Correlation Between State and Control Signals in Out-of-Order Issue Logic

Kenneth Barr and Kenneth Conley
Massachusetts Institute of Technology
Advanced VLSI Computer Architecture, Fall 2000
{kbarr, conley}@mit.edu

ABSTRACT

Current out-of-order control logic is optimized for performance rather than low-energy operation. Key out-of-order logic structures, including register renaming logic and superscalar issue logic, are examined for patterns and correlation between state and asserted control signals. We study the behavior of a register renaming scheme, showing potential to increase instances of reused renaming tags. We demonstrate that an issue logic prediction scheme would be successful and could be used to reduce wakeup window size. We estimate accuracy for such a scheme to exceed 80% accuracy in cases where it can be applied successfully and we suggest an explanation for cases in which it fails. By exploiting high correlation, predictive or memoization techniques could be implemented in the issue logic to eliminate redundant work and increase energy efficiency.

Keywords

Out-of-Order, Power consumption, prediction, control logic

1. INTRODUCTION

When manufacturers tout their latest superscalar processor designs, high performance often overshadows energy efficiency. Yet, the logical structures responsible for extracting performance-enhancing ILP are some of the most complex in a superscalar processor in terms of delay, scaling, and power. With predictive and memoizing structures dominating the front-end of most modern superscalar pipelines, it was logical to wonder whether these structures could be leveraged to project predictions further down the pipeline to eliminate the need to exercise complex out-of-order issue logic on each cycle. Specifically, this paper analyzes the relationship between out-of-order issue logic and the current program counter to see if the two are correlated. The regularity of the register renaming logic is also examined, motivated the increasing need to rename multiple instructions at a time. This paper explains a set of correlation statistics, explains how they were gathered, and suggests how the positive measured results can serve two power-saving goals. First, high correlation allows bypassing of pipeline stages similar to traditional and novel trace cache processors. In addition, the wakeup-window issue logic (one of the limiting factors in VLSI design) could be decreased in depth.

This paper describes a project undertaken by Kenneth Barr, Kenneth Conley, and Serhii Zhak as part of MIT's Course 6.893: Advanced VLSI Computer Architecture.

2. RELATED WORK

In dynamically scheduled superscalar processors, the size (and thus delay and power consumption) of the issue logic depends quadratically on the product of instruction issue width and instruction window size. This delay is mainly due to the associative search required by the issue logic [2] and performance-centric design philosophies [3]. The power consumption of such logic was measured at 18-46% by the designers of the Alpha 21264 [5] and studied in a recent paper [4] which modeled an out-of-order superscalar processor at the level of functional blocks. This study found that the renaming table, instruction queue, and reorder buffer, all necessary for extracting instruction level parallelism, are responsible for an average of 53% of the total power consumed by a processor. In addition, each of these three units considered individually consumes more energy than any other part of the processor (e.g., cache, branch predictor, functional units, and I/O). To address the power consumption of the instruction queue, the authors proposed a dynamically resized queue which saved 57% of the power consumed by the queue and reduced the total power consumption of the processor by about 15%.

Other authors have attempted reduce power usage in

the issue logic. Hiraki et al. [6] proposed a decoded instruction buffer that memoized control signals for execution with power savings of 40% for loops in an in-order processor; Wang and Roy [13] proposed a new encoding scheme for storing control signals in micro-programmed control units and claimed 4.8%-16.5% reduction in switching activity. Complexity effective design was surveyed in a paper by Palacharla, Jouppi, and Smith. [9]. Reducing delay by eliminating the associate search through the instruction window motivates their proposed organization of a wakeup window made up of just the heads of dependency-based issue queues. The issue prediction logic presented in section 3.3 can be used to achieve the same goal.

A renaming scheme relying on a modified ISA and compiler hints can be used to reduce the number of read ports in a register as shown by Sprangle and Patt in [11]. Jourdan [7] proposed physical register reuse in a value-identity detection scheme to improve performance. Despite this work, there remains great potential to reuse physical register mappings. If such potential could be exploited, a structure such as Vajapeyam's trace cache [12] (which stores register renaming tags) could be used to reduce the need to use power-intensive structures to rename multiple registers at once.

Research has also been done at the circuit level for saving power in control logic [8], but such results are beyond the scope of this paper.

