Reducing Delay and Power Consumption of the Wakeup Logic through Instruction Packing and Tag Memoization

Joseph Sharkey, Dmitry Ponomarev, Kanad Ghose

Department of Computer Science
State University of New York
Binghamton, NY 13902-6000
{jsharke, dima, ghose}@binghamton.edu

Oguz Ergin[1]

Intel Barcelona Research Center
Intel Labs, UPC
Barcelona, Spain
Oguzx.ergin@intel.com

Abstract. 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 result 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 two schemes to reduce the power consumption and the delays of the wakeup logic. Our first scheme – instruction packing – shares the associative part of an issue queue entry between two instructions, each with at most one non-ready source. As a result, the number of entries in the issue queue (and, hence, the length of the tag buses) can be reduced by a factor of two with almost no impact on the IPCs, because most instructions either enter the pipeline with at least one of their source operands ready, or do not make use of two source registers to begin with. Our second scheme – 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. While instruction packing results in the reduced length of the tag buses, tag memoization reduced the number of tag lines that need to be driven. We evaluate our designs using detailed microarchitectural simulations of the SPEC 2000 benchmarks and the SPICE simulations of the issue queue layouts.

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 address 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 driven on tag bus lines. A significant amount of energy dissipation results as the destination register address is driven on the tag busses. Energy dissipation occurs when the tag bus lines

[1] Currently with Intel Labs, UPC, Barcelona Spain. Work performed while at SUNY-Binghamton.

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 driving the tag busses. This is particularly true if comparators that dissipate energy only on a match are used within the issue queue [27].

The scope of this paper is to propose two fairly orthogonal techniques for reducing the energy dissipated in driving the tag lines. Our first approach reduces the effective length of the tag bus lines and the number of comparator bits driven by essentially reducing the number of issue queue entries through the opportunistic packing of two instructions into a single issue queue entry. Our second approach avoids the power dissipated in driving the tag lines by not driving the higher order bits in the tag bus if their value matches the corresponding values last driven on the same tag 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.

2. Instruction Packing

In a traditional RISC-like processor where each instruction can have at most two register source operands, each issue queue (IQ) entry has two comparators, which allow the instruction to track the arrival of both sources by monitoring the tag buses. In general, however, such a design results is a grossly inefficient usage of the CAM logic, because of two reasons: 1) Many instructions have only one source register operand, and therefore do not require the use of two tags (and two comparators) in the first place, and 2) of the instructions with 2 source operands, a large percentage have at least one of the source operands ready at the time of dispatch, again rendering the second comparator unnecessary. Our simulations showed that on the average across SPEC 2000 benchmarks, about 83% of the dynamic instructions enter the scheduling window with at least one of their source operands ready.

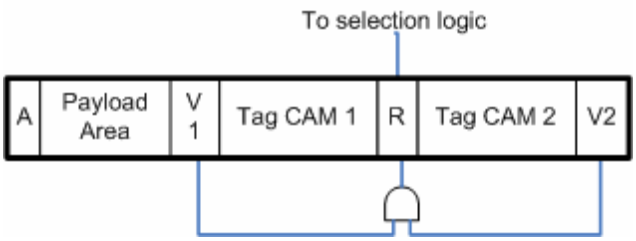

Fig. 1. Traditional IQ entry format.

These statistics have been presented before [4] and researchers have proposed different solutions to optimize the IQ design based on this inefficiency. In [4], the non-uniform IQ entry formats were used, i.e. some entries have a full set of tag comparators, other entries have just one comparator, and yet other IQ entries have no comparators. In [28], the tag buses were subdivided into the slow buses and fast

buses, such that the tag broadcast on the slow bus is delayed by one cycle. In this paper, we propose a different approach to optimizing the use of the CAM logic within the issue queue by packing multiple (two, for this paper) instructions into the same issue queue entry, effectively duplicating the RAM storage for these instructions (destination register addresses, literals, opcodes) and sharing the existing CAM logic. In effect, the aspect ratio of the issue queue changes: the number of issue queue entries become lower and the width of each entry goes up. In this section, we describe the details of our design.

Figure 1 shows a format of the issue queue entry used in traditional designs. The following fields comprise a single entry: a) entry allocated bit (A), b) payload area (opcode, FU type, destination register address, literals), c) tag of the first source, associated comparator (tag CAM word 1, hereafter just tag CAM 1, without the "word") and the source valid bit, d) tag of the second source, associated comparator (tag CAM 2) and source valid bit, and e) the ready bit. The ready bit, used to raise the request signal for the selection logic is set by AND-ing the valid bits of the two sources.

