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

Non-Uniform Instruction Scheduling

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. Dynamic instruction scheduling logic is one of the most critical and cycle-limiting structures in modern superscalar processors, and it is not easily pipelined without significant losses in performance. However, these performance losses are incurred only due to a small fraction of instructions, which are intolerant to the non-atomic scheduling. We first perform an empirical analysis of the instruction streams to determine which instructions actually require single cycle scheduling. We then propose a *Non-Uniform Scheduler* – a design that partitions the scheduling logic into two queues, each with dedicated wakeup and selection logic: a small Fast Issue Queue (FIQ) to issue critical instructions in the back-to-back cycles and a large Slow Issue Queue (SIQ) to issue the remaining instructions over two cycles with a one cycle bubble between dependent instructions. Finally, we propose and evaluate several steering mechanisms to effectively distribute instructions between the queues.

1 Introduction

It has been well documented in the recent literature that instruction wakeup and selection logic form one of the most critical loops in modern superscalar processors [17,20]. Unless wakeup and selection activities are performed within a single cycle, dependent instructions can not execute in consecutive cycles, which seriously degrades the number of instructions committed per cycle (IPC), by as much as 30% in a 4-way machine, according to our simulations. At the same time, both wakeup and selection logic have substantial delays [17], so if these activities are performed atomically within a single cycle, then the designers may be forced to use lower clock frequency or limit the size of the instruction issue queue.
Several schemes have been recently proposed to relax the scheduling loop without seri-ously compromising the processor's performance [3,4,14,20]. Most of these designs do somewhat mitigate the problem of IPC loss due to the inability to execute dependent in-structions in consecutive cycles with pipelined schedulers. However, all of these techniques result in significant additional complexities (as we detail later) and are not easy to retrofit into existing datapaths. In this paper, we investigate a much simpler solution. The idea is based on a distributed implementation of the issue queue in the form of two separate queues: a fast, small issue queue (FIQ) to perform a

1-cycle scheduling (with atomic wakeup/select) of some instructions, and a large, slow issue queue (SIQ) to perform pipelined 2-cycle scheduling of all other instructions. Each of these queues has a dedicated wakeup and selection logic, and only the dependent instructions from the FIQ are guaranteed to execute in the back-to-back cycles. In the rest of the paper, we refer to this design as the Non-Uniform Scheduler (NUS). The important feature of our design, and also the major difference from the previous proposals, is that at the time of dispatch, an instruction is steered to one of the queues and is eventually issued out of that queue. In this paper, we propose and evaluate several such steering heuristics.

2 Problem Characterization

Figure 1 presents the performance difference between the pipeline configurations with atomic (wakeup and select operations are performed within a single cycle) and pipelined (wakeup and select are pipelined over two cycles) schedulers. In general, the IPC degradation is as high as 30% for a 32-entry issue queue as seen from the graph. As the size of the issue queue is increased, this performance loss becomes smaller. For example, for a 64-entry issue queue, the performance degradation is reduced to 15% on the average. In any case, the performance impact due to the inability to execute instructions back to back is significant. Similar results were also presented by other researchers [4,20].

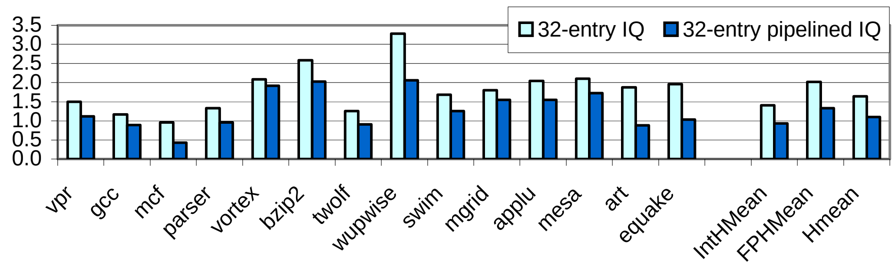

Fig. 1. IPC of 32-entry traditional queue and a 32-entry pipelined queue.

