In the Proceedings of the 23rd International Conference on Computer Design (ICCD), 2005

Power-Efficient Wakeup Tag Broadcast

Joseph J. Sharkey, Kanad Ghose, Dmitry V. Ponomarev, Oguz Ergin[‡]

Department of Computer Science
State University of New York at Binghamton
{jsharke, ghose, dima}@cs.binghamton.com

[‡]Intel Barcelona Research Center
oguzx.ergin@intel.com

Abstract

The dynamic instruction scheduling logic is one of the most critical components of modern superscalar microprocessors, both from the delay and power dissipation standpoints. The delay and energy requirement of driving the wakeup tags across the associatively-addressed issue queue accounts for a significant percentage of the scheduler's overhead and also limits the design scalability. We propose Tag Memoization and Tagline Folding - two schemes to reduce the power of wakeup tag broadcasts by reducing the number of tag-bits that are driven in each broadcast. Our results show that the combination of these mechanisms provides 22.3% average reduction of the wakeup tag broadcast power with no impact on the IPC.

1. Introduction

Modern superscalar processors use out of order execution to exploit instruction level parallelism. The dynamic scheduling engine employed in such processors often uses associative logic embedded into the issue queue entries to wakeup instructions that are awaiting a result. This is accomplished by storing the addresses of the source registers within the issue queue entries and using the comparators that match the stored source register values against the address of the result that is broadcast on tag bus lines. A significant amount of energy dissipation results as the destination register address is broadcast on the tag busses. Energy dissipation occurs when the tag bus lines are driven because of the charging and discharging of the wire capacitance of the tag line itself and the gate capacitance of the devices that implement the tag comparators. As wire capacitances dominate, a significant fraction of the energy spent in waking up instructions is attributed to the power used for broadcasting the tags. This is particularly true if comparators that dissipate energy only on a match are used within the issue queue [27]. Other researchers have reported the issue queue power to account for 20%-25% of total chip power [33, 34]. Few results have been published which break this down into components for the wakeup and select logic individually, but it has been reported that the wakeup power is dominated by the power spent in broadcasting the tags [32].

In this paper, we propose two schemes - *Tag Memoization* and *Tagline Folding* - to reduce the energy consumed by the wakeup tag broadcasts. Tag memoization avoids driving the upper portion of the tags, if those bits did not change their values from what was driven on the same tag bus during the most recent broadcast. Tagline folding avoids driving the upper order bits on a tag bus if those bits match the upper order bits driven in the same cycle on another bus. We validate the power savings achieved by using our techniques through the cycle-accurate simulations of SPEC 2000 benchmarks and the circuit simulations of the full-custom issue queue layouts.

The main contributions of this paper are as follows:

- We perform detailed layout-level simulations of the wakeup logic and establish that the power dissipated in the course of tag broadcasts amounts to almost 92% of the total wakeup power.
- We propose two complementary techniques – tag memoization and tagline folding – to reduce the power consumption of the wakeup logic.
- We perform detailed microarchitectural and circuit-level simulations of the proposed mechanisms. Our results show more than 22% reduction in the wakeup power with no IPC degradation on any of the benchmarks and only the slight increase in the wakeup delay.

The rest of the paper is organized as follows. Section 2 analyzes the power of the wakeup logic. Section 3 describes tag memoization and Section 4 describes tagline folding. Our simulation methodology is presented in Section 5 and our simulation results are presented in Section 6. We describe the related work in Section 7 and offer concluding remarks in Section 8.

2. Power Analysis of the Wakeup Logic

In this section, we analyze the sources of power consumption within instruction wakeup logic and present their percentage breakdown. For this analysis, we performed the circuit simulations using the actual hand-crafted and highly-optimized VLSI layouts of the issue queue and the associated logic. The complete details of our simulation framework can be found in Section 5.

There are two sources of power dissipation in the process of instruction wakeup:

- Power dissipated driving the destination tags of the selected instructions across the issue queue entries (hereinafter called *tag broadcast power*), and,

- Power dissipated in the course of performing associative matching of the broadcasted destination tags against the locally stored source tags for each not-yet-ready operand of every issued instruction, and setting the corresponding source valid bits. The power of setting the instruction's ready bit (which is used to drive the request signals to the selection logic) is also accounted for here. For simplicity, we call this component the *comparator power* in the rest of the paper.