If at least one of the source operands is ready at the time of dispatch, the tag CAM associated with this instructions IQ entry remains unused. To exploit this idle tag CAM, we propose to share one issue queue entry between two such instructions. An entry in the IQ can now hold one or two instructions, depending on the number of ready operands of the stored instructions at the time of dispatching this instruction. Specifically, if both source registers of an instruction are not available at the time of dispatch, the instruction is assigned an IQ entry of its own and makes use of both tag CAMs in the assigned entry to determine when its operands are ready. An instruction that has only one source register that is not available at the time of dispatch is assigned just one half of an IQ entry. The remaining half of the IQ entry may be used by another instruction that also has one of its source registers unavailable at the time of dispatch. Sharing an IQ entry between two instruction also requires the IQ entry to be widened to permit the payload parts of both instructions to be stored, along with the addition of flags that indicate whether the entry is shared between two instructions and the status of the stored instruction(s). Figure 2 shows the format of an issue queue entry that supports instruction packing. Each IQ entry is comprised of the "entry allocated" bit (A), the ready bit (R), the mode bit (MODE) and the two symmetrical halves: the left half and the right half. The structure of each half is identical, so we will use the left half for the subsequent explanations.

A left half of each IQ entry contains the following fields:

1. Left half allocated (AL) bit. This bit is set when the half-entry is allocated.
2. Source tag and associated comparator (Tag CAM). This is where the tag of the non-ready source operand for an instruction with at most one non-ready source is stored.
3. Source valid left bit (SVL). This bit signifies the validity of the source from part b), similar to traditional designs. This bit is also used to indicate if the instruction residing in a half-entry is ready for selection (as explained later)
4. Payload area. The payload area contains the same information as in the traditional design, namely: opcode, bits identifying the FU type, destination register address and literal bits. In addition, the payload area contains the tag of the second source.

Notice that the tag of the second source does not participate in the wakeup, because if an instruction is allocated to a half-entry, the second source must be valid at the time of dispatch. Compared to the traditional design, the payload area is increased by the number of bits used to represent a source tag.

The contents of the right half are similar. The ready bit (R) is used when an instruction with two non-ready source operands is allocated into the full IQ entry, as explained below. To summarize, each entry in the modified IQ is divided into a left half and a right half, each is capable of storing an instruction with at most one non-ready source operand, or the two halves can be used in concert to house an instruction with 2 non-ready source operands. In general, the issue queue entry can be in one of the following three states: 1) the entry holds a single instruction, both source operands of which were not ready at the time of dispatch, 2) the entry holds two (or one with another half free) instructions, each of which had at least one source operand ready at the time of dispatch, or 3) the entry is free. The "mode" bit, stored within each IQ entry as shown in Figure 2, identifies the state of the entry. If the mode bit is set to 1, then the entry maintains a 2-operand instruction, otherwise it either maintains one or two single-operand instructions or it is free.

Since each entry can hold up to two instructions, fewer IQ entries are needed. However, despite the fact that each entry in the modified IQ shown in Figure 2 is somewhat wider than the traditional queue entry (due to the replication of the Payload area and three extra bits – AL, AR, and MODE), the amount of CAM logic per-entry does not change. Each entry still uses only two comparators – those are either used by one instruction, which occupies full entry, or are shared by two instructions, each located in half-entry. In the next few subsections, we describe the details of this technique.

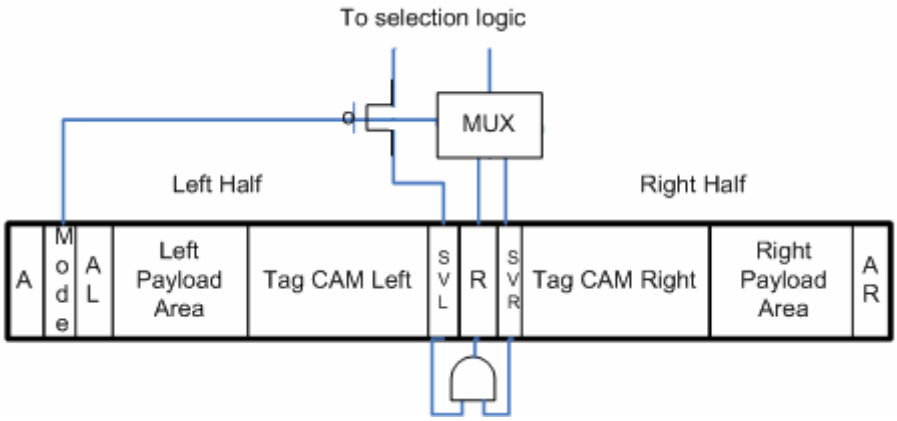

Fig. 2. Wakeup and Selection Logic Modified to Support Instruction Packing

Entry Allocation

To set up an issue queue entry for an instruction, the entry allocated bits corresponding to both halves (AL and AR), as well as the global "entry_allocated" bit (A) are associatively searched in parallel with register renaming and checking the

