Using Lazy Instruction Prediction to Reduce Processor Wakeup Power Dissipation

by

Houman Homayoun
M.A.Sc., University of Victoria, 2005

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming
to the required standard

University of Victoria

ABSTRACT

Power dissipation has become an important consideration in processor design.

One way to reduce power dissipation is to revisit modern designs and to redesign them for power efficiency. In this work we introduce two power optimization techniques to address this issue. Our optimizations rely on identifying lazy instructions, i.e., instructions that spend long periods in the issue queue. Moreover, we investigate lazy instruction predictability and exploit it to reduce activity and power dissipation. We use our findings to: a) reduce wakeup activity and power dissipation in the issue queue and b) reduce the number of in-flight instructions and the average instruction issue delay in the processor.

Our study shows that, by using these optimizations, it is possible to reduce wakeup activity and power dissipation by up to 34% and 29% respectively. This comes with a performance cost of 1.5%. In addition, we reduce average instruction issue delay and the number of in-flight instructions by up to 8.5% and 7% respectively with no performance cost.

Examiners:

Table of Contents

List of Figures

Acknowledgment

I would like to thank my supervisor, Professor Amirali Baniasadi, for all his guidance, assistance and time.

I would also thank my parents for their constant support and giving me the chance of doing my graduate studies at University of Victoria.

1. Introduction

RISC processors started to appear in 1970s (most notably the CDC-6600 and Cray-1) when advances in semiconductor technology began to reduce the difference in speed between main memory and processor chip [1]. The first generation of RISC machines were very simple single-chip processors. By the time CMOS technology improved, more area became available on the chip. This enabled designers to improve processor performance based on two techniques; using on-chip caches and instruction pipelining.

As resources continued to grow, more execution units became available. To justify the existence of such a large number of execution units, it was no longer sufficient to simply try to fetch and decode a single instruction at a time. To benefit from the extra resources it was necessary to fetch, decode and execute multiple instructions as well. This idea in fact initiated a new step in evolution of processors; the superscalar generation. Such processors are scalar processors that are capable of executing more than one instruction in each cycle. The key to superscalar execution is the capability of fetch, decode and execute multiple instructions per cycle. The superscalar design uses instruction level parallelism (ILP) to speed up the execution process.

Conventional superscalar processors designs are built to respond to the worst case demand; a rare cases during execution of a program. In particular information is fetched, dispatched to the pipelines structure, moved and re-written redundantly among them. Power and energy are becoming a first consideration in designing new processors. We expect to see that the processor cycle time will be limited by thermal limitations. While every single block in a processor will be capable of running fast the processor itself will not be able to run at this high frequency because it can not afford to cool the generated power.

To address this issue, microarchitectures seek ways to reduce overall energy consumption and power dissipation by inventing new paradigms or making existing architecture more power- aware. In this work we take a step towards reducing the power dissipation of the superscalar processor based on the later approach. In particular we

attempt to reduce the power dissipation of a conventional instruction queue which is estimated to be responsible for around 27% of the overall processor power.

2. Microarchitecture of a Typical Superscalar Processor

Figure 1 shows the structure of a typical superscalar processor. Fetch unit reads instructions from the Instruction Cache (I-cache). Fetch unit is capable of fetching multiple instructions in the same cycle. Next, instructions are decoded and their logical register operands are renamed to physical register (more on this later). Renamed instructions are dispatched to instruction queue (IQ). This is the beginning of the out-of-order execution. Before this stage everything is done in program order. Once instructions pass this stage they can be processed out-of-order. Instruction queue basically is a pool of instructions waiting for their source operands to become available. Once operands become available, the instruction is sent to an appropriate free functional unit. Once the instruction result is produced, all instructions waiting for the produced operand are informed. This is done by broadcasting the operand tag to all entries in the IQ. We refer to this as wakeup stage.

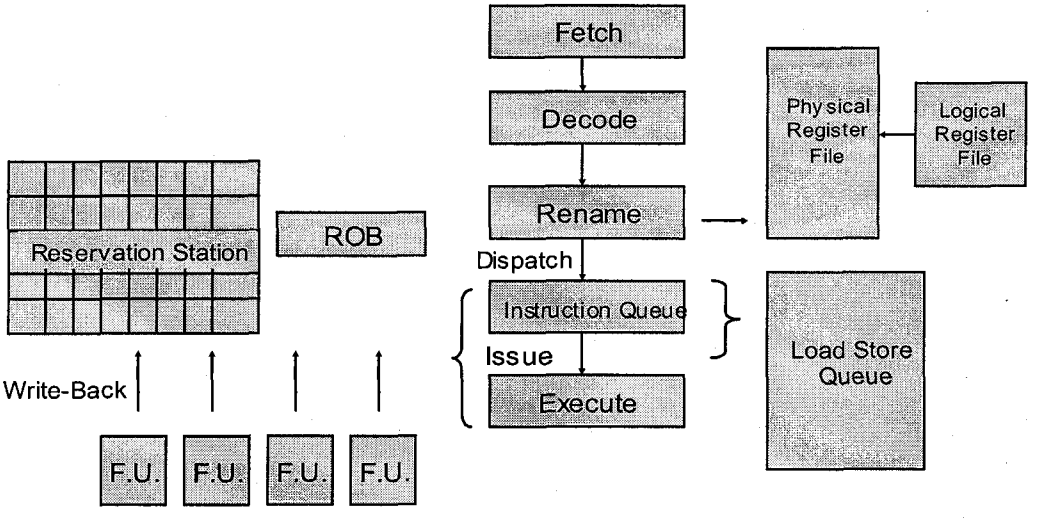


Figure 1: Typical superscalar architecture

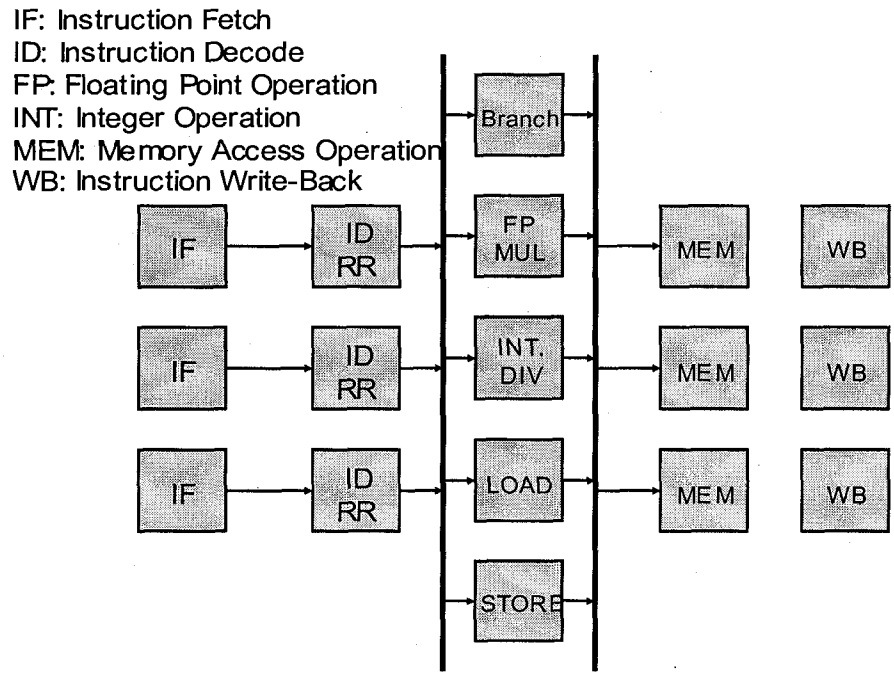


Figure 2: Snap-shot of superscalar Pipeline

As showed in figure 2 the major parts of the microarchitecture are as follows:

- Instruction fetch and branch prediction
- Instruction decode, register renaming and instruction dispatch stage
- Instruction queue; reservation stations and reorder buffer (ROB)
- Memory access stage
- Execution units and write-back stage
- Commit stage

In this section we study in more detail the hardware organization of each stage.

2.1 Instruction Fetch and Branch Prediction

In a superscalar processor, instruction fetch unit supplies instructions to the rest of the pipeline. To reduce instruction fetch latency processors use the instruction cache, which contains recently-used instructions. To supply the pipeline with enough instructions, superscalar processor should be able to fetch more than one instruction per cycle.

The basic instruction fetch process includes increasing the program counter (PC) by the number of instructions fetched and using the new PC to fetch the next block of instructions. This might not work for branch instructions which redirect the control flow. In this case the fetch stage must be redirected to fetch instructions from the branch target. As a result, processing the branch instructions in the fetch stage is somehow different from the rest of instructions. This process is done in the following steps:

- Identifying branch instructions
- Predicting the branch outcome (branch is taken or not taken)
- Computing branch target
- Transferring the flow of control to the branch target, if necessary

2.2 Instruction Decode, Register Renaming and Instruction Dispatch

In this stage, fetched instructions are decoded and their control and data dependences are detected for the register renaming pipeline phase. The register renaming stage is responsible for renaming the logical register (viewed by programmers) to physical register (viewed by processor). This process is necessary to increase the level of parallelism and to overcome artificial data dependences such as write-after-write (WAW) and write-after-read (WAR) hazards.

Due to the out-of-order nature of superscalar processor it is possible that multiple accesses happen to the same storage location. WAR hazard happens when an instruction wants to update the content of a storage location, but several instructions try to read the current value. WAW hazard occurs when several instructions wants to update the same

location. More number of physical registers than logical registers helps register renaming to overcome these kinds of data dependences. Figure 3 shows how register renaming can eliminate WAW and WAR hazards. In figure 3.a instruction 1 and 3 can not execute in parallel since they try to update same location (L3). By renaming logical register L3 to physical register P8 in instruction 3, now these two instructions can execute simulateously. In this case register renaming could solve WAW hazard. In figure 3.b instructions 1 and 2 can not execute in parallel since instruction 1 read the value of L1 and instruction 2 write into it. To eliminate WAR hazard, logical register L1 in instruction 2 is renamed to physical register P9. Overall in both scenarios more number of physical registers than logical registers eliminates data dependency hazards.

Finally decoded and renamed instructions are dispatched to instruction queue.