Our circuit simulations show that the tag broadcast power accounts for about 92% of the wakeup power. In these estimations, we assumed that the traditional comparators that dissipate energy on a mismatch are used within the issue queue. Since the majority of comparison situations result in a mismatch, more energy-efficient dissipate-on-match comparators could be used, resulting in further decrease of the comparison power [27]. We also assumed that if a source operand is ready or if the source operand is not used or if an issue queue entry is not allocated, the corresponding tag comparators are not activated in that cycle. Finally, we assumed that if no tag is driven on a tag bus within a given cycle, then this bus does not dissipate any power.

In the following sections, we describe two techniques to reduce the wakeup tag broadcast power by reducing the number of tag bits that are driven.

3. Tag Memoization

Tag memoization exploits the fact that the higher-order bits of the tags that are broadcasted within a short duration of each other are likely to be the same. The key idea here is to conserve power expended in broadcasting the tags by not driving the higher-order tag bits if they happen to match the higher-order tag bits that were driven on the same bus during the previous broadcast. The tag comparator used to match the tag on the bus is broken into two separate comparators, say U and L, to match the higher-order bits and the remaining lower-order bits, respectively. A 1-bit latch is inserted in between to remember if there was a match in the higher order bits with the previous broadcast. The match signal for an entry is derived by NAND-ing the output of the comparator for the lower order bits with either the latch output or the output of the comparator for the upper order bits. The modifications to the tag circuitry associated with a single tag bus to support tag memoization is shown in Figure 1. We now describe this in some detail.

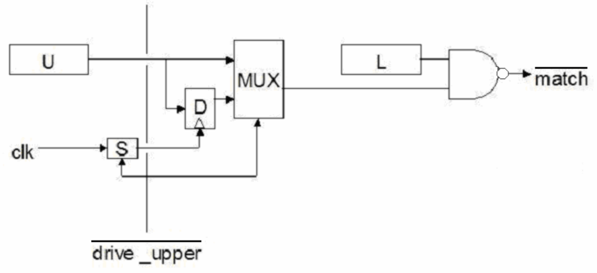

Figure 1: Tag comparator configuration for the tag memoization scheme

The comparator used for tag comparison is split into two parts – an upper part U that compares the higher order bits on the tag bus and a lower part L that compares the remaining bits against the respective parts of the locally stored source tags. When both the upper part and lower part of the tag buses are to be driven, the line ~drive_upper (complement of drive_upper) is driven low during the cycle. These signal lines run across the length of the issue queue. Doing so turns the transmission gate switch on and allows the clock signal clk to latch in the current output of the comparator U into a D-type latch, D. Simultaneously, the multiplexer MUX selects the output of the comparator U and feeds it to the NAND gate. In this case the NAND gate effectively combines the output of the two comparators U and L to produce the low-active match signal. Storing the result of U in the D-latch in this manner makes it possible for the logic shown to remember the result of a match of the upper bits of the locally-stored tag value and the upper order bits driven on the tag bus.

When the upper order bits to be driven on the tag bus match what was driven earlier on the same lines, the drive_upper line is maintained in its (default) high state, using a weak pullup device. The upper order tag bus lines are, of course, not driven at all. Doing so disables the clock to the latch D (so that its contents remain unaffected) and allows the multiplexer MUX to select the contents of D. In this case, the match signal is obtained by NAND-ing the output of the comparator L and the contents of D. A tag match signal is produced in this case only if the lower order tag bits match and if the upper order bits of the tag of the result matches the upper order bits driven on the corresponding lines of the tag bus in the immediate past.

The additional delay introduced in the path that generates the match signal is the propagation delay of the NAND gate and the propagation delay of a turned-on CMOS switch within the multiplexer. Our circuit simulations (0.18 micron CMOS) showed that these components add a delay of 73 ps (55 ps for the NAND gate and 18 ps for the turned on transmission gate within the multiplexor) to the critical path of the wakeup logic. Since smaller comparators, operating in parallel, are now used for matching the upper and lower order bits, the total comparator delay is reduced by 50 ps. Therefore, the overall critical path delay of the wakeup logic increases by 23 ps, which represents a 4 % increase compared to the 569 ps delay of our baseline case (as shown in Table 1).

Table 1: Delay breakdown of the traditional broadcast-based issue queue.