status of source physical registers. If the instruction is determined to have at most one non-ready source operand, the lowest numbered issue queue entry with at least one available half is allocated. If both halves are available within the chosen entry, then the instruction is written into the right half. After the appropriate half is chosen, both the "entry_allocated" bit of this half and the global A bits are reset. If an instruction is determined to have 2 non-ready source operands, then a full-sized entry is allocated, as dictated by the state of the A bits. The search for a full-sized and a half-sized entry occurs simultaneously, and the entry to be allocated is then chosen based on the number of non-ready source operands. This IQ entry allocation process is somewhat more complicated than similar allocation used in traditional designs, where just the A bits are associatively searched. However, there is no extra delay involved, because the searches occur in parallel. Similar issues with allocating the IQ entries are also inherent in other designs which aim to reduce the amount of associative logic in the queue by placing the instructions into the issue queue entries judiciously, based on the number of non-ready operands at the time of dispatch [4]. We will discuss what kind of information is written into the IQ for the various instruction categories later in the paper. But first, we describe how wakeup and selection are implemented in this scheme.

Instruction Wakeup

The process of instruction wakeup remains exactly the same as in traditional design for an instruction that occupies a full IQ entry (i.e. comes with 2 non-ready sources). Here, the ready bit (R) is set by AND-ing the valid bits of both sources. For instructions which occupy half of an IQ entry, the wakeup simply amounts to setting of the valid bit corresponding to the source that was non-ready when the instruction entered the IQ. The contents of the source valid bits are then directly used to indicate that the instruction is ready for selection (the validity of the second source is implicit in this case). The selection logic details are described next.

Instruction Selection

The process of instruction selection needs to be slightly modified to support instruction packing. To make the explanation easier, we assume that a 32-entry IQ is packed into a 16-entry structure, such that each entry is capable of holding two instructions with at most one non-ready source each, or one instruction with two non-ready sources. In a 32-entry IQ design, there are 32 request lines that can be raised by the awakened instructions – one line per IQ entry. In the instruction packing scheme, each of the two halves of each of the 16 entries requires a request line, thus retaining the same total number of request lines (32) and resulting in a similar complexity of the selection logic. In addition, the ready bits, used by the instructions allocated to full entries, also require request lines. Consequently, a straightforward implementation of the selection logic would require 48 (3x16) request lines, thus increasing the complexity, delay and power requirements of the select mechanism.

Such an undesirable elevation in the complexity of the selection logic can be avoided by sharing one request line between the R and the SVR bits. The shared request line is raised if at least one of the bits (the R or the SVR) is set. The R and the SVR bits are both connected to the shared request line through a multiplexor, which is controlled

by the “mode” bit of the IQ entry (Figure 2). Consequently, the overall delay of the selection logic increases only slightly – by the delay of a multiplexor. Notice also that the MUX control signal (the “mode” bit) is available in the beginning of the cycle when the selection process takes place (the “mode” bit is set when the issue queue entry is allocated). The request line driven by the SVL bit is controlled by the p-device, whose gate is connected to the “mode” bit. This request line will be asserted only if the “mode” bit is set to 0 (indicating that the IQ entry is shared between two instructions) and the SVL bit is set to 1.
Note that the only part of the selection logic that is modified is the process of asserting the request lines. The rest of the selection logic is unchanged compared to the traditional designs. The overall delay of the selection logic is thus increased by the delay of the multiplexor, whose control signal is preset (as the value of the “mode” bit is available as soon as the IQ entry is allocated).

