Using Lazy Instruction Prediction to Reduce Processor Wakeup Power Dissipation

Houman Homayoun[+]
houman@houman-homayoun.com

Amirali Baniasadi
University of Victoria, ECE Department
amirali@ece.uvic.ca

ABSTRACT

We study lazy instructions. We define lazy instructions as those spending long periods in the issue queue. Moreover, we investigate lazy instruction predictability and show how their behavior could be exploited to reduce activity and power dissipation in modern processors. We show that a simple and small 64-entry table can identify up to a maximum of 50% of lazy instructions by storing their past behavior. We exploit this to a) reduce wakeup activity and power dissipation in the issue queue and b) reduce the number of in-flight instructions and the average instruction issue delay in the processor.

We also introduce two power optimization techniques that use lazy instruction behavior to improve energy efficiency in the processor. Our study shows that, by using these optimizations, it is possible to reduce wakeup activity and power dissipation by up to 34% and 29% respectively. This comes with a performance cost of 1.5%. In addition, we reduce average instruction issue delay and the number of in-flight instructions by up to 8.5% and 7% respectively with no performance cost.

1. INTRODUCTION

Modern high-performance processors execute instructions aggressively, processing them in each pipeline stage as soon as possible. This requires fetching as many instructions as possible and processing them as fast as we can. A typical processor fetches several instructions from the memory, decodes them and dispatches them to the issue queue.

Instructions wait in the issue queue for their operands to become available. The processor associates tags with each source operand and broadcasts operand tags to all instructions in the issue queue every cycle. Instructions compare the tags broadcasted with the operand tags they are waiting for (referred to as instruction wakeup). Once a match is detected, instructions are executed subject to resource availability (referred to as instruction select). These are energy demanding tasks making the issue queue one of the major energy consumers in the processor (the issue queue is estimated to consume about 27% of the overall processor power [16]).

This aggressive approach appears to be inefficient due to the following:

1- In order to improve ILP, high-performance processors fetch as many instructions as possible to maximize the number of in-flight instructions. High-performance processors continue fetching instructions even when there are already many in-flight instructions waiting for their operands. A negative consequence of this approach is that some instructions enter the pipeline too early and long before they can contribute to performance. Nevertheless, they consume resources and energy.

2- Many instructions tend to wait in the issue window for long periods. An example of such instructions is an instruction waiting for data being fetched from the memory. Under such circumstances, the waiting instruction and consequently those depending on its outcome have to wait in the issue queue for several cycles. During this long period, however, the processor attempts to wakeup such instructions every cycle.

We exploit the two inefficiencies discussed above and use instruction behavior to address them. We study instruction issue delay (also referred to as IID). In particular, we study lazy instructions, *i.e.*, those instructions that spend long periods in the issue queue.

In this work we identify/predict lazy instructions. By identifying lazy instructions we achieve the following: First, by estimating the number of in-flight lazy instructions, we identify occasions when the front-end can be reconfigured to fetch fewer instructions without compromising performance.

[+] The author was with the University of Victoria, Electrical and Computer Engineering Department when this work was done.

Second, once lazy instructions are identified speculatively, we reduce wakeup activity by avoiding to wakeup lazy instructions every cycle.

By using the above approaches we reduce instruction wakeup activity, instruction wakeup power dissipation, the number of in-flight instructions and average issue delay by up to 34%, 29%, 7% and 8.5% respectively while maintaining performance.

The rest of the paper is organized as follows. In Section 2 we study issue delay prediction in more detail. In Section 3 we explain our optimization techniques. In Section 4 we present our experimental evaluation. In Section 5 we review related work. Finally, in Section 6 we summarize our findings.

2. ISSUE DELAY PREDICTION

In this work we adjust processor parameters dynamically to reduce processor activity and consequently power dissipation. We rely on instruction behavior to identify occasions where we can reduce the number of processed instructions while maintaining performance.

Many studies show that the behavior of an instruction in the issue queue is predictable [*e.g.,* 5-8]. In this paper we focus on predicting lazy instructions and the possible applications. Through this study we define lazy instructions as those spending more than 10 cycles in the issue queue. We picked this threshold after testing many alternatives.

