Defining Wakeup Width for Efficient Dynamic Scheduling

Aneesh Aggarwal
ECE Department
Binghamton University
Binghamton, NY 13902
aneesh@binghamton.edu

Manoj Franklin
ECE Department and UMIACS
University of Maryland
College Park, MD 20742
manoj@eng.umd.edu

Oguz Ergin
CS Department
Binghamton University
Binghamton, NY 13902
oguz@cs.binghamton.edu

Abstract

A larger Dynamic Scheduler (DS) exposes more Instruction Level Parallelism (ILP), giving better performance. However, a larger DS also results in a longer scheduler latency and a slower clock speed. In this paper, we propose a new DS design that reduces the scheduler critical path latency by reducing the *wakeup width* (defined as the effective number of results used for instruction wakeup). The design is based on the realization that the average number of results per cycle that are immediately required to wake up the dependent instructions is considerably less than the processor issue width. Our designs are evaluated using the simulation of the SPEC 2000 benchmarks and SPICE simulations of the actual issue queue layouts in 0.18 micron process. We found that a significant reduction in scheduler latency, power consumption and area is achieved with less than 2% reduction in the Instructions per Cycle (IPC) count for the SPEC2K benchmarks.

1 Introduction

In current microprocessor designs, dynamic scheduling is one of the most important techniques used to extract Instruction Level Parallelism (ILP). The dynamic scheduler (DS) logic consists of the wakeup logic, which marks the instructions when their dependencies are satisfied, and the select logic, which selects the instructions to execute from the pool of ready instructions. Both of these operations generally occur within a single cycle, forming a critical path [15, 17]. Studies [4, 19] have shown that an increase of a single cycle in this critical path decreases the performance dramatically.

To stay on the microprocessor industry's growth curve, future microprocessors must schedule and issue larger number of instructions from a larger instruction window [18]. While more ILP can be extracted from a larger DS, this increase in parallelism will come at the expense of a slower scheduler clock speed [6, 15]. Increasing wire latencies in smaller technologies may even further diminish the benefits obtained from a larger DS. The poor scalability of the DS logic results because of the wire delays in driving the register tags (identities of instructions' results) across the instruction window and the wire delays in driving the tag comparison (between the register tags and the source operand tags of the instructions) results across the width of the register tags.

In this paper, we propose a scheduler design that reduces the scheduler critical path latency by reducing the processor *wakeup width*. We define *wakeup width* as the number of register tags used to wakeup the dependent instructions. In a conventional DS, the wakeup width is usually equal to the issue width [15]. In our DS design, the processor wakeup width is reduced, without reducing the processor issue width. Reducing the wakeup width also reduces the scheduler energy consumption and area. The new scheduler design is based on the observation that the average number of immediately useful register tags produced per cycle is considerably less than the processor issue width. We found a significant reduction in scheduler latency, energy consumption, and area, with less than 2% reduction in the IPC for SPEC2K benchmarks.

The rest of the paper is organized as follows. Section 2 discusses the intuition behind the new scheduler design. Section 3 presents the scheduler design and its implementation. Section 4 discusses the experimental results. Section 5 presents enhancements to the basic design. Section 6 gives related work, and in Section 7, we conclude.

2 Program Behavior Study

Not all the instructions that are issued produce a result. Branch and store instructions (Figure 1(ii) later shows that they form about 30% of the total instructions) do not produce a register tag. The entire issue width of a processor is also not used in every cycle. This means that the average number of tags generated per cycle could be considerably less than the processor issue width. Figure 1(i) presents the data on register tags generated in each cycle. In Figure 1(i), the total cycles are divided into cycles where 0 or 1 tags are generated, where 2 or 3 tags are generated, and so on. For these measurements, we use the default processor parameters given in Table 1 (on page 4). However, to study a somewhat worst-case scenario (*i.e.* generate more tags per cycle), we increase the fetch, issue, and commit width to 12. From Figure 1(i), it is observed that, even for a very wide processor, almost 50% of the cycles (on an average) have either no or just 1 tag generated, and about 80% of the cycles have less than or equal to 3 tags generated. The number of tags generated are higher for the FP benchmarks because of their higher IPCs.

