Accepted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2004

Kanad Ghose_____Spring, 2004
Department of Computer Science

Nael Abu–Ghazaleh_____Spring, 2004
Department of Computer Science

Patrick Madden_____Spring, 2004
Department of Computer Science

Mark Fowler_____Spring, 2004
Department of Electrical and Computer Engineering

Acknowledgements

Abstract

In this dissertation, we propose several microarchitectural–level, technology–independent solutions for reducing the power requirements of a superscalar microprocessor without seriously impacting its performance. First, we focus on the issue queue which plays an important role in the out–of–order execution core. We apply several circuit– and microarchitectural–level techniques to minimize the power dissipation of this structure. First, after observing that the large percentage of mismatches occur in the tag–matching logic of the issue queue, we replace the traditional dissipate–on–mismatch type comparators with faster dissipate–on–match type comparators. Second, we introduce a circuit–level technique to avoid reading the bytes that contain all zeroes from the issue queue and also avoiding their writes to the same structure As the percentage of such bytes is quite large, the technique results in significant energy savings. Finally, we utilize bit–line segmentation to reduce the bitline read and write power in the issue queue. Combined, these three mechanisms reduce the power dissipated by the instruction issue queue in superscalar processors by 56% to 59% on the average across all SPEC 2000 benchmarks depending on the datapath design style.

Second, we focus on the next crucial datapath component: Reorder Buffer. The reorder buffer holds all in–flight instructions, and therefore its size is the second largest after the on–chip caches inside the processor. First, we propose a scheme to eliminate the ports needed for reading the source operand values for dispatched instructions. Second, we introduce the conflict–free ROB distribution scheme, where the conflicts over the use of the write ports are eliminated by allocating a small separate FIFO queue for holding the speculative results for each functional unit. On the average, the ROB power savings of as high as 51% are realized with only 1.7% average drop in performance.

Third, we propose two complementary techniques to reduce the energy dissipation within the register alias tables. The first technique uses the intra–group dependency checking logic already in place to disable the activation of sense amps within the RAT if the register address to be read is redefined by an earlier instruction dispatched in the same group. The second technique extends this approach one step further by placing a small number of associatively–addressed latches in front of the RAT to cache a few most recent translations. Again, if the register translation is found in these latches, the activation of sense amps within the RAT is aborted. The combined effect of the two proposed techniques is the energy reduction in the RAT of about 30%. This comes with no performance penalty, little additional complexity and no increase in the cycle time.

Our solutions are applicable in the domain of high–performance superscalar machines as well as in the domain of high–end embedded processors with the low degree of superscalarity. The results are validated by the execution of SPEC 2000 benchmarks on a true cycle–level microarchitectural simulator and the SPICE simulations of the actual layouts of the datapath components in 0.18u CMOS process.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern superscalar microprocessors exploit instruction–level parallelism (ILP) in the sequential codes by utilizing dynamic instruction scheduling, register renaming and speculative execution techniques. Instructions are executed inside the out–of–order execution core. To maximize the processor performance, contemporary designs aim to maximize the number of ready instructions that can be scheduled and executed in each cycle. This can be achieved by maintaining large window sizes for the buffers that hold the instructions. The register renaming is a well–known technique to eliminate data hazards that exist among dependent instructions. The technique maintains a Register Alias Table (RAT) and phsyical register files to map each logical destination register to a physical register. The speculative execution, on the other hand, deals with control hazards. Since, the result of the branch instructions are not available at dispatch time, the front–end of the pipeline need to be stalled until the branch is resolved. The resultant performance drop due to these type of pipeline stalls may be critical for a performance–oriented superscalar processor. The branch prediction unit is used to overcome this problem by predicting the branch results at the time of their dispatch. The front–end continues to bring instructions to the pipeline in the speculated path. Incase of a branch misprediction, the instructions in the mispredicted path are flushed and execution resumes on the correct path. The Reorder Buffer (ROB) is generally used to recover from branch mispredictions, since it maintains all the in–flight instructions in program order, allowing all the instructions in the mispredicted path to be

invalidated and flushed from the pipeline. To support precise interrupts, the back–end of the pipeline commits instructions strictly in program order.

This very complex instruction processing results in the utilization of several datapath structures such as Fetch Queue (FQ), Register Alias Table (RAT), Issue Queue (IQ), Branch Target Buffer (BTB), Prediction History Table (PHT), Reorder Buffer (ROB), Load/Store Queue (LSQ), Physical Register Files (PRFs) and Architectural Register Files (ARFs). These datapath artifacts are usually implemented in the form of large multi–ported register files. To minimize the latency, they are usually augmented with associative addressing capabilities. Additional complexities comes in the form of multiple Function Units (FUs), Translation Lookaside Buffers (TLBs) and sizeable on–chip caches.

Although all of these datapath components are vital for a performance–oriented superscalar processor, they do not come as free of charge. As a result, they drastically increase the complexity and energy dissipation of the processor [GH 96, Tiwari 98, ZK 00]. The complexity of a processor can be variously quantified in terms such as number of transistors, die area and the delay of the critical path through a piece of logic; and it directly effects the reliability, scalability, cost and performance of a processor. The energy dissipation, on the other hand, is also critical from the perspective of device reliability and cooling of the system. Up until now power consumption has not been of great concern since large packages, heat sinks and fans were capable of dissipating the generated heat. However, as the density of the chips continue to increase and systems are packaged into smaller and thinner enclosures it is becoming increasingly difficult to design adequate cooling system without incurring significant costs. Additionally, the reliability and operating lifetime of integrated circuits is greatly reduced when the ambient operating temperature increases because high temperature tends to exacerbate several silicon failure mechanisms such as electromigration, junction fatigue, and gate dielectric breakdown. Every 10 °C increase in operating temperature roughly doubles a component's failure rate [Small 94]. It is thus

imperative to operate these devices at safe temperatures to ensure reliable and long life performance.

In the rest of this chapter, we first examine the fundamental attributes to characterize a microprocessor. Then, we discuss the sources of power dissipations in CMOS circuits followed by the analysis of the current technology trends and consider the implications on the processor's energy and power consumption. Finally, we present the goals and contributions of this work.

1.1. Key Metrics for Characterizing a Microprocessor

Before going into more detailed discussions about microprocessors and current techology trends, it is essential to overview the key metrics that differentiates one microprocessor from another. These include: performance, power, cost (die area), and complexity.

Performance is measured in terms of the time it takes to complete a given task. It depends on many parameters such as the microprocessor itself, the specific workload, system configuration, compiler optimizations, operating systems, and more. The following equation defines microprocessor performance:

Performance = 1 / Execution Time = (IPC x Frequency) / Instruction_Count (1–1),

where *IPC* is the average number of instructions completed per cycle, *Frequency* is the number of clock cycles per second, and *Instruction_Count* is the total number of instructions executed. IPC varies depending on the environment – the application, the system configuration, and more. Instruction count depends on the ISA and the compiler used. For a given executable program, where the instruction stream is invariant, the relative performance depends only *IPC x Frequency.* In that case, the performance is measured in million instructions per second (MIPS).

Power is energy consumption per unit time, in watts. It is directly proportional to performance of the system. Higher performance requires more power. However, power is constrained due to the following.

- *Power density and Thermal:* Power density is described as the power dissipated by the chip per unit area. It is measured in watts/cm^2. Increases in power density causes heat to generate. In order to keep transistors within their operating temperature, the heat generated has to be dissipated from the source in a cost–effective manner.
- *Power Delivery:* Power must be delivered to a VLSI component at a prescribed voltage ad with sufficient amperage for the component to run. As the current increases, the cost and complexity of the voltage regulators/transformers that control current supplies increase as well.
- *Battery Life:* Batteries are designed to support a certain *watts x hours*. When the power requirements of the processor increase, the lifetime of the battery decreases.

Cost is primarily determined by the die area. Larger area means higher (even more than linear) manufacturing cost. Larger area also usually implies higher power consumption and may imply lower frequency due to longer wires. Manufacturing yield also has direct impact on the cost of each microprocessor.

Complexity reflects the effort required to design, validate, and manufacture a microprocessor. Complexity is affected by the number of devices on the silicon die and the level of aggressiveness in the performance, power and die area targets.

1.2. Sources of Power Dissipations in CMOS Circuits

CMOS circuits have an interconnection structure of complementary PMOS and NMOS FET transistors. The PMOS and NMOS transistors logically act as switches that turn ON and OFF under the control of appropriate gate voltages. Ideal complementary MOS circuits

have the unique property that there is never a direct path from the supply voltage to the ground terminal for all possible input combinations under steady state conditions. So the steady state power dissipation in CMOS circuits is ideally zero. However real CMOS circuits do dissipate energy in steady state because the MOSFETs are imperfect switches in reality with finite turn–on and turn–off time and nonzero leakage currents. The gates of the MOSFETs also present a capacitive load to the output of the gate that drive it. CMOS circuits thus also dissipate energy when their outputs make transitions in their voltage levels, because of the charging and discharging of the output capacitive load as its voltage levels vary. The different components of the total power dissipated in CMOS circuits are as follows:

- **Static Dissipation:** This is due to substrate leakage current or other current drawn continuously from the power supply.
- **Dynamic Dissipation:** This is due to a) transient short circuit current when both n–and p–transistors are momentarily ON simultaneously, which occurs due to finite switching time of these transistors, and b) charging and discharging of load capacitance at the output node during output transitions.

Figure 1–1 shows the steady state leakage current that exists in a simple CMOS inverter. The PMOS transistor Q_p has it's substrate connected to the voltage source V_{dd} and that of the NMOS transistor Q_n is connected to GND. $I_{leakage}$ is the leakage current that flows from V_{dd} to GND through these substrate connections. The numbers besides each transistor indicate the width and length of the FET channel for each transistor. Static dissipation represents about 7% of the the overall power in 0.18u technology, it is about 20% in 0.13u technology and is expected to grow to about 50% in the next technology generation. Figure 1–2 shows the speculated power of a 15mm. die, in detail [KN 02].

