Energy-Efficient Hybrid Wakeup Logic*

Michael Huang, Jose Renau, and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ABSTRACT

The instruction window is a critical component and a major energy consumer in out-of-order superscalar processors. An important source of energy consumption in the instruction window is the instruction wakeup: a completing instruction broadcasts its result register tag and an associative comparison is performed with all the entries in the window.

This paper shows that a very large fraction of the completing instructions have to wake up no more than a single instruction currently in the window. Consequently, we propose to save energy by using indexing to only enable the comparator at the single instruction to wake up. Only in the rare case when more than one instruction needs to wake up, our scheme reverts to enabling all the comparators or a subset of them. For this reason, we call our scheme *Hybrid*. Overall, our scheme is very effective: for a processor with a 96-entry window, the number of comparisons performed by the average completing instruction with a destination register is reduced to 0.8. The exact magnitude of the energy savings will depend on the specific instruction window implementation. Furthermore, the application suffers no performance penalty.

Categories & Subject Descriptors:

C.0 Computer System Organization: System Architectures.
C.1.1 Single Data Stream Architectures: RISC/CISC,VLIW Architectures
C.5.3 Microcomputers: Microprocessors.

General Terms: Design, Experimentation, Performance

Keywords: Low Power, Wakeup Logic, Issue Logic

1. INTRODUCTION

The instruction window is a central piece of the machinery required to support out-of-order execution in modern high performance processors. Instructions are inserted in the instruction window after their registers have been renamed to eliminate output and anti dependences. Instructions wait in the window until their operands are ready, at which point true dependences can be satisfied and the instructions can be issued to the functional units for execution.

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; and by gifts from IBM, Intel, and Hewlett-Packard.

ISLPED'02, August 12-14, 2002, Monterey, California, USA.

When an instruction finishes its execution, its destination register number is typically broadcasted to all the instructions in the window. The purpose is to inform all dependent instructions of the availability of the result. Each entry compares the broadcasted tag to its own source register tags. If there is a match, the source operand is latched and the dependent instruction may be ready to execute.

This tag broadcast and associative comparison is called instruction wakeup. It requires driving a register tag through long wires and comparing it to all the source register tags in the instruction window. Consequently, it can consume substantial energy.

To reduce the energy consumed by instruction wakeup, Folegnani and Gonzalez have recently proposed several optimizations [5]. Specifically, window entries that are either empty or contain instructions that already have all their source operands available, are not compared against the broadcasted register number. Furthermore, the size of the instruction window is dynamically adjusted to reduce empty area. According to [5], by disabling comparisons to empty and ready entries in a 128-entry window, we decrease the number of comparisons performed by the average completing instruction to only 14.2.

While this is a very significant reduction, our data indicates that we can further reduce the number of comparisons per finishing instruction. Indeed, we will show that the large majority of completing instructions have only one single dependent instruction in the window. Furthermore, the location of the dependent instruction can be easily recorded when the latter is inserted into the instruction window. Consequently, we propose to remember the location of the dependent instruction, and use indexing to enable only the comparator in the single dependent instruction. This is much more energy efficient. In the rare case when more than one instruction needs to wake up, we revert to enabling all the comparators or a subset of them. This scheme, which we call *Hybrid*, is very effective: for a processor with a 96-entry window, the number of comparisons performed by the average completing instruction is reduced to 0.8.

The rest of the paper is organized as follows: Section 2 discusses the rationale for our optimization, Section 3 shows the microarchitectural design, Section 4 presents the evaluation environment, Section 5 evaluates the design, Section 6 discusses related work and, finally, Section 7 concludes.

2. RATIONALE

Figure 1 shows the typical operation of the instruction window. An instruction window has many entries, each one corresponding to one instruction. The information recorded in an entry includes the opcode and the physical register numbers (also called tags) for the source operand(s) and the destination. There is one ready bit for each source operand. Once all the ready bits for the instruction are set, the instruction is ready to be executed. The arbiter selects ready instructions and sends them to the proper functional units

for execution. Right before an instruction finishes execution, the destination register number is broadcasted back to the instruction window to wake up the waiting instructions. This broadcast and the resulting many comparisons typically consume substantial energy.