Component	Delay (ps)
Tag-Bus Drive	224
Comparator Output	219
Final Match Signal	126
Total Delay	**569**

Determining if the upper order bits that are being broadcasted match the upper order bits that were last driven on the same bus requires us to store the values of the last-driven upper order bits in a latch. The logic for doing this is in the form of a fast combinational comparator (70 ps delay for a 3-bit combinational comparator), which has a much smaller delay than the pulldown comparators (110 ps for 3 bits). The delay of this logic can be absorbed by integrating this logic within the selection logic which not only selects instructions for issue but also performs tag bus assignments. While the grant signal propagates down the selection logic, which is usually a multi-level, tree-like structure, the comparison of the upper order bits can be completed. If the selection tree is three levels deep and already turned on transmission gates are used to route the grant signals down the tree to the requesting IQ entry, our layouts indicate that the delay of bringing down the grant signal is about 65 ps. Thus the overhead (70- 65 = 5 ps) of detecting whether to drive the upper or tag bits is a negligible part of the total selection process. If wakeup and selection are both performed in the same cycle, this overhead is even smaller.

Notice that with the logic just described, the line ~drive_upper is driven low only when the upper order tag bits are to be driven. Thus, when the upper order tag bits are not driven, no additional energy is expended in maintaining the line at this state, which is maintained in the high state by a small pullup device (we ignore the energy spent in replenishing the charge lost on this line due to leakage). Neither do we need to maintain or drive the complement of this signal - the complementary signal is derived locally within the multiplexers. Consider now a 7-bit tag, there are 14 bus lines in the tag bus as both the value of a tag bit and its complement have to be driven to avoid the need to generate the complement values locally within each comparator (typical pulldown comparators require both the true input and the complement input bits). Our memoization scheme thus requires an additional bus wire to be driven when the upper order tag bits are driven. The added energy overhead of the ~drive_upper is thus small.

One can force additional savings from the memoization scheme by assigning tag broadcasts to a bus whose U bits match the upper order bits of the tag value to be driven. We call this "intelligent" tag bus assignment. There is, of course, additional energy and delay overhead in assigning tag broadcasts to specific buses in this case. Another alternative is to assign the tag values sequentially to instructions. This is possible in datapaths that use the ROB slots as physical registers or have rename buffers that are assigned from a circular FIFO. *Tag memoization on these datapaths as well as datapaths with a unified register file is examined in the results section.*

The approach just described can be generalized to accommodate the segmentation of the tag comparator into more than two parts requiring an intervening latch in between consecutive segments. For example, a 7 bit tag comparator can be segmented into three parts: U1 (upper order two bits), U2 (next two bits), and L (remaining 3 bits). This arrangement requires two latches: one to remember the result of U1 and another for U2. These latches may be set independently, allowing for the gating off of either set of bits, or both. The match signal is derived by NAND-ing the contents of the intervening latches and the output of the comparator segment covering the lower order bits.

4. Tagline Folding

As observed with tag memoization, the high-order bits of the tags that are broadcast within a short duration of each other are likely to be the same. Tag memoization targets the case where there is a match in the upper order bits of the tag occurs across two successive broadcasts on the same tag bus. Tagline folding, on the other hand, targets the case where a match occurs across two different tag busses driven in the same cycle. The goal of tagline folding is to conserve power by only broadcasting the upper order bits of the tags on one of the busses.

Figure 2 presents the logic necessary for implementing tagline folding. Each comparator is broken into two parts: one for the upper-order bits of the tag and one for the lower-order bits. The number of bits in each of the two parts is determined by the folding width. For a folding width of w, the upper order comparator has w bits and the lower order comparator has n-w bits, where n is the number of bits in each tag.

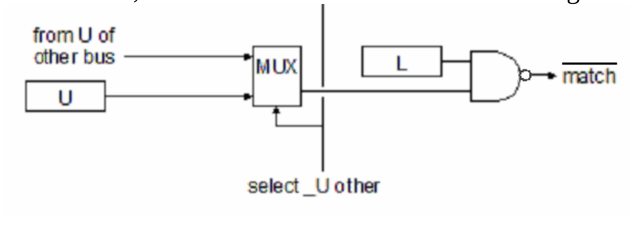

Figure 2: Tagline folding logic.

Tagline folding can be implemented on any subset of tag busses. As an example, let us assume that, in some cycle, the upper order bits of the tags driven on busses 1 and 2 match. The full tag will be broadcast on bus 1, and the comparator on bus one will NAND the output from its upper-order comparator and its lower order-comparator for detecting a match. Bus 2, however, will only broadcast the lower-order bits along with one additional select_U_other bit. Rather than NAND-ing both the upper-order and lower-order comparator outputs for the source tag on bus 2, the comparators will NAND the outputs for this source of the upper-order comparator on bus 1 and the lower-order comparator on bus 2. The additional delay introduced in path that generates the match signal is the propagation delay of the NAND gate and the propagation delay of a turned-on CMOS switch within the multiplexer. Similar to the tag memoization case, the overall critical path delay of the wakeup logic increases by about 28 ps, which represents a 5% increase compared to the 569 ps delay of our baseline case (as presented in Table 1). This is because the propagation delay of a transmission gate-based multiplexer is independent of the number of inputs; the added delay comes from the wire length needed to bring in the signal from the neighboring bus.