Figure 1–3 shows the currents in the CMOS inverter during output transitions and the corresponding transfer characteristics. V_{in} is the input voltage and V_{out} is the inverter output voltage. C_{load} is the effective load capacitance visible to the output node of the inverter. The

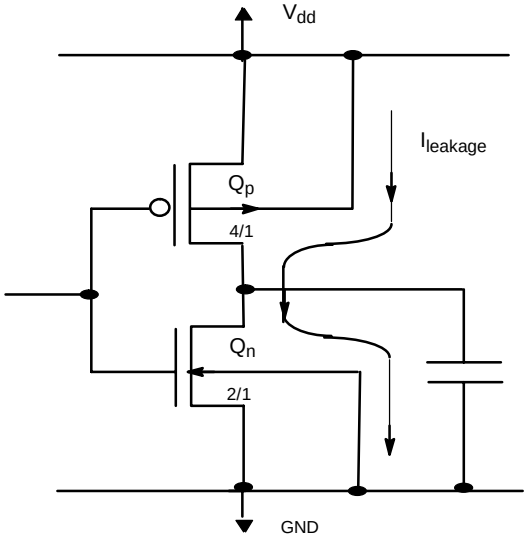


Figure 1–1. Leakage Current in a CMOS Inverter

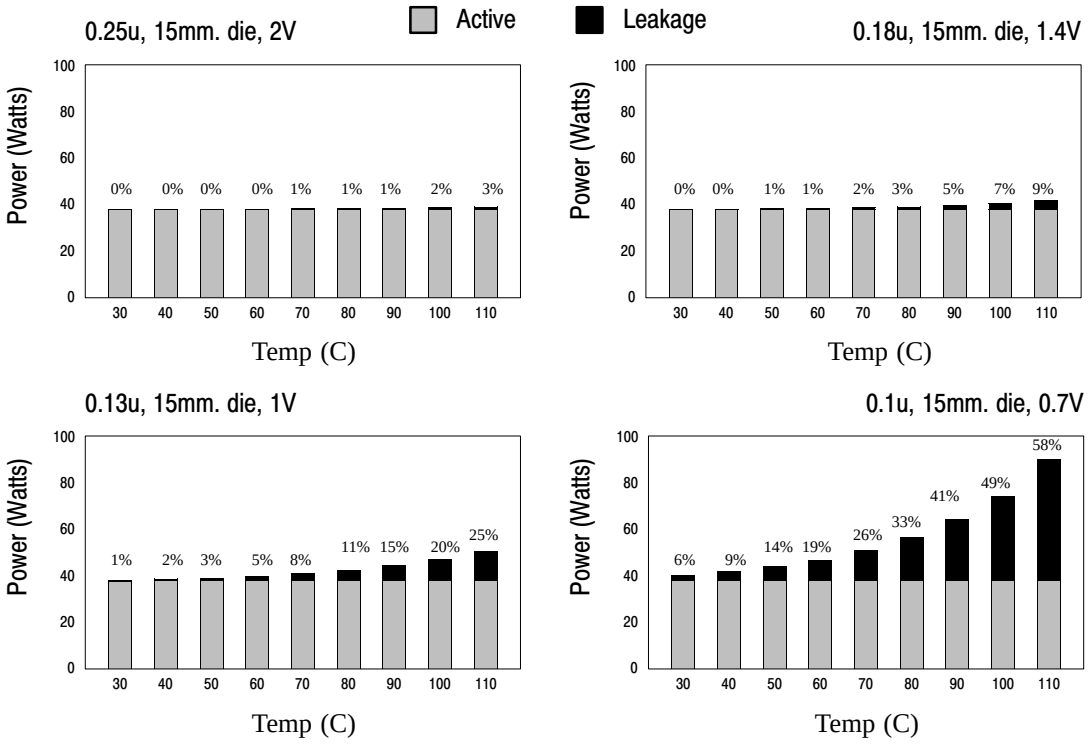


Figure 1–2. Projected power of a 15mm. uP

current I_{charge} is the charging current that flows to C_{load} when he output makes a transition to high voltage in response to the input's transition to a low value. The current I_{disch} is the

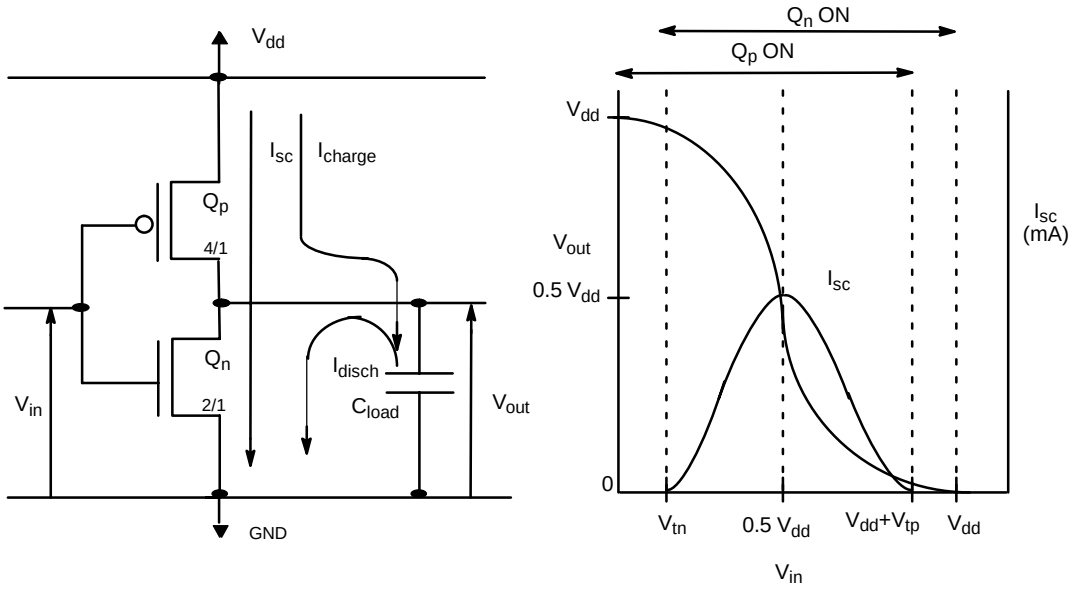


Figure 1–3. Dynamic Currents During Output Transitions in CMOS Circuits

discharging current from output load capacitor to GND when the output makes a transition from high to low voltages. The short circuit current I_{sc} is the short circuit that flows during the brief period of the time when both the transistors Q_n and Q_p are ON simultaneously. Short–circuit dissipation can be minimized by matching the rise/fall times of the input and output signals – this is called slope engineering. Short–circuit power does not represent a significant percentage of the overall power consumption in current technologies and is not expected to increase much more in the future. Therefore, we will ignore it in the forthcoming discussions.

At current feature sizes (0.18u and 0.13u at the time of this writing), the dynamic dissipation due to output transitions is the most dominant component. The power dissipated in CMOS digital circuits is thus directly dependent on the transition activity occuring in the circuit. Assuming that the output voltage changes every cycle, the dynamic power can be represented in terms of energy and frequency using the following expression:

$$P = \alpha * E * f = \alpha * C_{load} * V_{dd}^2 * f \qquad (1–2),$$

where α is the activity factor, C_{load} is the load capacitance at the output node, V_{dd} is the supply voltage and f is the frequency of input changes or the clock rate.

1.3. Technology Trends: the Ideal vs. the Practical

A new process generation is released every two to three years. It is usually identified by the length of a metal–oxide–semiconductor gate, measured in micrometeres (10^{-6} m., denoted as um.).

Ideally, process technology scales by a factor of 0.7 all physical dimensions of transistors and interconnects including those vertical to the surface and all voltages pertaining to the devices [Borkar 99]. Typical improvements are as follows:

- approximately 1.5 times faster transistors,
- two times smaller transistors,
- 1.35 times lower operating voltage,
- three times lower switching power.

Theoretically, there are two types of potential improvements when we consider the above figures:

1) **Ideal Shrink:** Use the same number of transistors to gain 1.5 times more performance, two times smaller die, and two times less power.
2) **Ideal New Generation:** Use two times the number of transistors of the previous process generation to gain three times more performance with no increase in die size and power.

In both ideal scenarios, there is three times gain in MIPS/watt and no change in power density.

In practice, it takes more than just process technology to achieve such performance improvements at much higher costs. One of the main reasons for this is the fact that the

performance does not scale linearly with frequency. Resource contention, data dependencies, memory delays and control dependencies are some of the performance limiting factors. To achieve better performance architects push the limits of the rules of the scaling theory. Assuming that a technology generation spans two to three years, the microprocessor frequency actually doubles with each new generation, rather than just increases by 50%, as predicted by the scaling theory estimates. Also, if a new microarchitecture is implemented in a new process technology then the new designs usually use three times more transistors (rather than two times as projected by the scaling theory) [Borkar 99].

As a result, every new process technology and microarchitecture generation, the absolute power increases by a factor of two. This is because three times as many transistors switch twice as often and the energy per transition is reduced by a factor of three. The power density increases somewhere between 30% and 80%, depending on the area needed by the new design. In addition, the supply voltage is expected to scale less aggressively in the future and the leakage power is becoming a serious concern.