Notice that it is only the dependency on a single-cycle latency operation that creates the pipeline bubble with pipelined schedulers. Moreover, it is the last arriving operand that truly awakens an instruction in the issue queue. Consequently, only the instructions with the last arriving operand produced by a single cycle latency instruction lose the ability to execute back-to-back with their parents in the presence of pipelined schedulers. In all other cases, the multi-cycle scheduling latency is completely hidden by the execution latency of the parent instructions (provided that the scheduling latency of a child does not exceed the execution latency of a parent).

To gauge the magnitude of this problem, we performed a study on the number of instructions whose last arriving operand is produced by a single-cycle latency instruction. Details of our simulation methodology are presented in Section 5. The results are shown in Figure 2. One can observe that on the average, about 60% of all instructions have a last arriving operand that is produced by a single-cycle latency instruction. These results show that there is a potential for optimizing traditional dynamic schedulers, as about half of all the instructions can tolerate 2-cycle scheduling latency.

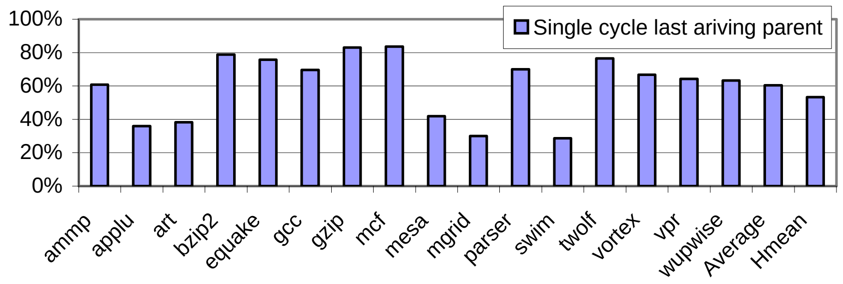

Fig. 2. Percentage of instructions, which have their last arriving operand produced by an instruction with single-cycle execution latency

3 Non-Uniform Scheduling Logic

Non-Uniform Scheduling (NUS) logic is different from the traditional scheduling logic in that a traditional monolithic issue queue (IQ) and its associated selection logic are divided into two parts. The first is a small, fast issue queue (FIQ) and the second is a large, slow issue queue (SIQ). Instructions are steered to one of these queues at the time of dispatch according to certain heuristics. Once dispatched to a queue, the instruction waits in that queue until it is ready to execute. The steering logic is activated in parallel with the rename stage and has negligible additional overhead, as the steering heuristics that we consider are very simple. The queues have separate wakeup and selection logic, and they share functional units. During selection, priority is given to the instructions in the FIQ. The FIQ's wakeup/select loop is atomic (takes one cycle) while the SIQ's wakeup/select loop is pipelined over two cycles. The datapath incurporating the NUS scheduler is shown in Figure 3.

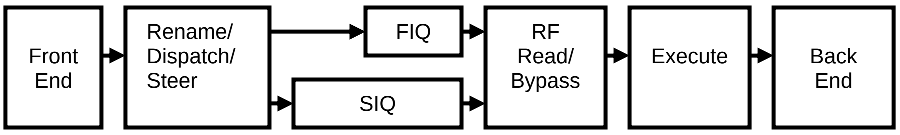

Fig. 3. Datapath Incorporating NUS Scheduling Logic

At the end of the selection cycle, combined W instructions are selected from both the FIQ and the SIQ. Then, the destination tags of all of the selected instructions are broadcast across both queues. Dependent instructions in the FIQ wakeup and get selected in the next cycle, thus allowing for the back-to-back execution. Instructions in the SIQ wakeup in the next cycle and get selected one cycle after that. Mechanisms similar to the ones described in [20] are used to ensure that an instruction does not issue prematurely, if the execution latency of its last arriving parent is higher than 2 cycles.

The selection logic in the NUS design needs to perform some arbitration between the instructions selected from both queues. For example, if N instructions are selected from the FIQ, then at most (W-N) instructions can be selected from the SIQ in the same cycle. This logic is part of the SIQ. There is enough slack in the SIQ's selection cycle (the entire cycle is still used for selection, but the SIQ's size is smaller than the size of the baseline issue queue) to perform this simple check and limit the number of instructions selected from the SIQ. The FIQ selection logic is unmodified, as all instructions selected from the FIQ are issued.