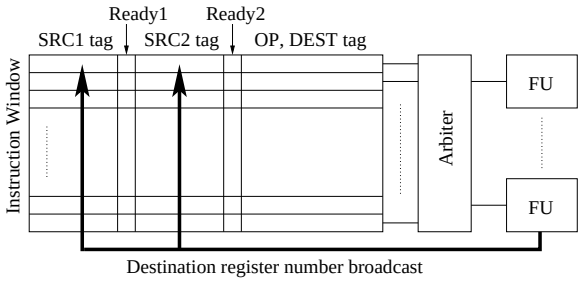

Figure 1: Typical operation of the instruction window.

Comparisons are needed because many instructions may depend on the result of a single instruction. However, if we look at ordinary code, not many instructions have multiple dependent instructions. For example, as shown in Figure 2, we find that in a normal dynamic instruction stream (of the applications detailed in Section 4), around 70% of the instructions with a destination register [1] have at most one dependent instruction. Furthermore, if we only consider dependent instructions that *are in the window* when the producer completes, the fraction of instructions with at most one dependent is significantly higher. This is because an instruction may have a dependent so far down the instruction stream that, by the time the instruction finishes, the dependent is not yet in the instruction window and, therefore, does not need to be woken up by the producer. We call the dependent instructions that are in the window *close-by* dependents.

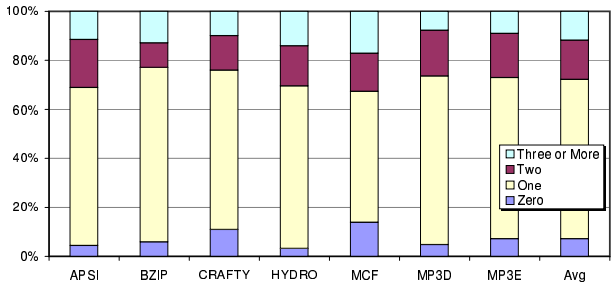

Figure 2: Breakdown of dynamic instructions based on the number of dependent instructions that they have.

The fraction of instructions that have one or no close-by dependent depends on the compiler and the actual processor. As an example, Figure 3 shows the fraction of instructions that have one or no close-by dependent in a 96-entry window. We consider two processor architectures, namely *Normal* and *Slow Memory*. *Normal* models a typical workstation with a high-end processor, as detailed in Section 4. *Slow Memory* simply doubles the latency and occupancy for each level of the memory hierarchy to factor in the widening speed gap between memory and processor. As can be seen, in *Normal*, on average 91.3% of the completing instructions have at most one close-by dependent instruction. Moreover, widening the gap between the processor and memory speed does not change the result much.

[1] Branches and stores do not have a destination register.

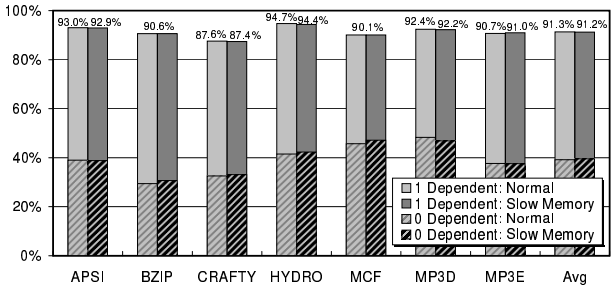

Figure 3: Percentage of completing instructions with no more than one close-by dependent instruction in a 96-entry window.

This measurement suggests that a plain broadcast-based wakeup mechanism can be improved upon. If each entry in the instruction window had a single pointer pointing to a close-by dependent instruction, in most of the cases, using that pointer would suffice to wakeup all the necessary instructions.

3. ENERGY-EFFICIENT WAKEUP LOGIC

To reduce the energy consumed by instruction wakeup, we propose to use indexing. Furthermore, given the measurements of Section 2, we optimize the micro-architecture for single-instruction wakeup. The resulting system is very energy-efficient because it eliminates the need to activate comparators in every entry of the instruction window. Instead, only one comparator is activated in the common case.

In the following, we present the micro-architecture and then consider instructions that have to wake up more than one instruction. We also examine other issues.

