An Energy-Efficient Instruction Scheduler Design with Two-Level Shelving and Adaptive Banking

Yu-Lai Zhao, Xian-Feng Li, Dong Tong, and Xu Cheng
Microprocessor Research and Development Center, Peking University, Beijing 100871, China
E-mail: {zhayulai, lixianfeng, tongdong, chengxu}@mprc.pku.edu.cn
Received January 19, 2006; revised October 19, 2006.

Abstract Mainstream processors implement the instruction scheduler using a monolithic CAM-based issue queue (IQ), which consumes increasingly high energy as its size scales. In particular, its instruction wakeup logic accounts for a major portion of the consumed energy. Our study shows that instructions with 2 non-ready operands (called 2OP instructions) are in small percentage, but tend to spend long latencies in the IQ. They can be effectively shelled in a small RAM-based waiting instruction buffer (WIB) and steered into the IQ at appropriate time. With this two-level shelving ability, half of the CAM tag comparators are eliminated in the IQ, which significantly reduces the energy of wakeup operation. In addition, we propose an adaptive banking scheme to downsize the IQ and reduce the bit-width of tag comparators. Experiments indicate that for an 8-wide issue superscalar or SMT processor, the energy consumption of the instruction scheduler can be reduced by 67%. Furthermore, the new design has potentially faster scheduler clock speed while maintaining close IPC to the monolithic scheduler design. Compared with the previous work on eliminating tags through prediction, our design is superior in terms of both energy reduction and SMT support.

Keywords content associative memory (CAM), energy-efficient architecture, instruction scheduler, tag elimination, waiting instruction buffer

1 Introduction

The improved performance of modern processors often comes at the cost of increased power or energy consumption. Today power has become the first constraint to the performance goal. Even worse, the power dissipation in the superscalar processor datapath is unevenly distributed, leading to “hot spot” problems on critical datapath components and severely reducing the reliability of the circuit. Therefore the researches that try to break the way the energy grows with increased issue width and instruction window size are very important.

One of the major contributors to the overall power consumption is the instruction scheduler which issues dynamic instructions out-of-order. Researchers have reported the instruction scheduler power to account for 20%-25% of the total chip power on average[1,2]. In particular, the wakeup function is the dominant power consuming part of the issue logic, which is reflected in high complexity circuits and logic activities[3].

The most prevalent way to implement the instruction scheduler is using a CAM-based issue queue (IQ), as Fig.1 shows. The IQ can store several instructions, but generally fewer than the total number of in-flight instructions. Each entry contains an instruction that has not been issued or has been issued speculatively but not yet validated and thus might need to be reexecuted.

In general, an entry contains payload RAM cells to store operations, destination operand tag, and flags indicating whether source operands are ready, and CAM cells to store source operand tags. When instructions are nearing the completion of their execution, they broadcast their result tags onto the result tag bus. The wakeup logic compares each source tag in the queue with multiple broadcast tags and marks the operand ready if there is a match. When all input operands are available, a request is made to the select logic which chooses instructions to execute using some policy such as oldest ready first. The selected instructions receive a grant signal from the selection logic and they will be sent forward to later pipeline stages. To priority oldest ready instructions, Alpha 21264 assigns an age to each in-flight instruction, which is encoded in the arbiter cells in select logic.

The high energy consumption of CAM-based IQ lies in its high circuit complexity and logic activity. Fig.2...
shows a single cell of the CAM array of the IQ. The
cell shown in detail compares a single bit of the operand
tag with the corresponding bit of the result tag. The
operand tag bit is stored in the RAM cell. Write word
lines which connect to the RAM cell provide the oppor-
tunity to write new instructions to the IQ. Since the
maximum number of instructions that are placed to the
queue equals the issue width, IW of a processor, IW
write ports are needed for sufficient write bandwidth.
The corresponding bit of the result tag is driven on the
tag lines.

![Diagram of CAM and instruction buffer](image)

Fig. 2. For a single CAM cell, a RAM cell is used to store the tag
bit, and the pull-down stack constitutes the comparator.