4 Steering Mechanisms for NUS

An important aspect of the NUS design is the set of heuristics used to steer the instructions between the two queues at the time of instruction dispatch. Optimally, a heuristic would be simple and easy to implement, but also be effective and schedule the key pairs of instructions back-to-back. In this paper, we examine several different steering heuristics. In all cases, if the destination queue is full, the instruction is placed into the other queue if such a possibility exists (i.e., the other queue is not full). The process of instruction dispatching blocks only when both queues saturate. The steering heuristics examined in this paper are as follows:

(1) FIQ Utilization (UTIL). Here, instructions are only steered to the SIQ if the FIQ is full. Otherwise, each instruction is steered to the FIQ. Notice that this is a greedy steering heuristic which, at first sight, may seem to be an optimal solution. However, this is not necessarily the case because the FIQ may become full with instructions that are in fact tolerant to the pipelined scheduling, forcing other instructions which are not tolerant of the pipelined scheduling to end up in the SIQ. Even in the presence of a free space in the FIQ, it could be more beneficial to steer some instruction into the SIQ so that instructions that cannot tolerate pipelined scheduling can later be placed in the FIQ. All subsequent heuristics attempt to do exactly that.

(2) Single Cycle Dependency (SCD). Here, all instructions dependent on a not-yet executed single cycle instruction are steered to the FIQ and all other instructions are steered to the SIQ.

(3) Multiple Non-ready Sources (MNR). Here, instructions with one or zero non-ready sources are steered to the SIQ and only instructions with 2 or more non-ready operands are steered to the FIQ. This heuristic is based on several observations. First, several researchers have shown that most instructions enter the scheduling window with at least one of their input operands already available [10,11,19], thus there are fewer instructions with two non-ready operands. Secondly, instructions waiting on two operands are likely to be waiting for a longer duration, and thus it could be advantageous to give these instructions scheduling priority when they do become ready to execute.

(4) Single Cycle Dependency with Multiple Non-ready Sources (SCD/MNR). This heuristic combines the previous two. An instruction is steered to the FIQ only if it is

dependent on multiple not-yet-executed instructions (i.e. both source operands are not ready) and one of those instructions is a single-cycle latency operation. All other instructions are steered to the SIQ.
(5) Single Non-ready Source with a Single Cycle Dependency (SNR/SCD). Here, an instruction is steered to the FIQ only if it has exactly one non-ready source at the time of dispatch and that source will be produced by an instruction with a single-cycle execution latency. All other instructions are steered to the SIQ.

5 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 [5] that implements separate structures for the issue queue, 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 7 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

For estimating the delay requirements, we designed the actual VLSI layouts of the issue queue and simulated them using SPICE. The layouts were designed in a 0.18 micron 6 metal layer CMOS process (TSMC) using Cadence design tools. A Vdd of 1.8 volts was assumed for all the measurements.

6 Experimental Results

The analyses in this section are focused on comparing the appropriate NUS configurations against a 32-entry traditional atomic issue queue. We first explain how we selected the appropriate sizes of the FIQ and the SIQ in the NUS design. In order to balance the delays of both queues, we performed circuit-level simulations of *complete*, hand-crafted issue queue layouts in 0.18-micron TSMC technology. We measured the delays of both the wakeup and selection logic for the schedulers of various sizes. Results are summarized in Table 2.

Table 2. Delays of the scheduling logic

	Tag-Bus Drive (ps)	Comparator Output (ps)	Final Match Signal (ps)	Total Wakeup Delay (ps)	Selection Delay(ps)	Total Delay(ps)
64-entry	313	223	131	667	489	1156
32-entry	224	219	126	569	370	939
16-entry	131	201	114	446	252	698
8-entry	72	194	110	376	136	512

The delays of the wakeup logic comprise of three components: the delay to drive the destination tags across the issue queue, the delays in performing tag comparisons, and the delays in setting the ready bit of the entry. We assumed that traditional pull-down comparators (whose outputs are precharged every cycle and are discharged on a mismatch) are used within the issue queue to perform tag matching. The delay of the tree-structured selection logic depends on the number of levels that must be traversed in the tree, both on the way to the arbiter and on the way back plus the additional wire delays on the way [17]. According to our estimations, the delay within a single level of the selection tree is about 60ps.