3.1 Proposed Micro-Architecture

Our scheme adds one field to the Register Alias Table (RAT), and one field and one bit to the instruction window. These extensions are shown in solid shaded pattern in Figure 4.

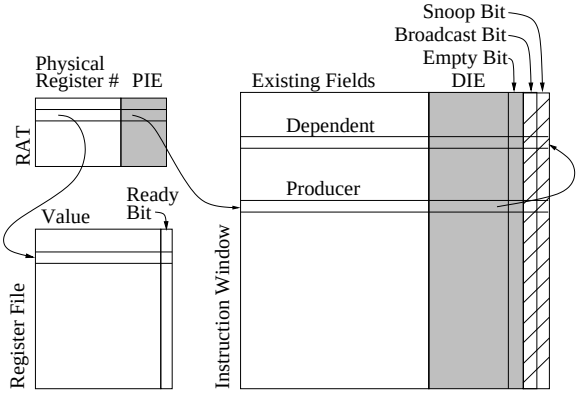

Figure 4: Proposed micro-architectural extensions.

Typically, when an instruction is about to be inserted in the instruction window, a check is made to determine whether or not its source registers are available. This can be done by checking some state in the register file or RAT. For illustration purposes, Figure 4 uses a ready bit per register in the register file. If any of the source registers is not ready, the instruction waits in the window until all its remaining source operands are produced, at which point the instruction is ready to execute.

In our optimized micro-architecture, when an instruction is inserted in the instruction window and any of its source registers is not ready yet, we take a special action. For each source register not ready, we identify the entry of the producer instruction in the window. Then, in that entry, we store a pointer to the consumer instruction. The pointer is simply the index of the consumer instruction in the window. The pointer is stored in a new *Dependent Instruction-window Entry* (*DIE*) field (Figure 4). In addition, the *Empty* bit of the producer instruction is reset.

Note that we can easily identify the producer instruction for any source register that is not ready. This is because, as shown in Figure 4, we add a new field in the RAT called the *Producer Instruction-window Entry* (*PIE*). The PIE points to the entry in the window for the instruction that will produce the corresponding register. The PIE is set when the producer instruction is inserted into the instruction window.

With this support, instructions proceed as follows. When an instruction is decoded and inserted into the instruction window, if any of its source registers is not ready, we follow the *PIE* pointer(s) to find the producer instruction(s). Then, the *DIE* pointers of such instructions are set to point to the entry in the window corresponding to the consumer instruction. Later, when a producer instruction completes, it passes its *DIE* pointer through a decoder to only enable the comparator in the consumer instruction and wake it up. Such a selective wakeup saves energy.

3.2 Waking Up More Than One Instruction

A scheme with a single *DIE* pointer per instruction window entry can only support instructions that have at most one dependent instruction in the window. To handle the much less likely case of instructions with more than one close-by dependent instruction, we propose two possible approaches.

The first solution we propose involves stalling. We call this scheme *Indexing-Only*. In this scheme, when an instruction that is about to be inserted into instruction window finds that at least one of its producers has the *Empty* bit reset, it is not inserted, and instruction decoding is stopped until the producer executes. This mechanism is similar to the case when the system runs out of renamed registers.

The second solution we propose involves reverting back to broadcasting. We call this approach *Hybrid*. For this approach, we need the additional hardware shown in the striped area in Figure 4: the *Broadcast* bit and (optionally) the *Snoop* bit in the instruction window. In this case, when an instruction is being inserted into instruction window, the *Empty* bit of its producer instruction(s) is checked. If the *Empty* bit is reset for one producer, the hardware sets the *Broadcast* bit of that producer. The same is done for the other producer, if any. This bit will force the producer instruction to broadcast its result tag to all the instructions in the window upon completion. We call this scheme *Hybrid-Plain*.

We can improve on the *Hybrid-Plain* scheme with the use of the *Snoop* bit. This bit can gate unnecessary comparisons. With this bit, when a result tag is broadcasted to all instructions, only the entries with the *Snoop* bit set will actually perform the comparison. Consequently, when we set the *Broadcast* bit for a producer instruction, we also set the *Snoop* bit for two instructions: the newly-inserted consumer instruction and the one pointed to by the *DIE* pointer of the producer. If the *Broadcast* bit was already set, then we only need to set the *Snoop* bit for the newly-inserted consumer. With this support, when a producer instruction broadcasts a result tag, only a very small number of the entries (those with the *Snoop* bit set) will actually perform the comparison. We call this scheme *Hybrid-Snoop*.