The match line is precharged high. If there is a mis-
match between the operand tag bit and the result tag
bit, the match line is pulled low by one of the pull-down
stacks. The pull-down stacks constitute the compar-
ators shown in Fig. 1. The match line extends across all
the bits of the tag, i.e., a mismatch in any of the bit
positions will pull it low. In other words, the match line
remains high only if the result tag matches the operand
tag. The above operation is repeated for each of the
result tags by having multiple tag and match lines as
shown in the figure. Finally, all the match signals are
ORed to produce the ready signal. For one operand
comparator in each issue queue entry, the number of
the bits in the tag is the 2-base logarithm of the num-
ber of physical registers. The energy of match operation
is dominant for CAM and considerably larger than read
and write operations of RAM. This is mainly due to the
large size of CAM cells and high switching activity of
comparators (since mismatches occur most of the time
and they are always precharged and pulled-down).

In this paper, we propose a new scheduler de-

gign based on two observations. First, only a small
percentage of instructions enter the issue queue with
two operands non-ready[4]. The second from our de-
tailed simulation is that these 2OP instructions gen-
elally spend a long latency waiting for the early-arriving
operand, and then a short period for the last-arriving
operand.

We propose to employ a small RAM-based waiting

2 Related Work

Researches have proposed several ways to reduce
the power consumption of the IQ. Dynamic adaptation

techniques[3–7] partition the IQ into multiple segments
and deactivate some segments periodically, when the
applications do not require the full IQ to sustain the
commit IPC. They are based on the observations that
occupancy of the IQ are averagely far below the full
size. While the adaptive techniques can capture the
single-thread workload characteristics, the SMT pro-
cesors are generally less amenable to such optimizations,
because the occupancy of the IQ is typically high as it
is shared among multiple threads. Moreover, schemes
to resize the IQ may be unsuitable to SMT processors
considering the distinct behaviors between threads.

A dependence-based scheme breaks the issue logic
into several FIFO queues in [9]. The dispatch logics for-
wards each instruction to the FIFO queue in which the
last instruction is the producer of a source operand; if no
FIFO queue meets this condition, the instruction goes
to an empty queue. If no empty queue is available, the
dispatch stage stalls. Placing instructions in this way
guarantees that the instructions in a given FIFO queue
execute sequentially; thus, this scheme monitors only
the youngest instruction in each queue for the potential
issue. The technique removes the wakeup logic but sacri-
fices an IPC degradation of 6.3%. They only considered
for clock speed improvement, and did not show power
characteristics. When applying to the SMT processors,
each dependence-based FIFO queue must be dedicated to a certain thread and cannot be shared to make the scheme work. Therefore, the number of FIFO queues and the total queue size are proportional to the number of threads, which introduces significant waste.

Ernst et al. proposed tag elimination technique to reduce the complexity and power of the IQ in [4]. They used the last tag predictor to predict the last-arriving operand for 20P instructions and speculatively issue them when that operand is ready. At the register read stage, the prediction is validated by reading the ready bit of the other operand. If the prediction is incorrect, the scheduler pipeline must be flushed and restarted, in a fashion identical to latency mispredictions. By employing a gshare-style predictor with appropriate sizing, they achieved a virtually 95% prediction accuracy for single-thread applications. Experiments on a 4-way superscalar processor show small impact on IPC, while gaining dramatic improvements in instruction per second and energy-delay product. They did no experiments on a SMT processor. In Section 6, we provide a quantitative comparison of our techniques with the last-tag predictor based scheme.

A resource-conscious IQ design is related with our work in that it also employs a waiting instruction buffer (WIB) to tolerate cache misses in [10]. The instructions that are directly or indirectly dependent on cache-missed loads are moved out of the conventional, small IQ to a much larger WIB at register read stage. When the long latency operation completes, the instructions are reinserted into the IQ. Their work is targeted at designing a large instruction window with a limited-sized IQ to expose more ILP. But their work is not energy-aware, since a large WIB may consume considerable energy, and the miss-dependent instructions that are moved out and reinserted into the IQ will also waste the issue slots.

