FP applications with respect to the high-complexity baseline is only 7.6%, whereas *IF_distr* loses 26%.

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] J. Abella, R. Canal, A. González. Power- and Complexity-Aware Issue Queue Designs. In IEEE Micro, September-October 2003.
- [2] J. Abella, A. González. Power-Aware Adaptive Issue Queue and Register File. In proceedings of the International Conference on High Performance Computing (HiPC'03), December 2003.
- [3] E. Brekelbaum, J. Rupley II, C. Wilkerson, B. Black. Hierarchical Scheduling Windows. In proceedings of the International Symposium on Microarchitecture (MICRO'02), November 2002.
- [4] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J.D. Wellman, V. Zyuban, M. Gupta, P.W. Cook. Power-Aware Microarchitecture: Design and Modelling Challenges for Next-Generation Microprocessors. In IEEE Micro, November-December 2000.
- [5] D. Brooks, V. Tiwari, M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In proceedings of the 27th International Symposium on Computer Architecture (ISCA'00), June 2000.
- [6] M.D. Brown, J. Stark, Y.N. Patt. Select-Free Instruction Scheduling Logic. In proceedings of the International Symposium on Microarchitecture (MICRO'01), December 2001.
- [7] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 3.0. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1999.
- [8] A. Buyuktosunoglu, D. Albonesi, P. Bose, P. Cook, S. Schuster. Tradeoffs in Power-Efficient Issue Queue Design. In proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'02), August 2002.
- [9] 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 proceedings of the 11th Great Lakes Symposium on VLSI (GLSVLSI'01), March 2001.
- [10] R. Canal and A. González. Reducing the Complexity of the Issue Logic. In proceedings of the International Conference on Supercomputing (ICS'01), June 2001.
- [11] R. Canal and A. González. A Low-Complexity Issue Logic. In proceedings of the International Conference on Supercomputing (ICS'00), June 2000.
- [12] S. Cotofana, B. Juurlink, S. Vassiliadis. Counter Based Superscalar Instruction Issuing. In proceedings of the Euromicro Conference, September 2000.
- [13] D. Ernst, T. Austin. Efficient Dynamic Scheduling through Tag Elimination. In proceedings of the 29th International Symposium on Computer Architecture (ISCA'02), June 2002.
- [14] D. Folegnani and A. González. Energy-Effective Issue Logic. In proceeding of the 28th International Symposium on Computer Architecture (ISCA'01), June 2001.
- [15] M. Huang, J. Renau, J. Torrellas. Energy-Efficient Hybrid Wakeup Logic. In proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'02), August 2002.
- [16] G. Kucuk, K. Ghose, D.V. Ponomarev, P.M. Kogge. Energy-Efficient Instruction Dispatch Buffer Design for Superscalar Processors. In proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'01), August 2001.
- [17] A.R. Lebeck, J. Koppalil, T. Li, J. Patwardhan, E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In proceedings of the 29th International Symposium on Computer Architecture (ISCA'02), June 2002.
- [18] P. Michaud, A. Sez nec. Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. In proceedings of the International Symposium on High Performance Computer Architecture (HPCA'01), January 2001.
- [19] S. Palacharla, N.P. Jouppi, J.E. Smith. Complexity-Effective Superscalar Processors. In proceedings of the 24th International Symposium on Computer Architecture (ISCA'97), June 1997.
- [20] S.E. Raasch, N.L. Binkert, S.K. Reinhardt. A Scalable Instruction Queue Design Using Dependence Chains. In proceedings of the 29th International Symposium on Computer Architecture (ISCA'02), June 2002.
- [21] 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.
- [22] E. Sprangle. Personal Communication.
- [23] K. Wilcox, S. Manne. Alpha Processors: a History of Power Issues and a Look to the Future. In Cool-Chips Tutorial, November 1999. Held in conjunction with MICRO-32.
- [24] SPEC2000. <http://www.specbench.org/osg/cpu2000>