A third alternative could be to add a second *DIE* pointer per window entry. However, such a solution is not very attractive: although we increase the number of cases handled without broadcast, we still have to handle the case of not having enough *DIE* pointers. Moreover, the hardware gets significantly more complicated. One way to reduce the hardware complexity would be to support only the case where one of the dependent instructions needs the result through the first source register, and the other dependent instruction needs it through the second source register. This case is easier to support than having the two *DIE* pointers point to any instruction without restrictions. Unfortunately, as shown in Figure 5, only around 1% of the dynamic instructions actually have exactly two close-by dependent instructions, each sourcing the result through a different source register. The figure corresponds to the *Normal* architecture of Figure 3. Consequently, we discard the approach of using multiple *DIE* pointers per window entry.

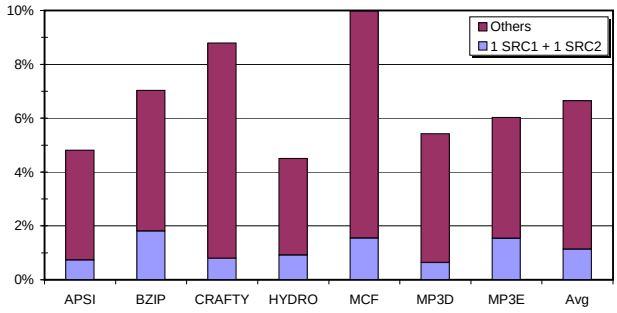

Figure 5: Percentage of instructions that have more than one close-by dependent instruction. *1 SRC1 + 1 SRC2* corresponds to instructions with exactly two close-by dependent instructions, one of which sourcing the result through *SRC1*, and the other through *SRC2*. *Others* corresponds to the other cases of two close-by dependents and more than two close-by dependents.

3.3 Handling Branch Mispredictions

The proposed schemes easily support branch misprediction. Indeed, when a branch is predicted, processors typically checkpoint the RAT [13] (or equivalent tables). In our case, we include the RAT's *PIE* field in the checkpoint. If the branch outcome is declared mispredicted, we follow the typical steps of restoring the RAT and squashing the wrong-path instructions in the window. Such squashing does not confuse our schemes. To see why, consider the cases when either a producer instruction or a dependent instruction are squashed.

If a producer instruction is squashed, then its dependent instructions in the window must have also been squashed. Therefore, the extra fields that we added to the instruction window do not become inconsistent. However, if a dependent instruction is squashed, its producer instruction(s) may not be squashed. In this case, the *DIE* field in its producer instruction(s) becomes a dangling pointer pointing to an incorrect entry.

To address this issue, we could include the *DIE* field and the extra bits from the instruction window (Figure 4) in the checkpoint, and restore them on misprediction. While this solution eliminates the dangling pointers, it is too costly in resources and energy.

Our proposed solution is to tolerate the dangling pointers. Such pointers do not cause any error such as a failure to wake up a correct instruction; they can only induce some unnecessary stall and/or comparisons. Specifically, when a true dependent instruction of a producer with a dangling pointer is about to be inserted into the window, it will have to stall (*Indexing-Only*) or mark the *Broad-*

cast bit (*Hybrid*) of the producer instruction. In either case, when the producer completes, it will enable the comparator for a useless instruction (in addition to the ones for the correct instruction in *Hybrid*). Overall, it can be shown that these unnecessary stalls and/or comparisons have a minimal impact on the performance and energy consumed.

3.4 Applicability to Other Organizations

While we have assumed a centralized instruction window, other window designs are possible. For example, some designs use multiple instruction windows [10]. In this case, our schemes work similarly. The main difference is that the *DIE* pointer now contains both a window ID and an entry ID so as to correctly index the dependent instruction. The resulting circuitry does not have a high line capacitance because the signals to the non-selected windows are gated. However, the overall circuitry is a bit more complicated.