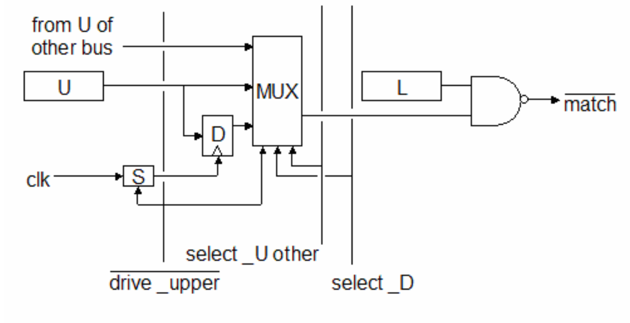

Figure 3: Combined Implementation of Tag Memoization and Tagline Folding

Figure 3 shows how the tag memoization logic (discussed in Section 3) can be augmented very simply to support tagline folding. The multiplexer shown in the figure now has the option of choosing one of three inputs – either the output of the local U comparator (select_upper driven high), the output of the local latch D (select_D driven high) or the output of the U comparator associated with an adjacent bus (select_Uother driven high). If the upper order bits driven on two adjacent buses happen to be the same, the select_upper line is asserted on one bus (say Bus A) and the select_U_other line is asserted on the other bus (say Bus B), to allow the comparator of Bus B to use the output of the U comparator of Bus A for the match. Only the upper order tag bus bits of Bus A are driven in this case.

The delay added in the path that produces the match signal for Bus B now has an additional component – the wire delay of the connection that brings in the output of the U-comparator of the adjacent bus to the input of the local multiplexer. This delay can be minimized by laying out the comparator logic for the two buses as close to each other as possible, by mirroring them symmetrically along a imaginary dividing line that runs across the length of the issue queue. The design of Figure 3 can be generalized to not only accept the output of the U comparator of the adjacent bus but also the output of the D latch for the adjacent bus or signals from comparators and latches of other buses if need be. Notice that the use of un-encoded lines to select the inputs of the multiplexer permits only one of the input selection lines to be activated at any time. The delay of the combined memoization and folding logic does not increase compared to either of the two designs individually.

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 [3] 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 8 integer and 7 floating-point benchmarks using their reference inputs (we had difficulties compiling other benchmarks in our framework, mostly those written in Fortran). In all cases, predictors and

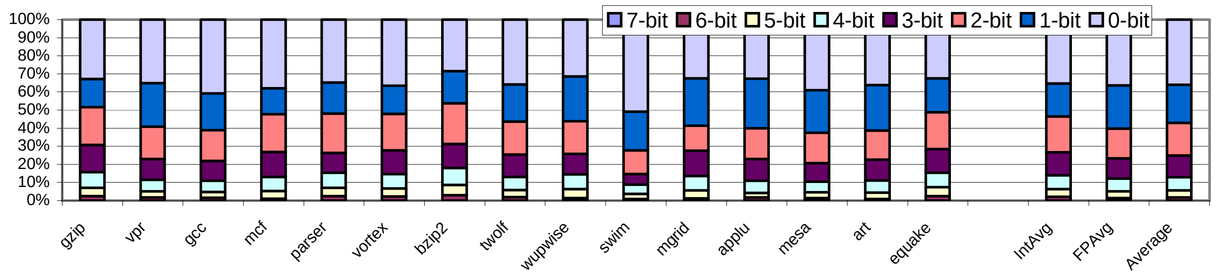

Figure 4: Number of most significant bits matching those of the previous tag broadcast on each tag bus for Datapath B.

caches were warmed up for 1 billion committed instructions and statistics were gathered for the next 500 million instructions.

Table 2 presents the configuration of the baseline simulated processor. Two separate datapaths have been simulated that use different physical register allocation mechanisms. Datapath A uses the PowerPC-style register file with rename buffers where registers are organized as a circular list [31]. Datapath B contains a unified architectural/physical register file similar to the Pentium 4 datapath. Both datapaths have the machine configuration as specified in Table 2.