To understand how we selected the sizes of the queues in the NUS scheduler, assume that the SIQ of 32-entries is used, which is equal in size to the baseline scheduler. The SIQ operates in a pipelined fashion over two cycles where the cycle time is constrained by the delay of the wakeup phase (569ps). To complement such a SIQ in the NUS design, the size of the FIQ should be chosen in such a way that the combined delays of the wakeup and selection logic in the FIQ are comparable to the wakeup delay of the SIQ. For this reason, as seen from the results presented in Table 2, an 8-entry FIQ is an appropriate match for a 32-entry SIQ. Likewise, a 16-entry FIQ is an appropriate match for a 64-entry SIQ. To summarize, the appropriate combinations of the FIQ and the SIQ sizes are such that the size of the FIQ is about one quarter of the size of the SIQ. Similar observations about the relationship of the FIQ and the SIQ sizes can be made by examining the delays presented in [17].

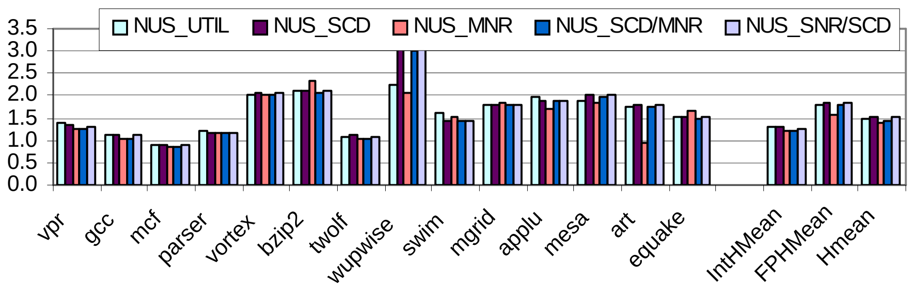

Fig. 4. Per-benchmark commit IPCs of the 8/32 NUS.

We then examined the performance of various steering heuristics to be used in conjunction with the NUS scheduler, as described in Section 5. The per-benchmark results for the 8/32 NUS (8-entry FIQ and 32-entry SIQ) for each steering heuristic are presented in Figure 4. As seen from the graph, the SCD steering provides the best results - it outperforms the simple UTIL steering (which tries to fill the FIQ first) by 1% on the average. The UTIL heuristic performs reasonably well on the average, but it represents a suboptimal choice for 8 of the 14 benchmarks. For example, for *bzip2*, the MNR steering provides 10.9% better performance than simple UTIL steering. For *mcf*, SCD and SNR/SCD provide a 1.2% and 2.7% performance benefit, respectively. For *twolf*, SCD and SCD/MNR also provide a performance benefit by 1.6% and 0.5%, respectively. For *wupwise*, SCD, SCD/MNR, and SNR/SCD steering all provide better performance than the UTIL steering by 36.8%, 34.7%, and 38.2%, respectively. For *mgrid*, MNR provides 1.7% better performance than the UTIL steering. For *mesa*, SCD, SCD/MNR, and SNR/SCD perform better than the UITL steering by 6.6%, 4.6% and 6.8%, respectively. For *art*, the SCD and SNR/SCD heuristics provide better performance by 3.2% and 3.0%. Finally, the SCD/MNR steering provides 10.1% better performance for *equake*.