Among the tags generated in a cycle, not all the tags are of immediate use. With branch mispredictions, many tags are either generated along the wrong execution path (which are not useful at all) or generated along the correct path but only used along the wrong execution path (which are not useful until the instructions from the correct path are fetched). Tags are also not immediately useful if the dependent instructions are either not present in the instruction window or are waiting for other operands. In effect, the average number of useful tags generated in a cycle are even less than the average number of tags generated in a cycle. Figure 1(ii) presents the statistics on the percentage of instructions that produce a tag that is immediately useful. In Figure 1(ii), the instructions are divided into different categories (shown as stacks in each bar) based on the types of tags generated by the instructions. Figure 1(ii) shows that only about 50-60% of the instructions produce a tag that is immediately required, except for a few exceptions in the FP benchmarks. About 8% of the instructions produce tags that are not required immediately, and about 8% produce tags that are either produced and/or only used along the wrong path.

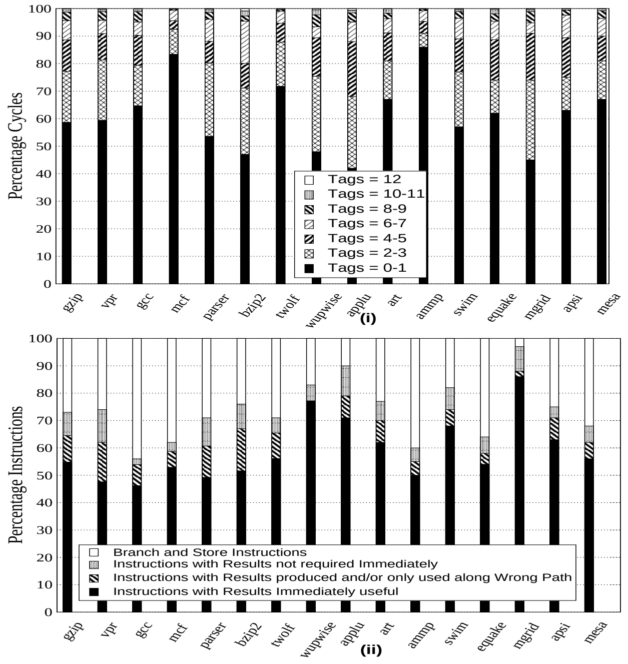

Figure 1: (i) Percentage Distribution of Clock Cycles based on the number of tags generated in a cycle; (ii) Percentage Distribution of Executed Instructions based on the requirements of their results

3 Reduced Wakeup Width

We define *wakeup width* as the number of register tags used to wakeup dependent instructions.

3.1 Basic Idea

In a conventional dynamic scheduler, the wakeup width is equal to the issue width [15], so that any register tag that is produced can immediately wakeup the dependent instructions. Section 2 showed that, even for a very wide machine, only about 50-60% of the instructions that are issued produce an immediately useful register tag. Hence, we propose a *Reduced Wakeup Width (RWW)* dynamic scheduler design, in which the wakeup width is reduced without reducing the processor issue width. This may result in some of the tags having to wait before waking up the dependent instructions. The performance impact of the delays due to waiting tags is not expected to be high because, as Figure 1(i) showed, soon there will be cycles with fewer tags produced, and the waiting register tags can use the available wakeup slots (the number of wakeup slots is equal to the processor wakeup width). On the other hand, the delayed tags that are not immediately useful may not even have any performance impact.