3. METHODOLOGY AND RESULTS

The SimpleScalar microarchitectural simulator [1] has been extended to provide detailed per-instruction statistics that explore trends in register renaming. In addition SimpleScalar pipetrace output is analyzed with custom parsers. Through the use of an oracle in the Fetch stage of the pipeline that, through prediction, allows complex issue logic to be disabled, we attempt to quantify the upper bounds of correlation between processor state and asserted control signals.

3.1 Pipetracing

Profile statistics were collected using programs from the SPECint95 benchmark suite with scaled-down datasets (Table 1). The first five million instructions of each programs were skipped to avoid initialization routines. To warm up the cache and predictors, an additional six million instructions were allowed to commit using the SimpleScalar default, cycle-accurate, out-of-order processor described in Table 2. At this point the pipeline tracer was activated to capture the state of the machine until an additional 250,000 instructions had been committed. While this represents a tiny portion of the program's execution, it was necessary due to disk space constraints. For the analysis, 250,000 instructions were sufficient to saturate the issue and renaming logic (which holds only 24 active physical destination registers at a time). Also, the instantaneous wakeup-set statistics (section 3.1.1) are gathered by static dataset analysis even though a hardware scheme would likely operate dynamically. This small dataset approximates the gains that could be achieved in a hardware structure of reasonable size. It does so better than measuring the performance of a predictor trained over several million instructions, over which the program state has much greater variance. Furthermore, it was not the goal of this paper to present estimated performance benefits, so a workload representing the complete execution would not have added to our analysis.

Program	Input Data
cc1	jump.i
compress95	bigtest.in
go	50 21 9stone21.in
ijpeg	penguin.ppm
li	*.lsp
m88ksim	ctrl.in
perl	scrabbl.pl
vortex	vortex.in

Table 1: Benchmarks and Inputs (from "reference" set).

Item	Size
Instruction fetch queue size (in insts)	4
Issue width	4
Commit width	4
Register Update Unit (window) size	16
Load/Store Queue (window) size	8

Table 2: Default simulator configuration

Two types of profile statistics were collected using SimpleScalar. We termed these "instantaneous" and "instruction consistency" statistics. For both types of profile statistics, cache misses, TLB misses, and branch mispredicts were ignored as actual hardware would avoid training against this worst-case behavior.

3.1.1 Instantaneous Statistics

Our instantaneous statistics examine the correlation between a cycle's Fetch program counter (fetchPC) and the instructions issued to functional units in that cycle. Figure 1 shows how often one could predict a set of dispatched instructions by predicting the window seen most frequently with the current fetchPC. The dual predictor bar shows improvement gained by keeping track of the *two* most popular wakeup sets for a particular fetchPC. The highest frequency correlation determines

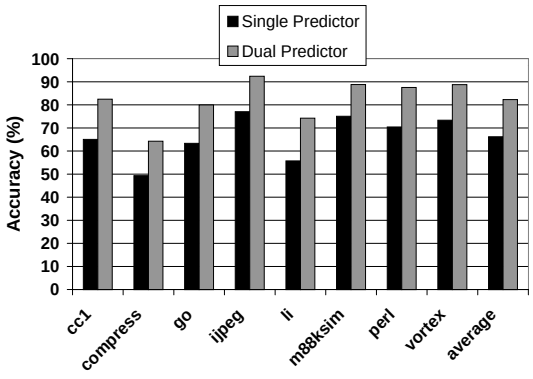

Figure 1: Issue Window Prediction Accuracy

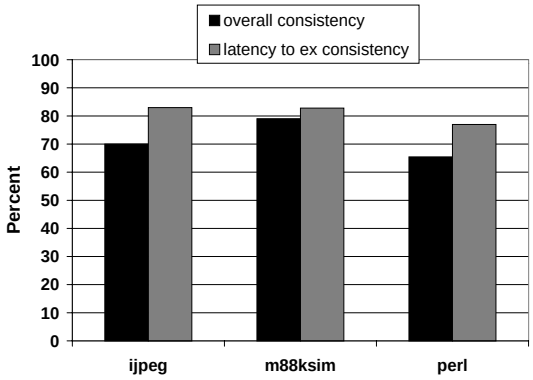

Figure 2: Latency to EX Consistency

a lower bound on the success rate of an issue logic prediction mechanism, assuming that a predictor can be properly trained. With the SPECint95 benchmarks this lower bound would be between 49.3% and 77.0% with the average lower bound being approximately 66%. Improving on this lower bound is discussed in sections 3.1.2 and 3.3.