Other designs use a compacting instruction window [4]. In this case, window entries move. Since our schemes use pointers for direct indexing, they do not work for these windows without additional support.

4. EVALUATION ENVIRONMENT

We evaluate our proposal on a simulated generic out-of-order processor with a centralized instruction window structure. Table 1 shows some parameters of the simulated system. Our execution driven simulator system models the contention and occupancy of all the resources [8].

Processor	
Frequency: 1 GHz Issue width: 6 Dynamic issue: yes I-window size: 96 Load/store units: 2 Int,FP units: 5,4 Pending loads,stores: 16,16	Branch units: 1 Branch penalty: 8 cycles RAS entries: 32 BTB entries: 2048 BTB assoc: 4 Branch predictor: GAp(10,8)
Caches	Bus & Memory
L1 size: 32 KB L1 OC,RT: 1,3 ns L1 assoc: 2 way LRU L1 line: 32 B L2 size: 512 KB L2 OC,RT: 4,12 ns L2 assoc: 8 way PLRU L2 line: 64 B I-cache: 32 KB 2way 32 B line	FSB freq: 333 MHz FSB width: 128 bits Mem: 2-channel Rambus DRAM bandwidth: 3.2 GB/s Mem RT: 108 ns

Table 1: System configuration. OC, RT, PLRU, FSB, and RAS stand for occupancy, contention-free round trip from the processor, pseudo LRU, front side bus, and return address stack respectively.

To show the impact of our proposal on different types of applications, we run seven applications from the multimedia, integer, and floating-point domains. The applications are compiled with the IRIX MIPSPro compiler version 7.3 with -O2 optimization. The applications are simulated from the beginning to the end, and last from hundreds of millions of cycles to over one billion cycles.

Table 2 shows the applications used. For the SPECint and SPECfp applications, we use a reduced input dataset. We verify that the reduced data set produces similar cache and TLB miss rates as the native execution with the reference data on a MIPS R12000-based workstation. For the multimedia applications, we use an mp3 decoder (MP3D) and an mp3 encoder (MP3E).

Domain	Application	Input Data Set
SPECint 2000	CRAFTY	Reduced ref
	BZIP	Reduced ref
	MCF	Reduced ref
SPECfp 2000	HYDRO	Reduced ref
	APSI	Reduced ref
Multimedia	MP3D (mpg123-0.59r)	Hifi
	MP3E (lame-3.85)	Voice

Table 2: Applications executed.

It is challenging to accurately estimate the energy consumed in the instruction window. The reason is that, since the window is one of the most critical components of the processor core, it is heavily tuned and has diverse implementations. Therefore, we do not attempt to report the absolute or relative energy consumptions of the different designs. Instead, we use the number of tag comparisons in the instruction window required by the different schemes as a *qualitative*, relative measure of the energy consumed.

5. EVALUATION

5.1 Indexing-Only Scheme

The main advantage of this scheme is that all instruction completions enable at most one single comparator. Consequently, the number of tag comparisons is approximately $\frac{1}{N_{entry}}$ of those in the unoptimized architecture, where N_{entry} is the number of entries in the instruction window. For our simulated architecture, this corresponds to a 99% reduction in tag comparisons.

The disadvantage of this scheme is that it sometimes stalls the processor and, therefore, slows down program execution. Figure 6 shows the increase in execution time of the applications caused by the *Indexing-Only* scheme. We can see that the scheme slows down the applications by an average of 8%. Some applications take quite a bit longer to complete. Note that, depending on the degree of clock gating, the longer execution time could diminish or even negate the energy savings achieved by the reduced number of comparisons. Consequently, *Indexing-Only* is not a good wakeup scheme.

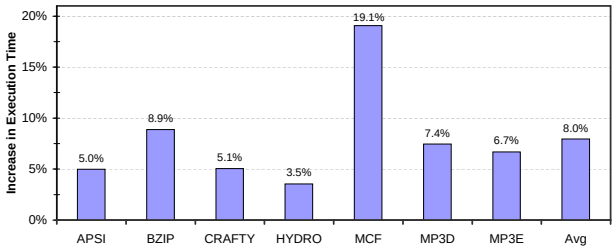

Figure 6: Increase in execution time caused by the Indexing-Only scheme.