Table 2: Configuration of the Simulated Processor

Parameter	***Configuration***
Machine width	4-wide fetch, 4-wide issue, 4 wide commit
Window size	issue queue – as specified,48 entry load/store queue, 96–entry ROB
FU's 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	128 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
Branch Mispred. Penalty	8 cycles minimum
Memory	128 bit wide, 150 cycles first chunk, 1 cycles interchunk
TLB	32 entry (I), 128 entry (D), fully associative, 12 cycles miss latency

For estimating the delay, energy and area requirements, we deigned 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. Results

Tag memoization does not impact IPC because it does not interrupt, hinder, or change the order of tag broadcasts. The power savings of tag memoization come from its ability to match the most significant bits of the tags on each bus from one broadcast to the next. Thus, it is important to consider how often these tag bits match. Figure 4 presents the number of most significant bits (MSBs), for each tag broadcast, that match those of the previous tag broadcast on the same bus for Datapath B (Datapaths A and B are defined in Section 5). The two most significant bits match 43% of the time even in this case, where the physical registers are not necessarily allocated from consecutive entries. If Datapath A is used, then physical registers are allocated from a FIFO-managed list, and thus the neighboring registers (which generally have the same most significant bits) are allocated to consecutive instructions. As a result, the percentage of matches in the two most significant bit positions during successive tag broadcasts on the same bus is even higher in this situation – 49% on the average across the benchmarks.

For Datapath A, on a configuration with two separate intermediate latches, the two MSBs match an average of 48.6% of the time, while the next two MSBs match an average of 23.5% of the time, as presented in Figure 5. Accounting for the extra line that must be driven every time one of these latches must be reset, the total power savings from such a variation of tag memoization is 16.4% of total tag-broadcast power. The savings are a little less for Datapath B, as expected – it amounts to 11.1% of the tag broadcast power.

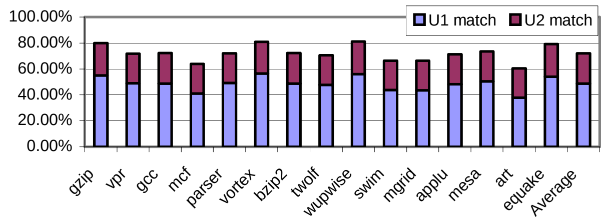

Figure 5: Percent of tag broadcasts in which there is a match on comparators U1 and U2 using tag memoization on Datapath A.

Similarly to tag memoization, tagline folding does not affect the IPC. As described previously, tagline folding can be implemented for any folding width that is less than or equal to the tag width of the processor. A folding width of 1, however, provides little benefit as the power saved is largely offset by the additional power dissipated in driving the select_U_other bit and the additional logic needed to support folding, as shown in Figure 2. Alternatively, a folding width that is equal to the tag width of the processor makes little sense since matches would only be detected when the same exact tag is broadcast on two busses at the same time. However, this is not possible because each instruction broadcasts its tag on only one of the available busses. Thus, only a folding width between 2 and n-1 for a processor with a tag width of n bits is in the practical realm.

As the tagline folding width increases, the power savings realized by each match goes up but the number of such matches decreases. Figure 6 presents the wakeup tag broadcast power savings for tagline folding widths of 2 through 6 on our simulated processor with 7-bit wakeup tags. The figure indicates that the folding width of 2 or 3 bits results in an optimal performance for the simulated processor.

Tag folding can be implemented to match the tags on any subset of the tag busses. Figure 7 presents the percentage of tag broadcasts in which the 2 upper order bits of the tags match on various sets of busses. The frequency of matches is impacted by the bus arbitration logic. We assume that the tag buses are assigned for the selected instructions in order starting with bus 1 and ending with bus 4 (for a 4-way machine with 4 wakeup tag buses). Thus, if only two tags are broadcast in a given cycle, then busses 1 and 2 are used. Bus 4 is only active when 4 tags are broadcast in a given cycle, which does not happen very often because the average broadcast rate is only slightly more than 1.5 instructions per cycle. As a result, the highest match rate is observed between busses 1 and 2, where a match occurs for 52.6% of all tag broadcasts, since these two busses are used most of the time. Busses 3 and 4 produce fewer overall matches, since they are used less frequently. We consider the additional hardware to support folding between busses 3 and 4 unnecessary since matches on these busses occur in less than 6% of the cases.

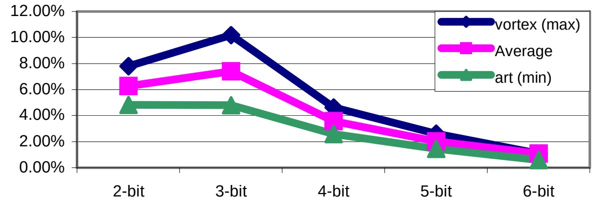

Figure 6: Tag bus broadcast power savings with Tagline Folding on 2 through 6 bits.

