In the IEEE/ACM Euro-Par Conference, Lisboa Portugal, 2005.

Instruction Recirculation: Eliminating Counting Logic in Wakeup-Free Schedulers

Joseph J. Sharkey, Dmitry V. Ponomarev

Department of Computer Science
State University of New York
Binghamton, NY 13902 USA
{jsharke, dima}@cs.binghamton.edu

Abstract. The dynamic instruction scheduling logic (the issue queue and the associated control logic) forms the core of an out-of-order microprocessor. Traditional scheduling mechanisms, based on tag broadcasts and associative tag matching logic within the issue queue are limited by high power consumption, large access delay and poor scalability. To address these inefficiencies, researchers have proposed various flavors of so-called *wakeup-free* scheduling logic. Such wakeup-free scheduling techniques remove the wakeup delay from the critical path, but incur other forms of complexity, essentially stemming from the need to keep track of the cycle when each physical register will become ready and when each instruction can be (speculatively) issued. We propose *instruction recirculation* – a wakeup-free instruction scheduler design that completely eliminates all counting and issue time estimation logic inherent in all previously proposed wakeup-free schedulers. This complexity reduction is also accompanied by 3.6% IPC improvement over the state-of-the-art wakeup-free scheduler.

1 Introduction

High-performance superscalar microprocessors rely on dynamic scheduling mechanisms to maximize instruction throughput across a wide variety of applications. The traditional scheduling logic operates in two phases: instruction wakeup and instruction selection. During instruction wakeup, the instructions stored within the issue queue (IQ) are associatively awakened by matching their source register addresses (called tags) against the destination tags of the instructions already selected for the execution. The selection logic, then, selects W out of N awakened instructions and issues them for execution. It has been well documented in the recent literature that the wakeup and selection logic form one of the most critical loops in modern superscalar processors [18,22]. Unless wakeup and selection activities are performed atomically (i.e. within a single cycle), dependent instructions cannot execute in consecutive cycles, which seriously degrades performance. At the same time, both wakeup and selection logic have significant delays [18], so if these activities are performed atomically, then the designers may be forced to either use the lower clock frequency or

limit the size of the IQ, neither of which is desirable. In addition, traditional broadcast-oriented schedulers suffer from high power consumption, which is mainly due to broadcasting the destination tags across long, highly capacitive wakeup buses. For example, the scheduling logic of the Alpha 21264 dissipates about 18% of the total chip power [11].

To address the aforementioned deficiencies, researchers have proposed wakeup-free scheduling schemes, where the traditional tag broadcast and associative tag matching mechanisms are replaced with the capability to predict the issue cycle of an instruction based on the availability information about the source registers. Such wakeup-free scheduling techniques [5,6,8,12,17,19] remove the wakeup delay from the critical path, but need to keep track of the cycle when each physical register becomes ready so that the instructions can be issued just in time to access the value as soon as it becomes available. This is typically accomplished with the use of counters that track the availability of physical registers and also control when instructions can be issued (we describe a generic wakeup-free scheduler in more detail in Section 3). Due to the presence of a large number of these multi-bit counters and the issue time estimation logic, wakeup-free schedulers still incur substantial design complexity.

In this paper, we attempt to improve the performance/complexity trade-offs in the design of wakeup-free schedulers by exploiting the observation that most instructions that are selected for issue are typically among the few oldest in the IQ. Specifically, we introduce a technique called *Instruction Recirculation*, which uses a compacting IQ, where only N instructions at the head of the queue are considered for execution each cycle. Instead of relying on the traditional tag matching mechanisms, these N instructions determine their readiness to issue by checking the status bits associated with their source registers. Instructions at the head of the queue are recirculated back to the tail of the queue if they were not able to issue for specified number of cycles. This allows the younger instructions to be considered for scheduling in the presence of the long-latency events, such as the cache misses. Our results show that instruction recirculation achieves 3.6% better performance on the average than a state-of-the-art wakeup-free scheduler, and at the same time avoids the need to implement the counting logic for predicting the instruction issue time.