We now compare the performance results for the 8/32 NUS with SCD steering against traditional schedulers, both atomic and pipelined. Figure 5 compares four different scheduler architectures: the leftmost bar shows the performance of a machine with a 32-entry atomic scheduler, the next bar shows the performance of a machine with a 32-entry scheduler such that wakeup and selection are pipelined over two cycles, the third bar shows the performance of a traditional, atomic 8-entry scheduler, and finally, the rightmost bar shows the performance of a 8/32 NUS with SCD steering, as described in section 5. As expected, the combination of the two queues outperforms either queue used in isolation. On the average, the NUS scheduler outperforms the 32-entry pipelined scheduler by 37.5% and the 8-entry atomic scheduler by 10.7%. *Most importantly*, the performance of the NUS comes within 9% of the performance of a 32-entry atomic queue, but the NUS achieves significant cycle time reduction, potentially by as much as 40% according to the figures presented in Table 2 (569ps for the NUS vs. 939ps for the traditional 32-entry issue queue). Compared to the atomic scheduler with 40 entries (which in this case is the combined size of the FIQ and the SIQ), the performance of the NUS is only 9.5%

lower. This is somewhat surprising because 80% of the NUS entries are implemented in the SIQ, which uses slow pipelined scheduling.

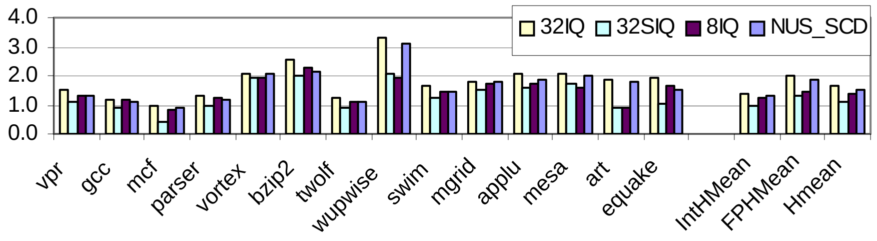

Fig. 5. Commit IPCs of the 8/32 NUS using the UTIL steering heuristic

Notice that the performance benefit of the 8/32 NUS compared to the 8-entry atomic queue is significantly higher for floating point benchmarks than it is for integer programs. This is because the floating point benchmarks significantly benefit from the presence of the larger queue and are generally more tolerant to the relaxation of the scheduling loop. Furthermore, the branch prediction accuracy is high for floating point benchmarks, which puts more pressure on the scheduler. The integer benchmarks, on the other hand, perform reasonably well even with the smaller schedulers and are significantly impacted by hindering the ability to execute instructions back to back. This is because these benchmarks are dominated by the long, serial dependency chains of mostly single-cycle latency instructions. Therefore, in many cases it may be beneficial to delay the dispatch of an instruction for a few cycles rather than dispatching it immediately to the slow queue. For example, *gcc, parser,* and *bzip2* show better performance with simply having the 8-entry issue queue compared to the NUS.

7 Related Work

Stark et.al. [20] pipelined the scheduling logic into wakeup and select stages and used the status of instruction's grandparents to wakeup the instruction earlier in a speculative manner. In [3], Brekelbaum et.al. introduced hierarchical scheduling windows (HSW) to support large number of in-flight instructions. The HSW design also relies on the use of fast and slow queues, but has a fairly complex logic for moving instructions from the slow queue (where all instructions are initially placed) to the fast queue. Our technique differs from HSW in that instructions are steered to the two queues upon dispatch. Lebeck et.al. [15] introduced a large waiting instruction buffer (WIB) to temporarily hold the load instructions that missed into the L2 cache as well as their dependents outside of the small issue queue (IQ). A somewhat similar technique, albeit implemented in a different manner, is also described in [8], where the instructions which are expected to wait for a large number of cycles before getting issued are moved to the secondary queue, called Slow Lane Instruction Queue. Brown et.al. [4] 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. Kim and Lipasti [14] proposed to group two (or more) dependent single-cycle operations into so-called Macro-OP (MOP), which represents an atomic scheduling entity with multi-cycle execution latency.
Scheduling techniques based on predicting the issue cycle of an instruction [1,6,7,10,13,16] remove the wakeup delay from the critical path, but need to keep track of the cycle when each physical register will become ready. In [9], the wakeup time prediction occurs in parallel with the instruction fetching. Additional mechanisms are needed in these schemes for handling issue latency mispredictions, as the instructions executed too early need to be replayed. In [18], the use of segmented issue queues is proposed, where the broadcast and selection are limited to a smaller segment.