Instruction Issue
We define instruction issue as a process of reading the source operand tags of the selected instructions and starting the register file access (effectively moving the instruction out of the IQ). When a grant signal comes back corresponding to the request line, which was shared between the R and the SVR, the issue logic has to know which physical registers have to be read. Conventionally, this information is conveyed by the contents of the tag fields. However, the register tags of an instruction with two non-ready sources (i.e. the instruction that occupies full IQ entry) and the register tags of an instruction with one non-ready source are generally stored in different locations within the IQ entry. In the former case, the tags are stored in the tag fields connected to both comparators – one tag is stored in the left half of the entry and the other tag is stored in the right half of the entry. In the latter case, both tags are stored in the right half of the entry, such that the tag of the non-ready operand is connected to the comparator and the other tag is simply stored in the payload area. Given this disparate locations of the source register tags, how would the issue logic know which tags to use when the grant signal corresponding to a shared request line comes back?
One solution is, again, to use the contents of the “mode” bit and a few multiplexors. This will, however, slightly increase the delay of the issue / register access cycle. A better solution, which avoids the additional delays in instruction issuing altogether, is as follows. When an instruction with two non-ready sources is allocated to the issue queue, the tag, which is connected to the left half comparator, is also replicated in the payload area storage for the second tag in the right half. As a result, both tags will be present in the right half of the queue, so these tags can be simply used for register file access, without regard for the IQ entry mode.

Benefits of Instruction Packing
Instruction packing, as described in this section, has several benefits over the traditional issue queue designs in terms of layout area, access delays and power consumption.
The area of the issue queue decreases, because compared to the traditional designs, the amount of RAM storage does not change (we use twice as fewer entries, but each

entry has about twice the amount of RAM), but the amount of associative logic is reduced by a factor of two.

The delay of the wakeup logic is reduced, because the tag buses become much shorter and the capacitive loading on these buses is also significantly reduced – the delay in driving the tag bus (which is a major component of the wakeup latency) is roughly reduced by half. Furthermore, shorter bitlines can potentially reduce the IQ access delays during instruction dispatching (setting up the entries) and issuing (reading out the register tags and literals). Finally, for the same reasons the power consumption is also reduced. Another potential reason for the reduction in the power consumption has to do with the use of fewer comparators. In the instruction packing, the tags of the source registers ready during dispatching are never associated with the comparators. In the traditional designs, each and every source tag is hooked up to a comparator. Unless these comparators are precharged selectively (based on whether or not a given IQ slot is awaiting for the result), unnecessary dissipations can occur than comparators associated with the already valid sources continue to fire.

In the result section, we quantify these savings using detailed simulations of SPEC 2000 benchmarks and also circuit simulations of the IQ layouts. Notice that all these benefits are achieved with essentially no degradation in the IPCs (committed Instructions Per Cycle). This is because most instructions (our results show 83%) have at least one of their sources ready at the time of dispatch, thus rendering the performance loss due to the smaller number of IQ entries negligible.

3. Tag Memoization

The tag memoization scheme 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 idea here is to conserve power expended in driving the tag 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 Cu and Cl, 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 AND-ing the output of this latch with the output of the comparator for the lower order bits. Figure 3 depicts this logic. We now describe this scheme in some detail.

Let Lb designate the latch used within an IQ entry to remember the match with the upper order bits driven on bus b with the tag value stored within a register operand field of the IQ entry. The tag driver logic for tag bus b also uses a latch array, Ub, to remember the upper order bits of the tag pattern that was driven onto the tag bus b. The following two cases arise when a tag value is to be driven on a tag bus:

If the upper bits driven on the bus b in the next broadcast match Ub, then only the lower order bits are driven on the tag bus. Entries that match the lower order bits and have their latch Lb set now produce a match signal. If, however, the upper order bits driven on the bus b do not match the contents of Ub, then the following actions are taken concurrently:

– The reset line shown in Figure 3 is driven to clear the contents of latch Lb in all of the IQ entries.
– Both upper and lower order bits of the tag are driven out on bus b
– Ub is updated

Clearing Lb in this case allows each entry to produce a match based on all of the tag bits - both upper order and lower-order bits.

The tag memoization scheme saves power by not driving the upper order bits of a tag bus whenever possible. The power savings are somewhat defeated by the need to drive the reset line on each tag bus, by the need to maintain the Ub latch, and dissipations within the Lb and the AND-gate used within each entry. One can save additional power dissipation by using the contents of Lb to disable Cu once Lb is set. Doing so prevents Cu from dissipating any power from false matches with the values floating on the upper order bits of the bus.

From a delay standpoint, the AND-ing of Lb with the output of Cl adds a slight delay in the generation of a request signal from matching entries. This added delay is however compensated to some extent by the smaller delay of Cl. (Cl has a smaller response time compared to that of a comparator that compares all bits of the tag value.)