2 Simulation Methodology

Our simulation environment includes a detailed cycle-accurate simulator of the microarchitecture and cache hierarchy. We used a modified version of the Simplescalar simulator [4] that implements separate structures for the IQ, re-order buffer, load-store queue, register files, and the rename tables in order to more accurately model the operation of modern processors. All benchmarks were compiled with gcc 2.6.3 (compiler options: -O2) and linked with glibc 1.09, compiled with the same options, to generate the code in the portable ISA (PISA) format. All simulations were run on a subset of the SPEC 2000 benchmarks consisting of 8 integer and 7 floating-point benchmarks using their reference inputs. In all cases, predictors and caches were warmed up for 1 billion committed instructions and statistics were gathered for the

next 500 million instructions. Table 1 presents the configuration of the baseline 4-way processor.

Table 1. Configuration of a simulated processor

Parameter	Configuration
Machine width	4-wide fetch, 4-wide issue, 4 wide commit
Window size	issue queue – as specified, 128 entry LSQ, 256–entry ROB
Function Units and Latency (total/issue)	4 Int Add (1/1), 2 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 2 FP Add (2), 2 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Physical Registers	256 combined integer + floating-point physical registers
L1 I–cache	64 KB, 1–way set–associative, 128 byte line, 1 cycles hit time
L1 D–cache	64 KB, 4–way set–associative, 64 byte line, 2 cycles hit time
L2 Cache unified	2 MB, 8–way set–associative, 128 byte line, 6 cycles hit time
BTB	2048 entry, 2–way set–associative
Branch Predictor	Combined with 1K entry Gshare, 10 bit global history, 4K entry bimodal, 1K entry selector
Memory	128 bit wide, 140 cycles first chunk, 2 cycles interchunk
TLB	32 entry (I), 128 entry (D), fully associative

3 Wakeup-Free Schedulers

Wakeup-free instruction scheduling schemes have recently emerged as a viable alternative to traditional broadcast-oriented scheduling logic in complexity-aware microprocessor designs. Several variations of wakeup-free (a.k.a. broadcast-free) schedulers have been proposed in the recent literature [1,5,6,9,12,16,17]. Instead of using slow, complex and power-hungry CAM-based wakeup logic, these solutions rely on the ability to predict the cycle in which all of an instruction's input operands will become ready and issue the instruction just in time to access the values as soon as they become available. In all wakeup-free designs proposed until now, the counters are used to count down the delay between dispatch and issue for each instruction and also to keep track of the register availability information. Although the various wakeup-free scheduling schemes differ in their implementation details, they are all based on the common concept that the latencies of most instructions are deterministic and that the instruction's issue time can be fairly accurately predicted at the time of instruction dispatching.

For the analysis in this paper, we implemented a generic wakeup-free scheduling scheme, loosely based on the Cyclone scheduler [9]. At the time of instruction dispatching, a *pre-scheduler* is used to predict the number of cycles until each instruction will become ready for issue. We consider the *pre-scheduler* which is similar to that of [9], with the addition of a bimodal load hit/miss predictor and a load/store dependence predictor as proposed in [12] to improve the accuracy in scheduling load-dependent instructions. Instructions passing through the pre-scheduling stage check the availability of their source operands (by reading the availability counters) and use the maximum of these values to determine the number of cycles that will elapse be-

fore the instruction is ready for issue. This result becomes the delay counter of the instruction and is placed, along with the instruction itself, into the allocated IQ entry. We assume that the delay counter calculation can be performed in parallel with register renaming and thus it does not add an extra stage to the front end of the pipeline. If this extra stage is accounted for, the performance of the generic wakeup-free scheduler will be slightly worse than what is reported here.

Every cycle, the delay counters associated with each instruction in the IQ are decremented by one. When the delay counter falls to zero, the instruction becomes speculatively ready to execute, but must check the register ready bits of its source operands to be certain before it can be selected for execution. The hardware support for such checks is in the form of a bit-vector with one bit for each physical register. If the check succeeds, indicating that all source operands are indeed ready, the instruction is selected. We limit the number of instructions that can check their ready bits in a single cycle to only 8, requiring a register ready bit-vector with 16 read ports. Simple logic is assumed to arbitrate for these ports, using positional priority.