Our proposed architecture employs a WIB similar to [10] by shelving the 20P instructions after they are renamed. However, our 20P instruction WIB is a small RAM-based structure at rename stage. It is targeted at reducing tag comparators in the IQ. To further downsize the IQ and reduce the bit-width of tag comparators, we use adaptive banking scheme which has instructions in banks with balanced usage. Inherently, reducing the number and bit-width of tags brings with it reduced load capacitance of CAM logic, which results in a faster scheduler clock speed. But we do not consider clock speed improvement because we focus on energy efficiency.

3 Methodology

3.1 Energy Estimator

We use the Wattch power model for the energy estimation of CAM and RAM structures[11]. The CAM energy estimator considers the energy to insert new instruction tags ($E_{insert}$) and the energy to match result tags ($E_{match}$) separately. The RAM energy estimator considers the read energy ($E_{read}$) and write energy ($E_{write}$) separately. However, the Wattch estimates the average access energy of CAM and RAM. To keep in line with contemporary processors, we use the process parameters for a 0.18µm process. All the results use Wattch’s aggressive clock-gating style (“cc3”). In this clock-gating model, power is scaled with port or unit usage and inactive units still dissipate 10 percent of the maximum power. The energy estimator is combined with the performance simulator to estimate the total energy consumption on the fixed amount of workloads. The overall energy is normalized to the baseline design. Other processor structures are not considered because we focus on the scheduler and the influences are insignificant.

3.2 Simulation

We use M-Sim simulator for estimating the performance impact of our proposed organization[12]. M-Sim is a significantly modified version of SimpleScalar 3.0d[13] that supports the SMT processor model, which also implements separate models of the IQ, the reorder buffer (ROB), and the physical register file. In the SMT model, the threads share the IQ, the pool of physical registers, the execution units, and the caches, but have separate rename tables, program counters, load/store queues and reorder buffers. Each thread also has its own branch predictor. The details of the baseline processor configuration are shown in Table 1. In the baseline SMT model, the I-Count fetch policy is implemented and fetching is limited to 2 threads per cycle.

We choose 7 SPEC2000 integer and floating point benchmarks for superscalar processor and 6 benchmark mixes for SMT processor. The multithreaded workloads contain a subset of the possible combinations of benchmarks. They are classified as low, medium, and high ILP, where the low ILP benchmarks are memory bound and the high ILP benchmarks are execution bound. All workloads are described in detail in Table 2. The initialization part of each benchmark is skipped and the single interval of 100 million instructions is simulated using the procedure prescribed by the SimPoint tool[14]. For multithreaded workloads, we measured a fixed amount of work. Every thread stopped after its own interval. When all threads had completed their intervals, the simulation stopped. We use the throughput IPC to evaluate the global performance.

4 Two-Level Shelving

4.1 Motivation

The two-level shelving architecture is motivated from the following two observations.

First, only a small percentage of instructions enter the IQ with 2 operands non-ready, which is shown in
Table 1. Simulated 8-Wide Issue Processor which Supports both Superscalar and SMT Execution Modes

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine Width</td>
<td>8-wide fetch, 8-wide issue, 8-wide commit</td>
</tr>
<tr>
<td>Window Size</td>
<td>128-entry issue queue, 64-entry load/store queue, 512-entry ROB</td>
</tr>
<tr>
<td>FU Latency (total/issue)</td>
<td>8 Int Add (1/1), 2 Int Mult (3/1)/Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 2 FP Mult (4/1)/Div (12/12)/Sqrt (24/24)</td>
</tr>
<tr>
<td>Physical Registers</td>
<td>512 physical registers</td>
</tr>
<tr>
<td>L1-Cache</td>
<td>64KB, 2-way associative, 128 byte line</td>
</tr>
<tr>
<td>L1 D-Cache</td>
<td>64KB, 4-way associative, 256 byte line</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2MB, 8-way set-associative, 512 byte</td>
</tr>
<tr>
<td>Unified</td>
<td>line, 8 cycles hit time</td>
</tr>
<tr>
<td>Front-End</td>
<td>64 entry fetch queue, 2K entry gshare, 10-bit global history per thread</td>
</tr>
<tr>
<td>Pipeline</td>
<td>24-entry, 2-way set-associative BTB</td>
</tr>
<tr>
<td>Structure</td>
<td>Fetch, decode, rename, issue, register</td>
</tr>
<tr>
<td>Memory</td>
<td>128 bit wide first chunk 150 cycles, next chunk 2 cycles</td>
</tr>
</tbody>
</table>

Fig. 3. Ernst et al. made the observation on a 4-wide processor\[4]. We confirm their results on the simulated 8-wide processor. Results also show that SMT workloads exhibit similar behaviors. On average, 15% dynamic instructions are 2OP instructions. The fewer 2OP instructions lead all IQ entries to have 2 tag comparators, which is inefficient.