3.2 Hardware Implementation

In a conventional scheduler (Figure 2(i)), every ready instruction requests a functional unit (FU) from the select logic. The select logic decides which instruction executes on which FU. The issued instructions and the register tags of their results, if any, are placed in *tag latches*. In the next cycle, the instructions from the *tag latches* are dispatched to the FUs, and their register tags (for the ones that produce a result) are driven (using tag-lines) to wakeup the dependent instructions. The *enable* signals of the drivers are controlled by some mechanism that indicates the presence of tags in the *tag latches*. For this, we assume one 1-bit latch (called *indicator latch*) for each *tag latch*, as shown in Figure 2(i), which is set when a tag is latched in the *tag latch* and reset when the tag is used. The instructions that now become ready request for FUs from the select logic, and the process goes on.

Figure 2(ii) shows the *RWW* dynamic scheduler design with wakeup width half the issue width. In this case, 2 *tag latches/FUs* share common tag-lines. If both the *tag latches* sharing the tag-lines are holding tags, only one of the tags is driven through the tag-lines, and the other remains latched. For this, the *enable* signals of the drivers are appropriately set, using control logic shown in the figure. In the *RWW* design, it may happen that a waiting tag may be over-written in the next cycle if the select logic issues a result-producing instruction to that *tag latch*. To avoid this, if the *indicator latch* of a *tag latch* is set, then no instruction is issued to that particular *tag latch*, wasting some of the issue slots.

In the select logic, each functional unit (FU) is provided with an arbiter which looks at requests from the instructions and decides which instruction is to be executed on that FU[1]. It then raises the *grant* signal for that request. Figure 3(a) shows the detailed circuit for *grant1* grant signal generation (corresponding to the *req1* request signal) inside a FU arbiter. This circuit implements:

$$grant1 = \overline{req0} \cap req1 \cap enable \qquad (1)$$

[1]Usually the giving the highest priority to the oldest instruction.

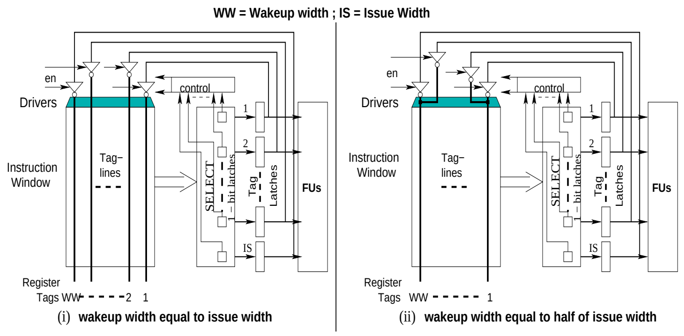
WW = Wakeup width ; IS = Issue Width
en
Drivers
Instruction Window
Tag–lines
control
SELECT
1–bit latches
Tag Latches
FUs
IS
Register Tags WW 2 1
(i) wakeup width equal to issue width
(ii) wakeup width equal to half of issue width

Figure 2: (i) Conventional Dynamic Scheduler; (ii) Reduced Wakeup Width (RWW) Dynamic Scheduler

In this equation, *req0* has a higher priority than *req1*. In Figure 3(a), the logic pre-computes the priority and if *req1* is the highest priority signal present, it raises the *grant1* signal when the enable signal is received [16].

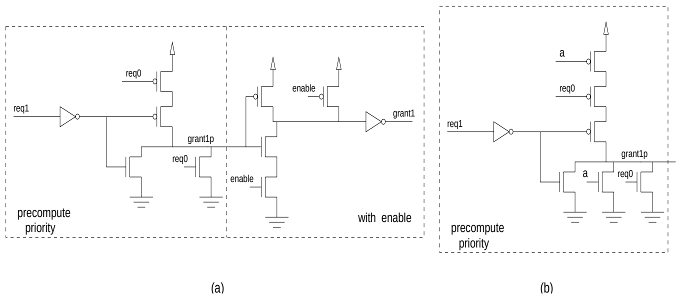

Figure 3: (a) Conventional FU Arbiter; (b) FU Arbiter in the RWW Scheduler Design