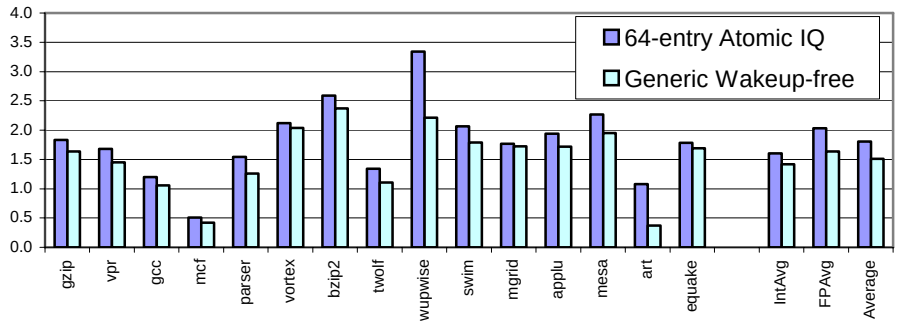

Fig. 1. IPCs of wakeup-free and traditional broadcast-based schedulers

The IPC results for the generic wakeup-free scheduler with a 64-entry IQ are presented in Figure 1, along with the results for a 64-entry traditional atomic IQ where wakeup and select activities are implemented as an atomic operation within a single cycle. The wakeup-free scheme, as described above, exhibits 16.5% performance degradation on the average as compared to the 64-entry atomic IQ. This is consistent with the results presented in both [9] and [12], where the performance losses compared to the baseline are 17% and 14% respectively (although each of those schemes is presented for a machine configuration slightly different from ours). This performance loss can be attributed mainly to the inaccuracy in the instruction issue time estimation due to variable latency operations such as memory accesses and possible delays during instruction selection. As a result of mispredictions, non-ready instructions may deny the issue bandwidth to the ready instructions, causing a delay in the issue of those and leading to further mispedictions for the instructions dependent on the delayed ones, leading to a so-called “snowball effect”. In the extreme case, all of the instructions in the queue can compete for issue in the same cycle, rendering the issue time estimation useless. In fact, we have observed such situation in our simulations on numerous occasions.

Although the wakeup-free scheme eliminates the tag broadcast and tag matching logic in the scheduler, it does not come without a cost. Additional circuitry must be added in the front end of the pipeline to make the delay predictions. This delay prediction logic can become quite complex when multiple instructions are co-dispatched. To take dependencies into account, adders with multiple inputs (K-1 inputs in the case of a K-way machine) or an adder tree have to be used to get the initial counter values of the co-dispatched instructions in the worst case. Such adder structures may well form a critical path with the increase of the issue width.

Another source of complexity in the wakeup-free schedulers is the inherent counting logic. Two sets of counters have to be maintained– the ones that track the register availability information (one counter per physical register) and the ones that control the number of cycles that each instruction waits in the IQ before attempting to issue (one per instruction in the queue). The hardware structure that maintains the register availability counters must be multi-ported and all of these counters need to be decremented every cycle.

In the next section, we propose *Instruction Recirculation* - a wakeup-free scheduler design which achieves a better performance and completely eliminates all counting and issue time estimation logic inherent in all previously proposed wakeup-free schedulers.

4. Instruction Recirculation

The motivation for instruction recirculation stems from the observation that even in an out-or-order machine, a large percentage of issued instructions are among the few oldest instructions in the IQ. As demonstrated by Figure 2, more than 60% of all dynamic instructions that are selected for execution in a processor with a 64-entry IQ are among the 8 oldest in the scheduling window. Simply reducing the IQ size, however, is not a sufficient solution to the problem of complexity reduction because a small queue quickly saturates under a long latency event, such as a cache miss, causing performance degradation.