Table 2. Simulated Single- and Multi-Threaded Workloads which Contains 4 SPECint, 3 SPECfp and 6 benchmark mixes of different ILP levels

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>SPECint</td>
</tr>
<tr>
<td>parser</td>
<td>SPECint</td>
</tr>
<tr>
<td>crafty</td>
<td>SPECint</td>
</tr>
<tr>
<td>grip</td>
<td>SPECint</td>
</tr>
<tr>
<td>equake</td>
<td>SPECfp</td>
</tr>
<tr>
<td>swim</td>
<td>SPECfp</td>
</tr>
<tr>
<td>vortex</td>
<td>SPECfp</td>
</tr>
<tr>
<td>mix1 (twolf, vpr, swim, parser)</td>
<td>4 low ILP</td>
</tr>
<tr>
<td>mix2 (applu, ammp, mgrid, galgel)</td>
<td>4 med ILP</td>
</tr>
<tr>
<td>mix3 (wupwise, grip, vortex, mess)</td>
<td>4 high ILP</td>
</tr>
<tr>
<td>mix4 (mcf, equake, meso, vortex)</td>
<td>2 low ILP +2 high ILP</td>
</tr>
<tr>
<td>mix5 (art, lucas, galgel, gcc)</td>
<td>2 low ILP+2 med ILP</td>
</tr>
<tr>
<td>mix6 (grip, wupwise, bsmd, apsi)</td>
<td>2 med ILP+2 high ILP</td>
</tr>
</tbody>
</table>

becomes ready. In our baseline configuration shown in Fig. 4, 2OP instructions spend about 20 cycles waiting for the early-arriving operand, and then about 3–4 more cycles waiting for the last-arriving operand on average. The other non-2OP instructions spend 10 cycles on average. Fig. 5 shows the latency distribution for 2OP instructions in more detail. It can be seen that dominant percentage of 2OP instructions wait more than 10 cycles before getting ready. By keeping these instructions waiting outside the IQ, we can also benefit from downsizing the energy- and cycle-critical IQ, since instructions with long waiting latency tend to block the IQ.

![Average Waiting Latency](image)

Fig. 4. On average, 2OP instructions tend to spend a long latency waiting for the first operand, and other instructions generally spending fewer waiting cycles.

![2OP Instr Latency Distribution](image)

Fig. 5. Dominant percentage of 2OP instructions waiting more than 10 cycles before getting ready.

4.2 Waiting Instruction Buffer

In our proposed two-level shelving architecture shown in Fig. 6, a RAM-based 2OP instruction WIB is employed as the first level shelving buffer, while the IQ is viewed as the second level shelving buffer. The traditional monolithic IQ is split into two hierarchical structures. The WIB stays in the Rename stage, while the IQ stays in the Issue stage.

In traditional design, 2OP instructions experience their whole waiting lifetime in the IQ. But in our two-level shelving architecture, they experience the first part of lifetime (usually long) in the WIB waiting for the first arriving operand, and the second part of lifetime

![2OP Instr vs IQ](image)

Fig. 3. Generally less than 20% of total dynamic instructions being dispatched with 2 non-ready operands. (Most instructions are dispatched with at most one non-ready operand.)

Second, most 2OP instructions will wait for a long latency in the IQ before one of their source operands
(usually short) in the IQ waiting for the second arriving operand. After the instructions are renamed and placed in the ROB in order, the dependences have been tracked through the map table. Each operand is known to be ready or not through the “valid bit” in the map table. After that, 2OP instructions are moved to the WIB, while the other and succeeding instructions are dispatched to the IQ normally. If the WIB is full, the rename stage stalls.