There are many factors influencing IID (instruction issue delay) including instruction dependency and resource availability. Figure 1(a) shows IID distribution for a subset of SPEC'2K benchmarks. On average, about 18% of the instructions are lazy instructions, *i.e.*, they spend at least 10 cycles in the issue queue (maximum: 32%).

We refer to the number of times an instruction receives operand tags and compares them to its operand tags as the instruction wakeup activity. Lazy instructions, while accounting for about 18% of the total number of instructions, impact wakeup activity considerably. This is due to the fact that they receive and compare the operands tags very frequently and during long periods. To explain this better in Figure 1(b) we report the relative share of total wakeup activity for each group of instructions presented in Figure 1(a). On average, lazy instructions, despite their relatively low frequency, account for more than 85% of the total wakeup activity.

Our study shows that lazy instructions tend to repeat their behavior. This provides an opportunity to identify them before they arrive in the issue queue. We use this opportunity to avoid the extra wakeup activity and to identify lazy instructions early enough. We use a small 64-entry, PC-indexed table to predict lazy instructions before they arrive in the issue queue. We refer to this table as the LI-table. While exploiting larger and more complex structures may improve prediction accuracy, we avoid such structures to maintain power and latency overhead at a low level.

To store lazy instructions we do the following: If IID is more than 10, we store the instruction PC in the LI-table (more on this later). We also associate each table entry with a 2-bit saturating counter. If the lazy instruction is already in the table we increment the corresponding saturating counter. For the non-lazy instructions with an entry in the table, we remove the corresponding entry. To predict whether an instruction is lazy, we probe the LI-table. The instruction is marked as lazy, if the corresponding counter is more than two. Note that the LI-table can be accessed in parallel to fetch/decode and therefore would not result in a deeper pipeline front-end.

We evaluate the proposed prediction scheme using two criteria, *i.e.*, *prediction accuracy* and *prediction effectiveness*.

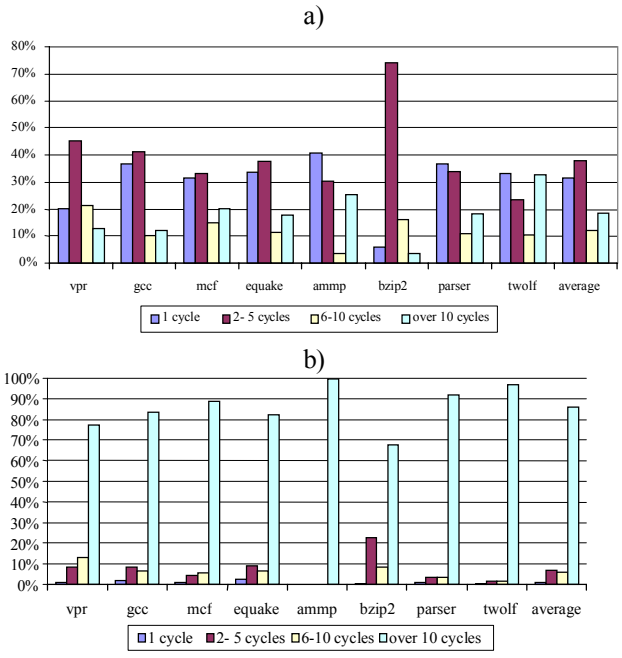

Figure 1: (a) Instruction issue delay distribution (b) Instruction wakeup activity distribution.

Lazy instruction prediction accuracy reports how often instructions predicted to have an issue delay more than 10 turn out to stay in the issue queue for more than 10 cycles. This, while important, does not provide enough information as it is silent regarding the percentage of lazy instructions identified. Therefore, we also report prediction effectiveness, *i.e.*, the percentage of lazy instructions identified.

While lazy instructions are identified by probing the LI-table at dispatch, the table can be updated at different stages. Two obvious update scenarios are *commit-update* and *issue-update*. We report prediction accuracy and effectiveness for both update scenarios.

In the first scenario, *commit-update*, lazy instructions are allowed to update the LI-table only after they have committed. Under this scenario wrong path instructions will not update the table.

Note that lazy instructions spend a long period in the pipeline and therefore update the LI-table long after they have entered the pipeline. As such, by the time a lazy instruction has committed, many lazy instructions have entered the pipeline without being identified. Also, it is quite possible that during this long period, the instruction behavior may change and therefore the stored information may not be valid by the time it becomes available. The second scenario, *issue-update,* allows lazy instructions to update the LI-table as soon as they issue. This, while making faster update possible, allows wrong path instructions to interfere.