Figure 3(b) shows the logic that precomputes priority in the FU-arbiter for the *RWW* scheduler design. Here "a" is the value of the *indicator* latch for the *tag latch* corresponding to this arbiter. This circuit implements:

$$grant1 = \overline{req0} \cap \overline{a} \cap req1 \cap enable \qquad (2)$$

The select logic is implemented as a tree of *arbiter cells* [16], where the request signals (from the instructions) travel towards the root of the tree and the grant signals in the opposite direction. In the new *arbiter cell* design, the additional transistors are only in the *pre-compute* part of the cells that are at the bottom of the tree and hence, the additional transistors will not affect the delay in the select logic. In the *RWW* design, we may have to slightly modify the generation of operand forwarding signal for instructions woken up by delayed register tags.

4 Experimental Evaluation

4.1 Experimental Methodology

For our experiments, we use the SimpleScalar [5] simulator simulating the PISA architecture. Table 1 gives the default parameters. For benchmarks, we use 7 INT (bzip2, gzip, gcc, vpr, mcf, parser, and twolf) and 9 FP (wupwise, applu, art, ammp, swim, equake, mgrid, apsi, and mesa) programs from the SPEC2K suite. Performance statistics are collected for 500M instructions after skipping the first 500M instructions. Loads and stores are issued out-of-order using dynamic memory disambiguation. For delay, energy, and area estimation of the CAM logic of the issue queue, measurements were made from the actual VLSI layouts using SPICE. CMOS layout for the CAM based issue queue and select logic in a 0.18 micron 6 metal layer CMOS process (TSMC) were used to get an accurate idea of the energy dissipations for each type of transition, assuming a Vdd of 1.8 volts.

For an issue width of 6, we experiment with wakeup widths of 3 and 2. Each scheduler configuration is given a name $I < issuewidth > W < wakeupwidth >$. For instance, I6W3 represents a configuration with an issue width of 6 and a wakeup width of 3. We compare the results of the *RWW* technique against the *I6W6* configuration, and against configurations with issue width equal to the reduced wakeup width (to compare the IPCs of configurations with almost the same scheduler latency). For instance, *I6W3* and *I3W3* configurations are compared.

4.2 Performance Results

We use the analysis in [16] and detailed logic layouts to evaluate the *RWW* scheduler latency. In the *RWW* design, the wires carrying the register tags and he wires carrying the result of tag comparisons become shorter, reducing the wakeup logic latency, and hence the scheduler latency. When compared to the I6W6 configuration, wakeup logic latency reduces by about 40% for I6W3, and by about 55% for I6W2, based on the analysis in [16]. If a stacked design is used for the select logic (which can be very expensive in terms of delay), the reduction in the overall DS logic delay is about 10% for the *I6W3* configuration and about 15% for the *I6W2* configuration. However, in our detailed layouts, the wakeup logic latency reduced by about 15% and 20%, respectively.

Figure 4 gives the IPCs as overlapping stacks in each bar. From Figure 4, it is observed that the IPC impact is higher for the FP benchmarks, mainly because of their higher IPCs. The IPC impact for I6W2 is also significantly higher than I6W3. *RWW* configuration performs significantly better when the IPCs obtained with configurations having almost the same scheduler latency are compared, (*e.g.* I6W3 with I3W3 and I6W2 with I2W2).

4.3 Performance Analysis

Figure 5(i) presents the percentage distribution of all the instructions (in terms of the reasons due to which the instructions are delayed) that are executed, and Figure 5(ii) presents the percentage distribution of all the tags (in terms of the effects of delayed tags) that are generated. The following observations can be made from Figure 5:

- With decreasing wakeup width, more number of tags and instructions are delayed, resulting in IPC reduction (in Figure 4) as the wakeup width is reduced.