Figures through 1–4 to 1–7 depict the projections for die size, transistor count, power and power density for Intel microprocessor family, respectively. It is easy to observe that power dissipation and power density have been growing rapidly in each successive generation. Assuming that the same trend continues in the future, it can be extrapolated that the power dissipation in microprocessors will increase to alarming levels enough to melt the chip itself if it is not cooled adequately. The issue of adequately dissipating the heat generated will be the most formidable challange to realize these projections. For most devices targeted towards the consumer market, the change in price point due to extra cost of the cooling system can directly affect the success of the product in the market. Environment related laws and standards such as the EPA Energy Star also demand reduction in the power consumed by computer systems with the ultimate goal of reducing environmental pollution. Reducing the power dissipated in system with external power

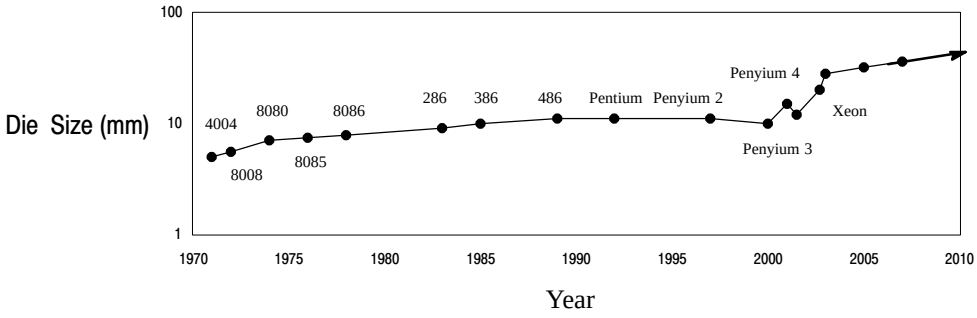


Figure 1–4. Die size increase in Intel microprocessor family

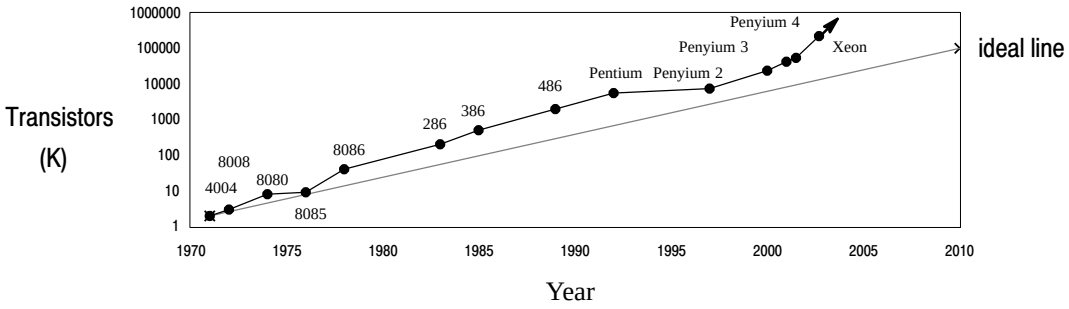


Figure 1–5. Transistor increase in Intel microprocessor family

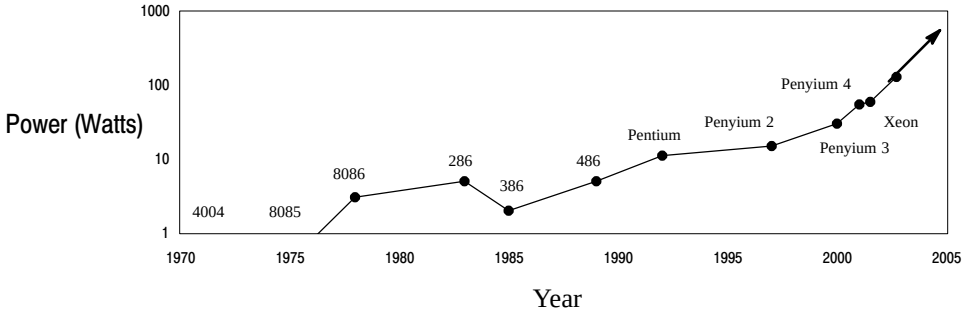


Figure 1–6. Power dissipations in Intel microprocessor family

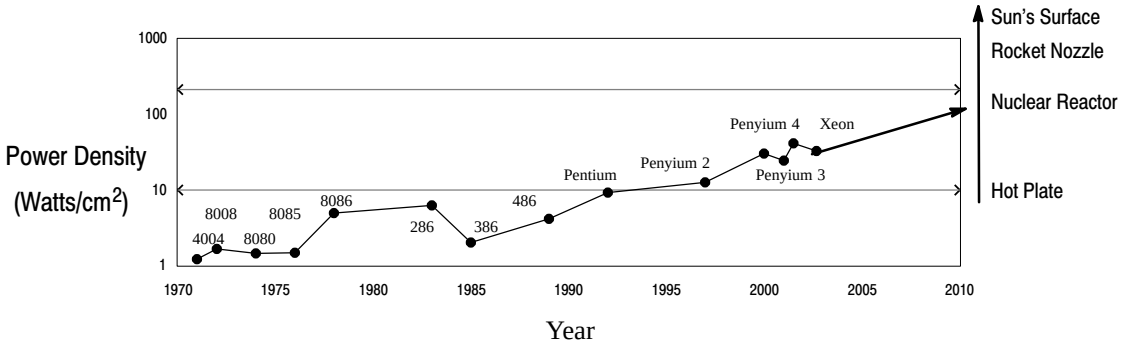


Figure 1–7. Power density increase in Intel microprocessor family

sources is also ultimately advantageous to the users because it reduces the operating expenses. Therefore, the power becomes a first–class, universal design constraint in the domain of high performance systems [Mudge 01]. In the past five years, a large body of work has been devoted to reducing the power dissipation in superscalar datapaths.

1.4. Goals and Contributions

The primary goal of this thesis is to propose and evaluate mechanisms for power and complexity reduction within superscalar microprocessors. Because power reduction is accompanied with little or no impact on the performance, similar amount of energy reduction is also achieved. Secondary goal of this thesis is to reduce the cost of design by proposing techniques that are technology–independent, microarchitectural–level solutions requiring a minimum amount of support at the circuit level. Because of technology–independent nature of our solutions, orthogonal solutions such as voltage and frequency scaling can also be used on top of our schemes to save additional energy and power. Specific contributions of this work are as follows:

First, we focus on the issue queue which plays an important role in the out–of–order execution core. According to some of the studies, the issue queue dissipates about 25% of the total power in the processor [FG 01]. We apply several circuit– and microarchitectural–level techniques to minimize the power dissipation of this structure:

- After observing that the large percentage of mismatches occur in the tag–matching logic of the issue queue, we replace the traditional dissipate–on–mismatch type comparators with faster dissipate–on–match type comparators.
- We introduce a circuit–level technique to avoid reading the bytes that contain all zeroes from the issue queue and also avoiding their writes to the same structure. As the

percentage of such bytes is quite large, the technique results in significant energy savings.

- We utilize bit–line segmentation to reduce the bitline read and write power in the issue queue.

Combined, these three mechanisms reduce the power dissipated by the instruction issue queue in superscalar processors by 56% to 59% on the average across all simulated SPEC 2000 benchmarks depending on the datapath design style.

Second, we focus on the next crucial datapath component: Reorder Buffer. The reorder buffer holds all in–flight instructions, and therefore its size is the second largest after the on–chip caches inside the processor. We apply following microarchitectural–level techniques to minimize the power dissipation and the complexity of this structure:

- We propose a scheme to eliminate the ports needed for reading the source operand values for dispatched instructions.
- We introduce the conflict–free ROB distribution scheme, where the conflicts over the use of the write ports are eliminated by allocating a small separate FIFO queue for holding the speculative results for each functional unit.

On the average, the ROB power savings of as high as 51% are realized. Taking into account that the ROB, as used in the considered datapath style, can dissipate as much as 27% of the total chip energy [FG 01], our techniques result in about 13% of total CPU power reduction with only 1.7% average drop in performance.

Third, we propose two complementary techniques to reduce the energy dissipation within the register alias tables. The first technique uses the intra–group dependency checking logic already in place to disable the activation of sense amps within the RAT if the register address to be read is redefined by an earlier instruction dispatched in the same group. The second technique extends this approach one step further by placing a small number of associatively–addressed latches in front of the RAT to cache a few most recent translations.

Again, if the register translation is found in these latches, the activation of sense amps within the RAT is aborted. The combined effect of the two proposed techniques is the energy reduction in the RAT of about 30%. This comes with no performance penalty, little additional complexity and no increase in the cycle time.

Finally, we describe the AccuPower – a toolset for accurate power analysis within high–performance microprocessors that we designed and used in our studies for evaluating the impacts of the proposed techniques on processor's power and performance.

1.5. Thesis Organization

The rest of this dissertation is organized as follows. In Chapter 2, we describe the baseline dynamic superscalar processor microarchitecture that we assume for this study. We also analyze the sources of complexity and power dissipation within the major datapath components. In Chapter 3, we overview the techniques that target low power and low complexity. Chapter 4 describes the AccuPower – an accurate power estimation toolsuite for superscalar microprocessors that we designed and used to evaluate our techniques. In subsequent chapters, we propose and evaluate novel power reduction mechanisms at the microarchitectural and circuit level. In Chapter 5, we describe and evaluate the techniques that targets energy–efficient issue queue. Chapter 6 proposes several technique for power and complexity reduction in the reorder buffer. In Chapter 7, we propose techniques for energy–efficient register alias table. Conclusions and directions for future research are presented in Chapter 8.

Chapter 2

Energy and Complexity Considerations for Superscalar Microprocessors

Contemporary superscalar microarchitectures aim performance by utilizing a number of aggressive techniques, such as dynamic instruction scheduling, register renaming and speculative execution. These techniques introduce new components into the processor core resulting in unavoidable increase in the die area and energy dissipation. In this chapter, we examine the very common variations of the superscalar datapaths and their key artifacts, in detail. We also take a brief look at the sources of complexity and energy dissipation within the major components of such superscalar datapaths.

2.1. Superscalar Datapath Architectures

A superscalar datapath consists of three major stages: 1) In–order front end, 2) Out–of–order execution core, and, finally, 3) In–order back end. Figure 2–1 shows an example superscalar datapath with these stages and major datapath components. Basically, it is an assembly line for instructions. The instructions are introduced by the *in–order front end*, executed by the *out–of–order execution core*, possibly out–of–order, and commited in order by the *in–order back end*. In the following subsections, we examine these stages and their datapath components in more detail.