The solution that we propose instead is to have a scheduler that examines only a small group of instructions in the window for execution, but is also capable of detecting long latency events (such as cache misses) and quickly move dependent instructions out of the front-end of the queue (if they cannot issue for a predetermined number of cycles) to allow possibly independent instructions down the stream the chance to execute. The older instructions can then be recirculated back into the tail end of the queue at a later time. We call this technique instruction recirculation. In the rest of this section we describe the details of our design.

The block diagram of instruction recirculation is shown in Figure 3. This design relies on the use of a compacting IQ, somewhat similar to the Cyclone scheduler [9], where instructions are dispatched to the tail block of the queue and work their way to the head block, from where they eventually get issued. The IQ is organized into several *n*-instruction blocks. Each row can compact independently of the other rows and each instruction can compact forward only one block at a time within its row. Only N instructions at the head of the queue participate in checking the register ready bit

(RRB) vector and selection each cycle. Finally, an *N*-entry *recirculation buffer* is used. Its purpose is explained below. Conceptually, the instruction recirculation scheduler can be viewed as a wakeup-free scheduler in which the instructions present in the head block in any given cycle are predicted as "ready" and thus check the register ready bit vector.

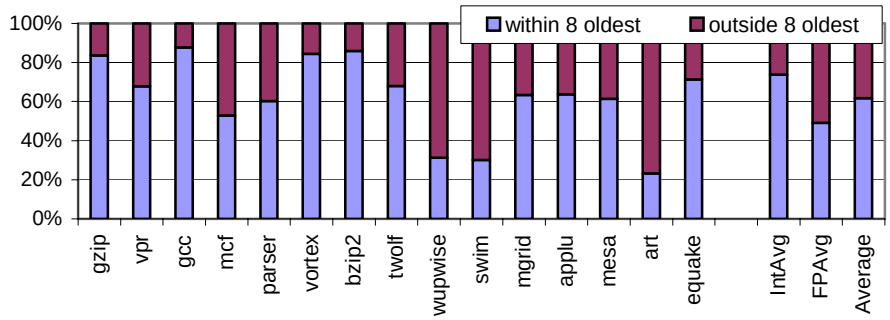

Fig. 2. Percentage of dynamic instructions issued that are among the 8 oldest in a traditional 64-entry atomic scheduler.

The scheduler of Figure 3 operates in the following manner. Every cycle, N instructions within the head block of the queue check the ready bits of their corresponding source physical registers. If both sources are ready, and the instruction succeeds in acquiring the issue slot, the instruction is issued and the corresponding row within the IQ is compacted. Note that the issue rate from the head block is still limited by the issue width of the processor, which is 4 in our experiments (same as in the baseline machine). The recirculation scheduler (just as any other wakeup-free scheme), thus, still requires the selection logic to arbitrate among the instructions in the head block. However, this selection logic is much simpler than similar logic in the traditional scheduler.

When the instruction issue rate falls below a certain number, called the *recirculation threshold*, N instructions in the head block are moved into the recirculation buffer and the queue is compacted forward (for all rows). Such a compaction gives the next *N* instructions in the queue the opportunity to execute if they are ready. The scheduler remains in this state (issuing from the head block and compacting the individual rows as needed) until the recirculation threshold is reached again. At this point, the instructions in the head block are again moved, as before, into the recirculation buffer and the instructions sitting in the recirculation buffer are moved into the tail end of the queue. As instructions are circulated through the queue, they are each given a chance to execute if they are ready.

Notice that it takes a different number of recirculations for each instruction to return to its original position at the head of the queue. In the worst case, an instruction returns to the head block after K recirculations, where K is determined as the size of the IQ divided by *n*. Some of the original instructions from the head block can return there early if the compaction within certain rows progresses at a faster rate, i.e. the rows are compacted individually in-between block recirculations. The normal instruction dispatching continues during instruction recirculation, but the instructions that

are already in the queue (i.e. recirculating) are always given higher priority over the newly dispatched instructions for the access to the tail block of the queue.
The goal of this mechanism is to quickly respond to the falling issue rates from the block of oldest instructions and consider other instructions for issue. This is useful if, for example, the instructions at the head block depend on a load that missed in the cache, but subsequent instructions are independent.