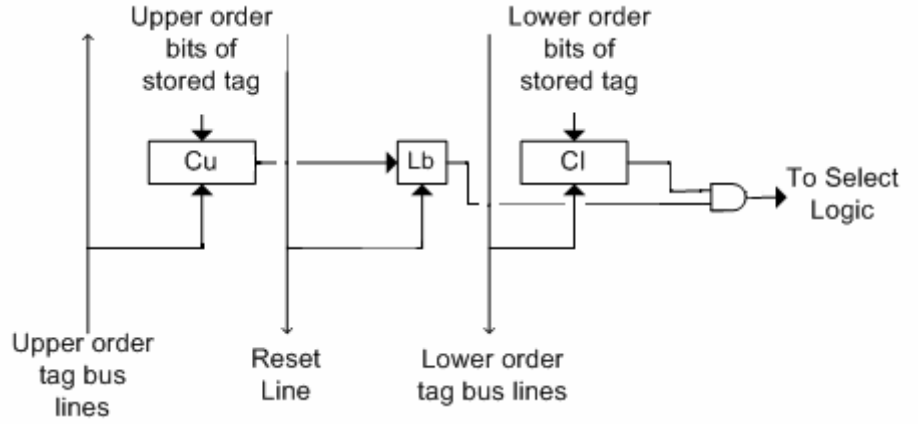

Fig. 3. Tag comparator configuration for the tag memoization scheme

One can force additional savings from the memoization scheme by assigning tag broadcasts to a bus whose Ub matches the upper order bits of the tag value to be driven. We call this "intelligent" tag bus assignment. There is, of course, some energy overhead in assigning tag broadcasts to specific buses in this case. Another possibility, and one that we have not explored here, 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.

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, an 7 bit tag comparator can be segmented into three parts: Ca(upper order two bits), Cb(next two bits), and Cr (remaining 3 bits). This arrangement requires two latches: one between Ca and Cb and another between

Cb and Cr; there is a reset line for each latch. These latches may be set independently, allowing for the gating off of either set of bits, or both. The match signal is derived by AND-ing the contents of the intervening latches and the output of the comparator segment covering the lower order bits.

4. Simulation Methodology

Our simulation environment includes a detailed cycle accurate simulator of the microarchitecture and cache hierarchy. While our simulator was developed from scratch, it uses the same binaries, system call interface and tools as the MIPS-like Simplescalar PISA ISA. All benchmarks were compiled with gcc 2.6.3 (compiler options: -O2) and linked with glibc 1.09, compiled with the same options. All simulations were run on a subset of the SPEC 2000 benchmarks consisting of 8 integer and 7 floating-point benchmarks. In all cases, predictors and caches were warmed up for 1 billion committed instructions and statistics were gathered for the next 200 million instructions. Table 1 presents the configuration of the baseline processor.

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.

Table 1. Configuration of the simulated processor.

Parameter	0Configuration
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
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	128 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 selector
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

5. Results

5.1 Instruction Packing

Table 2 shows the IPC loss due to instruction packing. These results are displayed in the form of a table rather than a graph because IPC differences are too small to be noticeable on the traditional bar graph. The columns, in order, show IPC results with a 32-entry issue queue, a 16-entry issue queue with instruction packing, an 8-entry IQ, and a 4-entry IQ with packing. The results show that a 16-entry issue queue utilizing instruction packing performs within 0.5% of a traditional 32-entry IQ. The configuration with an 8-entry queue packed into 4 wider entries is only shown to demonstrate that packing does not significantly degrade the performance even for very small issue queues. Here, for example, the performance loss is only 5.3% on the average.

Table 2. IPC for 32 and 8-entry traditional queues as compared to 16 and 4-entry queues supporting instruction packing.