2.1. Prediction Accuracy

In Figure 2(a) we report prediction accuracy. Bars from left to right report for commit and issue update for the subset of SPEC'2K benchmarks studied here. On average, prediction accuracy is 52% and 54% for commit-update and issue-update respectively. *Ammp* has the highest accuracy (97%) while *bzip2* and *vpr* fall behind other benchmarks. Our study shows that lazy instructions change their behavior frequently for these two benchmarks. This is consistent with the fact that *bzip2* and *vpr* have lower number of lazy instructions and lazy instruction activity compared to other benchmarks (see Figure 1). Note that the 50% average accuracy should be viewed in the light of the fact that only 18% of instructions are lazy instructions.

2.2. Prediction Effectiveness

In Figure 2(b) we report prediction effectiveness. On average, effectiveness is a bout 30%. Maximum effectiveness is achieved for *gcc* where we accurately identify more than half of the lazy instructions. Minimum effectiveness is achieved for *vpr*, where about 10% of lazy instructions are identified.

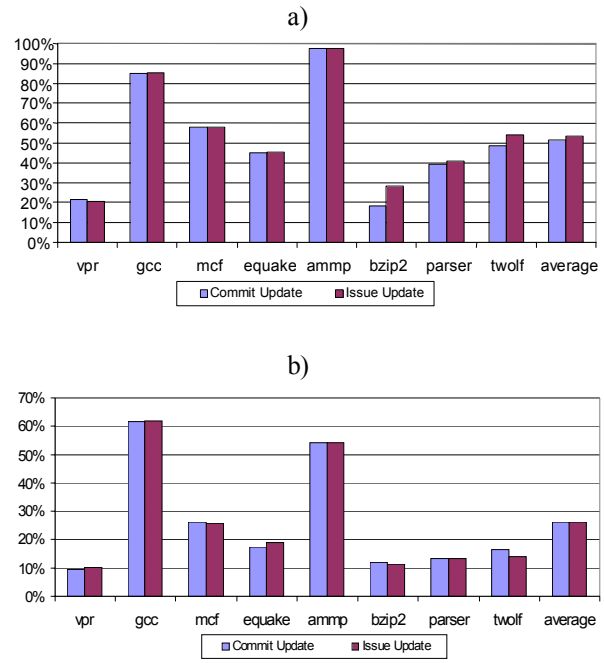

Figure 2: a) Lazy instruction prediction accuracy b) Lazy instruction prediction effectiveness.

3. OPTIMIZATIONS

In this section we introduce two optimization techniques which use information available by using an issue-update lazy instruction predictor. The two techniques are *selective instruction wakeup* and *selective fetch slowdown.* Selective instruction wakeup avoids waking up all instructions every cycle. Selective fetch slowdown reduces fetch speed if the number of lazy instructions in the pipeline exceeds a threshold. While the first technique impacts wakeup activity, the second one impacts more than one pipeline stage.

3.1. Selective Instruction Wakeup

As explained earlier, modern processors attempt to wakeup all instructions in the issue queue every cycle. As a result, instructions receive their source operands at the earliest possible. This consequently improves performance. However, it is unnecessary to wakeup lazy instructions as aggressively as other instructions.

Ideally, if we had an oracle and knew in advance when an instruction will issue, then a heuristic for selectively waking up lazy instructions would require waking up the lazy instruction only at the time it is supposed to issue. Of course, we cannot have such an oracle. An alternative is to predict instruction latency and consequently issue time [7] and restrict instruction wakeup to the predicted time. However, this will impose inherent limitations on performance by inaccuracies [2]. To avoid such complexities, we take a more conservative approach: Once we have predicted an instruction as a lazy instruction, instead of attempting to wake it up every cycle, we wake it up every two cycles.

The hardware structure for selective wakeup is shown in Figure 3. We add a multiplexer per issue queue entry to power gate the comparators every two cycle. As we wakeup lazy instruction in even cycle, we need to save the result tags produced in the odd cycle. This requires using registers to keep the result tags and broadcast them to lazy instructions when and if free broadcast slots are available. Our study of application behavior shows that broadcast results are often available. In fact, on average, only one broadcast slot is full every cycle.