Note that the matches on busses 1 & 2 and those on busses 2 & 3 are not disjoint; there is an overlap when the upper order tag bits match on all three busses. It is possible to extend the tagline folding logic to allow the folding of three busses. This, however, adds complexity to the logic since the match on the upper-order comparator can now come from three places. An alternative is to separate out the instances where there is a match on all three busses and those with a match on only busses 2 and 3. The match of all three, then, can be handled as a match on bus 1 & 2, and a mismatch on bus 2 & 3 (requiring the full tags to be driven on both busses 1 and 3). Discounting the situation where all three busses match, the percentage of matches on busses 2 and 3 drops from 22.92% to 8%, indicating that it is not very beneficial to support folding on busses 2 and 3 in this situation. An alternative is to fold busses 1 and 3, which yields a 29% match rate. Thus, for the remainder of this section, we consider this implementation of tag folding on busses 1 & 2, which yields a 52% match rate, and busses 1 & 3, which yields a 29% match rate. The combination of folding on these two sets of busses results in matches on 81% of all tag broadcasts. The combination of folding on these two sets of busses produces an average of 6.85% reduction in the total tag broadcast power, with the highest savings being 12.81% (vortex) and the lowest being 5.75% (swim). Per-benchmark power savings are given in Table 3 for Datapath A.

Tag memoization and tagline folding are complementary techniques. Both require the separation of comparators into upper and lower parts. Thus, they can easily be combined to extract the benefits of each while overlapping the costs through comparator modifications shared by both schemes (as described in Section 4). Combining tag memoization and 2-wide tagline folding results in 22.25% savings in the tag broadcast power with no change in IPC for the Datapath A. Notice that while the techniques are complementary, the total power savings is not simply the sum of the savings achieved by individual schemes, as there exists some overlap between them.

The baseline processor considered in this paper contains 128 physical registers, requiring 7-bit wakeup tags. We also performed experiments with 256-entry register file. In this case, tag memoization and tagline folding result in the combined power savings of 25.6% in the wakeup tag broadcast power. This implies that both mechanisms are scalable with the number of physical registers, as one would expect. Similar trends were observed when further increasing the number of physical registers.

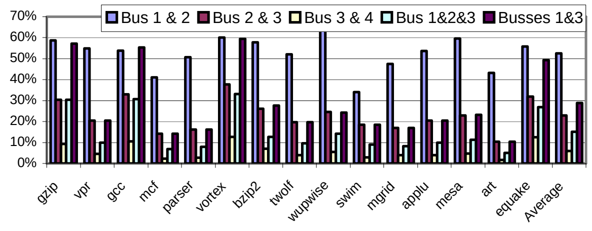

Figure 7: Percentage of all tag broadcasts in which the 2 upper-order bits of the tags match on the various sets of busses.

Table 3: Tag broadcast power savings with tagline folding.

	gzip	*vpr*	*gcc*	*mcf*	*parser*	*vortex*	*bzip2*	*twolf*	*wupwise*	*swim*	*mgrid*	*applu*	*mesa*	*art*	*equake*	**Avg**
Buses1+2	6.28	5.88	5.76	4.41	5.43	6.44	6.19	5.58	6.93	3.66	5.09	5.75	6.39	4.64	5.97	**3.75**
Busses1+3	6.13	2.20	5.92	1.52	1.74	6.37	2.96	2.12	2.60	1.99	1.82	2.19	2.49	1.11	5.29	**3.10**
Combination	12.41	8.08	11.69	5.93	7.17	12.81	9.15	7.70	9.53	5.64	6.91	7.95	8.88	5.75	11.27	**6.85**

7. Related Work

Researchers have proposed several ways to reduce the power consumption of the issue logic. Dynamic adaptation techniques [22,23,24,25] partition the queue into multiple segments and deactivate some segments periodically, when the applications do not require the full issue queue to sustain the commit IPCs. Energy-efficient comparators, which dissipate energy predominantly on a tag match were proposed in [26,27]. In [29], the associative broadcast is replaced with indexing to only enable a single instruction to wakeup. This exploits the observation that many instructions have only one consumer. In [30], the *wakeup width* is decreased to be smaller than the machine width, noting that the full machine width is rarely used for instruction wakeup.