Benchmarks	32IQ	16IQ_PACK	8IQ	4IQ_PACK
Gzip	1.544	1.587	1.594	1.412
Vpr	1.463	1.447	1.279	1.125
Gcc	1.128	1.128	1.105	1.032
Mcf	0.444	0.442	0.398	0.343
Parser	1.317	1.304	1.234	1.118
Vortex	1.996	2.001	1.908	1.672
bzip2	1.594	1.546	1.480	1.272
Twolf	1.209	1.161	1.104	0.968
Wupwise	2.212	2.212	1.924	2.212
Swim	1.511	1.511	1.412	1.269
Mgrid	1.218	1.218	1.210	1.218
Applu	1.337	1.337	1.345	1.278
Mesa	1.786	1.786	1.580	1.786
Art	0.399	0.400	0.332	0.243
Equake	1.724	1.723	1.522	1.312
IntAvg	**1.337**	**1.327**	**1.263**	**1.118**
FPAvg	**1.455**	**1.455**	**1.332**	**1.331**
Average	**1.368**	**1.363**	**1.279**	**1.211**

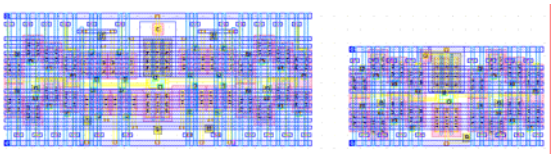

Fig. 4. Layout of a CAM bitcell (left) vs. an SRAM bitcell (right).

CMOS layouts of both the 32-entry traditional queue and the 16-entry packing queue show a 26.7% reduction in the issue queue area due to the use of instruction packing. Packing effectively reduces the number of CAM bitcells by half, while increasing the

number of SRAM bitcells in each row (but leaving the total number of SRAM bitcells in the IQ practically unchanged). Figure 4 presents the layouts of a CAM bitcell (left) and an SRAM bitcell (right).

As presented in Table 3, instruction packing achieves a 21.6% reduction in the wakeup delay (when a 32-entry IQ is packed into a 16-entry IQ). This delay reduction comes mainly from the shorter and lower-capacitance tag busses. Since the packing queue has half as many comparators, the tag busses are subsequently half as long and the delay of the tag bus drive is roughly reduced by half.

Table 3. Delays of a 16-entry queue supporting instruction packing compared to a 32-entry traditional queue.

	Tag-Bus Drive (ps)	Comparator Output (ps)	Final Match Signal (ps)	Total Delay (ps)
32-entry	224	219	126	569
16-entry Packing	131	201	114	446
Savings:	41.5%	8.2%	9.5%	**21.6%**

Finally, the instruction packing saves energy due to the presence of half as many tag comparators and shorter tag-busses. SPICE simulations show the 16-entry packing queue saves 37.99% of total wakeup power as compared to a traditional 32-entry queue, most of it coming from the savings in the tag bus drive energy (we present the detailed per-benchmark results in Figure 6, Section 5.3). We are currently in the process of performing complete analysis of the issue queue power dissipation, including the power of the selection logic as well as the power expended in establishing the IQ entries and also reading out the selected instructions. We will present these results in the final version.

5.2 Tag Memoization

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 comes 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 5 presents the number of most significant bits (MSBs), for each tag broadcast, that match those of the previous tag broadcast on that bus. Since two bits match 43% of the time, we consider 2-bit matches for the remainder of this discussion.

On a configuration with two separate intermediate latches, the two MSBs match an average of 48.1% of the time, while the next two MSBs match an average of 24.5% of the time. 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 11.1% of total tag-broadcast power. This power savings can be improved by selectively arbitrating for tag busses so as to maximize tag matches. Such an “intelligent” tag bus assignment is able to achieve tag broadcast power savings by as much as 16.1%.

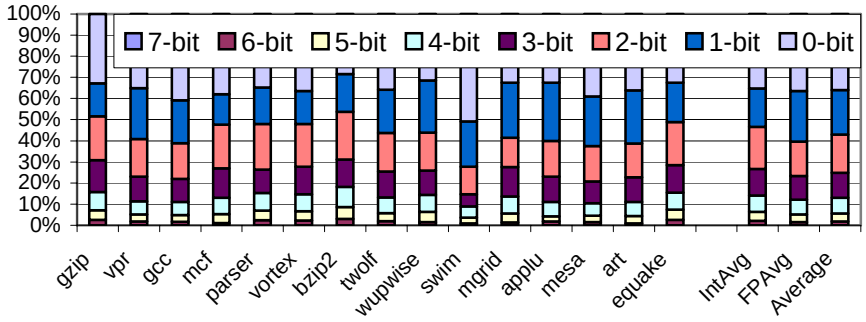

Fig. 5. Number of most significant bits matching those of the previous tag broadcast on each tag bus.