In our broadcast policy we assign higher priority to result tags produced in the previous cycle. In the rare case that all broadcast slots are full we stall issuing instruction until an empty broadcast slot becomes available. As presented in Figure 3, the hardware overhead is negligible.

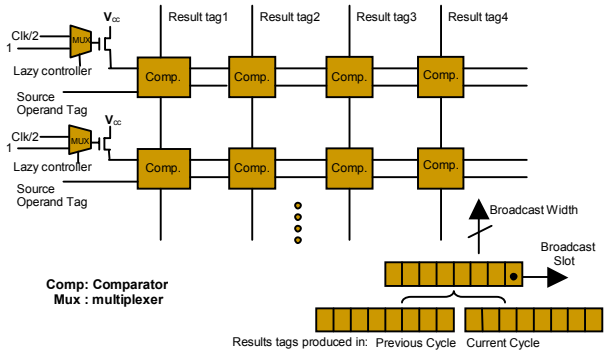

Figure 3: Hardware structure for selective wakeup.

3.2. Selective Fetch Slowdown

Modern processors rely on aggressive instruction fetch to maintain ILP. Instruction fetch is responsible for supplying the rest of the processor pipeline with instructions. Instruction fetch rate should at least match the instruction decode and execution rate otherwise the processor resources will be underutilized. Note that, if the instruction flow in the pipeline is too slow, it will be inefficient to fetch too many instructions. For example, if there are already many instructions waiting for their operands in the pipeline, we may be able to delay adding more instructions to the already high number of in-flight instructions without losing performance. This will reduce the number of in-flight instructions which in turn will result in less pressure on reservation stations and pipeline activity.

In this section we use our estimation of the number of in-flight lazy instructions to decide whether fetching instructions at the maximum rate is worthwhile. If the number of lazy instructions exceeds a dynamically decided threshold we assume that it is safe to slowdown instruction fetch. Accordingly, we reduce the maximum cache lines fetched from two to one.

To decide the dynamic threshold we record the number of instructions predicted to be lazy every 1024 cycles. If the number of lazy instructions exceeds one third of total number of in-flight instructions we reduce the threshold by five. If the number of lazy instructions drops below 5% of the total number of in-flight instructions we increase the threshold by five. Initially, we set this threshold to 15. Note that the design parameters picked here are selected to optimize energy efficiency for the configuration used. Alternative processor configurations may require different parameters to achieve best results.

Selective fetch slowdown needs minimal hardware modification as it only requires two counters and a register to keep the number of lazy instructions, all instructions and the dynamic lazy threshold respectively.

4. METHODOLOGY AND RESULTS

In this section, we report our analysis framework. To evaluate our techniques we report performance, wakeup activity, average issue delay, average number of in-flight instructions, power dissipation and how often we slowdown fetch. We compare our processor with a conventional processor that attempts to wakeup

all instructions every cycle and does not reduce the fetch rate.

Note that activity measurements are less technology- and implementation-dependent compared to power measurements. Nonetheless, we also report power analysis for the processor studied here. We detail the base processor model in Table 1. We used both floating point *(equake* and *ammp)* and integer *(vpr, gcc, mcf, bzip2, parser* and *twlf*) programs from the SPEC CPU2000 suite compiled for the MIPS-like PISA architecture used by the Simplescalar v3.0 simulation toolset [1]. We used WATTCH [15] for energy estimation. We modeled an aggressive 2GHz superscalar microarchitecture manufactured under a 0.1 micron technology. We used GNU's gcc compiler. We simulated 200M instructions after skipping 200M instructions.

Table 1: Base processor configuration.

Integer ALU	# 8	***Scheduler***	128 entries, RUU-like
FP ALU	# 8	***OOO Core***	any 8 instructions / cycle
Integer Multipliers/ Dividers	#4	***Fetch Unit***	Up to 8 instr./cycle. 64-Entry Fetch Buffer
FP Multipliers/ Dividers	#4	***L1 - Instruction Caches***	64K, 4-way SA, 32-byte blocks, 3 cycle hit latency
Instruction Fetch Queue	#64	***L1 - Data Caches***	32K, 2-way SA, 32-byte blocks, 3 cycle hit latency
Branch Predictor	2k Gshare bimodal w/selector	***Unified L2***	256K, 4-way SA, 64-byte blocks,16-cycle hit
Load/Store Queue Size	64	***Main Memory***	Infinite, 80 cycles
Reorder Buffer Size	128	***Memory Port***	#4

4.1. Results

In this section we report our simulation results. In 4.1.1 we report performance. In 4.1.2 we report activity and power measurements. In 4.1.3 we report average issue delay reduction and fetch slowdown frequency.