Energy savings on busses due to correlations among the past and immediate values of bits have also been explored in [35, 36, 37, 38]. In [35], bus invert coding is proposed that uses redundancy to reduce bus transitions. The scheme adds one line to the bus to indicate if the actual data or its complement is transmitted depending on the hamming difference between the current value and the previous one. A technique to reduce switching activity on the address busses through the use of Gray codes was proposed in [36]. The Gray code has the advantage that there is only a single transition on the address bus when consecutive addresses are accessed. A technique to compress data addresses and instructions by maintaining only significant bytes with two or three extension bits appended to indicate significant byte positions was proposed in [37]. In [38], a scheme to encode the bytes containing all zeros and save energy by only reading and writing one bit for each zero valued byte was proposed for caches. The zero byte encoding technique was extended to issue queues in [26]. The *T0* code was proposed in [39] to reduce the switching activity on the address buss by freezing the value on the bus if addresses are sequential and driving an additional *INC* line. Several combinations of bus-invert and *T0* encodings were proposed in [40]. A number of irredundant encoding techniques are presented in [21] that do not require any extra lines for encoding and decoding. None of these techniques specifically address the reduction in switching activity on the tag wakeup busses. In addition, we use the actual statistics about the bit patterns driven on these busses as obtained from the cycle-accurate microarchitectural simulations.

In [4], some comparators are removed from the issue queue entries to save power and last-tag speculation mechanisms are introduced for use in instruction wakeup. In [28], the tag buses were categorized into fast buses and slow buses, such that the tag broadcast on the slow bus takes one additional cycle.

Several works attempted to reduce scheduling complexity through pipelining the scheduling logic or reducing the issue queue size [2,7,8]. Other proposals have introduced new scheduling techniques with the goal of designing scalable dynamic schedulers to support a very large number of in-flight instructions [5,6,9,14,20]. Scheduling techniques based on predicting the issue cycle of an instruction [10, 11,12,13,15,16,17,18] remove the wakeup delay from the critical path and remove the CAM logic from the issue queue, but need to keep track of the cycle when each physical register will become ready.

8. Concluding Remarks

We proposed two schemes to reduce the power consumption of the wakeup tag broadcast. Tag memoization avoids driving the upper portion of the tags, if those bits did not change their values from what was driven on the same tag bus during the most recent broadcast. Tagline folding avoids driving the upper order bits on a tag bus if those bits match the upper order bits driven in the same cycle on another bus

The use of these two schemes results in 22.3% reduction in the wakeup tag broadcast power with only about 5% increase in the wakeup delay. If the wakeup and selection operations are combined within a single cycle to implement atomic dynamic scheduling in order to support back-to-back execution of dependent instructions, then the overall increase in the scheduling latency is only on the order of 2-3% (as the selection logic has about the same delay as the wakeup logic according to [1]). Additionally, there is no IPC degradation as neither mechanism interrupts, changes or hinders the order of tag broadcasts.

9. References

[1] S. Palacharla, N. Jouppi, J. Smith, "Complexity-Effective Superscalar Processors", in Proc. of Int'l Symp. on Computer Architecture, 1997.

[2] J. Stark, M Brown, Y Patt, "On Pipelining Dynamic Instruction Scheduling Logic", in Proc. of the International Symposium on Microarchitecture, 2000.

[3] Burger, D. and Austin, T. M., "The SimpleScalar tool set: Version 2.0", Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases.

[4] D. Ernst, T. Austin, "Efficient Dynamic Scheduling Through Tag Elimination", in Proc. of International Symposium on Computer Architecture, 2002.

[5] E. Brekelbaum et. al., "Hierarchical Scheduling Windows", in Proc. of the International Symposium on Microarchitecture 2002.

[6] A. Lebeck et. al. A Large, "Fast Instruction Window for Tolerating Cache Misses", in Proc. of ISCA, 2002.

[7] M. Brown, J. Stark, Y. Patt. "Select-Free Instruction Scheduling Logic", in Proc. Of International Symposium on Microarchitecture, 2001.

[8] I. Kim and M. Lipasti, "Macro-Op Scheduling: Relaxing Scheduling Loop Constraints", in Proc. of MICRO, 2003.

[9] A. Cristal, et.al., "Out-of-Order Commit Processors", in Proc. of HPCA, 2004.

[10] D. Ernst, A. Hamel, T.Austin, "Cyclone: a Broadcast-free Dynamic Instruction Scheduler with Selective Replay", in Proc. of ISCA, 2003

[11] Hu, J., Vijaykrishnan, N., Irwin, M., "Exploring Wakeup-Free Instruction Scheduling", in Proc. Of the International Symposium on High Performance Computer Architecture, 2004.