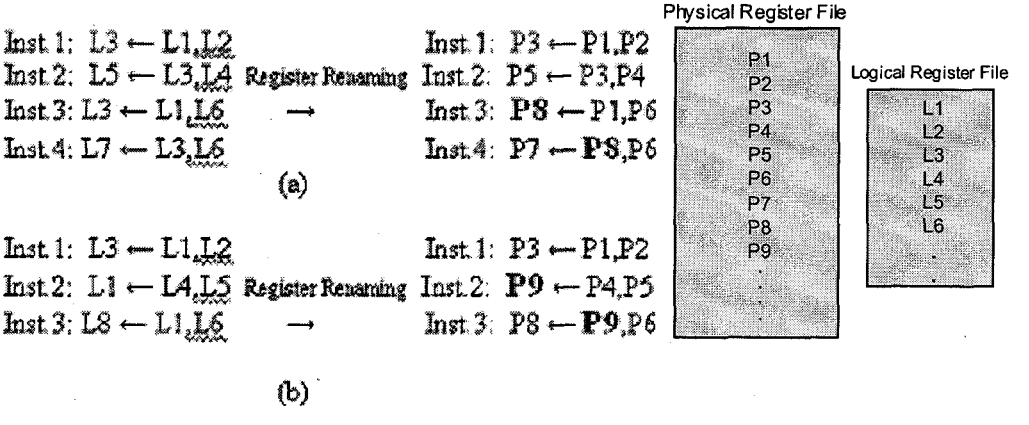


Figure 3: How register renaming eliminates a) WAW and b) WAR hazards

2.3 Instruction Queue; Reservation Station and Reorder Buffer

After instructions are decoded and their source operands are renamed they are dispatched to IQ. As explained earlier, prior to this stage everything is done in-order. After entering into the IQ, instructions can be executed regardless of their program order. The instructions reside in IQ until their source operands become ready. At this time they are issued to appropriate functional units.

Instruction queue constitutes of two major parts: reservation stations and reorder buffer. In traditional designs, reservation station keeps source operand value [2]. When an instruction is dispatched to IQ, available source operand values are read from instruction queue. Then each cycle reservation station compares functional unit result tags with its unavailable source operand tags. In the case of a match the result value is pulled into the appropriate entry. When all source operands become available, the instruction can enter the execution stage. Transferring operand value from register file to reservation station and from the reservation station to the functional unit is a redundant activity. To eliminate such redundancy, recent implementations of reservation stations only keep a pointer to where the actual operand value can be found [2]. In this case an operand value movement happens when instruction enters the execution stage. Reservation stations are mostly implemented as a CAM (Content-addressable memory) structure [4].

Reorder buffer basically keeps the original program order. This buffer maintains proper instruction ordering for precise interrupts which is caused by branch misprediction, system calls etc. In addition, in recent implementations, reorder buffer also serves in the register renaming process [2]. In such designs ROB is implemented as a circular buffer with head and tail pointers. When instructions are dispatched they are assigned an entry at the tail of the ROB. When they are executed, their results values are inserted into the corresponding entry.

2.4 Memory Access Stage

Unlike ALU instructions whose operands are identified during decode and renaming stage, load and store instruction operands can not be identified at decode. Identifying memory location requires an address calculation. Accordingly, load and store instruction need special treatment.

Such instructions are divided to two operations; address calculation and memory access. The address calculation part is dispatched to the instruction queue and the memory access part is dispatched to a separate queue referred to as the load store queue. When the address calculation operation is executed and the memory address resolved, the entry in load store queue is allowed to access memory. The complexity of load store queue is similar to the conventional reservation station [5]; it is designed as a set of CAM structures. Once an address is calculated, the correspondent tag is sent to all entries in load store queue to inform the dependent memory operation.

2.5 Execution and Write-back Stages

Instructions with all source operands ready are sent to the appropriate functional units. After the execution finishes the result is sent to instructions waiting in instruction queue. The result tag is sent to all entries in the reservation stations while its value is sent to correspondent entry in the reorder buffer. The reservation stations with TAGS similar to the result tag are marked as ready. Entries with both source operand tags marked as ready can be issued for execution. [more on this in chapter 3].

2.6 Commit Stage

This is the last stage of the pipeline. In this stage executed instructions are allowed to modify the logical process state. This stage keeps the in-order appearance of instruction stream. Due to the out-of-order execution this is necessary to recover from any kind of precise interrupt such as system calls and mispredicted branches.

3. Instruction Queue

The instruction queue of a superscalar processor is a complex structure which is dedicated to out-of-order execution. In any processor, only a limited number of instructions are allowed to issue/execute in a given cycle. This factor is referred to as Issue Width (IW). The hardware complexity of IQ depends on several microarchitectural factors including the size of instruction queue (IQsize) and IW [6]. Due to its high complexity, instruction queue is responsible for a significant amount of overall processor power dissipation [7].

There are four tasks involved in instruction queue stage [9]:

- Set an entry for a new dispatched instruction
- Read an entry to issue instructions to functional unit
- Wakeup instructions waiting in IQ once a result is produced by a functional unit
- Select instructions for issue when more ready instructions than issue width are available

The main complexity of the instruction queue stems from the associative search during the wakeup process [10]. During this stage produced results are by-passed from functional units to all entries in the instruction queue. Figure 4 shows the structure of wakeup logic. Tag drive lines are responsible to broadcast the results to all instructions waiting in IQ. Each instruction compares its operands tags with the broadcasted tags. If a match is detected the instruction source operand is marked as ready. Once all source operands of an entry are marked as ready (rdyL and rdyR flags) the instruction can enter the execution stage.

In each cycle as many as IW results might be broadcasted by functional units. Assuming an instruction has at most two source operands, there should be 2*IW number of comparators associated with an entry in the IQ to compare the results tags with the entry

source operands tags. Finally The OR logic which is responsible to OR the results of comparators sets the rdyL/rdyR flags.

Among all processor structures instruction queue is an extraordinary structure which is accessed during most pipeline stages; during dispatch to write an entry, during issue to read an entry, during write back to wakeup all entries and during commit to remove an entry. Because of these tasks instruction queue is one of the most power consuming parts of the processor. Ponomarev, et al estimated that more than 27% of the total power dissipation in a processor is dissipated in the instruction queue [8]. Moreover, Folegnani and Gonzalez have shown that wakeup is the most power consuming task among all four tasks involved which represent 63% of the total power dissipation of the IQ [9].

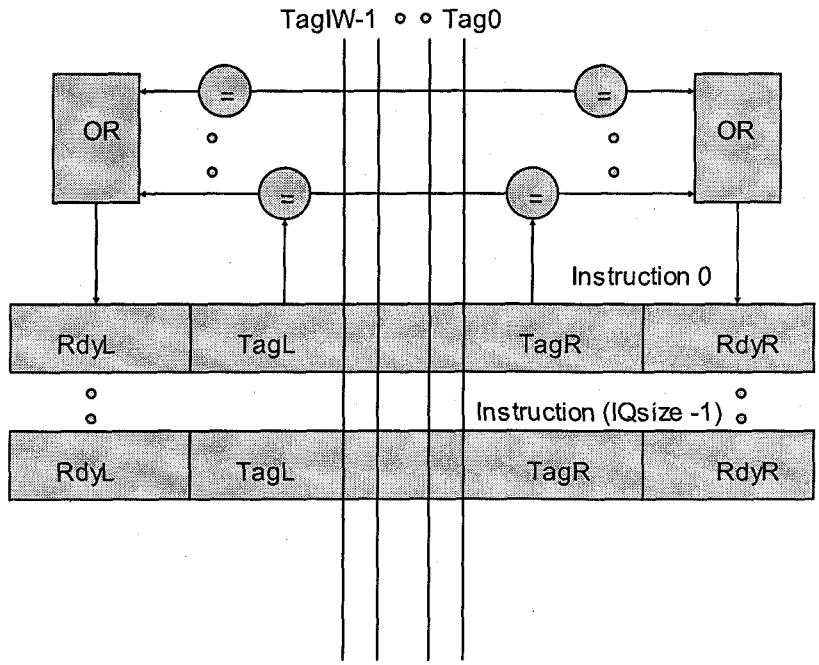


Figure 4: Conventional wakeup logic

In addition to high power dissipation, tasks involved in instruction queue (wakeup + select) have a high delay which affects the critical path delay [6 and 11]. In other words,

IQ delay is one of the major factors in impacting the clock cycle. Palacherla et al have shown that wakeup and select tasks together have the highest delay among all other structures in a four way superscalar processor and determine the clock frequency [6].

The associative search related to wakeup is the main source of its high power dissipation and delay. As discussed, in conventional IQ design a result is broadcast to all entries in the instruction queue. Past studies show that the majority of instructions wakeup at most one instruction in the IQ [11 and 12]. Accordingly waking up the rest of instruction queue entries is a redundant activity and a major source of power and energy dissipation.

4. Previous Work

Several approaches have been proposed to reduce the power dissipation of the associative search related to wakeup logic. Folegnani and Gonzalez [9] proposed a new scheme which avoids waking up empty entries in the instruction queue. Their approach also avoids waking up entries which already have been marked as ready.

Some other researchers proposed prescheduling instructions dynamically [13, 14, 15, 16 and 17]. These schemes attempt to schedule instructions in FIFO buffers using different heuristics. They wake- up only instructions in the head of each FIFO in each cycle. Accordingly the rest of instructions in each FIFO do not participate in wakeup activity. The heuristic should be chosen in a way that waking up only instructions in the FIFO head does not reduce the execution time.