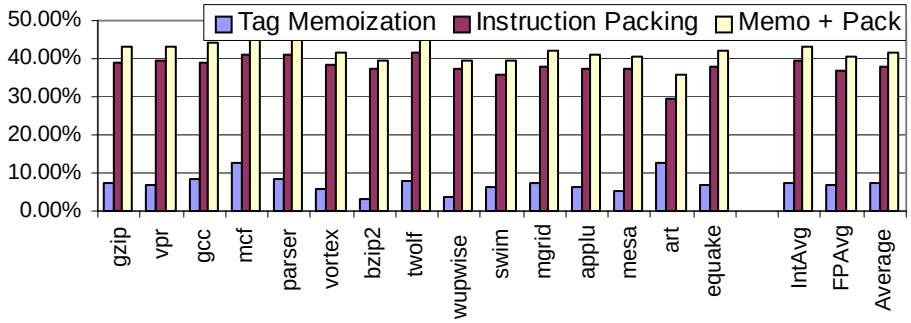

Fig. 6. Wakeup power savings.

5.3 Combining Instruction Packing with Tag Memoization

Instruction packing and tag memoization are two orthogonal approaches to reducing the power consumption of instruction wakeup. Instruction packing aims to reduce the length of the tag-broadcast while tag memoization aims to reduce the number of tag busses driven. The 16-entry queue that supports instruction packing and uses 2 + 2 tag memoization (where we segment the 7-bit comparator into 3 segments – two higher order segments, each 2 bits wide, hence 2+2, and a 3-bit segment for the lower order bits) with judicious bus arbitration (where the bus for a tag broadcast is selected to maximize the likelihood of matches in the most significant bits) reduces the wakeup power by 47.6% as compared to a traditional 32-entry queue. If the random bus selection is used, then the power reduction is about 44%. The per-benchmark results, showing power savings of both schemes in isolation, as well as the combined power savings, are presented in Figure 6. In this figure, the tag memoization results are presented for the scheme where the tag buses are randomly allocated. As seen from Figure 6, the combined power savings from these two schemes is not additive. This is simply because instruction packing changes the relative position of a source register

address over the tag busses, changing the pattern of tag matching, thus impacting the effectiveness of tag memoizatoin. (Higher power savings (not shown) are achieved in the combined scheme when intelligent tag bus assignment is used.)

6. 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]. Also in [26], the issue queue power was reduced by using zero-byte encoding and bitline segmentation. 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.

The observation that many instructions are dispatched with at least one of their source operands ready is not new – it was used in [4], where the scheduler design with reduced number of comparators was proposed. In that scheme, some IQ entries have two comparators, others have just one comparator, and yet others have zero comparators. Despite significant reduction in the number of comparators, the size of the issue queue, and thus the length of the tag busses, was not reduced. In addition, the last-tag speculation mechanism introduced in [4] requires the extra logic to handle possible mispredictions. 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. The design again relied on the last-arriving operand prediction to hook the last arriving operand (which actually identifies when the instruction wakes up) to the fast bus to avoid the wakeup delays.

One approach to reducing scheduling complexity involves pipelining the scheduling logic into separate wakeup and select cycles [2,8]. It is shown in both [2] and [8] that naively pipelining the scheduling logic doesn't provide for the back-to-back execution of dependent instructions and thus significantly degrades performance. To overcome this, [2] uses the status of an instruction's grandparents to wakeup the instruction earlier in a speculative manner. Kim and Lipasti [8] 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. A smaller issue queue can be used in this design, because the instructions forming the Macro-OP share the same issue queue entry. The concept of dataflow mini-graphs [21] is similar to Macro-Op scheduling in that groups of instructions are scheduled together. The order of instructions within the mini-graph are determined statically and the scheduler only considers "handles", or groups of instructions, for scheduling. This relies on re-compilation of code to generate these "handles" in the binary.

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]. Brown et.al. [7] 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.
Scheduling techniques based on predicting the issue cycle of an instruction [10, 11,12,13,15,16,18] remove the wakeup delay from the critical path and remove the CAM logic from instruction wakeup, but need to keep track of the cycle when each physical register will become ready. In [17], the wakeup time prediction occurs in parallel with the instruction fetching.

7. Concluding Remarks

We proposed two orthogonal schemes to reduce the power consumption of the wakeup logic. Instruction packing combines two instructions within the same issue queue entry if both instructions have at most one non-ready source operand at the time of dispatch. Consequently, the number of issue queue entries, and thus the length of and the capacitive loading on the tag busses, can be reduced substantially, leading to faster access and lower power dissipation. In addition, the layout area of the issue queue is also reduced. Tag memoization avoids driving the portion of the tag if it did not change from what was previously driven on the same tag bus. Combined, the two techniques result in about 47.6% reduction in the wakeup power. Additionally, instruction packing also achieves 26% reduction in the issue queue layout area and 21% reduction in the wakeup delay. The delay of the selection logic increases only slightly - by the delay of a single multiplexer (with the pre-set control signal). Thus, significant overall reduction in the scheduler delay, and thus higher frequency, can be also realized.