5.2 Hybrid Schemes

The advantage of the *Hybrid* schemes is that they do not stall the processor when instructions have multiple close-by dependent instructions. Consequently, the application does not suffer any performance penalty. However, the disadvantage of these schemes is that a completing instruction may induce more than one tag comparison. We will see, however, that the number of comparisons is still close to the *Indexing-Only* scheme, and much smaller than the full broadcast, baseline scheme.

With simple calculations, we can estimate the reduction in comparisons relative to the full broadcast scheme. Note that in instruction streams, there are instructions that do not have a destination register, such as stores and branches. Even in the baseline broadcast scheme, they do not require tag comparisons. Here we only consider instructions that have a destination register. We consider the *Hybrid-Plain* scheme first. In this scheme, the number of comparisons required relative to full broadcast scheme is given by

$$\frac{1}{N_{entry}} * P_{single} + 1 * P_{more}$$

In the formula, N_{entry} is the number of entries in the instruction window, P_{single} is the probability of instructions having a single close-by dependent instruction, and P_{more} is the probability of instructions having more than one close-by dependents. If we use $N_{entry} = 96$, as it corresponds to our simulated architecture (Section 4), $P_{single} = 0.521$, and $P_{more} = 0.087$ as shown in Figure 3 for the same architecture (*Normal*), we obtain 9.2% for the fraction of remaining comparisons. This means that *Hybrid-Plain* reduces the comparisons by 90.8% relative to the full broadcast scheme.

For the *Hybrid-Snoop* scheme, the fraction of remaining comparisons relative to the full broadcast scheme is

$$\frac{1}{N_{entry}} * P_{single} + \frac{N_{snoop}}{N_{entry}} * P_{more}$$

In the formula, N_{snoop} is the average number of entries with the *Snoop* bit set when an instruction broadcasts the result tag. In our experiments, we measure that N_{snoop} ranges from 2.2 to 3.4 for the different applications. Taking the average across applications, N_{snoop} is 2.8. Consequently, using these numbers, we obtain that the fraction of remaining comparisons is 0.8%. This means that *Hybrid-Snoop* reduces the comparisons by 99.2% relative to the full broadcast scheme. This is a very large reduction, which makes the *Hybrid-Snoop* scheme potentially very energy efficient.

Figure 7 shows the actual number of comparisons per completing instruction for different applications. The figure compares the two hybrid schemes: *Hybrid-Plain* and *Hybrid-Snoop*.

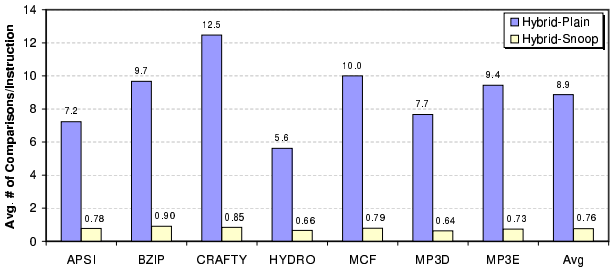

Figure 7: Average number of comparisons per completing instruction for *Hybrid* schemes. The instruction window has 96 entries.

We can see that, for a 96-entry window, the two schemes perform very few comparisons per instruction. *Hybrid-Snoop* has fewer comparisons – on average only 0.8 per completing instruction. As a result, this scheme is potentially very energy efficient. However, it also has a more complex hardware support.

If we remove the support for the *Snoop* bit, we get the *Hybrid-Plain* scheme. This scheme is simpler in hardware but results in an average of 8.9 comparisons per instruction.

Finally, for comparison purposes, we have also simulated a scheme like the one proposed by Folegnani and Gonzalez [5]. Specifically, we eliminate the support for indexing and simply gate the comparators for entries that are either invalid or that have all source operands ready. Note that we do not support the dynamic resizing of the instruction window proposed in [5] because it is orthogonal to our schemes. Overall, with such support, we obtain an average number of comparison per completing instruction equal to 23.7[2]. We see that while this scheme has a simpler hardware than our schemes, it requires many more comparisons and, therefore, it is potentially more energy consuming.