Inclusion the ones positioned behind the head entry in the WIB, can only be allocated to the normal entries. The instruction in the head of the WIB, say I, can be dependant on a predecessor, which is securely not dependent on any instructions in the WIB (directly or indirectly). Therefore, after I is inserted in the golden entry, it will be safely issued sooner or later. Under most cases, all instructions can be allocated in the normal entries. And only when the IQ reaches full, this golden entry will be strictly allocated to the head instruction of the WIB to avoid deadlocks.

5 Adaptive Banking

Banking is a well-known technique to resolve the bandwidth problem in cache memories or to reduce the size and port requirements in large, highly-ported structures such as the register file. The CAM-based IQ can also be divided into several banks while sharing the global select logic, since instructions in many banks may be simultaneously ready. Furthermore, using the age-ordered select logic as in Alpha 21264, the multi-banked IQs need not be compacted, since the age of each instruction is encoded in the arbiter cells. Allocation logic will pick holes in the IQs to allocate succeeding instructions.

Register tags are used to track the dependencies between instructions and placed in the CAM part in the IQ. The two-level shelving architecture enforces that each instruction enters the IQ with at most one operand non-ready. We call this operand tag the Waiting Tag. As our first allocation rule, some certain bits of the waiting tag, say high-order n bits, are used as the bank ID to hash instructions into different IQ banks. The low-order \( \log_2 (N - n) \) bits (\( N \) is the register file size) are placed in the CAM part for comparison, as Fig. 7 shows. When a value is produced, its tag only needs to be broadcast to the dedicated bank. An additional “S” bit in each

![Fig.6. Two-Level shelving architecture employing a small RAM-based buffer to temporarily hold all instructions that are dispatched with two non-ready operands.](image)

4.3 Avoiding Deadlocks

The scheme described in above section may cause a deadlock situation. The WIB permits the succeeding dependent instructions to enter the IQ ahead in an out-of-order manner. But if the IQ is already full, and each instruction in the IQ is directly or indirectly dependent on at least an instruction in the WIB, the deadlock occurs. No instruction in the IQ can be issued and no instruction in the WIB can enter the IQ. Although the deadlock occurs rarely, it needs to be addressed. This is inherently due to the reason that true dependences exist between leading and succeeding instructions in program order while our architecture permits an out-of-order insertion into the IQ.

To avoid this deadlock, we reserve a Golden Entry in the IQ which can be allocated only to the instruction which is positioned in the head of the WIB when all the other entries are occupied. All other instructions, including the ones positioned behind the head entry in the WIB, can only be allocated to the normal entries. The instruction in the head of the WIB, say I, can be dependant on a predecessor, which is securely not dependent on any instructions in the WIB (directly or indirectly). Therefore, after I is inserted in the golden entry, it will be safely issued sooner or later. Under most cases, all instructions can be allocated in the normal entries. And only when the IQ reaches full, this golden entry will be strictly allocated to the head instruction of the WIB to avoid deadlocks.

![Fig.7. Banked IQ organization with a small decoder before the banks. (Only the low-order bits need to compare with wakeup tags. S bit is used to indicate whether the left and right operands need to be switched when the values are read to the function units.)](image)
entry is indicating whether the left and right operands need to be switched to the original order when reading to the functional units.

If instructions are evenly distributed, the multi-banked IQs can perform close to a monolithic IQ which share entries between all renamed instructions. However, the performance of multi-banked IQ is compromised by IQ bank conflicts which refer to the situations where the instruction cannot acquire a position in the dedicated bank according to the allocation rule, i.e., the bank ID bits of 1OP instructions. Under these situations, a single bank is filled up while other banks may be still under-utilized.

As a second allocation rule to remedy the IQ bank conflicts problem, we selectively insert 0OP instructions into the Least Occupied bank on a cycle-by-cycle basis, because they need no comparators. There are abundant 0OP instructions which selectively fill up the least occupied bank, which are supposed to keep the occupancies between the banks evenly.

Choosing other n bits as the bank ID makes no improvement in hashing 1OP instructions into banks, since the registers allocated in sequence from the free list will experience variant-long lifetime, which results in a disordered free list after a short interval. The 0OP instructions can also be placed in a stand-alone bank with no CAM tag comparators. But this is trivial since the bit-widths of comparators have already been reduced. And a fixed-sized stand-alone 0OP bank may add to bank conflicts since they cannot accommodate 1OP instructions.