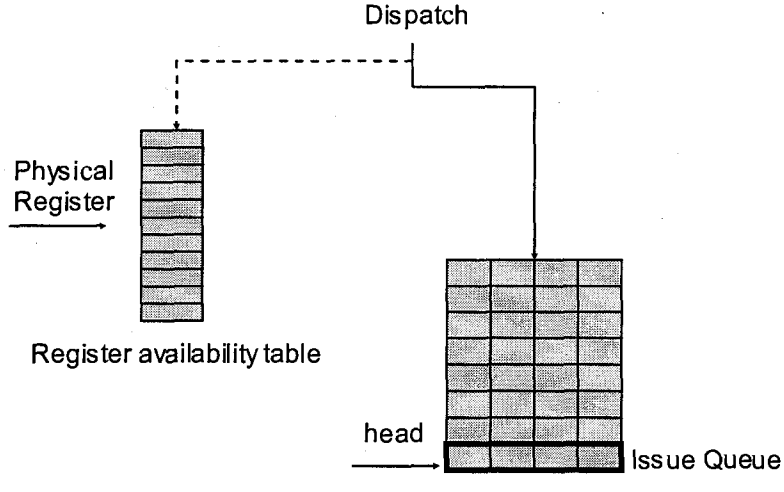


Figure 5: Base line latency based scheme[13,14]

Canal and Gonzalez [13 and 14] proposed a scheme, which schedules instructions based on their expected issue time. Figure 5 shows their base-line scheme. Head pointer points to instructions expected to be issued in the current cycle. In each cycle instructions at the head of each FIFOs are issued and the head pointer is increased by one. To schedule the

memory instructions which their issue time is non-deterministic during the decode stage they offer two solutions. In the first scheme [14] they schedule memory instructions based on an assumed memory access time. By doing so, such instructions are dispatched like other instructions. But if they found that they were scheduled too early they are sent to a small buffer which has the same hardware complexity as the conventional instruction queue; i.e. all entries participate in wakeup in each cycle. Figure 6 shows this scheme.

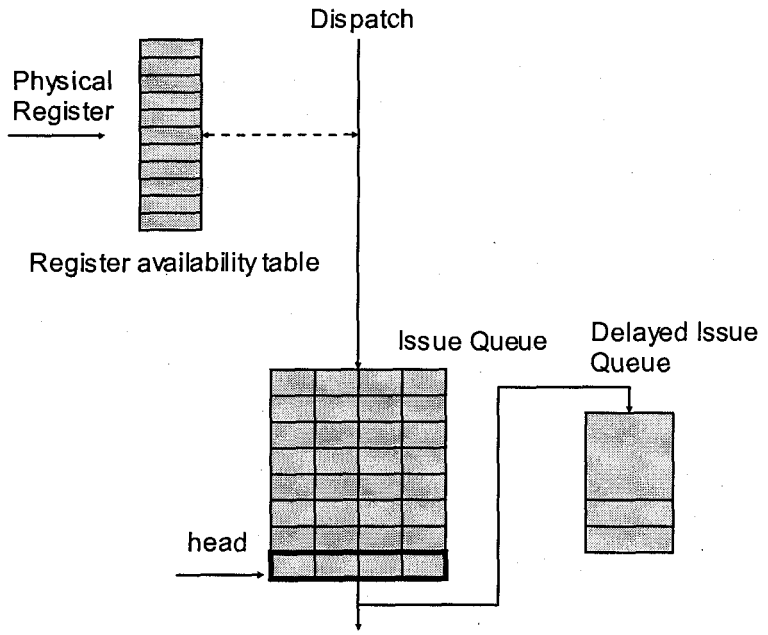


Figure 6: Deterministic latency scheme[14]

The second solution [13] is to dispatch such instructions to an extra hardware, called wait queue. These instructions will be kept in this queue as far as their availability time is not known. This buffer has the same structure as conventional instruction queue. Figure 7 shows this scheme which is referred to as distance latency scheme.

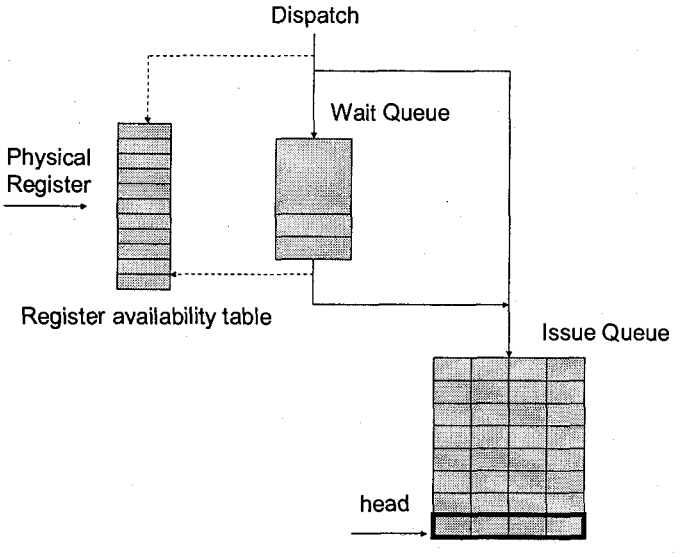


Figure 7: Distance Latency scheme[13]

A second group of techniques reduce the associtivity of instruction queue through linking the producer instruction to the consumer ones [6, 12, 13, 14, 18 and 19]. These schemes are referred to as dependence schemes. Most of these schemes are based on the observation that the majority of producer instructions have just one consumer.

Canal and Gonzalez have observed that in a typical instruction stream the majority of register values are read at most once by other instructions before being overwritten [14]. Based on this characteristic they modified instruction scheduling by using a physical register indexed table (Figure 8). The instructions are dispatched to entries corresponding to their source operands. Consequently when a result is produced, the associated tag is sent directly to its own entries in the N-use table instead of going to all entries as in the traditional issue logic. If there are not enough free entries in the N-use table the instructions are sent to a small buffer which has the same complexity as conventional instruction queue.

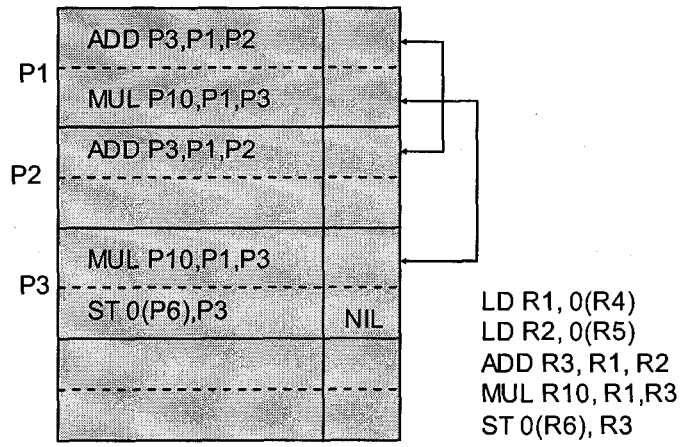


Figure 8: Physical register indexed instruction queue[14]

Huang et al propose a dependence scheme which links the producer to at most one consumer [12]. In the rare case of having more than one consumer, their scheme attempt to wakeup all entries in the instruction queue.

Another approach is to dynamically resizing the IQ [9, 20, 21, 22, 23 and 24]. Ponomarev et al, Dropsho et al and Buyuktosunoglu et al attempt to adjust instruction queue size based on the number of occupied entries [21, 22 and 24]. Folegnani and Gonzalez proposed resizing instruction queue based on the contribution of its younger part to IPC [9]. Abella and Gonzalez resize instruction queue based on the ratio of the time instructions spent in reservation station and reorder buffer [23].

Brown et al., introduced methods to remove the select logic from the critical path [30]. Brekelbaum et al., introduced a new scheduler, which exploits latency tolerant instructions in order to reduce implementation complexity [31]. Strak et al., used

"grandparent" availability time to speculate wakeup [32]. Ernst et al., suggested a wakeup free scheduler which relied on predicting the instruction issue latency [27]. Finally, Hu et al., studied wakeup-free schedulers such as that proposed in [27] and explored how design constrains result in performance loss and suggested a model to eliminate some of those constrains [28].

5. Energy Consumption Analysis of Conventional Instruction Queue

A conventional instruction queue is designed as a CAM structure. As discussed, among four tasks involved in Instruction queue, studies have shown that wakeup is the most power consuming.

Each time a result is produced its associated tag is propagated through all entries in the CAM cell (reservation station part). Then each entry matches its tag with the result tag. In the case of a match the correspondent entry is marked as ready. Figure 9 shows the structure of wakeup logic which is implemented as CAM cells[6]. To broadcast the instructions tags IW * (tag-size) tag lines run across the instruction queue. Along with tag lines, match lines run across the width of instruction queue. An entry's tag is compared with all tags carried by the tag lines. Accordingly, there are IW match lines per entry. Finally all match lines are OR-ed together to set the ready flag in the case of a match.

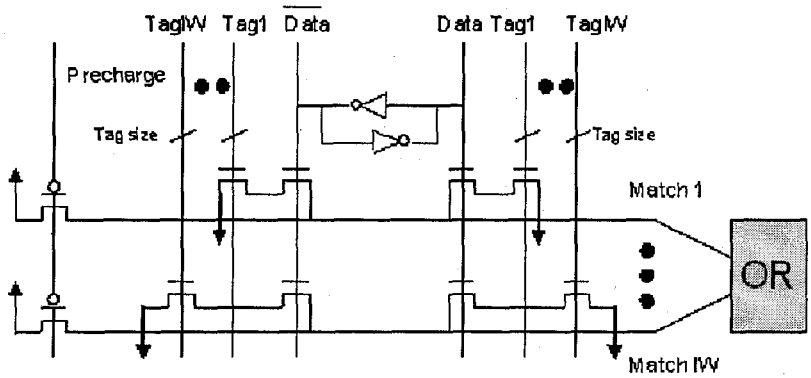


Figure 9: CAM cell wakeup logic[6]

As a result tasks in the wakeup process are categorized as [6 and 25]:

- Tag broadcast
- Tag match
- Match-line OR

And hence the wakeup activity energy is [6, 25 and 35]:

$$Energy_{\text{wakeup}} = Energy_{tag\text{-}broadcast} + Energy_{tag\text{-}match} + Energy_{match\text{-}OR}$$

5.1 Tag Broadcast Energy

This process includes driving the tag into the tag lines and writing the tag into each entry in the instruction queue. Accordingly tag broadcast energy can be expressed as [6, 25 and 35]:

$$Energy_{tag\text{-}broadcast} = Energy_{tag\text{-}drive} + Energy_{tag\text{-}write}$$

Tag Drive Energy is a function of total length of tag lines. Considering the instruction queue has IQsize entries and the processor issue width is IW [6, 25 and 35]:

$$Taglinelength = IQsize * (Cellheight + 2 * IW * Matchlinespacing)$$

Where Matchlinespacing is the spacing between the match lines and Cellheight is the height of a CAM cell.