5.3 Analysis of Stall Cycles

In the *Indexing-Only* scheme, if an instruction already has a dependent instruction, we stall the insertion of a second dependent instruction until the producer completes. For these stalled instructions, Figure 8 shows the distribution of the number of cycles that they stall. In the figure, each curve represents one application, while the thicker line with large marks shows the average of all applications.

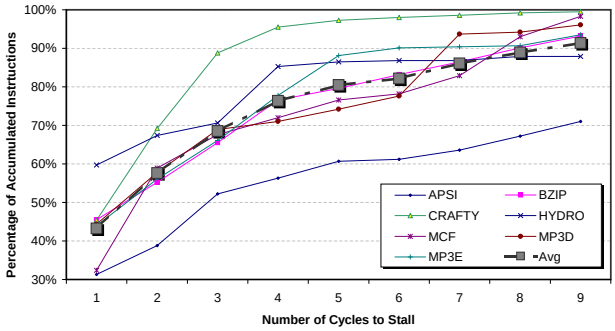

Figure 8: Distribution of the number of cycles that stalled instructions wait due to their producer already having one dependent instruction.

The average line shows that about 45% of these instructions stall for only one cycle. This suggests two optimizations. One of them is to further reduce the number of comparisons in the *Hybrid* schemes by always stalling for one cycle the decode and insertion of an instruction that is about to set the *Broadcast* bit. In 45% of the cases, it will not be necessary to set the *Broadcast* bit one cycle later. Of course, the drawback is that the application will take longer to execute. Indeed, Figure 9 shows the increase in execution time of the applications when we apply this optimization to any of the *Hybrid* schemes. On average, the applications take 2.1% longer to run. Overall, given that the number of broadcasts in the original *Hybrid* schemes is already very low (Figure 3) and that any slowdown usually involves additional energy consumption, this stalling policy is unattractive.

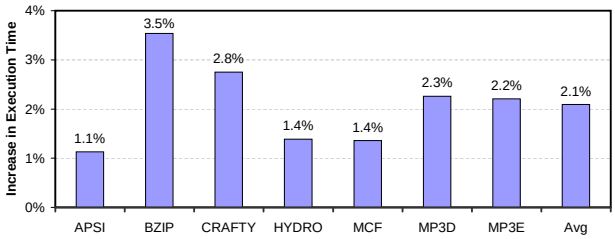

Figure 9: Increase in execution time of the *Hybrid* schemes when the instructions that would set the *Broadcast* bit are forced to stall for one cycle.

[2]This number is slightly different than the one obtained by Folegnani and Gonzalez because we are simulating a different system architecture.

A second optimization that can benefit the schemes just described and the *Indexing-Only* scheme is to use the compiler to schedule the code to reduce these stalls. Scheduling the stalling instructions only one cycle later is likely to make a major difference. Exploiting this optimization is beyond the scope of this paper.

6. RELATED WORK

The work most related to ours is [5], where comparisons to empty and fully ready entries in the instruction window are gated out to reduce power consumption. In their simulated architecture, this optimization reduces the average number of comparisons performed by each completing instruction in a 128-entry instruction window to 14.2. In our work, we exploit a different fact: on average, over 91% of the dynamic instructions need to wake up no more than one single instruction currently in the window. This leads us to propose a scheme that combines direct indexing (most of the time) and broadcasting (occasionally). For a 96-entry window, the number of comparisons performed by the average completing instruction is only 0.8.

Using indexing to replace associative search at instruction wakeup has been examined in two papers. In [12], an algorithm similar to our *Indexing-Only* scheme called Direct Tag Search (DTS) is proposed. The goal is to reduce hardware complexity. In [7], two-level RAM bitmap arrays are proposed, which support indexing-based wakeup for all instructions, regardless of their number of dependent instructions. The goal is to have a scalable design. However, the design is complex and is likely to be energy-inefficient. Overall, in our paper, we combine frequent indexing and occasional broadcast to obtain fast and energy-efficient schemes (our *Hybrid* algorithms). We also show experimental data that justify why single-indexing works most of the time.