Parameter	*Value*	*Parameter*	*Value*
Issue Width	6 instructions/cycle	*Instruction Queue Size*	128 instructions
Physical Register File	128 Int/128 FP	*Fetch/Commit Width*	8 instructions/cycle
L1 - I-Cache	32K, direct-mapped 2-cycle latency	*L1 - D-Cache*	32K, 4-way assoc. 2-cycle latency
L2 - Cache	unified 512K, 8-way assoc. 6-cycle latency	*Mem. Reorder Buffer*	64 entries
realistic branch pred.	bimodal 4096 entries 20-cycle mispred. pen.	*Functional Units*	Int. : 3 ALU, 1 Div/Mult. 2 load/store FP : 3 ALU, 1 Div/Mult.

Table 1: Default Parameters for the Experimental Evaluation

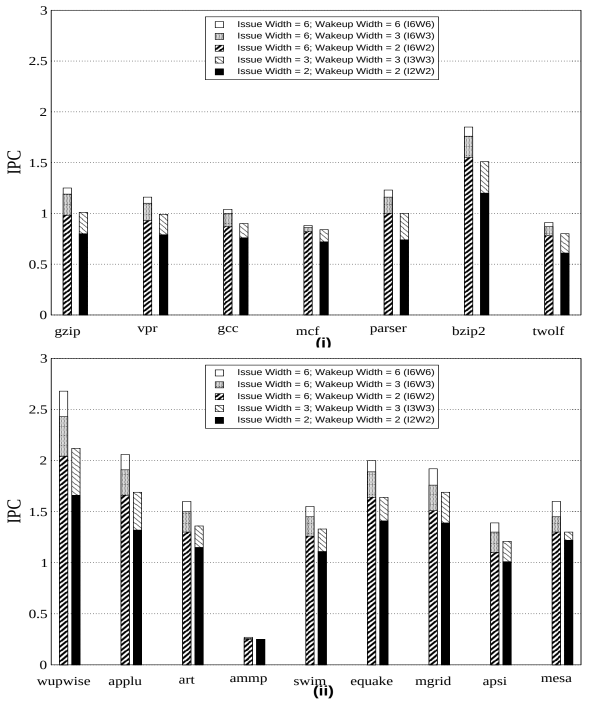

Figure 4: IPC obtained with different Configurations of the *RWW* Design; (i) Integer, and (ii) FP Benchmarks

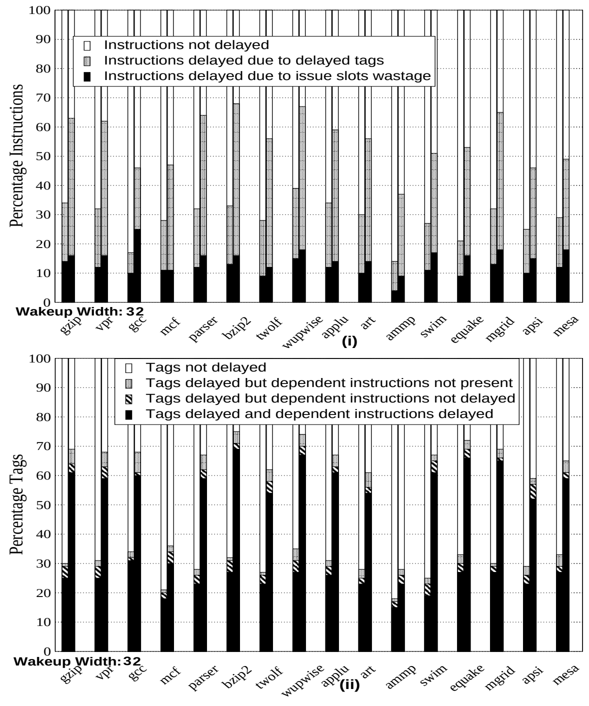

Figure 5: (i) Percentage Distribution of Instructions based on their reasons of delay; (ii) Percentage Distribution of Generated Tags based on the effects of the tags

- Lower IPC benchmarks have lower percentage of tags and instructions delayed, resulting in smaller IPC impact. High IPC benchmarks generally have a higher percentage of tags and instructions delayed, resulting in a larger IPC impact. Exceptions are `gcc` and `equake`, that have a high percentage of tags delayed, but a low percentage of instructions delayed. This is because, as Figure 1(ii) shows, both `gcc` and `equake` have a higher percentage of branches and stores, which results in less tag generation.