There is a tag line per each bit of the tag and a tag has totally log (IQsize) bits. As a result energy of tag-drive is proportional to [6, 25 and 35]:

$$Energy_{tag\text{-}drive} \propto IQsize * IW * log(IQsize)$$

In the tag write process the bypassed result writes its associative tag to all entries in the instruction queue. This is done by raising the word-lines and bitlines and then writing the tag content to the bitlines [6, 25 and 35].

$$Energy_{tag\text{-}write} = Energy_{cam\ wordline} + Energy_{cam\ bitline}$$

$$Cam_{wordlinelength} = log\ (IQsize) * (Cellwidth + IW * Taglinespacing)$$

Taglinespacing is the space between the tag lines and cellwidth is the width of the CAM cell [6, 25 and 35]..

$$Cam_{bitlinelength} = IQsize * (Cellhigth + IW * Matchlinespacing)$$

Since we have IQsize bitlines and wordlines [6, 25 and 35]:

$$Energy_{cam\ wordline} \propto IQsize * log(IQsize) * IW$$

$$Energy_{cam\ bitline} \propto IQsize * log(IQsize) * IW$$

And therefore [6, 25 and 35]:

$$Energy_{tag\text{-}broadcast} \propto IQsize * log(IQsize) * IW$$

5.2 Tag match Energy

Tag match Energy is a function of match line length. The tag match energy consumption can be acquired the same way as wordline energy consumption [6, 25 and 35]:

$$Cam_{matchlinelength} = log(IQsize) * (CellWidth + IW * Taglinespacing)$$

$$Energy_{tag\text{-}match} \propto IQsize * log(IQsize) * IW$$

5.3 Match line OR Energy

To find if there was a match between the broadcasted tags and the cell tag, each cycle all matchlines are ORed. This energy is quite small compared to the tag broadcast and tag match energy [6, 25 and 35].

6. Using Lazy Instruction Prediction to Reduce Processor Wakeup Power Dissipation

Modern high-performance processors execute instructions aggressively, processing them in each pipeline stage as soon as possible. A typical processor fetches instructions from the memory, decodes them and dispatches them to the instruction queue. Instructions wait in the IQ for their operands to become available. The processor associates tags with each source operand and broadcasts operand tags to all instructions in the IQ every cycle. Instructions compare the tags broadcasted with the operand tags they are waiting for (wakeup stage). Once a match is detected, instructions are executed subject to resource availability (select stage). This aggressive approach appears to be inefficient due to the following:

1- In order to improve ILP, high-performance processors fetch as many instructions possible to maximize the number of in-flight instructions. High-performance processors continue fetching instructions even when there are already many in-flight instructions waiting for their operands. A negative consequence of this approach is that some instructions enter the pipeline too early and long before they can contribute to performance. Nevertheless, they consume resources and energy.

2- Many instructions tend to wait in the instruction queue for long periods. An example of such instructions is an instruction waiting for data being fetched from the memory. Under such circumstances, the waiting instruction and consequently those depending on its outcome have to wait in the IQ for several cycles. During this long period, however, the processor attempts to wakeup such instructions every cycle.

We exploit the two inefficiencies discussed above and use instruction behavior to address them. We study the time instructions stay in the instruction queue (also referred to as instruction issue delay or in brief as IID). In particular, we study lazy instructions, i.e., those instructions that spend long periods in the instruction queue. We attempt to identify/predict these instructions. By identifying lazy instructions we achieve the

following: First, by estimating the number of in-flight lazy instructions, we identify occasions when the front-end can be reconfigured to fetch fewer instructions without compromising performance. Second, once lazy instructions are identified speculatively, we reduce wakeup activity by avoiding to wakeup lazy instructions every cycle.

6.1 Simulation Tools

In this section we explain simulation tools that are used for this research. To measure performance and power we use Simplescalar [29] and WATTCH [25] respectively.

6.1.1 Simplescalar Tool Set

Simplescalar is an open source simulation tool that is written in C programming language and simulates a generic superscalar processor. For every stage of the processor pipeline an associated function is implemented. The program accepts a set of benchmarks as input as well as parameters to configure processor resources. (such as cache size, branch prediction table size and etc.). The benchmarks are in form of binaries. The generated output is a text file that gives information about a particular benchmark.

6.1.2 WATTCH

WATTCH is a tool that uses Simplescalar as its backbone to estimate the processor power dissipation. Every cycle access to all hardware resources are collected in Simplescalar environment and are sent to WATTCH. WATTCH then calculates the power based on the reported results. WATTCH is a very fast tool compare to other power simulation tool. It is also accurate compares to existing lay-out level industry tools.

6.2 Methodology

In this Section, we report our analysis framework. We detail the base processor which models an aggressive superscalar processor in Table 1. We used both floating point (equake and ammp) and integer (vpr, gcc, mcf, bzip2, parser and twlf) programs from the SPEC CPU2000 suite compiled for the MIPS-like PISA architecture used by the Simplescalar v3.0 simulation tool set. Table 2 shows the benchmarks input files. The

benchmarks studied here include different programs including high and low IPC and those limited by memory, branch misprediction, etc. We used GNU's gcc compiler with the –O3 compiler flag. We simulated 200M instructions after skipping 200M instructions.

We used WATTCH for energy estimation. We modeled an aggressive 2GHz superscalar microarchitecture manufactured under a 0.1 micron technology.

Table 1: Base processor configuration

Integer ALU	# 8	***Scheduler***	128 entries, RUU-like
FP ALU	# 8	***OOO Core***	any 8 instructions / cycle
Integer Multipliers/ Dividers	#4	***Fetch Unit***	Up to 8 instr./cycle. 64-Entry Fetch Buffer
FP Multipliers/ Dividers	#4	***L1 - Instruction Caches***	64K, 4-way SA, 32-byte blocks, 3 cycle hit latency
Instruction Fetch Queue	#64	***L1 - Data Caches***	32K, 2-way SA, 32-byte blocks, 3 cycle hit latency
Branch Predictor	2K GShare, bimodal w/selector	***Unified L2***	256K, 4-way SA, 64-byte blocks,16-cycle hit
Load/Store Queue Size	64	***Main Memory***	Infinite, 80 cycles
Reorder Buffer Size	128	***Memory Port***	#4

We modified Simplescalar to gather statistics. We modified functions which models instruction queue, wakeup stage (writeback function), dispatch stage and issue stage. We add one bit to each reservation station entry to indicate lazy instruction. We also modified

Table 2: Benchmarks input file

Benchmark	Input File
mcf	inp.in
vpr	net.in, arch.in, place.in
ammp	inp.in
gcc	cccp.i
equack	inp.in
Bzip2	input.random
Parser	2.1.dict, ref.in
twolf	ref

the recovery function (to recover from a branch misprediction) as the added bit in the reservation station needs to be flushed in the "recovering branch misprediction" state.

6.3 Lazy Instruction Characteristics

Many studies show that the behavior of an instruction in the instruction queue is predictable [e.g. 13 and 26]. In this section we study lazy instruction characteristics in the instruction queue. Through this study we define lazy instructions as those spending more than 10 cycles in the instruction queue. We picked this threshold after testing many alternatives. (More on this in section 6.7.2). There are many factors cause laziness such as data cache miss, TLB miss and instruction dependency. We take into account all these factors by defining the laziness based on the time instructions spend in the instruction queue.

Figure 10 shows IID distribution for a subset of SPEC'2K benchmarks. On average, about 18% of the instructions are lazy instructions, i.e., they spend at least 10 cycles in the IQ (maximum of 32%).

We refer to the number of times an instruction receives operand tags and compares them to its operand tags as the instruction wakeup activity. Lazy instructions, while accounting

for about 18% of the total number of instructions, impact wakeup activity considerably. This is due to the fact that they receive and compare the operands tags very frequently and during long periods. To explain this better in Figure 11 we report the relative share of total wakeup activity for each group of instructions presented in Figure 10. On average, lazy instructions, despite their relatively low frequency, account for more than 85% of the total wakeup activity.

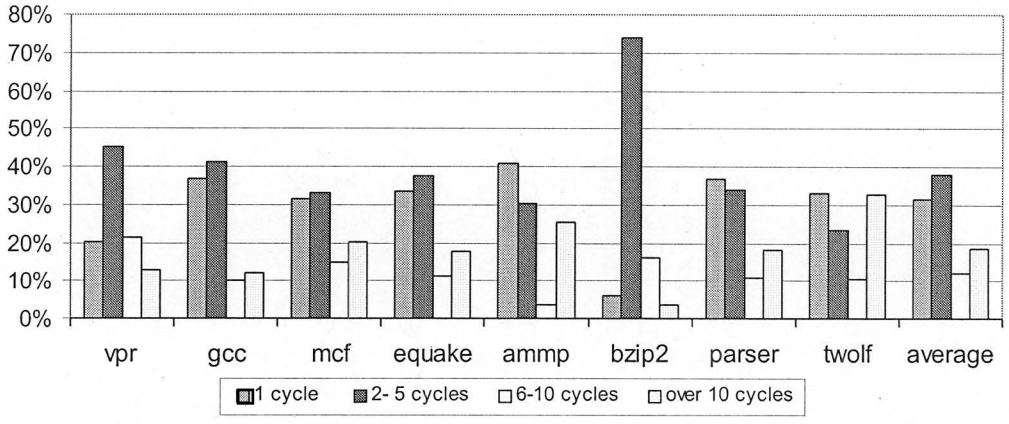


Figure 10: Instruction issue delay distribution

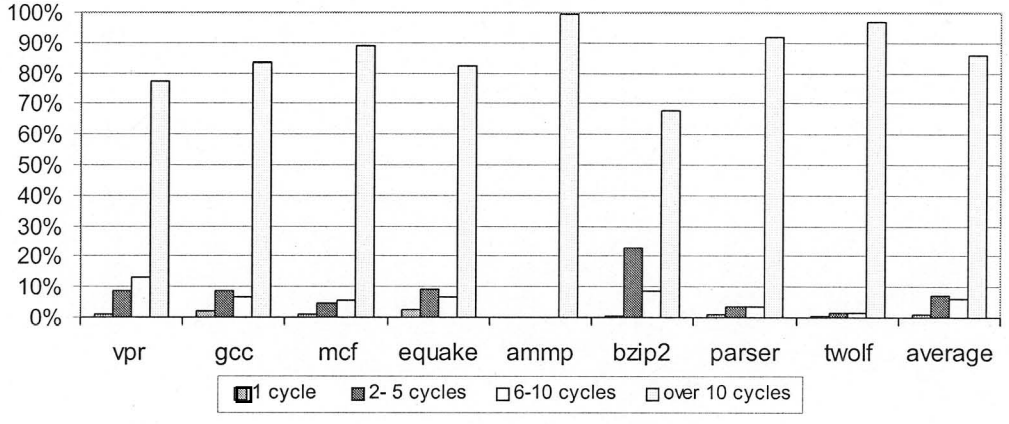


Figure 11: Instruction wakeup activity distribution

6.4 Lazy Instruction Repetition

In this section, we study lazy instruction repeatability; i.e. how often a committed lazy instruction repeats its behavior in the instruction queue.