3.1.2 Instruction Consistency Statistics

Another type of profile statistic that was collected studied the consistency of an individual instruction's path through the stages of the pipeline. We define a consistent instruction to be one that spends the same amount of time in each stage of the pipeline every time it executes. Figure 2 shows that instructions are consistent roughly 65%-80% of the time. This percentage grows to 75-85% when restricting the definition of consistency to measure only the time it takes an instruction to move from Fetch to Execute stages of the pipeline.

The instruction consistency statistics tend to indicate a high amount of instruction interdependence in determining the issue windows. Probabilistically, the issue windows would only have a correlation of $0.75^4 = 31.6\%$ if each of the instructions in the issue window was independently determined with a consistency rate of 75% (the minimum latency-to-Execute consistency shown in Figure 2). We found this correlation to be much higher, which shows that instructions tend to be issued with the same instructions for a given fetchPC.

Furthermore, the instantaneous statistics found a correlation of roughly 50-75% between the fetchPC and issued instructions, and a narrower 70-75% for the three sample benchmarks (ijpeg, m88ksim, perl). Given that cache and TLB misses are filtered out, we found that the instruction consistency data bounds functional unit and data dependencies between 15-25% (the opposite of the latency-to-Execute consistency) for the three benchmarks. Comparing this bound with the equivalent instantaneous statistics, it was expected that the majority of the inaccuracy in the instantaneous statistics is due to functional unit and data dependencies, not errors in branch prediction. However, other factors that may also affect instruction consistency, such as buffer sizes in the Fetch and Dispatch stages, were not accounted for.

Unlike the instataneous statistics, the data for the three sample benchmarks were collected for instruction traces that ran for several million cycles. Due to the volume of data we had to process, SimpleScalar's pipetrace facility had to be re-written to produce binary pipetraces, and the output had to be separated into multiple traces of one million instructions. Summary data from the separate traces was then aggregated in the final statistics. Although the data from the traces were not necessarily independent, as the same PC may have been present in multiple traces, the traces were sufficiently long to reflect what a processor would view before retraining its data set.

Statistics for all eight SPECint95 benchmarks were not collected because, unlike the instantaneous statistics, we did not envision an actual power-saving prediction scheme that correlated well with this concept of instruction consistency. An analysis using these three benchmarks was sufficient for our needs and is validated by the results presented in section 3.3.

3.2 Renaming

The SimpleScalar RUU scheme [10] ties renaming to all aspects of out-of-order execution. Each RUU serves as a physical destination register for renaming as well as a reservation station and part of the reorder buffer and

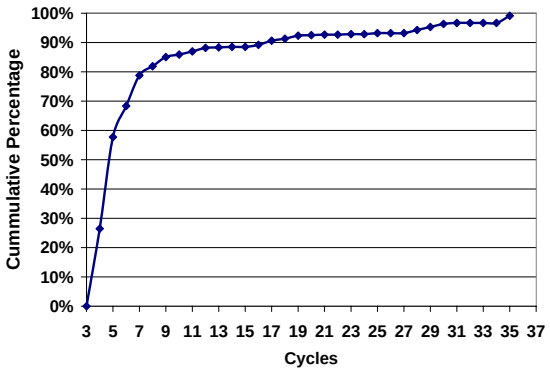

Figure 3: Register Mapping Reservation Duration

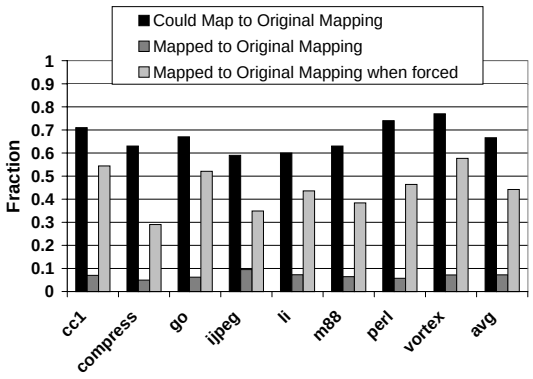

Figure 4: Register Renaming Behavior

wakeup window. Unfortunately, this makes it difficult to quantify the independent effects of register renaming or to see how renaming effects other aspects of out-of-order control. Nevertheless, we were able to extended SimpleScalar to manage a "free list" of physical registers and added extensive per-instruction renaming statistics including minimum, maximum, and average length of time a logical destination register stays mapped to particular physical register; number of dynamic instances of an instruction; and the number of times an register is renamed to its initial mapping.