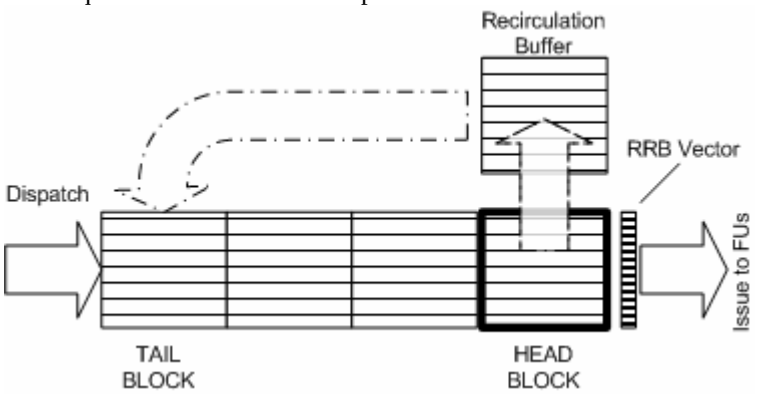

Fig. 3. Instruction Recirculation

The selection of the recirculation threshold is critical to the performance of this scheme. In some cases, it may be more beneficial to wait rather than recirculate the entire block and encounter the full latency of multiple recirculations to bring these instructions back into the head block of the queue. We evaluated many configurations and present the results (Figure 4) for a few configurations that were representative of the rest. All of the presented configurations have a 56-entry IQ with an 8-entry recirculation buffer for a total of 64-entries in the scheduler to match that of the baseline case. Notice that in the interests of space we only present the averages across all benchmarks in this figure. The configurations presented are marked as x/y and can be interpreted as follows: the recirculation cycle occurs if, for *x* consecutive cycles, the issue rate is *y* instructions or less. Recirculation can sometimes degrade performance because it moves the oldest instructions out of the head of the queue for several cycles and thus does not allow them to be considered for execution. Presumably, many instructions deeper in the instruction window will be dependent on the oldest instructions, either directly or indirectly. Thus, it is important that the recirculation parameters be chosen carefully to allow independent instructions the opportunity to execute, but at the same time bring the oldest instructions back to the head of the queue quickly. In the graph of Figure 4 we only show the configurations with y=0, experiments with y=1 showed similar trends.
The best performance is achieved with the use of a 16/0 configuration, as shown by the graph, which degrades performance by 12.9% compared to a 64-entry traditional broadcast-based scheduler. The larger thresholds (20/0 for example) suffer because recirculations occur less often, and thus opportunities for independent instructions to execute are lost. The configurations with smaller threshold values (2/0, 4/0, 8/0) exhibit lower performance because instructions are recirculated too eagerly and may

take several cycles (depending on compaction patterns, as discussed above) to return to the head block of the queue.

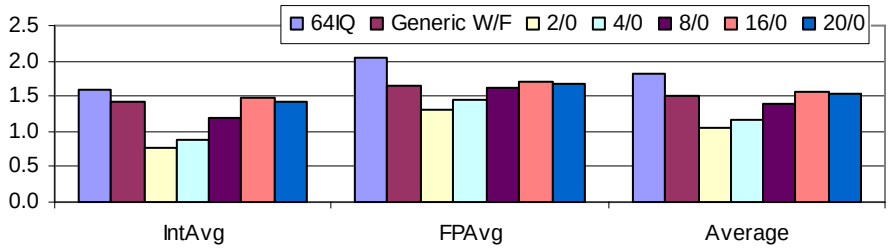

Fig. 4. IPC results of various configurations of the *instruction recirculation* scheduler. The configurations presented are marked as x/y and can be interpreted as follows: the recirculation cycle occurs if, for *x* consecutive cycles, the issue rate is *y* instructions or less.