To keep track of lazy instructions, we use a PC-index table. To measure an upper bound for lazy instruction predictability we choose an infinite-size table. Later in this work, we will use a limited-size table to account for practical restrictions. We refer to this table as the LI-table

We use LI-table to track lazy instructions history. If IID for an instruction is more than 10 we store the instruction PC in LI-table. Otherwise, for instructions already stored in the LI-table, we remove them from the table if IID is less than 10.

In Figure 12 We report how often a lazy instruction appears to be lazy next time encountered. We refer to this value as Lazy instruction repeatability (LIR).

As shown floating point benchmarks (ammp and equake) have higher LIR compared to integer benchmarks (mcf, gcc, , bzip2 and vpr, parser and twolf). On average, LIR is as high as 86%.

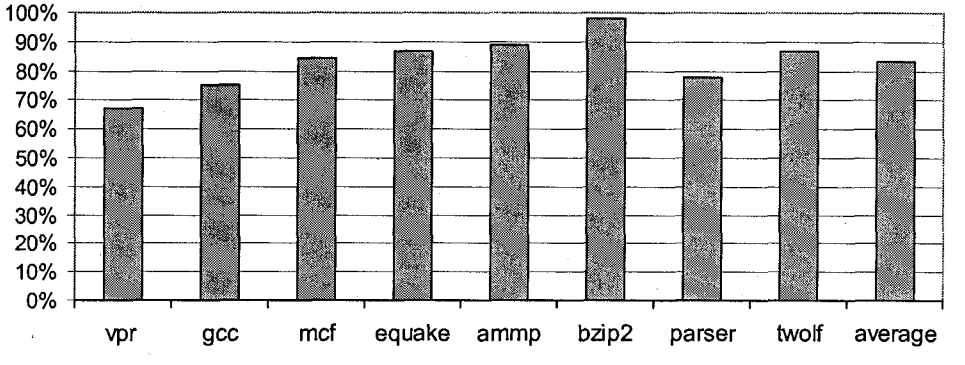


Figure 12: Lazy Instruction Repeatability

We have observed that even when the LI-table size is reduced to 64 average LIR does not change significantly. Exploiting smaller LI-table size results in evicting previously committed lazy instructions to allocate space for recently committed instructions more

often. The fact that a small table can accurately predict IID indicates that lazy instruction have strong temporal locality. In other words, over short periods of time, a small subset of lazy instruction is executed repeatedly.

6.5 Lazy Instruction Prediction

As explained earlier lazy instructions, despite their relatively low frequency, account for more than 85% of the total wakeup activity. To address this inefficiency it is important to identify lazy instructions early enough. Previous section illustrated that lazy instructions are highly predictable. In this section we use a small 64-entry, PC-indexed table to predict such instructions before they arrive in the instruction queue. While exploiting larger and more complex structures may improve prediction accuracy, we avoid such structures to maintain power and latency overhead at a low level.

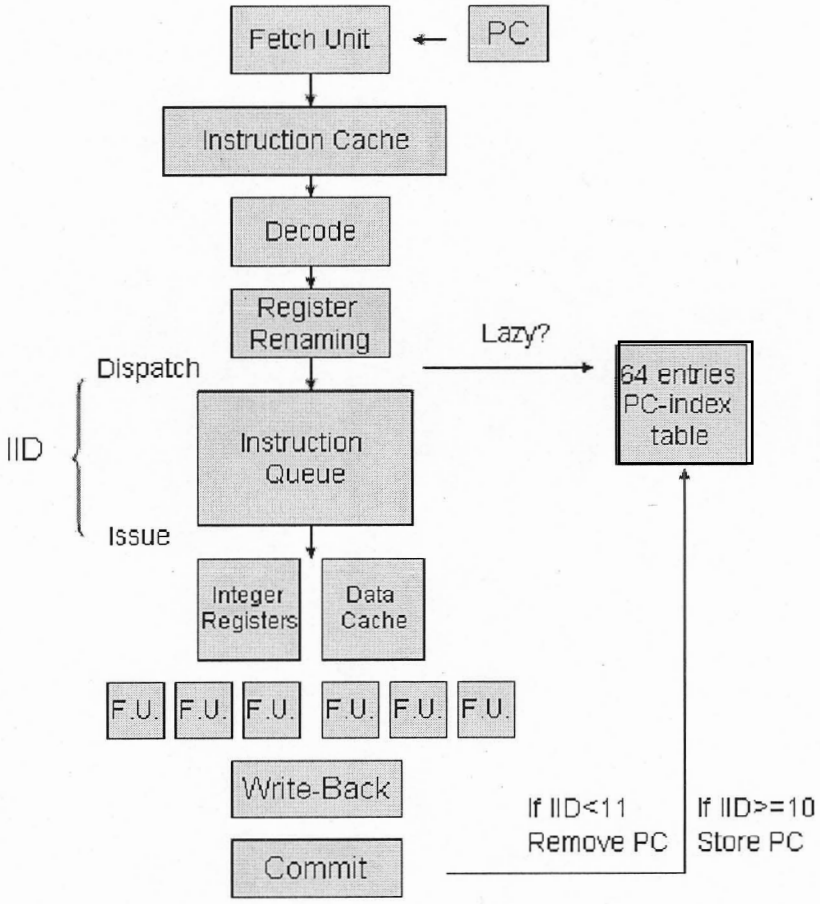
To predict lazy instructions we do the following: If IID is more than 10, we store the instruction PC in the LI-table. We also associate each table entry with a 2-bit saturating counter. If the lazy instruction is already in the table we increment the corresponding saturating counter. For the non-lazy instructions with an entry in the table, we remove the corresponding entry. To predict whether an instruction is lazy, we probe the LI-table at dispatch. The instruction is marked as lazy, if the corresponding counter is more than two. Figure 13(a) shows the configuration we propose. Figure 13(b) shows the pseudo code to identify lazy instruction.

We evaluate the proposed prediction scheme using two criteria, i.e., prediction accuracy and prediction effectiveness.

Lazy instruction prediction accuracy reports how often instructions predicted to have an issue delay more than 10 turn out to stay in the instruction queue for more than 10 cycles. This, while important, does not provide enough information as it is silent regarding the percentage of lazy instructions identified. Therefore, we also report prediction effectiveness, i.e., the percentage of lazy instructions identified.

While lazy instructions are identified by probing the LI-table at dispatch, the table can be updated at different stages. Two obvious update scenarios are commit-update and issue-update. We report prediction accuracy and effectiveness for both update scenarios.

In the first scenario, commit-update, lazy instructions are allowed to update the LI- table only after they have committed. Under this scenario wrong path instructions will not update the table.



(a)

```
Dispatch()
 {
        ....

        if (instruction PC is in the table && related saturating counter>=2)
            instruction_behaviour = predicted_lazy
        instruction_dispatch_time = dispatch_cycle
        ....
 }

Issue()
 {
        ....
        If (instruction_issued = true)
           instruction_issue_time = issue_cycle
        ....
 }

commit()
 {
        ....
        instruction_issue_delay = instruction_issue_time –
                                  Instruction_dispatch_time
        if (instruction_issue_delay >=10 )
           {
             save instruction PC in the LI Table
             if (instruction_behaviour = = predicted_lazy)
                 lazy instruction was correctly predicted
           }

        else
          {
              if (instruction_behaviour = = predicted_lazy)
                instruction was incorrectly predicted to be lazy
              if (instruction PC is in the table)
                  remove instruction PC
          }
        ....

   }
```

(b)

Figure 13: (a) Superscalar pipeline with logic to predict lazy instructions

(b) Pseudo code to identify lazy instruction.

Note that lazy instructions spend a long period in the pipeline and therefore update the LI-table long after they have entered the pipeline. As such, by the time a lazy instruction has committed, many lazy instructions have entered the pipeline without being identified. Also, it is quite possible that during this long period, the instruction behavior may change and therefore the stored information may not be valid by the time it becomes available.

The second scenario, issue-update, allows lazy instructions to update the LI-table as soon as they issue. This while making faster update possible, allows wrong path instructions to interfere.

6.5.1 Prediction Accuracy

In Figure 14 we report prediction accuracy. Bars from left to right report for commit- and issue-update. On average, prediction accuracy is 52% and 54% for commit- update and issue-update respectively. Ammp has the highest accuracy (97%) while bzip2 and vpr fall behind other benchmarks. Our study shows that lazy instructions change their behavior frequently for these two benchmarks. This is consistent with the fact that bzip2 and vpr have lower number of lazy instructions and lazy instruction activity compared to other benchmarks (see Figure 10 and 11). One might argue that it is possible to achieve 50% prediction accuracy for lazy instructions with coin flipping but considering the weight of lazy instruction in the total number of instructions (18%) it is not possible to achieve such accuracy (50%).

6.5.2 Prediction Effectiveness

In Figure 15 we report prediction effectiveness. On average, effectiveness is a bout 30%. Maximum effectiveness is achieved for gcc where we accurately identify more than half of the lazy instructions. Minimum effectiveness is achieved for vpr, where about 10% of lazy instructions are identified.