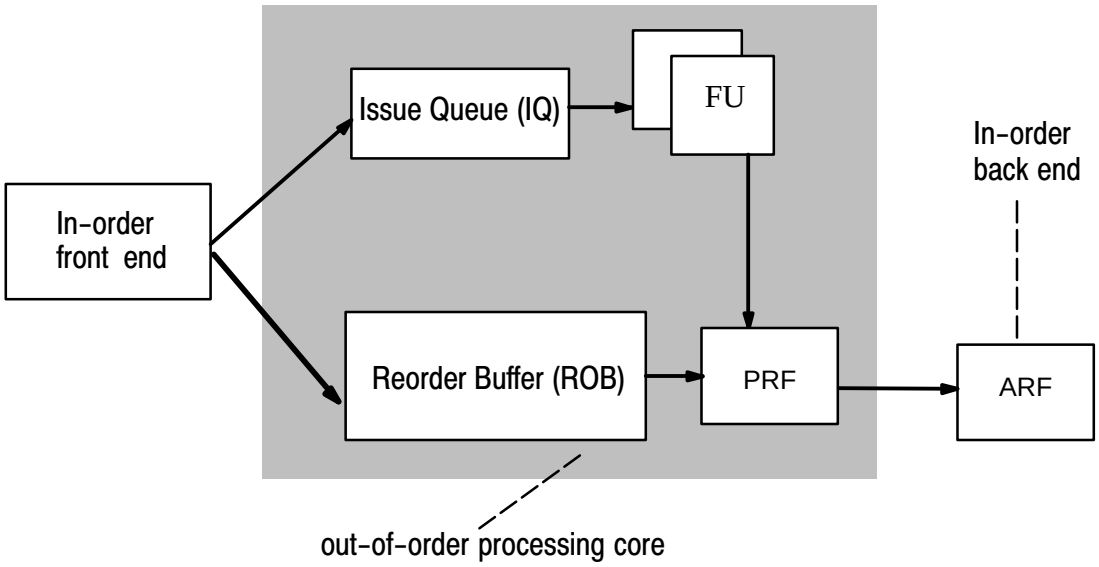


Figure 2–1. An example superscalar datapath

2.1.1. In–Order Front End

This very first stage of the processor introduces instructions to the superscalar datapath. This stage contains the fetch and decode stages of the pipeline. First, the instructions are put into a FIFO queue which is called as Fetch Queue (FQ). Second, they are decoded and prepared for the out–of–order execution stage. In the decode stage, Register Alias Table (RAT) is accessed for renaming the source and destination operands of the instructions. Register renaming is a well–known technique to avoid data hazards. Therefore, we will briefly examine it in the end of this section.

2.1.1.1. Fetch Stage

The fetch stage supplies instructions to the Fetch Queue (FQ). The typical size of a FQ is 2*W for a W–way superscalar datapath. This allows simultaneous access to this structure from *Fetch* and *Decode* stages. In the fetch stage, the instructions are read from the instruction cache (I–cache) and copied to the FQ in order; while in the decode stage, they are read from the FQ and prepared for the IQ. Incase of a miss to the I–cache, the instructions have to be read from the memory and brought to the cache to satisfy further possible

references. The usual hit ratio to the I–Cache is more than 95% on the average. Therefore, the penalty of a miss to the I–Cache is very small.

To avoid any pipeline stall in front–end of the pipeline, fetch process continue fetching instructions after encountering conditional branch instructions by utilizing a branch prediction unit. When a conditional branch is reached, the address of the instruction is sought in Branch Target Buffer (BTB) and Prediction History Table (PHT) to retrieve the predicted branch direction and its target address. If it is not found in there, the default prediction is used. Otherwise, fetch mechanism continues fetching instructions starting from the predicted branch target address corresponding to that conditional branch. When, the branch is resolved in further stages of the pipeline, the computed branch direction is compared with the predicted direction which is used by the fetch mechanism. If the prediction result turns out to be incorrect, the FQ is flushed and the fetch process continues fetching instructions from the correct direction. The speculative execution increases the performance drastically, if the branch misprediction rate is reasonably low. Otherwise, if the misprediction rate is high, the scheme results in some severe performance drop. Moreover, many instruction are scheduled and flushed in the pipeline, resulting in unnecessary data movements among the datapath components. This considerably increases the energy dissipations in the processor front end.

2.1.1.2. Decode Stage

This pipeline stage checks several structures (issue queue, reorder buffer, load/store queue, and physical register files) depending on the instruction type to make sure if there is enough space to allocate the instruction(s). If there is not enough space available, decode stage stalls; otherwise, the usual decode process takes place. For a W–way processor, at most W instructions may be decoded in each cycle.

The register renaming is an important part of this stage. It is used to cope with false data dependencies by assigning a new physical register to each produced result.

2.1.1.3. Key Datapath Components in In–Order Front End

In this subsection, we examine the key datapath components in detail:

The Register Alias Table

The mappings between logical and physical registers are kept in the Register Alias Table (RAT, a.k.a rename table), so that each instruction can identify its physical register sources by performing the RAT lookups indexed by the addresses of the source logical registers.Both RAM and CAM–based implementations of RAT are possible. In this study, we assume the RAM–based implementation of the RAT, where for each logical register the table maintains the most recently assigned physical register. The number of entries in such RAT is thus equal to the number of logical registers, which are typically few. Each entry is $\log_2 P$ bits wide, where P is the number of physical registers. The RAT is multi–ported to support the access for both reads and writes by multiple instructions co–dispatched in the same cycle. Additional area in the RAT is occupied by the bits needed for checkpointing the state to support rapid recovery from branch mispredictions. The number of extra bits needed for checkpointing depends on the number of unresolved branches that are allowed to be in–flight simultaneously. Consequently, a significant amount of power (about 4% in the Pentium Pro rename unit [MKG 98]) is dissipated in the disproportionately small area, thus creating a hot spot. Even higher percentage of the overall power – 14% – is attributed to the RAT in the global power analysis performed by *Folegnani* and *Gonzalez* in [FG 01].

For this study, we used RISC–type ISA, where instructions may have at most two source registers and one destination register. In a W–way superscalar machine, up to W instructions may undergo renaming in the same cycle. Thus, 2*W register address translations may have to be performed in a cycle to obtain the physical addresses of the source registers. In addition, up to W new physical registers may have to be allocated to hold the new results.

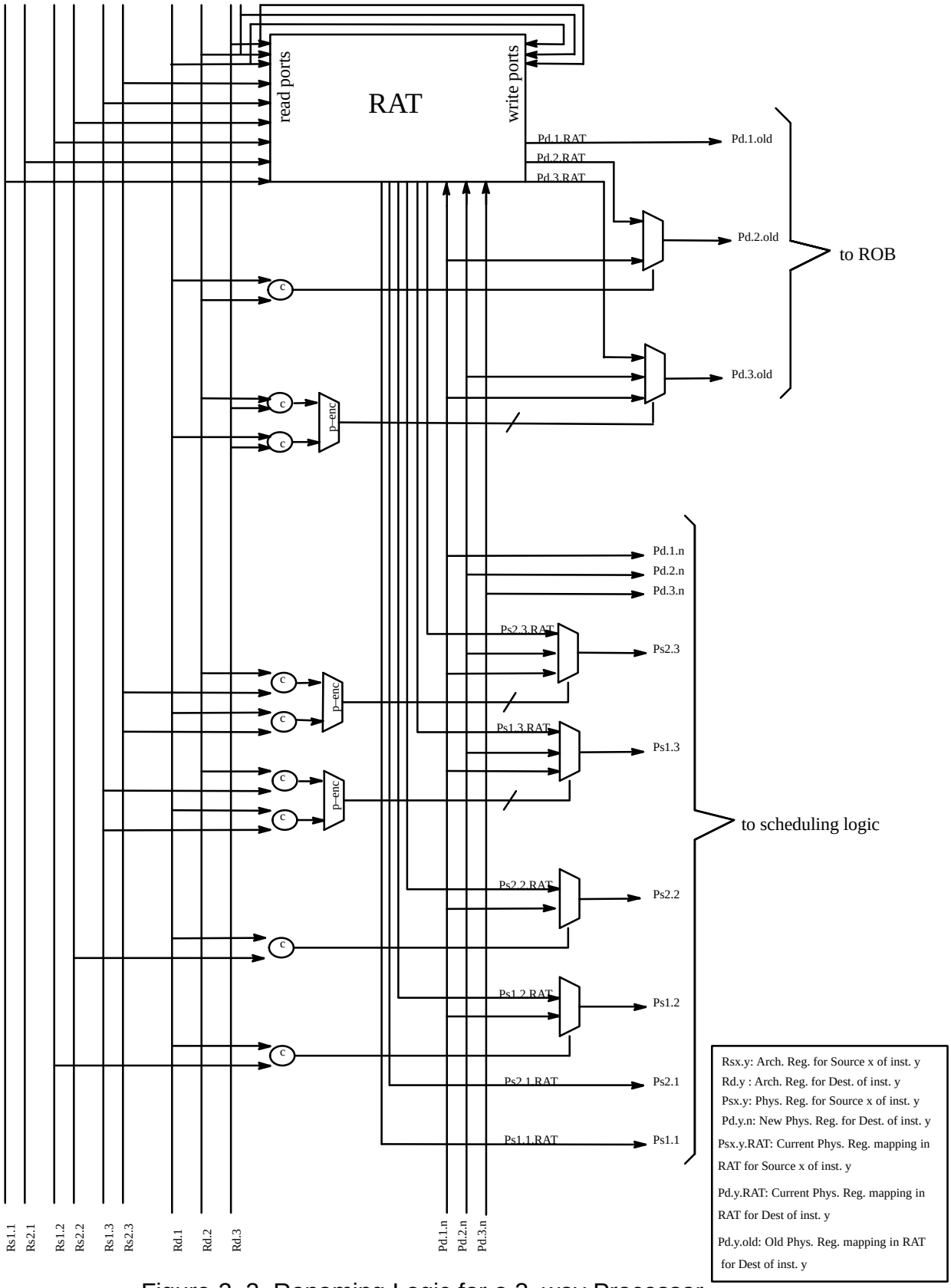


Figure 2–2. Renaming Logic for a 3–way Processor

In our studies, we assumed that the RAT is implemented as a multi–ported register file, where the number of entries within the RAT is equal to the number of architectural general–purpose registers in the ISA. The width of each RAT entry is equal to the number of bits in a physical register address. An alternative design is to have the number of entries in the RAT equal to the number of physical registers, such that each RAT entry stores the logical register corresponding to a given physical register. In this scheme, as implemented in Alpha 21264 [Kessler 99], the RAT lookup is performed by doing the associative search using logical register address as the key. We did not consider this variation of the RAT in this study, because it is inherently less energy–efficient than the RAM–based design, due to excessive amounts of energy dissipated in the course of associative comparisons. This is because in most of the cases the comparisons result in a mismatch, but the traditional pulldown comparators dissipate significant energy on mismatches. Their output is precharged and if a mismatch occurs in at least one of the bit positions in the comparands, the output is discharged, causing the energy dissipations. One way to address this problem is to use a recently proposed dissipate–on–match comparator [EGK+ 02] in the associative logic within the RAT as disccussed in Chapter 5.