- Figure 5(i) shows that more instructions are delayed due to delayed register tags than due to the wastage of issue slots. This implies that delayed register tags have more impact on the IPC than wastage of issue slots. In addition, with decrease in the wakeup width, instructions delayed due to delayed tags increases dramatically. This is because, with less wakeup width, fewer instructions get ready every cycle (also depicted by lower IPC) and more register tags are delayed. The small number of instructions that do get ready usually find a functional unit to execute and get issued, but the large number of delayed tags delay many more instructions.

4.4 Energy and Area Results

In order to study the energy savings in the wakeup logic (which forms the major fraction of the total scheduler energy consumption [8]) achieved with the *RWW* design, we use the number of accesses to the wakeup logic obtained using [5] and the energy-consumption values from our detailed layouts. The main reason for high wakeup logic energy consumption is the high capacitance tag-lines and match-lines [3]. Energy saving of about 10% is observed in the wakeup logic for the *I6W3* configuration and about 15% for the *I6W2* configuration. Intuitively, reducing the wakeup width reduces the lengths of tag-lines (because of reduction in the CAM cell height) and match-lines which in turn reduces their capacitance. The *RWW* design also reduces the number of high capacitance tag-lines and match-lines, and the number of comparators for tag comparison. Energy saving values obtained are not very high because of conditional clocking, where the tag-lines and match-lines that are not active are assumed to consume almost no energy. Hence, removal of these lines does not save as much energy as expected.

For the I6W3 configuration, the area of the CAM cells, and hence the area of the tag part of the instruction window, reduces by about 30%, and by about 40% for the I6W2 configuration. However, other fields in the instruction window such as the ROB index, and the literal fields reduce the area savings to about 20% for the I6W3 configuration and about 30% for the I6W2 configuration.

5 Enhancing the RWW Design

5.1 Reduced Issue Slots Wastage (RWIS)

Wastage of issue slots results because the select logic does not issue any instruction to the FUs with waiting tags. We propose a technique that classifies the instructions into: *tag-producing* instructions (that generate a register tag) and *non-tag-producing* instructions (that do not generate a register tag), and even if a tag is waiting, the select logic still issues a *non-tag-producing* instruction to that FU. In doing so, the waiting register tag is not over-written. To implement this technique, an additional bit (we call it *type* bit) is used for each instruction in the instruction window. The *type* bit is set (*tag-producing*) or reset (*non-tag-producing*) when the instruction is dispatched. When an instruction sends a request (for an FU) to the select logic, it is also accompanied by the *type* bit, the equation (2) for the *grant1* signal now becomes:

$$grant1 = \overline{req0} \cap (\overline{a \cap b}) \cap req1 \cap enable \qquad (3)$$

where *b* is the *type* bit. In this case as well, no additional delays were observed in the select logic.