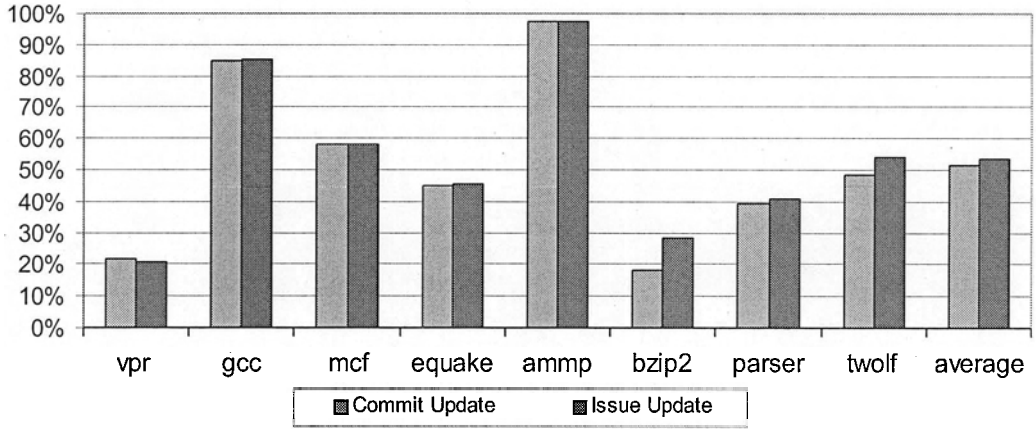


Figure 14: Lazy instruction prediction accuracy

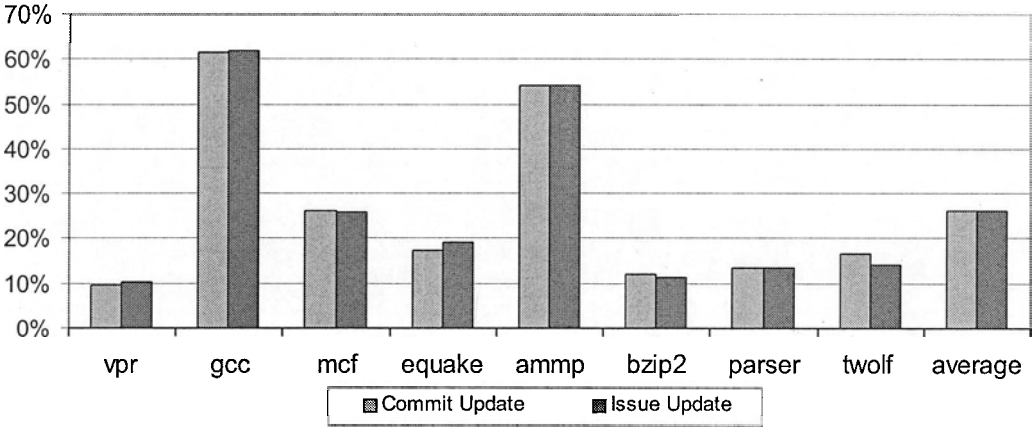


Figure 15: Lazy instruction prediction effectiveness

6.6 Optimization Based on Lazy Instruction Prediction

In this section we introduce two optimization techniques which use information available from a commit-update lazy instruction predictor. The two techniques are selective instruction wakeup and selective fetch slowdown. Selective instruction wakeup avoids waking up all instructions every cycle. Selective fetch slowdown reduces fetch speed if the number of lazy instructions in the pipeline exceeds a threshold. While the first technique impacts wakeup activity, the second one impacts more than one pipeline stage.

6.6.1 Selective Instruction Wakeup

As explained earlier, modern processors attempt to wakeup all instructions in the instruction queue every cycle. As a result, instructions receive their source operands at the earliest possible. This consequently improves performance. However, it is unnecessary to wakeup lazy instructions as aggressively as other instructions.

Ideally, if we had an oracle and knew in advance when an instruction will issue, then a heuristic for selectively waking up lazy instructions would require waking up the lazy instruction only at the time it is supposed to issue. Of course, we cannot have such an oracle. An alternative is to predict instruction latency and consequently the time instruction is issued [27] and restrict instruction wakeup to the predicted time. However,

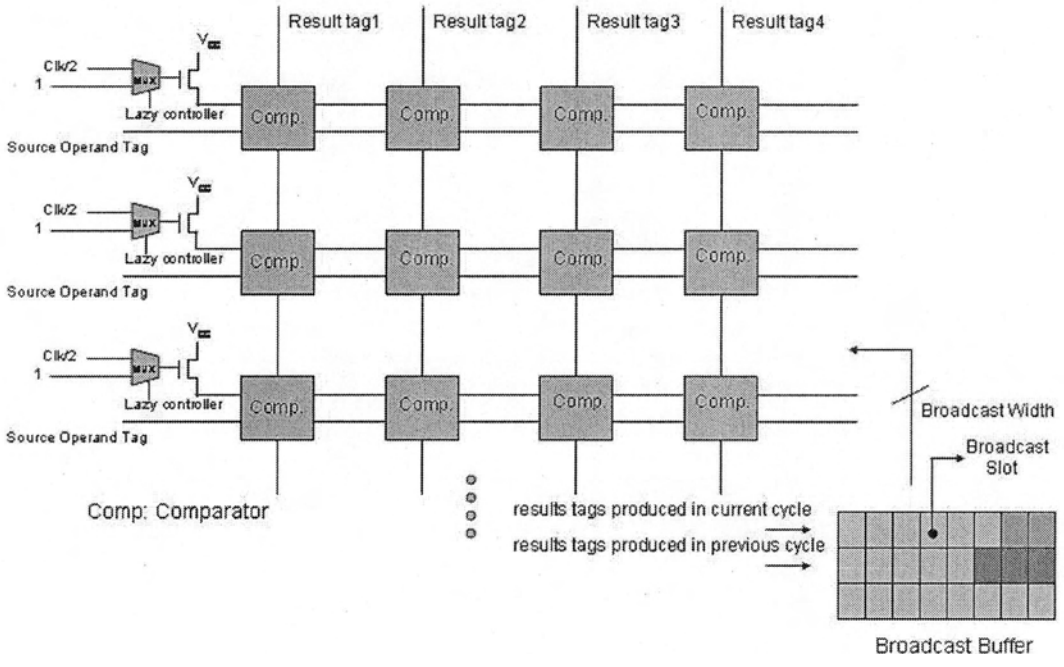


Figure 16: Hardware structure for selective wakeup

this will impose inherent limitations on performance by inaccuracies [28]. To avoid such complexities, we take a more conservative approach: Once we have predicted an instruction as a lazy instruction, instead of attempting to wake it up every cycle, we wake it up every two cycles. To model this, we have modified wakeup (writeback function) and issue stages in the Simplescalar. The hardware structure for selective wakeup is shown in figure 16. We add a multiplexer per instruction queue entry to power gate the comparators every two cycles. As we wakeup lazy instruction in even cycles, we need to save the result tags produced in the odd cycles. As such, we require some registers to keep the result tags and broadcast them to lazy instructions when free broadcast slot is available. Since the average broadcast width is one, in each cycle we can broadcast the result tags produced in the previous cycles along with the tags produced in the current cycle. In the rare case that all registers are full we can stall issuing instruction until empty entries become available. As the average broadcast width is one, it is very rare that all registers become full.

6.6.2 Selective Fetch Slowdown

Modern processors rely on aggressive instruction fetch to maintain ILP. Instruction fetch is responsible for supplying the rest of the processor pipeline with instructions. It is logical that the instruction fetch rate should at least match the instruction decode and execution rate otherwise the processor resources will be underutilized. Note that, if the instruction flow in the pipeline is too slow, it will be inefficient to fetch too many instructions. For example, if there are already many instructions waiting for their operands in the pipeline, we may be able to delay adding more instructions to the already high number of in-flight instructions without losing performance. This will reduce the number of in-flight instructions which in turn will result in less pressure on reservation stations and pipeline activity.

In this work we use our estimation of the number of in-flight lazy instructions to decide whether fetching instructions at the maximum rate is worthwhile. If the number of lazy instructions exceeds a dynamically decided threshold we assume that it is safe to

slowdown instruction fetch. Accordingly, we reduce the maximum cache lines fetched from two to one. We refer to this technique as selective fetch slow down (or front-end slow down). To model this, we have modified fetch stage.

To decide the dynamic threshold we record the number of instructions predicted to be lazy every 1024 cycles. If the number of lazy instructions exceeds one third of total number of in-flight instructions we reduce the threshold by 5. If the number of lazy instructions drops below 5% of the total number of in-flight instructions we increase the threshold by 5. Initially, we set this threshold to 15.

6.7 Results

In this section we report our simulation results. To evaluate our techniques we report performance, wakeup activity, average issue delay, average number of in-flight instructions, power dissipation and how often we slowdown fetch. We compare our processor with a conventional processor that attempts to wakeup all instructions every cycle and does not reduce the fetch rate. Note that activity measurements are less technology- and implementation-dependent compared to power measurements. Nonetheless, we also report power analysis for the processor studied here. We detail the base processor model in Table 1.

In 6.7.1 we report performance. In 6.7.2 we report activity and power measurements. In 6.7.3 we report average issue delay reduction and fetch slowdown frequency.

6.7.1 Performance

We measure performance in average number of committed instruction per cycle.(also referred to as IPC). In Figure 17 we report how selective wakeup and selective fetch slowdown impact performance. To provide better insight we also report performance for a processor that never fetches more than one cache line (referred to as the single line processor). In Figure 17 bars from left to right report performance for selective wakeup,

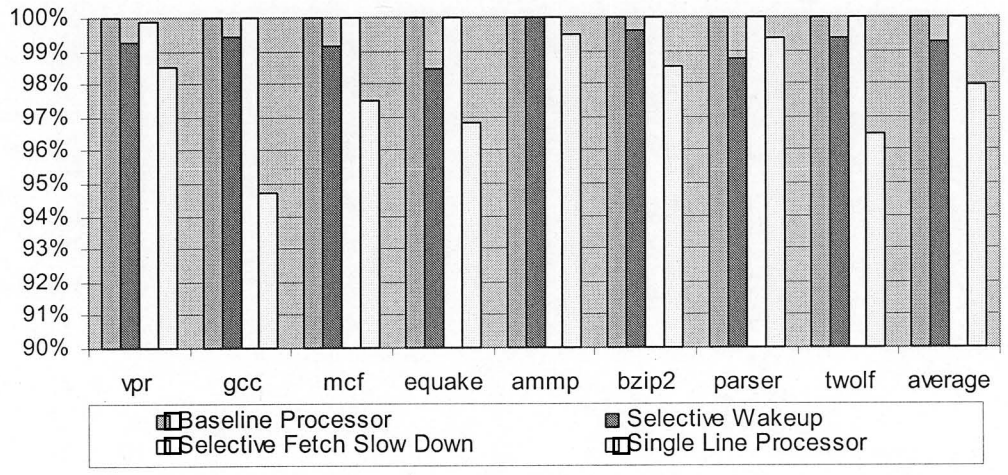


Figure 17: performance for selective instruction wakeup, selective fetch slowdown and single line processor respectively.

selective fetch slowdown and the single line processor. Across all benchmarks performance cost is below 1.5% for selective wakeup. Selective fetch slowdown, however, does not impact performance. On the other hand, the single line processor comes with a maximum performance cost of 5.5%. This shows the importance of smart fetch slow down which is achieved by using lazy instruction prediction.