The energy reduction comes from the following aspects. First, the IQ is down-sized, meaning that each insert or wakeup operation will activate the dedicated bank. Second, the accesses can be distributed, which brings more energy reduction opportunities if properly gated. Third, the bit-width of the CAM tag comparator is reduced in all entries of the dedicated banks.

To avoid deadlocks in combination with the two-level shelving architecture, each bank must also maintain a reserved golden entry.

6 Evaluation

6.1 Performance

First, we evaluate the performance impact of the two-level shelving scheme which splits the monolithic IQ into the first-level WIB and the second-level IQ. There are mainly two reasons leading to the performance loss in this scheme. First, the WIB or the IQ may be filled up while the other is under-utilized which may increase dispatch stage stalls. Second, the instructions behind the 8 leading ones in the WIB can only be examined when they move to the head. This restricts the out-of-order issue capabilities by introducing semi-in-order semantics by the WIB.

Considering that 2OP instructions tend to spend a long latency waiting for the first-arriving operand, and only instructions positioned in the front of the WIB can be examined for inserting into the IQ, the WIB must be sized to an extent that will not introduce outstanding dispatch stage stalls when the WIB is full. Fig.8 presents the normalized IPC obtained by varying the size of the WIB and the IQ. As Fig.8 shows, when a smaller 16-entry WIB is employed, the average IPC degradation is 3%. The slowdown is mainly due to the under-sized WIB which often stalls dispatching instructions. However, a processor with a 24- or 32-entry WIB degrades IPC within 1% of the baseline for most workloads. These two configurations achieve few dispatch stage stalls which are close to that of a monolithic IQ. If the WIB is sized larger and the IQ is adjusted smaller accordingly, we find the WIB is often under-utilized, and the IQ becomes the performance bottleneck, which is obviously not an option. Therefore, a properly-sized WIB (in a proper range of the total scheduler size) can minimize the impact of imbalanced usage of resources.

Fig.8. Performance impact by varying the size of the WIB and the IQ accordingly. In our experiments, a processor with 24- or 32-entry WIB degrades IPC within 1% of the baseline.
Even for the best configurations, certain degree of performance loss is observed. This is mainly due to the semi-in-order dispatch restriction by the WIB for simplified, low-power circuits. We observe that the slowdown is at the worst within 4% budgets (equake, swim, and mix 4 perform worst). The semi-in-order restriction by the WIB incurs no significant performance loss because only a small percentage of dynamic in-flight instructions (which are positioned behind the leading eight ones) might not be monitored. In most cases, their waiting latency can be overlapped with that of the leading instructions within the WIB. Fig. 9 shows the proportion of WIB instructions that get ready before entering IQ. These hindered instructions account for only 15% of total WIB instructions and 3% of total dynamic instructions across the workloads.

Second, we evaluate the performance impact of the adaptive-banking scheme. The choice of the number of IQ banks must guarantee no significant bank conflicts, since every dedicated bank contains fewer entries. Fig. 10 gives the normalized IPC obtained by varying the number of IQ banks with a fixed-sized 32-entry WIB. We observed 1.5% and 1.9% IPC degradation by harmonic mean when the IQ is divided into 2 and 4 banks respectively. However, an 8-bank IQ causes a considerable IPC loss, especially for SMT workloads. In our experiments, the processor with 8 banks drops IPC by 8.1% by harmonic mean.

We feel that the performance loss within 2% is well tolerable noting that our primary goal is to achieve a scalable and energy-efficient scheduler.

### 6.2 Energy

Table 3 shows the max energy per access estimation for different structures under some typical configurations. As we have described before, CAM tag comparators account for a major energy dissipation source. It is also reported in [10] using circuit-level estimation that wakeup operation consumes ten times more energy than reading the RAM part of the IQ. That is why a tag-less WIB consumes much lower energy. We also observed that with the ability of two-level shelving and adaptive banking, the energy on the small, tag-reduced IQ continues to shrink superlinearly.