Initial results were compiled by examining these statistics after running 500,000 instructions of a Perl benchmark on a default SimpleScalar configuration (with 16 RUUs). Figure 3 shows that every instruction needs a physical register for at least three cycles, but 80% relinquish their mapping within seven cycles. This gives hope that a scheme could quickly reassign a mapping.

Figure 4 quantifies this hope. We run all eight spec benchmarks using reduced datasets shown in Table 1. Each benchmark was run for 50 million instructions. Just before a logical register is assigned an RUU, we check to see if its original RUU is available. In 59%-77% of the cases, the original RUU available. Despite this, less than 10% of renamed registers map to their original mapping.

To finalize the upper bound we *forced* this remapping to occur if the physical register first used for this instruction was free. The results are also displayed in Figure 4. Examining the average case, one sees that 45% of instructions can be renamed to their original physical register when such a renaming is forced. Interestingly, this is less than the 67% potential shown in the

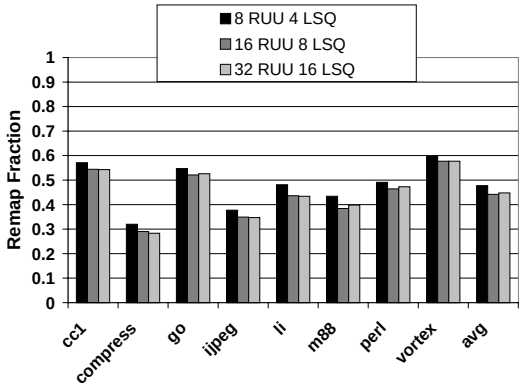

Figure 5: Registers Remapped When Forced

first bar. We attribute this difference to the occasional "stealing" of a mapping desired by a future instruction. Fortunately, as explained earlier, mappings are reserved for a very short period of time, so the impact is obvious but not severe.

Intuition suggests that fewer choices of physical registers would lead to higher correlation at the expense of performance. Surprisingly we found the number of RUUs (which correspond to available physical registers) to play a very minor role in the remapping frequency. Testing various configurations yields Figure 5.

Since modern superscalar processors will rely increasingly on renaming multiple registers at a time [12], the results we have gathered suggest there is lots of potential to improve the register renaming structures in an out-of-order machine. If the logical structures could

coax registers into using their original mapping with high accuracy (greater than the 45% that naturally occurs), one could attempt to speculate the rename process, checking its correctness in parallel. This would allow for a physical register file with fewer ports saving both power and performance. Currently, if such a speculation scheme were implemented to save power, it would fail 55% of the time and require a *very* fast recovery scheme to justify the lost performance.

3.3 Issue Oracle

As discussed previously (section 3.1.1), the fetchPC-versus-issue window correlation statistics provide a lower bound for the prediction success rate of a properly trained issue window predictor. However, this lower bound can be improved upon because the correlation statistics do not differentiate between genuine mispredicts and false mispredicts. We implement an *oracle* that simulates a hardware prediction scheme residing in the Fetch stage of an out-of-order microprocessor that sends predicted issue instructions directly into the wakeup window. The oracle is an ideal predictor trained with profile data from previous executions of an instruction trace. In a non-ideal hardware realization of this scheme, the Dispatch stage (broadcast and select) would be disabled when possible, allowing for a shorter "hot pipeline."

3.3.1 Background

A *genuine* mispredict may either be a branch mispredict, where the oracle prediction does not correspond with the instruction path chosen by current branch direction; or a synchronization mispredict, which occurs when an instruction the oracle is predicting has already been issued prior to the oracle turning on. We define a *false* mispredict to be one where the oracle is predicting along the correct branch path and the predicted instruction has not yet issued, but the predicted issue window does not match the instructions that are actually issued. We call this a false mispredict because this type of mispredict would generally be caused by functional unit dependencies and can be resolved through the dynamic wakeup window logic. Thus, this type of mispredict does not affect the performance of the oracle, though it causes the wakeup window size to increase temporarily while the dependency is being resolved.

If an issue window prediction scheme were implemented, a true lower bound on the prediction success rate would be one without false mispredicts. In a loop with a large number of executions, the false mispredicts will dominate the genuine mispredicts once a predictor successfully synchronizes with the processor. Thus, the lower bound should be higher than was found in section 3.1.1. In section 3.1.2 we estimate that this improvement should be around 15-25%.