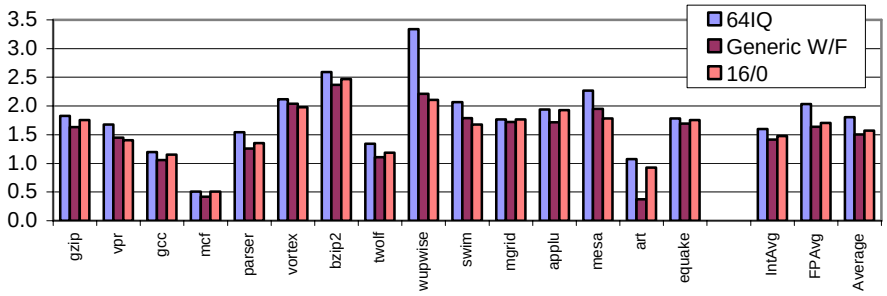

Fig. 5. Per-benchmark IPC of the best configuration of Instruction Recirculation.

As a further comparison, the generic wakeup-free scheduler (as described in the previous section) degrades performance by 16.5% as compared to the same baseline, indicating that recirculation can perform 3.6% better than an aggressive state-of-the-art wakeup-free scheduler.

Figure 5 presents the per-benchmark IPC values for the 64-entry atomic queue, the generic wakeup-free scheduler from Section 3, and the best configuration for instruction recirculation. Instruction Recirculation outperforms the generic wakeup free scheduler for 10 of the examined benchmarks. The largest differences are seen on *art* and *mcf*, where recirculation shows a 149.6% and 21.3% improvement over the generic wakeup-free scheduler, respectively. This is because the poor memory behavior of these two benchmarks significantly impacts the wakeup-free scheduler's ability to predict wakeup times. Instruction Recirculation, however, does not rely on predictions and can dynamically adapt to the memory behavior of individual benchmarks. The generic wakeup-free scheduler provides higher IPC than instruction recirculation for five of the benchmarks (*vpr*, *vortex*, *wupwise*, *swim*, *mesa*), with the largest difference (8.5%) observed for the *mesa* benchmark.

5. Related Work

Scheduling techniques based on predicting the issue cycle of an instruction [5,6,8,9,12,16,17] remove the wakeup delay from the critical path, but need to keep track of the cycle when each physical register will become ready. [5] proposed the "distance scheme issue logic" that reorders instructions during dispatch time based on predicted wakeup times. In [8], the wakeup time prediction occurs in parallel with the instruction fetching. [9,12] remove the counters from the IQ and instead use *pre-scheduling* predictions to determine the placement of instructions in the IQ which, in turn, determines the number of cycles until the instruction is considered for execution. [16] also removes the counters from the IQ and uses *pre-scheduling* predictions to distribute instructions amongst the variable sized FIFOs.
Alternative mechanisms have also been proposed to reduce the complexity and access delay of the dynamic scheduling logic. To pipeline the scheduling logic without hindering the ability to execute dependent instructions back-to-back, Stark et.al. [22] proposed to use the status of an instruction's grandparents to wakeup the instruction earlier in a speculative manner. Kim and Lipasti [14] proposed grouping of two (or more) dependent single-cycle operations into so-called Macro-OP (MOP), which represents an atomic scheduling entity with multi-cycle execution latency. As a result, the scheduling logic can be pipelined with much smaller impact on the IPC. Other proposals have introduced new scheduling techniques with the goal of designing scalable dynamic schedulers [2,15,7,19,21]. Brown et.al. [3] proposed to remove the selection logic from the critical path by exploiting the fact that the number of ready instructions in a given cycle is typically smaller than the processor's issue width. Ernst et.al. [10] introduced specialized IQ entries for instructions with various numbers of non-ready operands. In [20], instruction packing was proposed to dynamically assign instructions to either a full IQ entry or a half IQ entry, depending on the number of ready sources. In [13], half of the tag comparators are offloaded from the fast wakeup bus and are connected to the slow wakeup bus, where the tags are broadcast one cycle later. In [1], instructions are issued from multiple FIFO buffers such that multiple dependency chains may be intermixed within a single FIFO.