5.2 Reduced Tag Delays (RTD)

Register tags are delayed if multiple *tag-producing* instructions are issued to FUs sharing common tag-lines (we call such a group as an *FU-group*). If most of the *tag-producing* instructions are concentrated in a few *FU-groups*, the number of register tags getting delayed increases. Ideally, the *tag-producing* instructions should be evenly distributed among the *FU-groups*. To implement the *RTD* technique, we limit the number of *tag-producing* instructions that are issued to a single *FU-group*. It may be difficult to simultaneously count and limit the number of tag-producing instructions issued to an *FU-group* in the same cycle. Hence, the count of waiting tags of the previous cycle is used. If the count for an *FU-group* is equal to or more than the limit, then the *indicator* bit a is set for all the *tag-latches* in that *FU-group*. The counting is done in parallel to instruction selection in each cycle, and the setting of the *indicator* bits is done in parallel to instruction wakeup in each cycle. The *non-tag-producing* instructions can still be issued to FUs with the *indicator* bit set. If the count for an *FU-group* is less than the maximum, then the *indicator* bits are untouched. This may occasionally result in the number of register tags waiting in an *FU-group* being more than the limit.

5.3 Performance

Figure 6 presents the IPCs with the *RWIS* and the *RTD* techniques. For the *RTD* technique, we experiment with a limit of 1 waiting tag per *FU-group* (*RTD-1*) and 2 waiting tags per *FU-group* (*RTD-2*). Figure 6 shows that the IPC impact of the *RWW* design reduces with these techniques. In fact, with the *RTD-1* technique, compared to the I6W6 configuration, the IPC of the I6W3 configuration is less than 2% lower and the IPC of the I6W2 configuration is less than 5% lower (for many benchmarks).

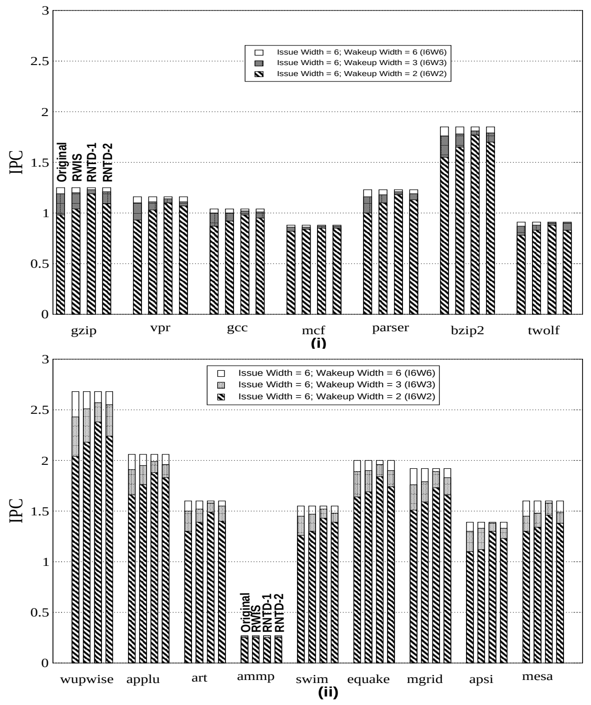

Figure 6: IPCs with *RWIS* and *RTD* techniques applied to the *RWW* Design (i) Integer; (ii) FP benchmarks

Figure 6 shows that the *RTD-1* technique is more effective than any other technique, because *RWIS* helps in only reducing the wastage of issue slots whereas, *RTD* helps in reducing the number of delayed register tags as well, and with *RTD-2* more tags are allowed to wait per *FU-group* resulting in more instructions getting delayed. Our studies also found that, with the *RTD-1* technique the total number of instructions delayed is the least resulting in the least impact on IPC when using this technique. We also found that, for the *RTD-1* technique, even though the number of instructions delayed due to waiting tags decrease significantly, the instructions delayed due to wastage of issue slots increase when compared to the *RWIS* technique, because more issue slots are wasted due to the limit on the tags that can wait in any *FU-group*.

6 Related Work

Both micro-architectural and circuit-level solutions have been provided to implement reduced latency dynamic schedulers. In [1, 2, 7, 15, 21, 13], the instruction queue is distributed among multiple clusters. This solution trades global communication for fast local communication. A recently proposed circuit-level approach [11] merges the reorder buffer and the issue buffer, and uses parallel-prefix circuits for wakeup and selection phases.