The instantaneous correlation statistics do not concern themselves with bounding the number of consecutive cycles a prediction scheme can be expected to run. This statistic must necessarily be high in order to amortize the costs of switching into a prediction mode. We provide measurements of the oracle average run length in section 3.3.3. The results of the oracle experiment provides better bounds on this type of data, and helps estimate the power-savings possible by predicting issue windows.

3.3.2 Implementation Notes

In the oracle prediction logic we have implemented, the dispatch stage is not disabled, but the oracle only allows *predicted* instructions to enter the issue wakeup window. SimpleScalar does not have a distinct issue wakeup window, so, in order to simulate the effects on wakeup window size with an issue oracle, an issue wakeup window was implemented.

The oracle is designed to measure prediction accuracy, run length, and wakeup window size of an ideal prediction scheme. Performance costs of mode switching are assumed to be zero, because reasonable design targets for these costs can be estimated from the other data. However, we do speculate that the performance cost can be kept reasonably small. Assuming that predictions are sent directly from the Fetch stage to the wakeup window, then only instructions in the functional units and wakeup window, as well as some instructions in the reorder buffer, will have to be squashed. The extra cycles saved by this shorter pipeline, versus a branch mispredict in the full pipeline, could be used for the more complex cleanup the reorder buffer may require.

3.3.2.1 Wakeup Window

We chose to implement an oracle that simulates utilizing dynamic wakeup window logic. However, it would also be possible to implement an oracle that uses no issue wakeup window, which would correspond to even further power savings and eliminate another stage in the microprocessor pipeline. Such a scheme, though, can be less desirable because it makes the oracle predictions more sensitive to functional unit dependencies, thus causing false mispredictions. From the instruction consistency measurements of the latency to the Execute stage (section 3.1.2), we believe that a bound on these dependencies could range from 15-25% in certain benchmarks. In the event of a false mispredict, the oracle must either stall the entire frontend to wait for the dependency to be resolved, or it must stop predicting.

In an oracle scheme that utilizes an issue wakeup window, these data and functional unit dependencies will only cause the issue wakeup window usage to grow temporarily, and will eventually be resolved by the dynamic

wakeup window logic. Furthermore, the oracle does not have to resolve which functional unit the instruction is issued to, which decreases the occurrence of these dependencies.

3.3.2.2 Oracle Datasets

Storing out-of-order data with in-order data requires consistent traces for the oracle dataset. To have a correct dataset for an oracle to predict from, all of the in-order instructions must eventually be predicted to be issued by the oracle. Otherwise, a non-predicted instruction would eventually cause the pipeline to stall due to data dependencies. It would also cause instructions to fill up the wakeup window, which is counter to the goal of power-saving.

Using the fetchPC as the index, the oracle dataset was filled with "consistent" traces. The requirements for a consistent trace were very stringent. We examined fetchPCs and their issue windows from the previous data collection. Using this data, a consistent trace can be collected by examining each fetchPC in the SimpleScalar pipetrace. If two consecutive occurrences of a fetchPC match in their issue window, then this data is deemed to be part of a "consistent trace" and is stored in the oracle dataset. In our implementation of this algorithm we chose the simplifying assumption that the dataset would be trained off of the first instance of the fetchPC rather than using the mode found in the full profile statistics.

While a consistent trace does not perfectly ensure that non-predictions are prevented, it is a likely indicator they will not occur. However, we found even this non-perfect requirement to be too stringent. For ijpeg, which only had 100 unique fetch PCs out of the 250,000 committed instructions, this algorithm only returned one fetchPC for the dataset. Other benchmarks reported better results, but the average cycle coverage still was still only 24%, even though the majority of the benchmarks had less than 700 unique fetchPCs. The cycle coverage is defined to be the ratio of the number of cycles the fetchPC occurs to to the total number of cycles. It is an indication of the number of cycles during which an oracle could make a prediction during the trace. We sought to improve the cycle coverage of the dataset by relaxing the dataset algorithm to allow for a dual predictor oracle.

Instead of requiring the second occurrence of the fetchPC have the same issue window mode as the first occurrence, we extended our dataset collection to also look at the third occurrence. If the third occurrence matched and the second occurrence did not, then the second occurrence was stored as an alternate prediction. These requirements would have met the need to have a consistent trace, but also allow for some variance between consecutive loop executions. Further, the issue window correlation statistics show that the mode of the issue windows generally represented more 65% of all issue windows, so with high probability we expected that two out of the three occurrences would match.