<table>
<thead>
<tr>
<th>Configurations (shown in different rows)</th>
<th>WIB and IQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-Entry IQ</td>
<td>232.4 nJ</td>
</tr>
<tr>
<td>32-Entry WIB, 96-Entry IQ</td>
<td>0.4 nJ</td>
</tr>
<tr>
<td>32-Entry WIB, Two 48-Entry IQ Banks</td>
<td>2.4 nJ/bank</td>
</tr>
<tr>
<td>32-Entry WIB, Four 24-Entry IQ Banks</td>
<td>0.4 nJ</td>
</tr>
<tr>
<td></td>
<td>1.1 nJ/bank</td>
</tr>
</tbody>
</table>

The max energy estimation corresponds to the peak power of the circuit, which does not take into account the runtime effects such as conditional clock and the processor usage by the workloads. We measure the total scheduler energy consumption for the given amount of work to evaluate the runtime energy reduction effects. The values of the total scheduler energy consumption are normalized to the baseline design. Combining the two-level shelving and adaptive banking achieves significant energy reductions.

As Fig. 11 shows, employing a 32-entry WIB alone reduces the scheduler energy by 36%, while further splitting the IQ into 2 and 4 banks reduces the energy by 57% and 67% respectively. By assuming that the instruction scheduler consumes 20% of the total processor energy, the above energy reductions can be approximately translated to 7%, 11% and 13% reduction of the total processor energy.
6.3 Comparison with Previous Work

Finally, we quantitatively compare our scheme against the work of [4], which is also aimed at reducing scheduler tags. Their scheme depends on a gshare-style predictor at front-end to predict the last-arriving operand tag of 2OP instructions, and speculatively dispatches them into the IQ with a single tag comparator. These instructions can be speculatively issued, and the prediction is validated at register read stage. If the prediction is validated to be wrong, the scheduler pipeline must be flushed and restarted. In addition, they employed a queue with no comparators to accommodate those ready instructions. We model the prediction-based scheme in detail, including detailed simulation of flushing instructions upon a mis-prediction and reexecution. For SMT workloads, only thread-specific instructions behind the mis-speculated instruction are flushed and refetched. We use the most aggressive configuration described in [4] in our simulation, which employed a 64-entry 0-tag queue and a 64-entry 1-tag queue. We use an 8192-entry PHT with 8-bit global history, which achieves a high prediction accuracy of 94% for single-thread workloads on average. For the SMT processor, each thread maintains a separate predictor to prevent table interferences between multiple threads. However, the predictor achieves only 82% prediction accuracy on SMT workloads. The reduced prediction accuracy is inherently caused by the interferences between multiple resource competing threads. For example, sharing IQ entries between multiple threads can easily change the timing of instruction issue within a certain thread, and therefore the value producing order. Therefore, SMT execution makes the arriving order of the operands within a thread hard to predict.

Fig.12 shows the performance comparison. For single-threaded workloads, the prediction-based scheme performs closely to our scheme, but it brings significant IPC loss on SMT workloads. Generally, the optimized design of prediction-based scheme degrades IPC by 4% for the simulated workloads. However, our scheme does not suffer mis-speculation penalties and performs similarly well on SMT workloads. Therefore, our scheme is superior in terms of SMT support.

Table 4. Max Energy per Access for Different Structures of the Most Aggressive Configuration in the Last-Tag Predictor Based Design.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>0-Tag Queue</th>
<th>1-Tag Queue</th>
<th>Last-Tag Predictor</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-Entry 0-Tag IQ</td>
<td>0.6</td>
<td>3.4</td>
<td>0.8</td>
</tr>
<tr>
<td>64-Entry 1-Tag IQ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4 shows the max energy per access estimation for the scheduler components used by the prediction-based scheme. We observed that the predictor table, although not on the scheduler timing path, still consumes considerable energy due to its large table size. Therefore, the predictor must be included for the energy evaluation. For SMT modes, each thread must maintain a separate predictor which account for additional energy consumption. The work of [4] only compares the energy consumption of the instruction scheduler on a per access basis, which precludes the energy consumption on the predictor table. In addition, they did not take into account the runtime effects such as conditional clock and the processor usage by the workloads.