The number of ports needed on the RAT in a W–way superscalar machine is quite significant. Specifically, 2*W read ports are needed to translate the source register addresses from logical to physical identifiers and W write ports are needed to establish W entries for the destinations. In addition, before the destination register mapping is updated in the RAT, the old value has to be checkpointed in the Reorder Buffer for possible branch misprediction recovery. Consequently, W read ports are needed for that purposes, bringing the total port requirements on the RAT to 3*W read ports and W write ports. Figure 2–2 shows the details of renaming logic for a 3–way processor.

The energy dissipations take place in the RAT in the course of the following events:

1) **Obtaining physical register addresses of the sources:** This is in the form of reads from the register file implementing the RAT.

2) **Checkpointing the old mapping of the destination register.** This, again, is in the form of reading the register file that implements the RAT. This read is necessary in machines that support speculative execution and read out the old mapping of the destination register, which is then saved into the reorder buffer. If the instruction that overwrites the entry is later discovered to be on the mispredicted path, the old mapping saved within the reorder buffer is used to restore the state of the RAT.

3) **Writing to the RAT for establishing the new mapping for the destination register.** If the dispatched instruction has a destination register, a new physical register is assigned for that destination register, and the RAT entry for the destination architectural register is updated with the address of the allocated physical register by this write. If there is no physical register available for the destination register at the time of dispatch, the dispatch stalls.

2.1.2. Out–of–Order Execution Core

The out–of–order execution core exploits instruction–level parallelism (ILP) to increase performance of the superscalar processor. It combines three pipeline stages: dispatch, issue and writeback. Now, we examine these stages and their key components, in detail:

2.1.2.1. Dispatch Stage

This dispatch stage places instructions into the out–of–order execution core. This process is called as *instruction dispatch*. Each instruction is unconditionally placed into two well–known datapath structures: Issue Queue (IQ, a.k.a. dispatch buffer) and Reorder Buffer (ROB). The IQ is the heart of the out–of–order execution core. It holds the instructions that are waiting for execution. The order of the instructions are not important in this structure,

since instruction may execute in any order as long as their source operands are valid and the instruction is ready. The ROB is a FIFO queue that keeps track of the instruction order. It is used at the time instruction retirement to commit the instructions in order. If the instruction being dispatched is a memory instruction, an entry is also need to be allocated in the Load/Store Queue (LSQ). The LSQ is used to disambugiate memory references and enable *load*s to bypass instructions if their address do not match with the addresses of the previously dispatched *store* instructions.

2.1.2.2. Issue Stage

When all of the source operands of an instruction becomes valid, the instruction becomes ready for execution. Issue stage moves ready instructions from issue queue to available function units for execution. It consists of two stages: wakeup and select. In the wakeup stage (as shown in Figure 2–3), the ready field of the instructions whose source operands becomes valid are set. Consequently, the select logic selects at most W ready instruction from issue queue for execution for a W–way processor. The number of selected instruction may be less than W depending on the total number of ready instructions sitting in the issue queue and the availability of the corresponding function units that will execute those instructions.

2.1.2.3. Writeback Stage

When the function units execute instructions and produce results, those results are forwarded back to issue queue through forwarding buses to activate the dependent instructions. The results are also copied to the corresponding result repositories, in this stage. Since, these results may be in speculative path, they are not written to Architectural Register File (ARF), directly. They are kept in the result repositories until all the previous conditional branch instructions are resolved. The result repositories may have different names in

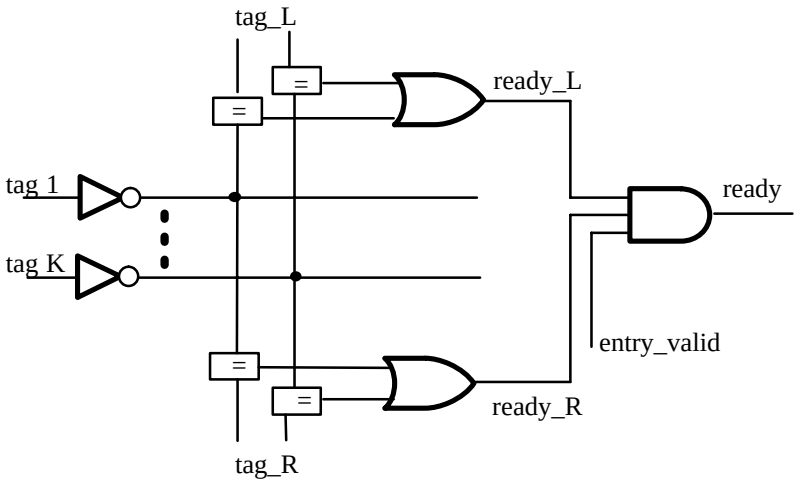


Figure 2–3. Wakeup logic of the issue queue

different datapath variations. They are called Physical Register File (PRF) or Rename Buffers (RB), if they are implemented as seperate structures. They may also be integrated either into ROB or ARF, in some other datapath variations.

2.1.2.4. Key Datapath Components in Out–of–Order Execution Core

In this subsection, we examine the key datapath components and possible datapath variations, in detail:

Issue Queue

In modern superscalar processors, the issue queue is a complex multi–ported structure that incorporates associative logic in the form of comparators for data forwarding, wakeup logic to identify the instructions ready for execution and additional logic for selecting ready instructions. In aggressive superscalar designs, the size of the issue queue ranges from the 20s to more than 100 entries [Emer 01]. It is therefore not surprising to see that a significant fraction of the total CPU power dissipation, often as much as 25% [FG 01] is expended within the issue queue.

Figure 2–4 depicts the black box view of a IQ as described above. This is essentially a multi–ported register file with additional logic for associative data forwarding from the

forwarding buses and associative addressing logic that locates free entries and entries ready for issue. When we assume a W–way superscalar processor, the IQ is assumed to have read ports, W write ports and W forwarding buses. W write ports are used to establish the entry for up to w instructions simultaneously in the IQ at the time of dispatching. W read ports are used to select up to w ready instructions for issue and move them out of the IQ to the FUs. If the IQ, shown in Figure 2–4, has N entries and each instruction can have up to two source operands, then 2*W comparators are needed for each IQ entry (for a total of 2*W*N comparators for the entire IQ). This is because each source operand can be delivered through any of the W result/tag buses.

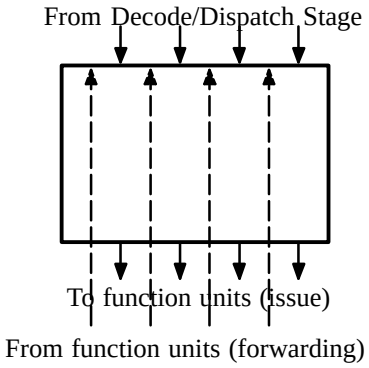


Figure 2–4. Black box view of the Issue Queue

Figure 2–5 shows an example superscalar datapath. Bold lines shown in this and subsequent datapath diagrams correspond to buses (or multiple set of connections). Here, input registers that contain valid data are read out while the instruction is moved into the IQ. As the register values required as an input by instructions waiting in the IQ (and in the dispatch stage) are produced, they are forwarded through forwarding buses that run across the length of the IQ [PJS 96]. Figure 2–6 depicts the IQ structure. It has one data field for each input operand, as well as an associated tag field that holds the address of the register whose value is required to fill the data field. When a function unit completes, it puts out the result produced along with the address of the destination register for this result on a forwarding bus. Comparators associated with each IQ entry then match the tag values stored

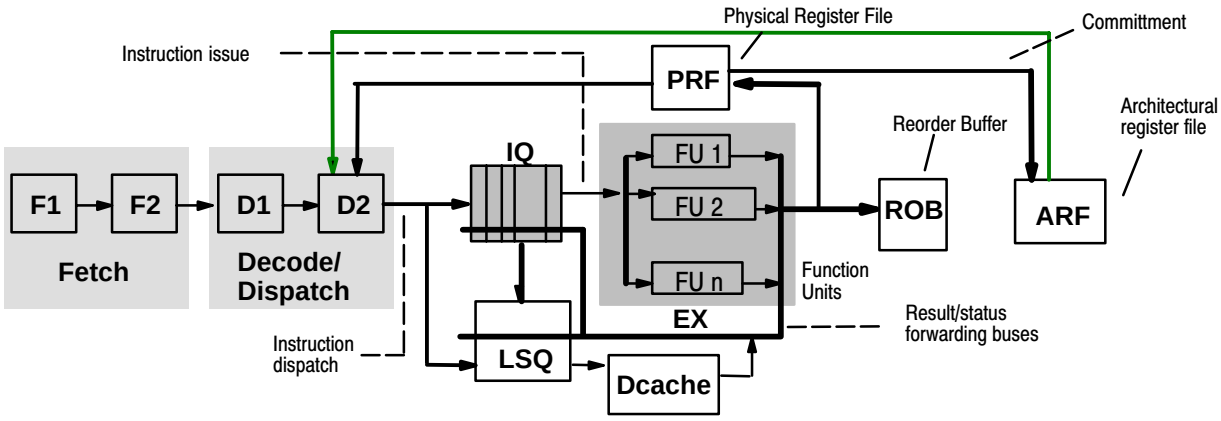


Figure 2–5. Superscalar datapath with dispatch–bound operand reads

in the fields (for waited–on register values) against the destination register address floated on the forwarding bus [PJS 96]. On a tag match, the result floated on the bus is latched into