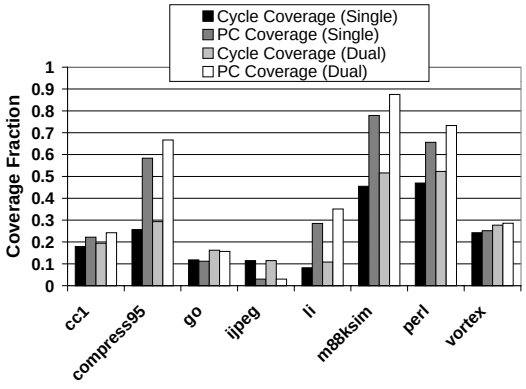

Figure 6: Oracle Dataset Coverage

Even this dual predictor scheme, though, did not significantly expand the dataset. The dual predictor datasets did not expand the cycle coverage of any of the datasets by more than 5%, at what would be double the storage cost and increased complexity in a hardware implementation. However, this does not indicate that a dual predictor scheme is not without merit; rather, it suggests that the presented algorithm for gathering a consistent trace requires improvement in order to collect larger, high-coverage datasets.

Because the dual predictor scheme did not significantly increase the size of the datasets, we chose to calculate our results using a single predictor scheme. Figure 6 shows how many of the fetchPCs were included in a consistent trace. These coverage statistics were compared with the number of unique fetchPCs seen in a trace to give more relevant, normalized coverage data. The

3.3.3 Oracle Prediction Results

The prediction success rate (along with the misprediction and no prediction rate) is shown in Figure 7. For several of the benchmarks, cc1, compress95, and li in particular, the success rate is much higher than the lower bound found in section 3.1.1. In section 3.1.2, our analysis showed that we could expect the rate of false mispredicts to range between 15-25% for ijpeg, perl, and m88ksim. In the actual oracle prediction data,

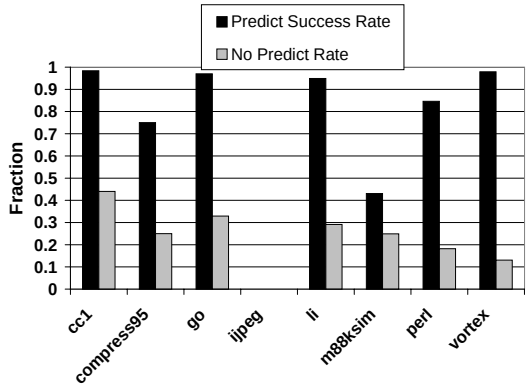

Figure 7: Oracle Prediction Statistics

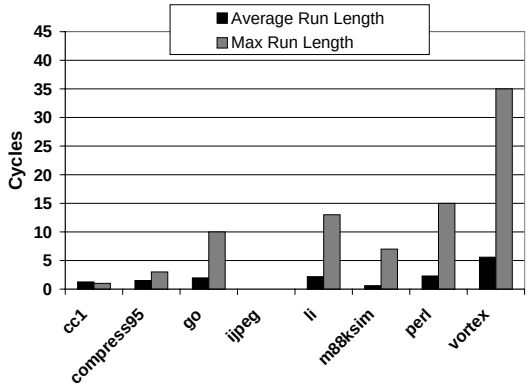

Figure 8: Oracle Average and Maximum Run Length

this rate appeared to be even higher for some of the SPECint95 benchmarks. li had a success rate of 94.9%, which is nearly 40% higher than the single predictor lower bound given by the instantaneous issue statistics discussed in section 3.1.1. However, our lower bound was not verified for all benchmarks because ijpeg and m88ksim both showed inconsistent results. The results for ijpeg were understandable given that the dataset only had a single fetchPC. However, the extremely small size of this dataset and the abysmal predict success rate of m88ksim (43%) were surprising considering the loop-oriented nature of both benchmarks. The oracle predictions were expected to be especially accurate for these types of benchmarks.

The *run length* of the oracle is the number of consecutive cycles during which it attempted a prediction. Figure 8 shows a table of the average run lengths compared to the maximum run length for each of the benchmarks. The average run length of the oracle was much lower than we anticipated. Even instructions with fairly high prediction success rates, such as cc1, had average run lengths below five, with most having average run lengths between one and three. However, we believe that both the short average run length and high mispredict rate for certain benchmarks may be due to the attempt to apply the oracle to very long data runs. For a much shorter run of cc1, we were able to find an average run length of 18.76 cycles over 41 uses of the oracle. While this may be a spurious result, it may also indicate much more positive results for an oracle implemented with dynamic retraining.