8. References

[1] S. Palacharla, et. al., "Complexity-Effective Superscalar Processors", in the Proc. of the Int'l Symp. on Computer Architecture, 1997
[2] J. Stark, et. al., "On Pipelining Dynamic Instruction Scheduling Logic", in the Proc. of the Int'l Symp. 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 the Proc. of the Int'l Symp. on Computer Architecture, 2002.
[5] E. Brekelbaum et. al., "Hierarchical Scheduling Windows", in the Proc. of the Int'l Symp. on Microarchitecture 2002.
[6] A. Lebeck et. al. A Large, "Fast Instruction Window for Tolerating Cache Misses", in the Proc. of the Int'l Symp. on Computer Architecture, 2002.
[7] M. Brown, J. Stark, Y. Patt. "Select-Free Instruction Scheduling Logic", in the Proc. of the Int'l Symp. on Microarchitecture 2001.
[8] I. Kim and M. Lipasti, "Macro-Op Scheduling: Relaxing Scheduling Loop Constraints", in the Proc. of the Int'l Symp. on Microarchitecture 2003.
[9] A. Cristal, et.al., "Out-of-Order Commit Processors", in the Proc. of the Int'l Symp. on High Performance Computer Architecture, 2004.

[10] D. Ernst, A. Hamel, T.Austin, “Cyclone: a Broadcast-free Dynamic Instruction Scheduler with Selective Replay”, in the Proc. of the Int’l Symp. on Computer Architecture’03
[11] Hu, J., Vijaykrishnan, N., Irwin, M., “Exploring Wakeup-Free Instruction Scheduling”, in the Proc. of the Int’l Symp. on High Performance Computer Architecture, 2004.
[12] R.Canal, A. Gonzalez, “A Low-Complexity Issue Logic”, in the Proc. of the Int’l Conference on Supercomputing, 2000.
[13] R.Canal, A.Gonzalez, “Reducing the Complexity of the Issue Logic”, in the Proc. of the Int’l Conference on Supercomputing 2001.
[14] S. Raasch, N.Binkert, S.Reinhardt, “A Scalable Instruction Queue Design Using Dependence Chains”, in the Proc. of the Int’l Symp. on Computer Architecture, 2002.
[15] J. Abella, A.Gonzalez, “Low-Complexity Distributed Issue Queue”, in the Proc. of the Int’l Symp. on High Performance Computer Architecture, 2004.
[16] P. Michaud, et.al. “Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors”, in the Proc. of the Int’l Symp. on High Performance Computer Architecture, 2001.
[17] T. Ehrhart, S. Patel, “Reducing the Scheduling Critical Cycle using Wakeup Prediction”, in the Proc. of the Int’l Symp. on High Performance Computer Architecture, 2004.
[18] Y. Liu, et. al., “Scaling the Issue Window with Look-Ahead Latency Prediction”, in the Proc. of the Int’l Conference on Supercomputing 2004.
[19] Z. Chishti, T. Vijaykumar, “Wire Delay Is Not a Problem for SMT”, in the Proc. of the Int’l Symp. on Computer Architecture 2004.
[20] S. Srinivasan et. al. “Continual Flow Pipelines”, in the Proc. of the Int’l Conference on Architectural Support for Programming Languages and Operating Systems, 2004.
[22] A. Buyuktosunoglu, et.al.,”A Circuit-Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors”, GLSVLSI, 2001.
[23] D.Folegnani, A.Gonzalez, “Energy-Effective Issue Logic”, in the Proc. of the Int’l Symp. 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 the Proc. of the Int’l Symp. on Microarchitecture 2001.
[25] A. Buyuktosunoglu et.al., “Energy-Efficient Co-adaptive Instruction Fetch and Issue”, in the Proc. of the Int’l Symp. 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 the Proc. of the Int’l Symp. on Computer Architecture, 2003.
[29] M.Huang et.al., “Energy-Efficient Hybrid Wakeup Logic”, in the Proc. of the Int’l Symp. on Low-Power Electronics and Design, 2002.
[21] A. Bracy, et. al. “Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth”, in the Proc. of the Int’l Symp. on Microarchitecture 2004.