6.7.2 Activity and Power

In this section we report activity and power measurements.

In Figure 18 we report how selective instruction wakeup impacts wakeup activity. On average, we reduce wakeup activity by 12% reaching a maximum of 34% for ammp.

In Figure 19 we report average reduction in the number of in-flight instructions for selective fetch slowdown and the single line processor. Selective fetch slowdown reduces the average number of in-flight instructions by 4% (maximum 7%) without compromising performance (see figure 17). The single line machine reduces average number of in-flight instructions by 8.5% (maximum 16%), however, this can be as costly as 5.5% performance loss as presented earlier.

In Figure 20 we report wakeup power reduction as measured by WATTCH. Bars from left to right report power reduction for selective wakeup, selective fetch slowdown and the combination of both techniques.

Selective wakeup reduces wakeup power dissipation up to a maximum of 29% (for ammp). Note that this is consistent with Figure 18 where ammp has the highest activity reduction. Minimum wakeup energy reduction is about 2% for bzip2. Again this is consistent with Figure 18 where bzip2 has the lowest activity reduction.

Selective fetch slowdown reduces wakeup power up to a maximum of 12% (for equake) and a minimum of 1% (for bzip2 and ammp). This is consistent with Figure 19 where equake hake has the highest reduction in the number of in-flight instructions and bzip2 and ammp have the lowest.

Using both techniques simultaneously, on average, we reduce wakeup power by about 14%.

Average wakeup power reduction is 8.3% and 6.7% for selective wakeup and selective fetch slowdown respectively.

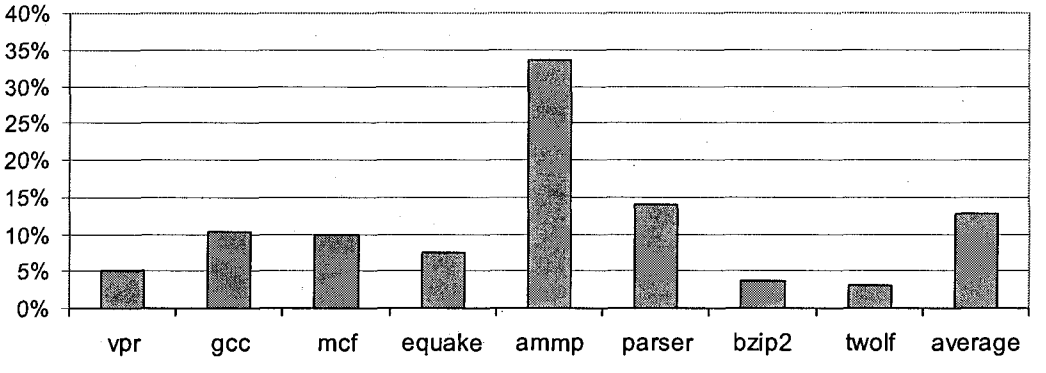


Figure 18: Selective wakeup: activity reduction

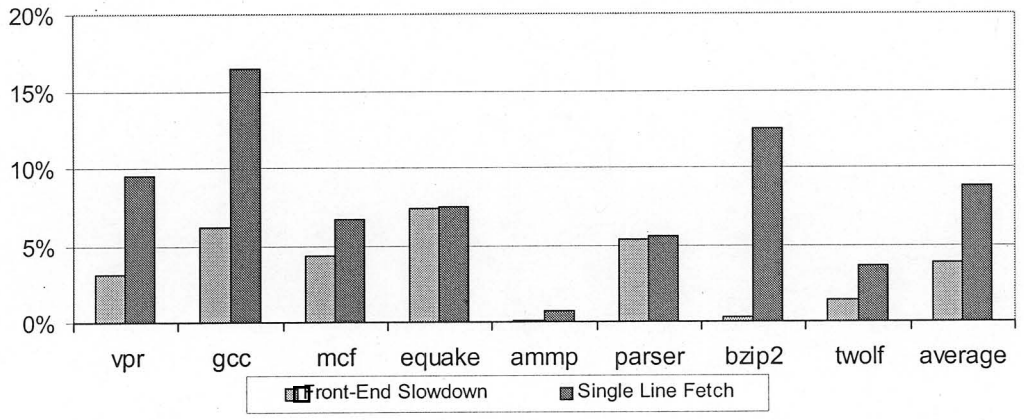


Figure 19: Selective fetch slowdown: average in-flight instruction reduction

Using both techniques simultaneously, on average, we reduce wakeup power by about 14%.

Average wakeup power reduction is 8.3% and 6.7% for selective wakeup and selective fetch slowdown respectively.

Recalling from section 6.3, we choose 10 cycles for lazy instruction threshold. Increasing the threshold reduces the power savings. On the other hand, decreasing the threshold

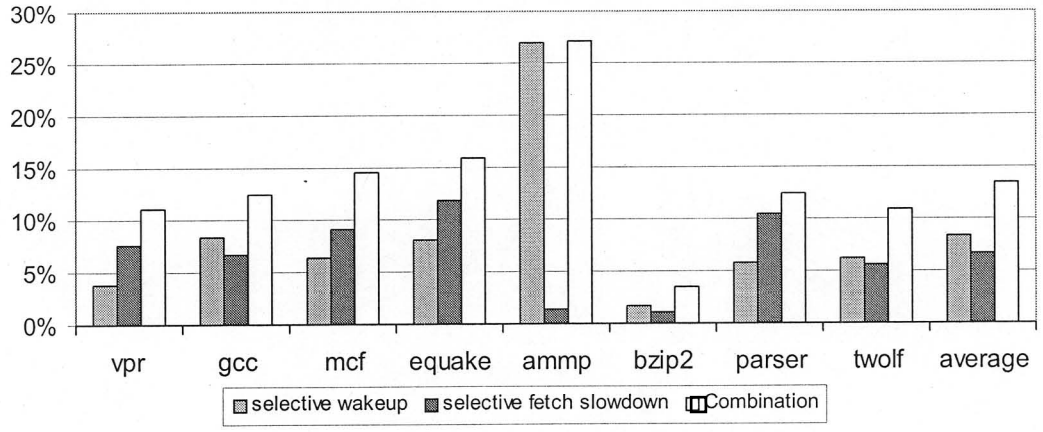


Figure 20: Wakeup power reduction.

increases performance degradation. 10 cycles threshold results in considerable power savings and at the same time negligible performance reduction.

6.7.3 Issue Delay and Slowdown Rate

In Figure 21 we report average reduction in IID achieved by selective fetch slowdown. On average, we reduce IID by 4% (maximum 8%) compare to the baseline processor.

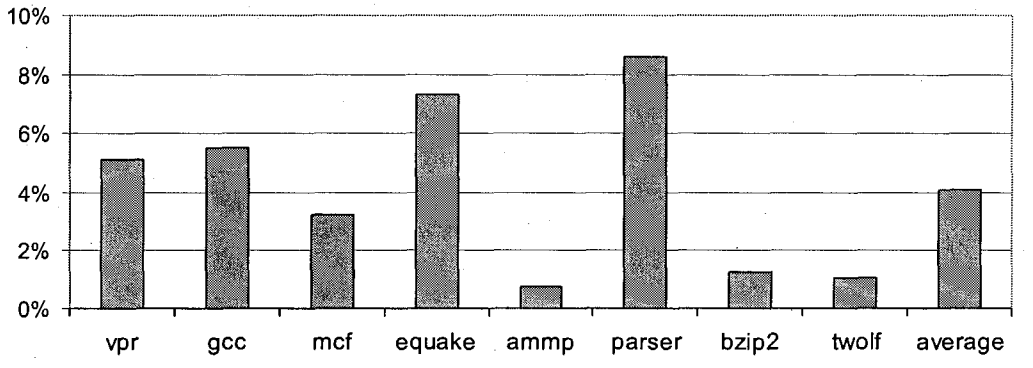


Figure 21: Selective fetch slowdown: average issue delay reduction

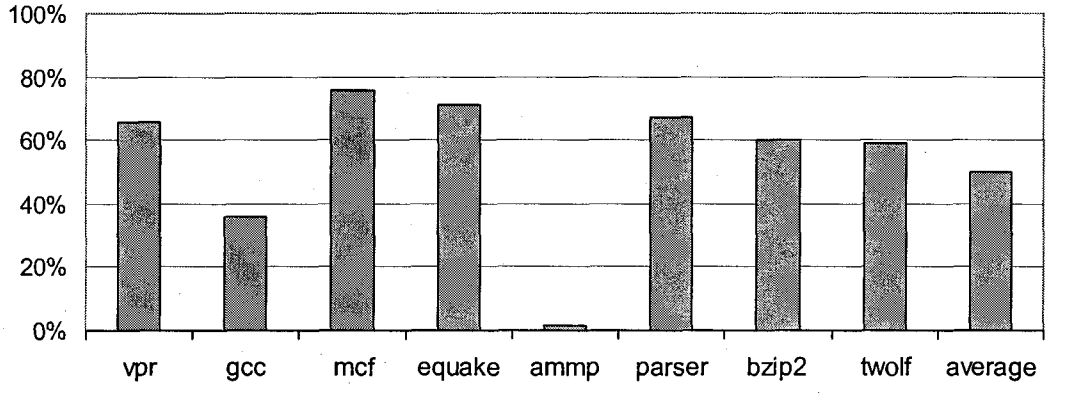


Figure 22: Selective fetch slowdown: slowdown rate

Finally, in Figure 22 we report how often selective fetch slowdown reduces fetch rate. On average we reduce fetch rate about 50% of the time. Note that for ammp we rarely reduce the fetch rate. This explains why we do not witness any reduction in average number of in- flight instructions or IID for ammp as reported in Figure 19.

6.8 Discussion

As discussed, lazy instruction prediction requires a 64-entry PC-index table. Waking up lazy instructions every two cycles also requires some modifications in the conventional instruction queue and broadcast buffer (figure 16). Selective fetch slow down needs least hardware modification as it only requires two counters and a register to keep number of lazy instructions, all instructions and the dynamic lazy threshold respectively. We will refer to all these extra hardware logics as overhead since they increase the total chip power dissipation.