6. Concluding Remarks

The wakeup logic of dynamic instruction schedulers has significant delay and power consumption. To address this scalability of the schedulers, and/or support higher clock frequencies, researchers have proposed wakeup-free scheduling solutions where the traditional wakeup logic is replaced by the capability to estimate instruction issue time through the use of counters. In this works, we extended these proposals and introduced instruction recirculation – a wakeup-free instruction scheduler design, which completely eliminates all counting and issue time estimation logic inherent in all previously proposed wakeup-free schedulers. This complexity reduction is accompanied by a 3.6% IPC gain over the state-of-the-art wakeup-free scheduler.

7. Acknowledgements

We would like to thank Matt Yourst for help with the microarchitectural simulation environment. We would also like to thank Kanad Ghose and Deniz Balkan for useful comments on earlier drafts of this paper.

8. References

[1] J. Abella, A.Gonzalez, "Low-Complexity Distributed Issue Queue", HPCA, 2004.
[2] E. Brekelbaum et. al., "Hierarchical Scheduling Windows", in Proc. of MICRO, 2002.
[3] M. Brown, J. Stark, Y. Patt. "Select-Free Instruction Scheduling Logic", in the 34th International Symposium on Microarchitecture, 2001.
[4] D. Burger and T. Austin, "The SimpleScalar tool set: V. 2.0", Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases.
[5] R.Canal, A. Gonzalez, "A Low-Complexity Issue Logic", in Proc. of the International Conference on Supercomputing (ICS), 2000.
[6] R.Canal, A.Gonzalez, "Reducing the Complexity of the Issue Logic", in Proc, of the Int'l. Conf. on Supercomputing (ICS), 2001.
[7] A. Cristal, et.al., "Out-of-Order Commit Processors", in Proc. of HPCA, 2004.
[8] T. Ehrhart, S. Patel, "Reducing the Scheduling Critical Cycle using Wakeup Prediction", in HPCA 2004.
[9] D. Ernst, A. Hamel, T.Austin, "Cyclone: a Broadcast-free Dynamic Instruction Scheduler with Selective Replay", in Proc. of Int'l. Symp. On Computer Architecture (ISCA), 2003.
[10] D. Ernst, T. Austin, "Efficient Dynamic Scheduling Through Tag Elimination", in the 29th Int'l. Symp. on Comp. Architecture, 2002.
[11] M.K. Gowan, L.L. Biro, D.B. Jackson, "Power considerations in the Design of the Alpha 21264 microprocessor", in the Proceedings of the 35th ACM/IEEE Design Automation Conference (DAC 98), 1998.
[12] J. Hu, N. Vijaykrishnan, M. Irwin, "Exploring Wakeup-Free Instruction Scheduling", in Proc. of the Int'l. Symp. on High Perf. Computer Architecture (HPCA), 2004.
[13] I.Kim, M.Lipasti, "Half-Price Architecture", in Proceedings of ISCA 2002.
[14] I. Kim and M. Lipasti, "Macro-Op Scheduling: Relaxing Scheduling Loop Constraints", in the 36th International Symposium on Microarchitecture, 2003.
[15] A. Lebeck et. al. "A Large, Fast Instruction Window for Tolerating Cache Misses", in the 29th Intl. Symp. on Comp. Arch. (ISCA), 2002.
[16] Y.Liu, et. al.,"Scaling the Issue Window with Look-Ahead Latency Prediction" ICS 2004.
[17] P. Michaud, A. Seznec, "Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors", HPCA 2001.
[18] S. Palacharla, et.al. "Complexity-Effective Superscalar Processors", in ISCA 1997.
[19] S. Raasch, N.Binkert, S.Reinhardt, "A Scalable Instruction Queue Design Using Dependence Chains", in Proc. of ISCA, 2002.
[20] J. Sharkey, et. al. "Instruction Packing: Reducing Power and Delay of the Dynamic Scheduling Logic", in Proc. ISLPED 2005.
[21] J. Sharkey, D. Ponomarev, "Non-Uniform Instruction Scheduling", in Proc. Euro-Par, 2005.
[22] J. Stark, M Brown, Y Patt, "On Pipelining Dynamic Instruction Scheduling Logic", in 33rd Int'l. Symp. on Microarchitecture, 2000.