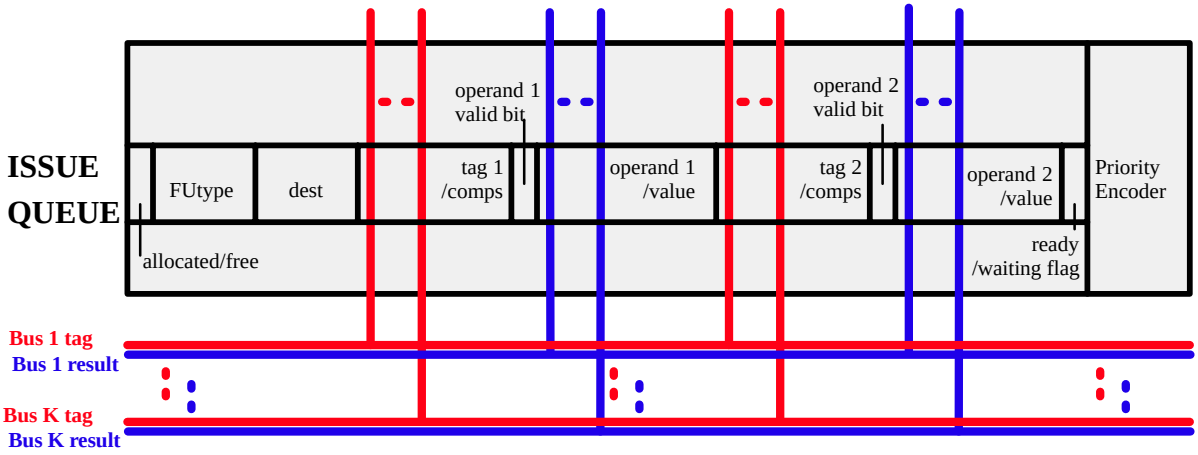


Figure 2–6. IQ structure for datapath where IQ entries hold instruction operand values (Dispatch–bound operand reads)

the associated input field and the corresponding source operand is marked valid. Once both operands are marked as valid, the wakeup logic of the IQ marks the corresponding instruction as ready for issue. The select logic then selects ready instructions for execution. Since multiple function units complete in a cycle, multiple forwarding buses are used; each input operand field within an IQ entry thus uses a comparator for each forwarding bus.

Examples of processors using this datapath style are the Intel Pentium Pro, Pentium II, IBM Power PC 604, 620 and the HAL SPARC 64 [MR 9X].

Figure 2–7 shows an alternative datapath. Here, even if input registers for an instruction contain valid data, these registers are not read out at the time of dispatch. Instead, when all the input operands of an instruction waiting in the IQ are valid and a function unit of the required type is available, all of the input operands are read out from the register file (or as they are generated, using bypassing logic to forward data from latter pipeline stages) and the instruction is issued. Figure 2–8 shows the IQ structure for this datapath style. In this case,

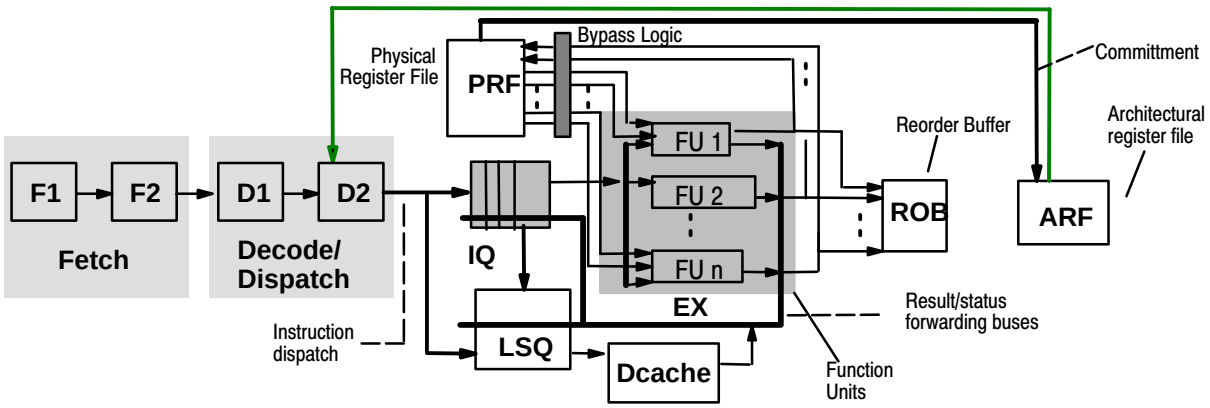


Figure 2–7. An alternative superscalar datapath with issue–bound operand reads

the IQ entry for an instruction is considerably narrower compared to the IQ entries of previously described datapath, since entries do not have to hold input register values. Examples of processors using this datapath style are the MIPS 10000, 12000, the IBM Power 3, the HP PA 8000, 8500, and the DEC 21264 [Bha 96, MR 9X].

The main sources of energy dissipation in the IQ are as follows:

a) Dispatch: Energy is dissipated in the process of establishing IQ entries for dispatched instructions in locating a free entry associatively and in writing into the selected entry.

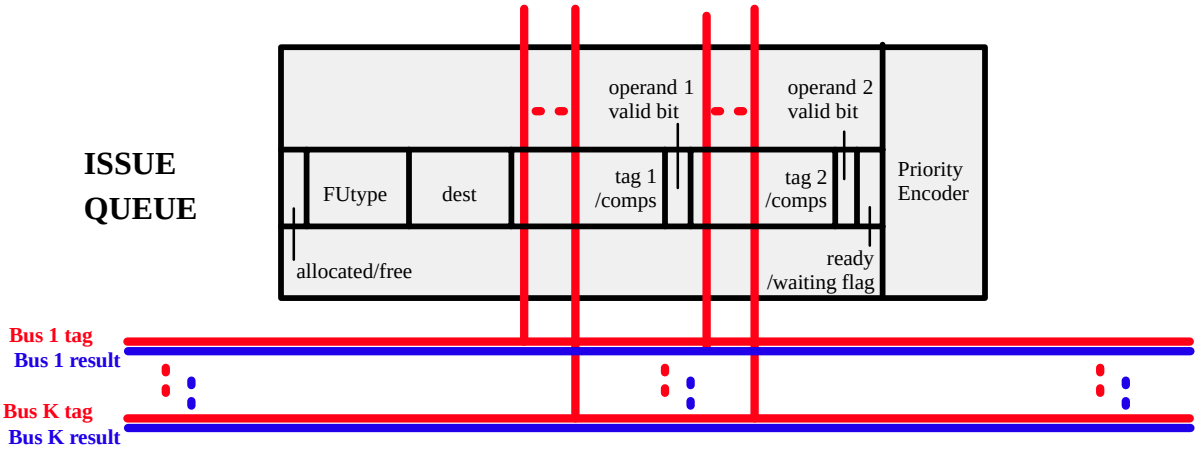


Figure 2–8. IQ structure for alternative superscalar datapath where IQ entries do not hold instruction operand values (issue–bound operand reads)

b) **Data Forwarding:** Energy is dissipated when FUs complete and forward the results and/or status information to the IQ entries. A significant fraction of this energy dissipation is due to the tag comparators used for associative matching to pick up forwarded data.

c) **Issue:** Energy is dissipated at the time of issuing instructions to the FUs in arbitrating for the FUs, enabling winning instructions for issue and in reading the selected instructions from the IQ. Here, additional dissipations occur to read the source operands from the IQ for the datapath with dispatch–bound operand reads.

d) **Branch Misprediction:** Energy is dissipated when flushing the IQ entries along the mispredicted paths.

The IQ for high–end superscalar processors can be quite sizable; for example, in the planned EV8 implementation of the Alpha processor, the IQ has 112 entries [Emer 01]. The energy dissipation within the IQ is a significant power dissipation component for modern superscalar CPUs, as much as 25% according to some estimates [FG 01].

Physical Register Files

Result repositories are used to hold the renamed registers. One possible implementation puts them at the end of function units as stand–alone structures. In this form, they are called as Physical Register Files (PRFs) or Rename Buffers (RBs). The typical size of a PRF is usually chosen as much as the number of reorder buffer entries to maximize the number of in–flight instructions. The typical implementation of PRFs is in the form of a multi–ported register file (RF). In a W–way superscalar machine, the PRF has the following ports:

- At least 2*W read ports for reading source operands for each of the W instructions dispatched/issued per cycle, assuming that each instruction can have up to 2 source operands.
- At least W write ports to allow up to W FUs to write their result into the PRF slots.
- At least W read ports to allow up to W results to be retired into the ARF per cycle.
- At least W write ports for establishing the PRF entries for co–dispatched instructions.

When PRFs are implemented as stand–alone structures, a list structure is also needed to keep track of free physical registers that are used by the decode stage. In some other implementations, the PRFs may be integrated into either reorder buffers or architectural register files. Those schemes completely eliminate the need for the free physical register list; however, they may introduce hot spots in the die. We discuss these implementation possibilities in detail, when we examine reorder buffer structure in the next section.

The main sources of energy dissipation in the IQ are as follows:

a) **Dispatch/Issue:** Energy is dissipated in the process of reading the input operands from the register files (ARF or PRF) at dispatch or issue time depending on the datapath style. This causes energy dissipation in the register files.

b) **Writeback:** Energy is dissipated when the FUs complete and write their results into the register files.

c) Retirement: Energy dissipations occur in the PRF at the time of committing instructions, as reads from the PRF to the ARF. No such dissipations occurs in datapath with combined ARF and PRF, since there is no data movement between PRF and ARF at the time of committment.

Load/Store Queue

For every instruction accessing memory, an entry is also reserved in the Load/Store Queue (LSQ) at the time of instruction dispatch. As the address used by a load or a store instruction is calculated, this instruction is removed from the IQ, even if the value to be stored (for store instructions) has not yet been computed at that point. In such situations, this value is forwarded to the appropriate LSQ entry as soon as it is generated by a function unit. All memory accesses are performed from the LSQ in program order with the exception that load instructions may bypass previously dispatched stores, if their addresses do not match. If the address of a load instruction matches the address of one of the earlier stores in the LSQ, the required value can be read out directly from the appropriate LSQ entry.