The advantages of our scheme over the prediction-based scheme can only be drawn from runtime energy estimation. First, we characterize the conditional clock effects on the scheduler runtime energy. The result tag bus across the IQ entries can accommodate IW tags in a certain cycle. If there are fewer values produced, the load capacitances on the unused bus line are not activated (we used the cc3 conditional clock model, which is described in Subsection 3.1). In our scheme, the IQ is partitioned into smaller banks, and the result tag bus of each bank can accommodate IW tags to sustain the peak requirements. However, there are at most IW tags
to be broadcast every cycle, and these tags are distributed among the banks. In other words, each result tag selectively wakes up only one bank. Second, we characterize the last-tag mis-speculation on the scheduler runtime energy. Mis-speculations waste energy on the scheduler as well as other datapath components, since all the instructions behind the mis-speculated one have to be reexecuted. However, we only take into account the scheduler energy. As shown in Fig.13, our proposed architecture consistently achieves more runtime energy reductions on the instruction scheduler than the prediction-based scheme. For single-threaded workloads, our architecture achieves 7%-14% more energy reductions. For multi-threaded workloads, our architecture achieves 17%-24% more energy reductions. The prediction-based scheme is less energy-efficient on SMT workloads due to additional energy consumptions on multiple predictor tables for each thread and extra mis-speculation overhead due to degraded prediction accuracy.

7 Conclusion

The instruction scheduler has become one of the primary power and complexity bottlenecks towards large instruction window design to tolerate long memory latency or to support multiple threads. Although some energy-efficient scheduler design choices have been proposed, they are less applicable to the SMT processors, such as the dynamic adaptive IQ and the dependence-based IQ.

We have introduced more energy-efficient instruction scheduler design for both superscalar and SMT processors through two complementary techniques: two-level shelving and adaptive banking. These optimizations reduce both the number and the bit-width of tag comparators in the CAM logic while maintaining IPC that are close to a monolithic scheduler design. Compared with previous work on reducing tags through prediction, our architecture is superior in terms of both energy reduction and SMT support. In addition, some other optimizations can be lucratively combined with our architecture, such as circuit-level optimizations[3,15], the clustered architectures[9,16], the banked-select logic or the select-free logic[17].

There are still many ideas to be explored in reducing the energy and complexity of superscalar datapath components. Especially, the design choices which favor the single-thread workload behaviors may not be efficient for the SMT workloads. Our proposal made an effort towards energy-efficient datapath favoring both single- and multi-threaded processors.

Acknowledgement We thank all the reviewers for their valuable advices which help to improve this paper.

References


Yu-Lai Zhao received the B.Sc. degree in computer science from the Dept. of Computer Science and Technology of Peking University, in 2002. He is currently a Ph.D. candidate advised by Prof. Xu Cheng in the Microprocessor R&D Center of Peking University. His main research interests are in computer architecture, particularly in the optimizations of high-end microprocessors for energy efficiency. He has developed several simulation environments for microarchitecture exploration and HW/SW co-design purposes for the UNITY SoC project. He has also partitioned in the verification of the UNITY microprocessor.

Xian-Feng Li received his B.Sc. degree from Beijing Institute of Technology, in 1993 and Ph.D. degree from National University of Singapore, in 2005. From 1993 to 2000 he was an engineer at Norinco (G) Information Center. He is currently a post-doctoral researcher in the School of Information Science and Technology of Peking University. His research interests include micro-architecture, real-time systems, and System-on-Chip.

Dong Tong is currently an assistant professor in the School of Information Science and Technology of Peking University. His research interests include computer architecture, storage system, interconnection network, and System-on-Chip. He is the co-founder of the UNITY system architecture. He also led the design of the UNITY microprocessor, SoC chip and IP cores.

Xu Cheng is currently a professor and Ph.D. advisor in the School of Information Science and Technology of Peking University. His main research fields include high performance microprocessor, System-on-Chip, embedded system, instruction level parallelism, HW/SW co-design and compiler optimization. He is the founder of the Microprocessor R&D Center and the UNITY system architecture. He also led the design of the UNITY system software, the UNITY microprocessor, SoC chip, IP cores and network computer.