4.1.1. Performance

In Figure 4 we report how selective wakeup and selective fetch slowdown impact performance. To provide better insight we also report performance for a processor that never fetches more than one cache line *(referred to as the single line processo*r). In Figure 4, bars from left to right report performance for selective wakeup, selective fetch slowdown and the single line processor. Across all benchmarks performance cost is below 1.5% for selective wakeup. Selective fetch slowdown, however, does not impact performance. On the other hand, the single line processor comes with a maximum performance cost of 5.5%.

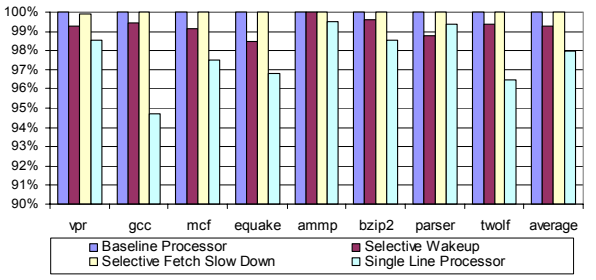

Figure 4: Bars from left to right report performance for selective instruction wakeup, selective fetch slowdown and single line processor respectively.

4.1.2. Activity and Power

In Figure 5 we report activity and power measurements. In Figure 5(a) we report how selective instruction wakeup impacts wakeup activity. On average, we reduce wakeup activity by 12% reaching a maximum of 34% for *ammp*.

In Figure 5(b) we report average reduction in the number of in-flight instructions for selective fetch slowdown and the single line processor. Selective fetch slowdown reduces the average number of in-flight instructions by 4% (maximum 7%) without compromising performance (see Figure 4). The single line machine reduces average number of in-flight instructions by 8.5% (maximum 16%); however, this can be as costly as 5.5% performance loss as presented earlier. In Figure 5(c) we report wakeup power reduction as measured by wattch. Bars from left to right report power reduction for selective wakeup, selective fetch slowdown and the combination of both techniques.

Selective wakeup reduces wakeup power dissipation up to a maximum of 27% (for *ammp*). Note that this is consistent with Figure 5(a) where

ammp has the highest activity reduction. Minimum wakeup energy reduction is about 2% for *bzip2*. Again this is consistent with Figure 5(a) where *bzip2* has the lowest activity reduction.

Selective fetch slowdown reduces wakeup power up to a maximum of 12% (for *equak*e) and a minimum of 1% (for *bzip2 and ammp*). This is consistent with Figure 5(b) where *equake* has the highest reduction in the number of in-flight instructions and *bzip2* and *ammp* have the lowest.

Using both techniques simultaneously, on average, we reduce wakeup power by about 14%. Average wakeup power reduction is 8.3% and 6.7% for selective wakeup and selective fetch slowdown respectively.