Some other works try to use a direct addressing table to maintain the dependence chain and, as a result, reduce the required instruction window size. Among those, Dynamic Data Forwarding (DDF) uses the *Wait Memory* to complement the *Match Unit*, an equivalent of the instruction window [9]. DDF brings only some of the instructions from the *Wait Memory* to the *Match Unit* for associative check. In [3], two different schemes are studied that use some auxiliary structure to reduce the size of a fully-associative instruction window. One of the schemes uses a table that keeps the first consumer instructions of register results. When these instructions are ready to execute, they move to the ready queue, bypassing the instruction window. Like our proposal, such a scheme also exploits the fact that most instructions have no more than one dependent. However, it uses this fact to reduce the size of the window.

Select-free scheduling logic selects for execution all the instructions being woken up [2]. Extra circuitry handles the situation when more than one instruction wakes up and thus a collision occurs. This work leverages the fact that there is often only one instruction waking up inside each wakeup array. Such a fact is consistent with our observation that most instructions have no more than one close-by dependent. However, their work focuses on modifying the selection logic, while we change the wakeup logic.

There are many other related works that propose clustered windows or improvements to the selection logic [6, 10, 14]. In general, these works are orthogonal to ours.

Finally, in other works, the issue logic is dynamically adjusted to reduce the number of idle window entries or issue slots and, therefore, save energy [1, 5, 11]. These optimizations are largely orthogonal to our schemes and can still be applied to further reduce energy consumption.

7. CONCLUSIONS

This paper presented a simple hardware extension to improve the energy efficiency of the wakeup logic in a superscalar processor. The idea is based on the observation that a very large fraction of the completing instructions have no more than one dependent instruction currently in the window. For example, this fraction is on average 91.3% for a 96-entry window. Consequently, we proposed to save energy by using indexing to only enable the comparator at the single dependent instruction. When more than one instruction needs to wake up, our support automatically reverts to enabling all the comparators (*Hybrid-Plain* scheme) or a subset of them (*Hybrid-Snoop* scheme). Overall, the two schemes that we propose are shown to be very effective: for a system with a 96-entry window, the number of comparisons performed by the average completing instruction under *Hybrid-Plain* is 8.9, while under the slightly more complicated *Hybrid-Snoop*, it is only 0.8. The exact magnitude of the energy savings will depend on the specific instruction window implementation. Furthermore, in both schemes, the application suffers no performance penalty.

8. REFERENCES

[1] R. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *International Symposium on Computer Architecture*, pages 218–229, May 2001.

[2] M. Brown, J. Stark, and Y. Patt. Select-Free Instruction Scheduling Logic. In *International Symposium on Microarchitecture*, pages 204–213, Dec. 2001.

[3] R. Canal and A. Gonzalez. A Low-Complexity Issue Logic. In *International Conference on Supercomputing*, pages 327–335, June 2000.

[4] J. Farrell and T. Fischer. Issue Logic for a 600-MHz Out-of-Order Execution Microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, May 1996.

[5] D. Folegnani and A. Gonzalez. Energy-Effective Issue Logic. In *International Symposium on Computer Architecture*, pages 230–239, May 2001.

[6] K. Ghose. Reducing Energy Requirements for Instruction Issue and Dispatch in Superscalar Microprocessors. In *International Symposium on Low Power Electronics and Design*, pages 231–233, Aug. 2000.

[7] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita. A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors. In *International Symposium on Microarchitecture*, pages 225–236, Dec. 2001.

[8] V. Krishnan and J. Torrellas. A Direct Execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, Oct. 1998.

[9] S. Onder and R. Gupta. Superscalar Execution With Dynamic Data Forwarding. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 130–135, Oct. 1998.

[10] S. Palacharla, N. Jouppi, and J. Smith. Complexity Effective Superscalar Processors. In *International Symposium on Computer Architecture*, pages 206–218, June 1997.

[11] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *International Symposium on Microarchitecture*, pages 90–101, Dec. 2001.

[12] S. Weiss and J. Smith. Instruction Issue Logic in Pipelined Supercomputers. *IEEE Transactions on Computers*, 33(11):1013–1022, Nov. 1984.

[13] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–41, Apr. 1996.

[14] V. Zyuban and P. Kogge. Optimization of High-Performance Superscalar Architectures for Energy Efficiency. In *International Symposium on Low Power Electronics and Design*, pages 84–89, Aug. 2000.