The dominant reason for the oracle runs ending was lack of an available prediction. We do not believe that the this indicates the program is moving to a different path of execution due to a branch, as this statistic might suggest, because the average run length of the oracle is too short. Furthermore, we ran a simple "Hello World" type program that executed a simple loop with no other branches; this loop had a similarly high exit rate due to no predictions. While this may indicate problems with the dataset collection, we also believe that this may indicate a problem with our oracle implementation in SimpleScalar. Because our implementation of the oracle sits on top of the SimpleScalar model, it is possible that this high "no predict" rate is being caused by second-order effects, such as fetch buffer saturation, that may not be properly handled.

3.3.4 Wakeup Window Usage Statistics

The wakeup window is not emptied when the oracle initiates in order to show the size of the wakeup window over time once the oracle begins prediction. It is assumed that the wakeup window size during the first cycle of oracle execution is a good representation of the initial wakeup window state. Figure 9 shows the length of the wakeup window queue over time. Time "zero" is when the oracle begins execution. As expected, once the oracle starts, the wakeup window usage quickly decreases over time until it approaches actual average oracle utilization, which is appears to be less than two. The oracle is able to achieve this low utilization because it does not have to place instructions into the wakeup window until they are predicted to be issued. Thus, the oracle will insert instructions at a rate much closer to the IPC of the program, which is generally below two, rather than use the full dispatch bandwidth.

Future implementations may want to examine the effect of squashing the entire wakeup window when the

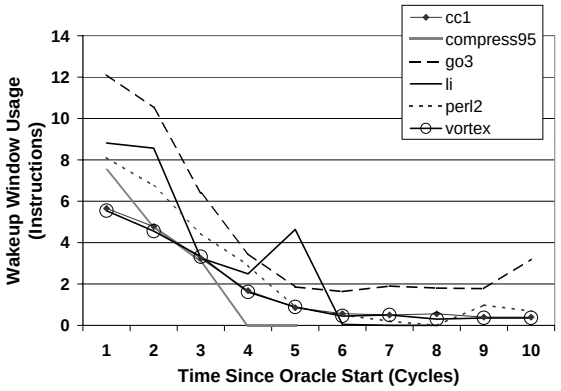

Figure 9: Wakeup Window Usage

oracle begins predicting. Squashing the entire wakeup window is beneficial because it immediately reduces the size of the wakeup window utilization and allows power-savings to be realized as soon as the oracle starts predicting. However, there are two opposing effects that would also need to be measured. It is possible that the current wakeup window contains instructions that need to be issued before the oracle begins predicting in order to prevent the oracle from halting prematurely due to data dependencies. Although this is essentially a misprediction by the oracle, it is one that could be resolved by the wakeup window logic. On the other hand, it is also possible that by not squashing the current wakeup window, an instruction may be issued before it is predicted to, leading to a later mispredict. Both of these effects would have to be measured to determine which is a more desirable implementation.

3.4 Oracle Coverage

The oracle coverage (the ratio number of cycles of oracle prediction to the total number of cycles) was below 1% for all eight benchmarks Although the oracle does not predict if it is seeing a PC for the first time, we expected that the oracle coverage would still be close to the cycle coverage. The dataset collection issues and buffer saturation effects discussed in section 3.3.3 may be the cause of this low coverage, as it could be the high "no predict" rates that are causing the oracle to prematurely exit.

Assuming that the ISA is not specifically extended to enable issue logic prediction, hardware for training the predictors would most likely be running every cycle. Even if the oracle coverage matched the cycle coverage rate, this would most likely be too low to amortize the power consumption costs of the training hardware. In order for power savings to be realized, better algorithms for training the dataset would need to be found.

4. HARDWARE IMPLICATIONS

The issue window oracle presented in this paper resembles a simple branch predictor in that a fetchPC is used to index an array of predictions. While this scheme may have its merits, we also believe that a trace cache-style microprocessor could be modified to include issue logic prediction at minimal marginal cost of hardware and power. Modifying a trace cache-style system, however, requires that the issue of storing in-order and out-of-order data be resolved. One potential scheme we envision a would be to store decode instruction data in out-of-order, with four to five-bit virtual reorder tags stored with the instruction to allow the instruction trace to be correctly reassembled into the reorder buffer. The order in the trace cache would correspond to the issue window order, with stop bits denoting the beginning and end of a predicted issue window.