In a W–way processor, the load–store queue has the following ports and associative addressing logic:

- W write ports for setting up the LSQ entries for co–dispatched load and store instructions. The use of fewer than W ports may cause a noticeable performance degradation as the load and store instructions are frequently dispatched in bursts. One reason for this is the need to access multiple words in the stack memory for procedure calls and returns.
- K read ports for reading K instructions from the LSQ for the D–cache access, where K is the number of D–cache ports.
- Comparators, similar to the ones used within the issue queue, are deployed to compare the tags of the results produced by the execution units against the tags of the source data registers associated with the store instructions.

- Address comparators for performing the comparison of the load addresses against the previously dispatched store addresses (for load bypassing).

Energy dissipations taking place within the LSQ have the following components:

a) Dispatch: Dissipations occur in the LSQ at the time of setting up the entry for dispatched instructions.

b) Address Write: Dissipations occur when writing computed effective addresses into a LSQ entry.

c) Store Forwarding: Dissipations for forwarding the result of a pending store in the LSQ to a later load.

d) Data Forwarding: Dissipations for forwarding the data in a register to be stored to a matching entry in the LSQ.

e) Address Read: Dissipations that occur when initiating D–cache accesses from the LSQ.

f) Branch Mispredictions: Dissipations occur in clearing the LSQ on mispredictions. This involves clearing the valid bit of all LSQ entries.

2.1.3. In–Order Back End

This last stage of the datapath takes care of the retirement of the instructions. The instructions need to commit to architectural register files in order. This is important for having a precise processor state all the time. The Reorder Buffer (ROB) is the heart of the in–order back end.

Reorder Buffer and Register Files

A Reorder Buffer (ROB) is typically used to reconstruct a *precise state* – a state corresponding to the sequential program semantics. While the physical registers are updated when instructions complete, possibly out of program order, the ROB maintains results (in

some implementations) and updates the precise state – typically implemented as a set of *architectural registers* – strictly in program order. The ROB is essentially a FIFO queue, with entries set up in program order at the time of instruction dispatch, at the tail of the ROB, and retired to the architectural registers in program order from the head of the ROB. The ROB allows the processor to start from a precise state following an interrupt or a branch misprediction [SP 85].

The typical implementation of a ROB is in the form of a multi–ported register file (RF). In a W–way superscalar machine where the physical registers are integrated within the ROB, the ROB has the following ports:

- At least 2*W read ports for reading source operands for each of the W instructions dispatched/issued per cycle, assuming that each instruction can have up to 2 source operands.
- At least W write ports to allow up to W FUs to write their result into the ROB slots acting as physical registers in one cycle.
- At least W read ports to allow up to W results to be retired into the ARF per cycle.
- At least W write ports for establishing the ROB entries for co–dispatched instructions.

The actual number of ports can double if a pair of registers are used to hold double–width operands. Alternatively, double–width operands can be handled by using a pair of register files in parallel, each with the number of ports as given above – this approach is wasteful when double and single width operands are intermixed – as is typical. Complicated allocation and deallocation schemes have to be used if such wastages are to be avoided. For our studies we assume that two consecutive ROB entries are allocated for an instruction producing a double–precision result.

Figure 2–9 summarizes the port requirements for an ROB of a W–way superscalar processor. The widths (number of bits) for each port are also shown in this figure, where *p* is the number of bits in the physical register and *a* is the number of bits in the architectural

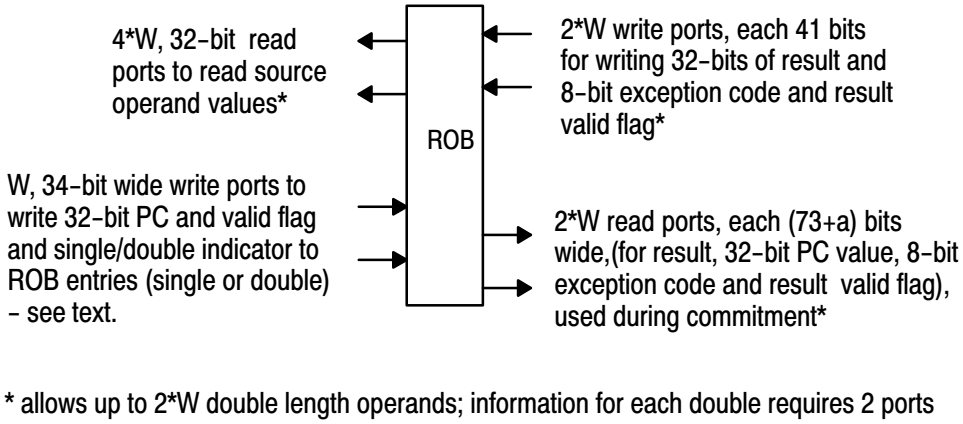


Figure 2–9. Complexity of a ROB in a W–way processor

register address. Two operand widths are assumed: 32 bits and 64 bits; 64–bit operands are generated into two 32–bit architectural registers. Each ROB slot is also capable of holding only a 32–bit result, so two consecutive ROB slots are used to accommodate a 64–bit result. Only one of the two consecutive entries for a 64–bit result will use the exception codes and PC–value fields of the entry.

At low clock frequencies, the number of ports needed on the ROB can be reduced by multiplexing the ports – this, alas, is a luxury not permissible in the designs that push the performance envelope. As the number of ports increases, the area of the RF grows roughly quadratically with the number of ports. The growth in area is also accompanied by a commensurate increase in the access time stemming from the use of longer bit lines and word lines, bigger sense amps and prechargers.

The large number of ports on the ROB results in two forms of penalties that are of significance in modern designs. The first penalty is in the form of the large access delay that can place the ROB access on the critical path; the second is in the form of higher energy/power dissipations within the highly multi–ported RFs implementing the ROB.

In some architectures, the physical registers are implemented as slots within the ROB entries and a separate register file – an architectural register file (ARF) is used to implement the architectural registers that embody the precise state. The Pentium III is an example of a processor that uses this specific datapath, which is depicted in Figure 2–10. Here, the

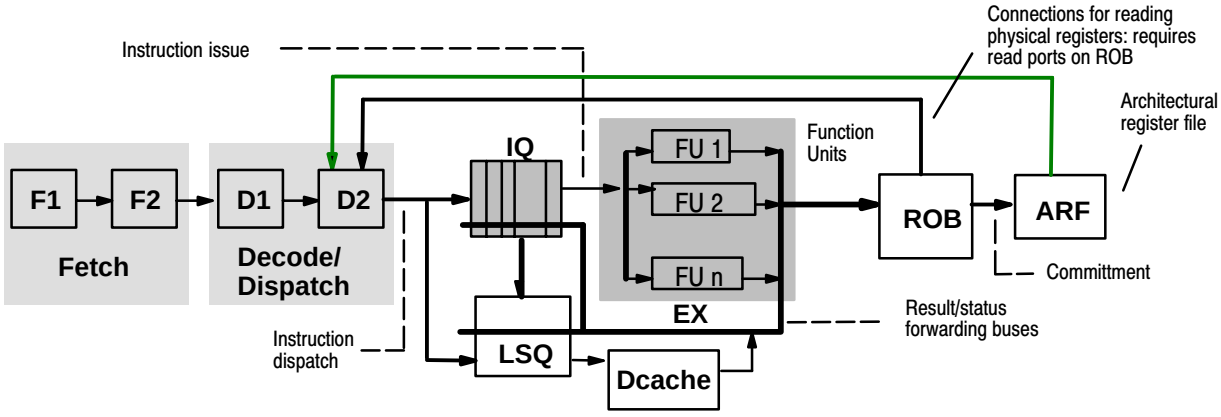


Figure 2–10. The superscalar datapath where ROB slots serve as physical registers

results produced by functional units (FUs) are written into the ROB slots and simultaneously forwarded to dispatched instructions waiting in the Issue Queue (IQ) at the time of writeback. To enable back–to–back execution of the dependent instructions, result tags are broadcast one cycle before the actual data. The result values are committed to the ARF at the time of instruction retirement. If a source operand is available at the time of instruction dispatch, the value of the source register is read out from the most recently established entry of the corresponding architectural register. This entry may be either an ROB slot or the architectural register itself. If the result is not available, appropriate forwarding paths are set up.

In some contemporary architectures, the architectural register file and physical register files can be merged. MIPS R10000, Alpha 21264 and Pentium 4 are some examples of this datapath style, which is depicted in Figure 2–11. The implementations that use a separate register file (the ARF) for holding committed values have several advantages over the scheme with a unified register file for holding both speculative and committed register

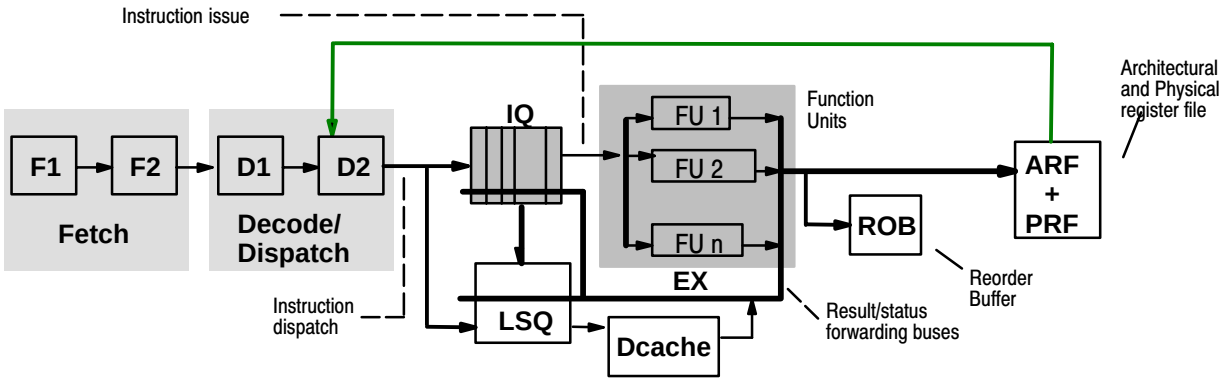


Figure 2–11. The superscalar datapath where architectural register file and physical register file are combined

values. First, if the ROB slots are used as physical registers, allocation and deallocation of physical registers are trivial – there is no need to maintain an explicit list of free physical registers, because the registers are allocated and deallocated in order as instructions are dispatched and committed. Second, only one register mapping table needs to be maintained. In contrast, the design with the unified register file often requires the use of two mapping tables. One is used to track the data dependencies in the course of instruction dispatching and the other maintains the retirement map, which is updated as instructions commit. The use of two mapping tables can be avoided by maintaining the old register mapping for destination register in the ROB entry and using this information to rollback to a precise state by walking through the ROB and undoing the modifications performed on the mapping table by all instructions on the mispredicted path. In such design, however, branch mispredictions may incur significant additional delay, as only a fixed (and small) number of instructions on the mispredicted path can be retrieved from the ROB for inspection in one cycle. Our experiments show the average performance (IPC) degradation of as high as 5% in this case. Third, if a separate architectural register file (ARF) is used to hold the committed register values, these values can be accessed quickly because of the small size of the ARF. Our measurements indicate that the percentage of such reads is quite large (in excess of 30% on

the average) and therefore delaying these reads even by one cycle has a detrimental effect on the overall IPC.