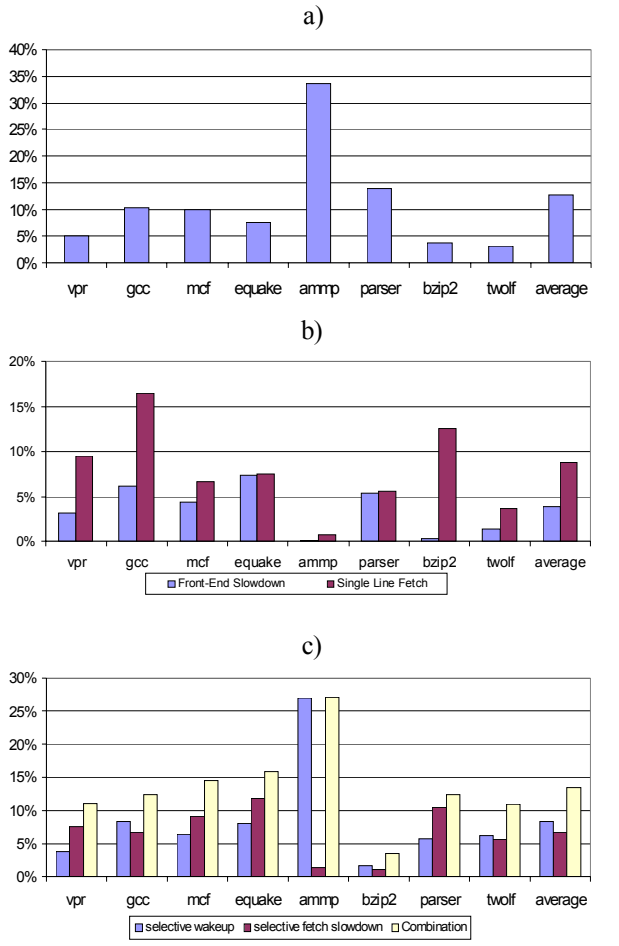

Figure 5: Selective wakeup: activity reduction b) Selective fetch slowdown: average in-flight instruction reduction c) Wakeup power reduction.

Note that, compared to other processor structures, the LI-table is a small structure and therefore comes negligible power overhead.

4.1.3. Issue Delay and Slowdown Rate

In Figure 6(a) we report average reduction in IID achieved by selective fetch slowdown. On average, we reduce IID by 4% (maximum 8%).

Finally, in Figure 6(b) we report how often selective fetch slowdown reduces fetch rate. On average we reduce fetch rate about 50% of the time. Note that for *ammp* we rarely reduce the fetch rate. This explains why we do not witness any reduction in average number of in-flight instructions or IID for *ammp* as reported in Figure 5(b).

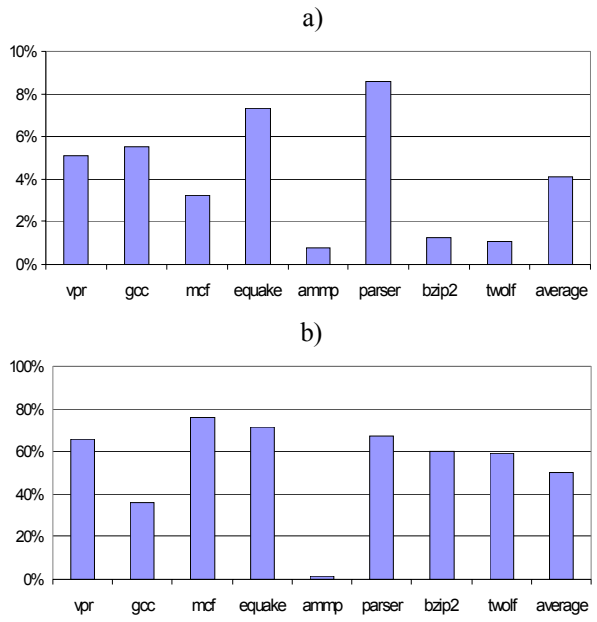

Figure 6: a) Selective fetch slowdown: average issue delay reduction b) Selective fetch slowdown: slowdown rate.

5. RELATED WORK

Many studies have used instruction behavior to optimize processor design. Such studies either redesign different processor parts, such as the issue queue, or reconfigure the processor dynamically.

Raasch *et al.* suggested adapting the issue queue size and exploiting partitioned issue queues to reduce the wakeup activity [11]. Our work is orthogonal to

and could be used on top of their approach to increase savings.

Many studies use dependency chain information to reduce issue queue complexity [5,6,8-10]. Our work is different from these studies as it relies on issue delay estimation without considering data dependency.

Brown *et al*., introduced methods to remove the select logic from the critical path [4]. Brekelbaum *et al*., introduced a new scheduler, which exploits latency tolerant instructions in order to reduce implementation complexity [3]. Stark *et al*., used "grandparent" availability time to speculate wakeup [12]. Ernst *et al*., suggested a wakeup free scheduler which relied on predicting the instruction issue latency [7]. Hu *et al*., studied wakeup-free schedulers such as that proposed in [7] and explored how design constrains result in performance loss and suggested a model to eliminate some of those constrains [2]. Our technique is different from all these studies as it takes a more conservative approach and uses a less complex scheme.