The additional hardware cost of this implementation would be minimal because it would only require an additional five to six bits per instruction to store the stop and reorder bits. Trace caches are designed to maximize the fetch bandwidth for a superscalar processor. However, because a issue logic prediction scheme effectively throttles the introduction of instructions to the IPC rate, we envision that this extra fetch bandwidth could instead be used to amortize the cost of the extra bit storage. This additional bandwidth could be used to fetch multiple issue window predictions, which would allow the trace cache fetch to be disabled the next cycle(s).

As our difficulties with collecting appropriate oracle datasets have shown, backend logic for filling the trace cache lines is likely to be much more sophisticated, and thus more power-consuming, then traditional trace cache techniques. Further research is necessary to investigate a scheme that provides both good predictions as well as good coverage in order to make a trace cache issue logic predictor scheme worthwhile.

Due to the low register remapping rate we found in our results, we currently do not envision an issue prediction scheme that would allow static renaming, which would be necessary if renaming logic were integrated into an issue prediction scheme. As shown in section 4, a circular register renaming allocation scheme is not able to achieve a high remapping rate, and a different scheme would be necessary to realize effective renaming prediction. If such a scheme were created, it would allow further power savings by allowing the register renaming units to be powered off.

5. CONCLUSION

Increasing complexity in superscalar processors demands increasing attention to low-energy designs. We have shown that there is opportunity to improve the regularity of register renaming which may lead to physical register files with fewer read ports or allow the implementation of speculative and/or static renaming which could be tied into a trace cache processor. We measured a high correlation between the fetchPC and issue window; and we implemented a single-predictor issue window oracle to verify that an architectural model can take advantage of this high correlation. The issue window oracle used a static prediction scheme based on profile statistics for 250,000-instruction sequences of code. Our results suggest that the issue window statistics do indeed give a lower bound on a issue logic prediction scheme for most instruction traces, although we have not resolved why we were unable to gather sufficient datasets for m88ksim and ijpeg. We also find that while our issue prediction scheme has poor coverage, it can be successful for a majority of benchmarks when used over short regions of code. We believe a dynamic prediction scheme could provide the predictor with timely and consistent data and could greatly improve upon our results. Finally, we found that use of an issue window predictor can allow for dynamic reducing the size of the wakeup window to reduce power consumption. It is suggested that the extra power required by incorporating an issue window predictor in an already-existing front-end speculation structure would be more than offset by the savings in the complex issue logic.

6. CODE

All tables, figures, and performance results included in this paper were generated by code produced by Kenneth Barr, Kenneth Conley, or Serhii Zhak and are available upon request.

7. ACKNOWLEDGEMENTS

We thank our partner, Serhii Zhak, for his initial research and direction on the subject of memoizing control logic, for helping to generate the Issue Window Prediction Accuracy figure, and for his contributions to the hardware implications section of this paper. We also appreciate the help of Professor Krste Asanovic who provided insightful direction throughout the evolution of the project.

8. REFERENCES

[1] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] R. Canal and A. Gonzalez. A low-complexity issue logic. In *Proceedings of the 14th International Conference on Supercomputing (ICS-00)*, May 2000.

[3] J. A. Farrell and T. C. Fischer. Issue logic for a 600-Mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, May 1998.

[4] D. Folegnani and A. Gonzalez. Reducing power consumption of the issue logic. In *Workshop on Complexity Effective Design held in conjunction with ISCA 27*, June 2000.

[5] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *35th Annual Conference on Design Automation*, pages 726–731, June 1998.

[6] M. Hiraki, R. Bajwa, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Sasaki, and K. Seki. Stage-skip pipeline: A low power processor architecture using a decoded instruction buffer. In *Proceedings of the 1996 International Symposium on Low Power Electronics and Design*, pages 353–358, 1996.

[7] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *31st annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[8] U. Ko, A. Hill, and P. Balsara. Design techniques for high-performance, energy-efficient control logic. In *International Symposium on Low Power Electronics and Design*, 1996.

[9] S. Palacharla, N. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings 24th International Symposium on Computer Architecture*, pages 206–218, June 1997.

[10] G. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3), March 1990.

[11] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *27th Annual International Symposium on Microarchitecture*, 1994.

[12] S. Vajapeyam and T. Mitra. Improving superscalar dispatch and issue by exploiting dynamic code sequences. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[13] C. Wang and K. Roy. An activity-driven encoding scheme for power optimization in microprogrammed control unit. *IEEE Transactions on VLSI Systems*, Vol 7(1), March 1999.