[12] R.Canal, A. Gonzalez, "A Low-Complexity Issue Logic", in Proc. International Conference on Supercomputing, 2000.

[13] R.Canal, A.Gonzalez, "Reducing the Complexity of the Issue Logic", in Proc. International Conference on Supercomputing, 2001.

[14] S. Raasch, N.Binkert, S.Reinhardt, "A Scalable Instruction Queue Design Using Dependence Chains", in Proc of the International Symposium on Computer Architecture, 2002.

[15] J. Abella, A.Gonzalez, "Low-Complexity Distributed Issue Queue", in Proc. of International Conference on High Performance Computer Architecture, 2004.

[16] P. Michaud, et.al. "Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors", in Proc. of International Conference on High Performance Computer Architecture, 2001.

[17] T. Ehrhart, S. Patel, "Reducing the Scheduling Critical Cycle using Wakeup Prediction", in Proc. of International Conference on High Performance Computer Architecture, 2004.

[18] Y. Liu, et. al., "Scaling the Issue Window with Look-Ahead Latency Prediction", in Proc. International Conference on Supercomputing, 2004.

[19] Z. Chishti, T. Vijaykumar, "Wire Delay Is Not a Problem for SMT", in Proc. Of International Symposium On Computer Architecture 2004.

[20] S. Srinivasan et. al. "Continual Flow Pipelines", in Proc. Of International Conference on Architectural Support for Programming Languages and Operating Systems, 2004.

[21] Y. Aghaghiri, F. Fallah, M. Pedram, "A Class of Irredundant Encoding Techniques for Reducing Bus Power", in Journal of Circuits, Systems, and Computers, Vol 11, No. 5 (2002) pp 445-457.

[22] A. Buyuktosunoglu, et.al.,"A Circuit-Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors", in Proc of Great Lakes Symposium on VLIS, 2001.

[23] D.Folegnani, A.Gonzalez, "Energy-Effective Issue Logic", in Proc of International Symposium On Computer Architecture 2001.

[24] D.Ponomarev, G.Kucuk, K.Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources", in Proc. Of International Symposium on Microarchitecute, 2001.

[25] A. Buyuktosunoglu et.al., "Energy-Efficient Co-adaptive Instruction Fetch and Issue", in Proc. International Symposium On Computer Architecture, 2003.

[26] D.Ponomarev, et.al., "Energy-Efficient Issue Queue Design", in IEEE Transactions on VLSI Systems, November 2003.

[27] D.Ponomarev, et.al., "Energy-Efficient Comparators for Superscalar Datapaths", IEEE Transactions on Computers, July 2004.

[28] I.Kim, M.Lipasti, "Half-Price Architecture", in Proc of International Symposium On Computer Architecture, 2003.

[29] M.Huang et.al., "Energy-Efficient Hybrid Wakeup Logic", in Proc of International Symposium on Low-power Electronics Design, 2002.

[30] A. Aggarwal, et. al., "Defining Wakeup Width for Efficient Dynamic Scheduling", in Proc. of International Conference on Computer Design, 2004.

[31] S. Song, et. al., "The PowerPC 604 Microprocessor", IEEE Micro, 14(5), pp. 8-17, Oct 1004.

[32] V. Zyuban, "Inhernently Low-Power High-Performance Superscalar Architectures", Ph.D. thesis, University of Notre Dame, 2000.

[33] 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.

[34] K. Wilcox and S. Manne. "Alpha processors: A history of power issues and a look to the future", in Cool-Chips Tutorial, November 1999.

[35] M. Stan and W. Burleson, "Bus-Invert Coding for Low-Power I/O", IEEE Trans. On VLSI, 3(1), 1995, pp. 49-58.

[36] C. Su, C. Tsui, and A. Despain, "Saving Power in the Control Path of Embeded Processors", IEEE Design and Test of Computers, 11(4), 1994, pp. 24-30.

[37] R. Canal, A. Gonzales, J. Smith, "Very Low Power Pipelines using Significance Compression", in Proc. MICRO, 2000.

[38] L. Villa, M. Zhang, and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction", in Proc. MICRO 2000.

[39] L. Benini, et al "Asymptotic Zero-Transition Activity Encoding for Address Busses in Low-Power Microprocessor-Based Systems." In Proc. GLSVLSI, pp. 77-82, 1997.

[40] L. Benini, et al "Address Bus Encoding Techniques for System-Level Power Optimization", in Proc. Design Automation and Test in Europe, pp. 861-866, 1998.