Dependence-based pre-scheduling is proposed in [15, 6, 14, 17], with different implementations, to keep the top-level buffer used for waking up dependent instructions small. Hrishikesh et al [12] also propose segmented instruction windows to reduce scheduler latency, where instruction wakeup is pipelined among the segments. Folegnani and Gonzalez [8] focus on "gating off" entries to reduce power consumption in dynamic schedulers.

Our design reduces hardware complexity for efficient dynamic scheduling, which can also be used in conjunction with clock gating and data dependent analysis.

7 Conclusions

Larger dynamic schedulers are required to exploit more Instruction Level Parallelism (ILP) to increase the Instruction per Cycle (IPC) count, and hence increasing performance. While more ILP can be exploited by larger dynamic schedulers, this increase in parallelism comes at the expense of a slower scheduler clock speed.

In this paper, we proposed a reduced wakeup width (*RWW*) dynamic scheduler design to reduce the scheduler critical path latency by reducing the maximum number of register tags that can be used for identifying data-ready instructions. The design exploits the observation that the average number of useful register tags produced every cycle is much less than the processor issue width. Our studies showed that significant reduction in wakeup logic latency can be achieved with less than 5% reduction in IPC. The design also reduced the wakeup logic energy consumption by about 15%, and the instruction window area by about 30%. Our enhancements to the basic *RWW* scheduler design, reduce the IPC impact to less than 2%.

References

[1] A. Aggarwal and M. Franklin, "An Empirical Study of the Scalability Aspects of Instruction Distribution Algorithms for Clustered Processors," *Proc. ISPASS*, 2001.

[2] A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors," *Proc. MICRO-33*, 2000.

[3] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," *Proc. ISCA*, 2000.

[4] M. Brown, J. Stark and Y. Patt, "Select-free Instruction Scheduling Logic," *Proc. Micro-34*, 2001.

[5] D. Burger and T. Austin, "The Simplescalar Tool Set," *Technical Report*, Computer Sciences Department, University of Wisconsin, June 1997.

[6] R. Canal and A. Gonzalez, "A low-complexity issue logic," *Proc. ICS*, 2000.

[7] K. Farkas, et. al., "The Multicluster Architecture: Reducing Cycle Time Through Partitioning," *Proc. Micro-30*, 1997.

[8] D. Folegnani and A. Gonzalez, "Energy-Effective Issue Logic," *Proc. ISCA-28*, 2001.

[9] M. K. Gowan, L. L. Biro and D. B. Jackson, "Power Considerations in the Design of the Alpha 21264 Microprocessor," *Proc. Design Automation Conference*, 1998.

[10] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Microprocessors," *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 9, September 1996.

[11] D. S. Henry, et. al., "Circuits for wide-window superscalar processors," *Proc. ISCA-27*, 2000.

[12] M. S. Hrishikesh et. al., "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," *Proc. ISCA*, 2002.

[13] D. Leibholz and R. Razdan, "The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor," *Proc. Compcon*, pp. 28-36, 1997.

[14] P. Michaud and A. Seznec, "Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors," *Proc. HPCA*, 2001.

[15] S. Palacharla, et. al., "Complexity-Effective Superscalar Processors," *Proc. ISCA*, 1997.

[16] S. Palacharla, "Complexity-Effective Superscalar Processors," *PhD thesis*, University of Wisconsin, 1998.

[17] S. Raasch, et. al., "A Scalable Instruction Queue Design Using Dependence Chains," *Proc. ISCA*, 2002.

[18] Y. N. Patt, et al., "One Billion Transistors, One Uniprocessor, One Chip," *IEEE Computers*, pp. 51-57, Sept. 1997.

[19] E. Sprangle and D. Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines," *Proc. ISCA-29*, 2002.

[20] V. Tiwari, et al., "Reducing Power in High-performance Microprocessors," *Proc. DAC*, 1998.

[21] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, pp. 28-40, April 1996.