To provide better insight on lazy instruction prediction hardware complexity we compare the LI-table with the typical branch predictor logic structure as they both can be modeled as a SRAM based structure. We compare the number of one-bit cells in typical branch prediction logic with number of one-bit cells in the 64 entry LI-table. As Parikh and Skadron have shown, [34] the branch predictor, including BTB, dissipates 7% to 10% of total chip power dissipation. We assume a typical 8K predictor with 512-entry BTB. Assuming the PC has 32 bits the total number of bits in branch prediction logic is 48K. The total number of bits in the 64-entry LI-table with two bit saturating counter is 64*32 + 64*2 or 2.1K. Consequently the area and power consumption of our structure might be as small as 1/23 of branch predictor logic or less than 0.3% of total chip power dissipation which is negligible.

To provide better insight for power dissipation of the modified instruction queue which enables selective wakeup, we compare the number of added transistors with number of transistors used in comparators. A typical 128-entry instruction queue in an 8 width superscalar processor has 2 * 128 * 8 * log (128) one-bit comparators. Assuming that we can build a single bit comparator with 3 transistors, we have 42k transistors for all 14k comparators in the instruction queue. On the other side, assuming that we can build a multiplexer with two transistors, the extra hardware cost is 384 transistors which is negligible compare to 42k transistors used in comparators.

As shown in figure 16, we need some registers to save the produced results tag and broadcast them to all lazy instructions next cycles. Theoretically we might have a case that for long consecutive cycles processor produces as many as issue width number of tags. To respond to such case we require infinite number of registers to save the results produced in the previous cycles and broadcast them when a broadcast slot become available. To keep the power overhead of this structure in a low level, we can use few numbers of registers and stall the issue logic when all registers become full. This doesn't impact the performance as it is a very rare case during execution of a program (considering that the average broadcast width of typical programs is one).

The disadvantage of selective wakeup and selective fetch slowdown might be the complexity of their redesign and verification. To meet market deadline, manufacturers mostly prefer straightforward and simple solution rather than a complex and more effective one.

In this section we have discussed the hardware cost of our proposed optimization techniques to reduce instruction queue power dissipation which is based on lazy instruction prediction. It is also possible to use lazy instruction prediction outcome to reduce power in other processor structures such as ROB, register file, etc. The more places we can use lazy instruction prediction, the better we can amortizes its extra power dissipation cost.

7. Conclusion and Future Work

In this work we studied lazy instructions and introduced two related optimization techniques. We showed that it is possible to identify a considerable fraction of lazy instructions by using a small and simple 64-entry predictor. By predicting and estimating the number of lazy instructions we reduced wakeup activity, wakeup power dissipation, average instruction issue delay and average number of in-flight instructions while maintaining performance. We relied on limiting instruction wakeup for lazy instructions to even cycles and reducing the processor fetch rate when the number of lazy instructions in the pipeline exceeds a dynamically decided threshold. Our study covered a subset of SPEC'2k benchmarks.

As mentioned, several approaches have been proposed to reduce power dissipation of the instruction queue. As these techniques target different base-line configurations, it is not possible to directly compare their results with ours. Future work will include implementing these techniques on our base-line configuration and study the possibility of their combination with our proposed techniques.

While throughout this work we only focused on lazy instructions it is possible to study fast instructions; i.e. instructions which are issued quickly. In addition we can use the information of lazy/fast instruction prediction to reduce power dissipation in other structure of a superscalar processor such as select unit, register renaming unit, fetch unit and etc. Lazy/fast instruction prediction benefits doesn't limited to superscalars and can be applied to any kind of processors such as SMTs and embedded processors.

8. References

[1] Paul R. Woodward: Perspectives on Supercomputing: Three Decades of Change. IEEE Computer, Vol.9, Issue: 10 , pp. 99-111, Oct. 1996.

[2] J.E. Smith and G.S. Sohi,: *The Microarchitecture of Superscalar Processors*, Proceedings of the IEEE, vol. 83, no. 12, pp. 1609--24, December 1995.

[3] D. Folegnani and A. Gonzalez: *Energy-effective issue logic*. In Proceedings of the 28th Annual International Symposium on Computer Architecture, pages 248--59, June. 2001.

[4] Alper Buyuktosunoglu, David H. Albonesi, Pradip Bose, Peter W. Cook, Stanley E. Schuster: Tradeoffs in power-efficient issue queue design, Proceedings of the 2002 international symposium on Low power electronics and design.

[5] Il Park, Chong Liang Ooi, T. N. Vijaykumar : Reducing Design Complexity of the Load/Store Queue, Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, Dec 2003.

[6] Subbarao Palacharla, Norman P. Jouppi, J. E. Smith: Complexity-effective superscalar processors, Proceedings of the 24th annual international symposium on Computer architecture, May 1997.

[7] Dmitry Ponomarev, Gurhan Kucuk, Kanad Ghose : Energy–Efficient Design of the Reorder Buffer, *12th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'02)* Seville, Spain, September 2002.

[8] Gurhan Kucuk, Dmitry Ponomarev, Kanad Ghose :Low-Complexity Reorder Buffer Architecture, *16th ACM International Conference on Supercomputing (ICS'02),* New York, June, 2002, pp. 57-66.

[9] Daniele Folegnani, Antonio González: Energy-effective issue logic, Proceedings of the 28th annual international symposium on Computer architecture, May 2001.

[10] Jaume Abella, Ramon Canal, Antonio González: Power- and complexity-aware issue queue designs,Micro, IEEE , Volume: 23 , Issue: 5 , Sept.-Oct. 2003, Pages:50 – 58.

[11] Aneesh Aggarwal, Manoj Franklin, Oguz Ergin: Defining Wakeup Width for Efficient Dynamic Scheduling, IEEE International Conference on Computer Design (ICCD'04), pp. 36-41,October 2004.

[12] Huang, M.; Renau, J.; Torrellas, J.: Energy-efficient hybrid wakeup logic, Proceedings of the 2002 International Symposium on Low Power Electronics and Design Aug. 2002.

[13] R. Canal and A. González: A Low-Complexity Issue Logic, Proc. ACM Int'l Conf. Supercomputing (ICS 00), ACM Press, 2000, pp. 327-335.

[14] R. Canal and A. González: Reducing the Complexity of the Issue Logic, Proc. ACM Int'l Conf. Supercomputing (ICS 01), ACM Press, 2001, pp. 312-320.

[15] P. Michaud and A. Seznec: Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors, Proc. Int'l Symp. High-Performance Computer Architecture (HPCA 01), IEEE CS Press, 2001, pp. 27-36.

[16] S.E. Raasch, N.L. Binkert, and S.K. Reinhardt: A Scalable Instruction Queue Design Using Dependence Chains, Proc. 29th Int'l Symp. Computer Architecture (ISCA 02), IEEE CS Press, 2002, pp. 318-329.

[17] J.P. Grossman: Cheap Out-of-Order Execution Using Delayed Issue, Proc. Int'l Conf. Computer Design 2000 (ICCD 00), IEEE CS Press, 2000, pp. 549-551.

[18] S. Önder and R. Gupta: Superscalar Execution with Dynamic Data Forwarding, Proc. Int'l Conf. Parallel Architectures and Compilation Techniques, IEEE CS Press, 1998, pp. 130-135.

[19] T. Sato, Y. Nakamura, and I. Arita: Revisiting Direct Tag Search Algorithm on Superscalar Processors, Proc. Workshop Complexity-Effective Design, 2001.

[20] D. Albonesi: Dynamic IPC/Clock Rate Optimization, Proc. 25th Int'l Symp. Computer Architecture (ISCA 98), IEEE CS Press, 1998, pp. 282-292.

[21] A. Buyuktosunoglu et al.: A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors, Proc. 11th Great Lakes Symp. VLSI (GLSVLSI 01), ACM Press, 2001, pp. 73-78.

[22] D. Ponomarev, G. Kucuk, and K. Ghose,: Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources, Proc. 33rd Int'l Symp. Microarchitecture (Micro-33), IEEE CS Press, 2001, pp. 90-101.

[23] J. Abella and A. González: Power-Aware Adaptive Issue Queue and Register File, Proc. Int'l Conf. High-Performance Computing (HiPC), 2003, to appear.

[24] S. Dropsho et al.: Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power,Proc. 11th Parallel Architectures and Compilation Techniques, IEEE CS Press, 2002, pp. 141-152.

[25] D. Brooks, V. Tiwari M. Martonosi "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations", Proc of the 27th Int'l Symp. on Computer Architecture, 2000.

[26] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In Proc. of the 29th Annual International Symposium on Computer Architecture, May 2002.

[27] D.Ernst, A.Hamel, and T.Austin. Cyclone:a broadcast-free dynamic instruction scheduler selective replay. In Proc. of the 30th Annual International Symposium on Computer Architecture, June 2003.

[28] J. S. Hu, N. Vijaykrishnan, and M. J. Irwin: Exploring Wakeup-Free Instruction Scheduling,In Proc. of the 10th International Conference on High-Performance Computer Architecture (HPCA-10 2004), 14-18 February 2004, Madrid, Spain.

[29] D. Burger, T. M. Austin, and S. Bennett: Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin Madison, July 1996.

[30] M. D. Brown, J. Stark, and Y. N. Patt: Select-free instruction scheduling logic. In Proc. of the International Symposium on Microarchitecture, Dec. 2001

[31] E. Brekelbaum, J. R. II, C. Wilkerson, and B. Black: Hierarchical scheduling windows. In Proc. of the 35th Annual IEEE/ACM International Symposium on Microarchitecture, Nov. 2002.

[32] J. Stark, M. D. Brown, and Y. N. Patt: On pipelining dynamic instruction scheduling logic, In Proc. of the International Symposium on Microarchitecture, Dec. 2000.

[33] S. Manne, A. Klauser and D. Grunwald.: Pipeline Gating: Speculation Control For Energy Reduction, In Proc. Intl. Symposium on Computer Architecture, Jun., 1998.

[34] Dharmesh Parikh, Kevin Skadron, Yan Zhang, Mircea R. Stan: Power-Aware Branch Prediction: Characterization and Design. IEEE Trans. Computers 53(2): 168-186 (2004)

[35] Himabindu Kakaraparthi: Low Energy Wakeup Logic, Master of Science Thesis, University of Maryland, 2003.