Huang *et al*., [13] showed that a large fraction of instructions wake up no more than a single instruction. They proposed an indexing scheme which only enables the comparator for a single dependent instruction. Our technique is different as it uses prediction to decide which comparators to disable.

Previous study has introduced front-end gating techniques to stop fetching instructions when there is not enough confidence in the in-flight instructions [14]. Our work is different as it relies on lazy instruction prediction.

6. CONCLUSION AND FUTURE WORK

In this work we studied lazy instructions and introduced two related optimization techniques. We showed that it is possible to identify a considerable fraction of lazy instructions by using a small and simple 64-entry predictor. By predicting and estimating the number of lazy instructions we reduced wakeup activity, wakeup power dissipation, average instruction issue delay and the average number of in-flight instructions while maintaining performance. We relied on limiting instruction wakeup for lazy instructions to even cycles and reducing the processor fetch rate when the number of lazy instructions in the pipeline exceeds a dynamically decided threshold. Our study covered a subset of SPEC'2k benchmarks.

As mentioned, several approaches have been proposed to reduce power dissipation of the issue queue. One possible future avenue is to study the possibility of their combination with our proposed techniques.

While throughout this work we only focused on lazy instructions it is possible to study fast instructions; *i.e.,* instructions which are issued quickly. In addition we can use the information of lazy/fast instruction prediction to reduce power dissipation in other structure of a superscalar processor.

7. ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada, Discovery Grants Program and Canada Foundation for Innovation, New Opportunities Fund.

REFERENCES

[1] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. *Technical Report CS-TR-96-1308, University of Wisconsin-Madison,* July 1996.

[2] J. S. Hu, N. Vijaykrishnan, and M. J. Irwin. Exploring Wakeup-Free Instruction Scheduling. *In Proc. of the 10th International Conference on High-Performance Computer Architecture (HPCA-10 2004),* 14-18 February 2004, Madrid, Spain.

[3] E. Brekelbaum, J. R. II, C. Wilkerson, and B. Black. Hierarchical scheduling windows. In *Proc. of the 35th Annual IEEE/ACM International Symposium on Microarchitectur*e, Nov. 2002.

[4] M. D. Brown, J. Stark, and Y. N. Patt. Select-free instruction scheduling logic. In *Proc. of the International Symposium on Microarchitectur*e, Dec. 2001.

[5] R. Canal and A. Gonzalez. A low-complexity issue logic. In *Proc. of 2000 International Conferences on Supercomputin*g, May 2000.

[6] R. Canal and A. Gonzalez. Reducing the complexity of the issue logic. In *Proc. of 2001 International Conferences on Supercomputin*g, June 2001.

[7] D.Ernst, A.Hamel, and T.Austin. Cyclone:a broadcast-free dynamic instruction scheduler selective replay. In *Proc. of the 30th Annual International Symposium on Computer Architectur*e, June 2003.

[8] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proc. of the 29th Annual International Symposium on Computer Architectur*e, May 2002.

[9] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proc. of the 7th International Symposium on High Performance Computer Architectur*e, Jan. 2001.

[10] S. Palacharla, N. P. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proc. of the 24th Annual International Symposium on Computer Architectur*e, pages 206–218, June 1997.

[11] S. Raasch, N. Binkert, and S. Reinhardt. A scalable instruction queue design using dependence chains. In *Proc. of the 29th Annual International Symposium on Computer Architectur*e, May 2002.

[12] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *Proc. of the International Symposium on Microarchitectur*e, Dec. 2000.

[13] M. Huang, J. Renau, and J.Torrellas Energy-Efficient Hybrid Wakeup Logic. In Proc. *of the international symposium on Low power electronics and design-ISLPED'02,* August 2002.

[14] S. Manne, A. Klauser and D. Grunwald. *Pipeline Gating: Speculation Control For Energy Reduction.* In Proc. Intl. Symposium on Computer Architecture, Jun., 1998.

[15] D. Brooks, V. Tiwari M. Martonosi "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*", Proc of the 27th Int'l Symp. on Computer Architectur*e, 2000.

[16] Gurhan Kucuk, Dmitry Ponomarev, Kanad Ghose :Low-Complexity Reorder Buffer Architecture, 16th ACM International Conference on Supercomputing (ICS'02), New York, June, 2002, pp. 57-66.