8 Concluding Remarks

The capability to execute the dependent instructions in the back-to-back cycles is important for sustaining high instruction throughput in modern out-of-order microprocessors. If the dependent instructions cannot execute in consecutive cycles, then the IPC impact can be very significant. We described a non-uniform scheduler design which significantly reduces the IPC penalties by using a pair of issues queues, one implementing single-cycle scheduling and the other implementing 2-cycle scheduling with pipelined wakeup and select. We evaluated several mechanisms for intelligent instruction placement between the two queues.
For a 4-way machine, the performance our design comes within 9% of an idealized atomic scheduler with potentially as much as 40% reduction in cycle time. To compare, if the idealized scheduler is simply pipelined into separate stages for wakeup and selection, then the performance loss compared to the idealized atomic situation is 30% for a 32-entry scheduler. Consequently, the NUS reduces this performance degradation by almost 70%. We evaluated several steering heuristics for the NUS and found out that, for a 32-entry queue, the single cycle dependency steering performs the best. We also found out that the simple greedy steering based on the utilization of the queues is not the optimal solution for the majority of the simulated benchmarks. Finally, we found out that most of the benefits of the NUS scheduler are achieved for floating-point benchmarks. For the majority of integer benchmarks, it is more beneficial to delay the dispatch of instructions rather than processing them through a slow queue.

9 Acknowledgements

We would like to thank Oguz Ergin for assistance with the VLSI layouts and 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.

10 References

[1] J. Abella, A.Gonzalez, “Low-Complexity Distributed Issue Queue”, in Proc. of HPCA, 2004.
[2] H. Akkary, R. Rajwar, S. Srinivasan, “Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors”, in Proc. of MICRO 2003.
[3] E. Brekelbaum et. al., “Hierarchical Scheduling Windows”, in 35th Int’l. Symp. on Microarchitecture, 2002.
[4] M. Brown, J. Stark, Y. Patt. “Select-Free Instruction Scheduling Logic”, in the 34th International Symposium on Microarchitecture, 2001.
[5] D. Burger and T. Austin, "The SimpleScalar tool set: Version 2.0", Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases.
[6] R.Canal, A. Gonzalez, “A Low-Complexity Issue Logic”, in Proc. of the International Conference on Supercomputing (ICS), 2000.
[7] R.Canal, A.Gonzalez, “Reducing the Complexity of the Issue Logic”, in Proc, of the Int’l. Conf. on Supercomputing (ICS), 2001.
[8] A. Cristal, et.al., “Out-of-Order Commit Processors”, in the International Symposium on High-Perf. Comp. Arch. (HPCA), 2004.
[9] T. Ehrhart, S. Patel, “Reducing the Scheduling Critical Cycle using Wakeup Prediction”, in HPCA 2004.
[10] 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.
[11] D. Ernst, T. Austin, “Efficient Dynamic Scheduling Through Tag Elimination”, in the 29th Int’l. Symp. on Comp. Architecture, 2002.
[12] B. Fields, R. Bodik, M. Hill. “Slack: Maximizing Performance Under Technological Constraints”, in the 29th International Symposium on Computer Architecture, 2002.
[13] 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.
[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] P. Michaud, A. Seznec, “Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors”, HPCA 2001.
[17] S. Palacharla, N. Jouppi, J. Smith, “Complexity-Effective Superscalar Processors”, in 24th Intl. Symposium on Computer Architecture, 1997.
[18] S. Raasch, N.Binkert, S.Reinhardt, “A Scalable Instruction Queue Design Using Dependence Chains”, in Proc. of ISCA, 2002.
[19] J. Sharkey et.al., “Instruction Packing: Reducing Power and Delay of the Dynamic Scheduling Logic”, in Proc. of ISLPED 2005.
[20] J. Stark, M Brown, Y Patt, “On Pipelining Dynamic Instruction Scheduling Logic”, in 33rd Int’l. Symp. on Microarchitecture, 2000.