The design with the unified register file also has its advantages. Here, no data movement is involved in the course of instruction committment thus saving energy. Although the retirement mapping table needs to be updated after each committment, this is expected to be less energy–consuming event than the reading of the value from the ROB and writing this value into the ARF. Perhaps even more alarming feature of the architecture that uses the ROB as a storage for physical registers is the high ROB complexity and power dissipation. In addition, the accesses to such ROB for reading and writing physical registers stretch over several cycles in high–end designs. Therefore, it is not surprising that with the current trend to decentralization, multiclustering and the use of smaller components in general, the most recent superscalar implementations moved away from this scheme and instead used the unified register file to store both speculative and committed register values. If, however, appropriate mechanisms are designed to simplify the ROB and reduce its complexity, especially its access delay and power dissipation, a scheme that uses a separate ARF could again re–emerge as an attractive design choice for future processor generations.

In another variant, the IQ, ROB and the PRF are all merged into integrated structures such as RUUs or DRIS (as used in the Lightning Metaflow processor, [Pop 91]). Unfortunately, the long wire delays of these structures make such schemes unattractive.

Energy dissipations taking place within the ROB have the following components:

a) Dispatch: Dissipations occur in the ROB at the time of setting up the entry for dispatched instructions. Energy is dissipated at the time of writing the address of the instruction (PC value), flags indicating the instruction type (branch, store, register–to–register), the destination register id, and predicted branch direction (for a branch instruction) into the appropriate slot of the ROB. Overall, this dissipation is really in the form of energy spent in writing to the register file that implements the ROB.

b) **Commit:** Dissipations take place when ROB entries are committed. This dissipation is the energy spent in reading the register file that implements the ROB.

c) **Branch Mispredictions:** Dissipations occur in clearing the ROB on mispredictions. This involves clearing the valid bit of all ROB entries. Compared to all of the other ROB dissipation components, this is quite small.

When the Physical Register File is integrated into the ROB, the following additional energy dissipations occur:

d) **Register Read:** Dissipations occur when memory instructions are moved to the LSQ or when the valid data value for the most recent entry for an architectural register is read out.

e) **Register Write:** Dissipations take place when results are written to the ROB from the function units.

Chapter 3

Existing Techniques for Reducing Complexity and Power in Superscalar Microprocessors

As the high–end superscalar microprocessors become more energy–hungry and complexity–ineffective in the last decade, a large number of solutions targeting the reduction of microprocessor power and complexity have been proposed. These techniques include the microarchitectural level, circuit–level solutions as well as the compiler–oriented designs for developing new paradigms for power–aware processors as well as the efforts to make existing processors more power and energy efficient. This chapter overviews some of these mechanisms.

3.1. Microarchitectural Techniques

The Microarchitectural level techniques reduce the dynamic power consumption by reducing the activity factor (α) and the switching capacitance (C_{load}) in the Equation (1–2). These techniques can also reduce leakage power by either putting some portions of the on–chip storage structures in a low–power stand–by mode or even completely shutting off the power supply to these portions. When a processor frequency is limited by thermal issues,

saved power and energy can be used to increase frequency and, hence, the performance [SRG 02].

The proposed solutions target almost all the stages and the structures in a processor pipeline such as the front–end of the processor's pipeline, the instruction scheduler (the issue queue), the register files (including the ROB that serves as the repository of physical registers), the instruction and data caches and the execution units.

3.1.1. Front–End Throttling Techniques

The front–end of the pipeline does not always supply valuable resources to the pipeline, since the speculation may not give perfect results all the time. Sometimes, it is beneficial to throttle the front–end of the pipeline and temporarily prevent the new instructions from entering the pipeline. In this way, the activity factor (α) in the front–end of the pipeline is reduced resulting in power savings. In [MKG 98], Manne et.al. proposed pipeline gating to avoid speculation beyond the branches with low prediction accuracy. They proposed the branch confidence estimators to assess the accuracy of each branch prediction. These confidence estimators are used to decide if the processor is likely to fetch and execute the instructions along the wrong path. The counter of unresolved low confidence branches is used to determine when to gate the pipeline and for how long. The counter is incremented when a low confidence branch is encountered at the time of decode, and decremented when that branch is resolved at the time of writeback. When the number of low confidence branches exceeds the predetermined threshold, the fetching and decoding stalls. The results showed that the gating intervals are typically short, in the order of 2 to 4 processor cycles, allowing to achieve significant power savings with small impact on performance. The study also covered a number of confidence estimators with a range of implementation costs. The authors report a 38% reduction in wrong–path instructions with a marginal performance loss (approximately 1%).

In [BM 01b], Baniasadi and Moshovos proposed a number of instruction flow–based heuristics to throttle the front–end of the pipeline in a cycle–by–cycle granularity. These heuristics are: 1) Decode/Commit Rate (DCR): If the decode rate exceeds the commit rate in the pipeline with a certain factor (in the study, this factor was 3), it is assumed that sufficient parallelism already exists in the pipeline, and as a result, the authors triggers the throttling for the next three cycles. 2) Dependence–Based (DEP): If the number of true data dependencies over registers in a group of co–decoded instructions exceeds the predetermined threshold, the front–end is stalled for the next three cycles. The best results were obtained if the threshold was set to be half of the decode width. The intuition behind this approach is that a high number of dependencies is an indication of a long and probably critical computation path. Consequently, it is unlikely that injecting additional instructions as quickly as possible will have a significant impact on performance. 3) Adaptive (ADAP): The results show that DCR works best for some benchmarks whereas DEP works best for some others. The adaptive method attempts to exploit the best of both techniques. It selects the DCR approach when the commit rate is high and selects the DEP approach when the commit rate is low.

In [KSB 02], Karkhanis et.al. proposed throttling of the fetch stage depending on the number of in–flight instructions. A single counter, maintaining the total number of instructions in the pipeline is used to control throttling. If the counter exceeds a predetermined threshold, the fetch stage is throttled. The threshold is dynamically adjusted at periodical intervals through performance monitoring. The threshold is set to the minimum possible number that does not limit the performance. This is accomplished through so–called “tuning cycle”, where the threshold is first set to the maximum allowable number of instructions, as determined by the window size, and the IPC performance is recorded after an interval of 100K instructions. In the next interval, the threshold is set to the minimum number of in–flight instructions (e.g. 8), and then the threshold is incremented by eight until the performance for an interval is within a predetermined bound of the performance with the

maximum number of in–flight instructions. The threshold is then kept at this value until the program phase change occurs, which is indicated by either the changes in the IPC or in the number of dynamic branches. The overhead of implementing this algorithm in hardware is certainly non–negligible. The alternative is to implement it is a low–level software. Such "just–in–time" instruction delivery mechanism results in energy savings for two reasons: first, fetching of some instructions on the mispredicted path that would have been fetched in the traditional design, may never occur, and 2) instructions spend fewer cycles in the issue queue, resulting in the lower utilization of the issue queue and in fewer comparators being activated for instruction wakeup in an average cycle.

In [BKA+ 03], where Buyuktosunoglu et.al. introduced a technique to combine fetch gating and adaptive issue queue resizing to aggressively reduce the front–end power. The scheme proposes a fetch gating scheme that attempts to match the size of the instruction window resident in the issue queue to application ILP characteristics, while keeping the utilization of the queue high enough to avoid back–end starvation. The authors distinguish the parallelism in the program into close parallelism, where most of the issued instructions are located closer to the head end of the ROB, and distant parallelism, where most of the issued instructions are located near the tail end of the ROB. Fetch gating occurs when the occupancy of the issue queue is high and the program exhibits close parallelism. In addition, the unused issue queue partitions are deactivated to further reduce power consumption. The reported results indicate a 20% better energy–delay compared to the flow–based mechanisms which use the disparity between the dispatch and commit rates to gate the instruction fetching on a cycle–by–cycle basis.

Selective Throttling, a way of triggering different power–aware techniques, is proposed by Aragon et.al. in [AGG 03]. According to the confidence level assigned to each branch prediction, different processor blocks are dynamically throttled: fetch unit, decode unit or selection logic (from more to less aggressive). Aggressive throttling is applied for those branches with high probability of being mispredicted (at the expense of reducing

performance if the branch hits). On the other hand, when the estimator is not sure about the correctness of the prediction, less aggressive techniques, both in terms of power reduction and performance degradation, are applied. The selection throttling is implemented by maintaining an additional bit within each entry of the issue queue, indicating if the entry should be considered for issue. Results for *Selective Throttling* obtain average enery savings of 13.5%.

3.1.2. Energy–Efficient Branch Predictors and Renaming Logic

In [PSZ+ 02], Parikh et.al. introduced banking and the usage of branch prediction probe detector as two techniques that reduce branch predictor power dissipation without harming accuracy. Banking lowers the switching capacitance of the bitlines by reducing the active portion of the predictor. The Prediction Probe Detector (PPD) eliminates unnecessary BTB and predictor accesses by using pre–decode bits. These bits are used to recognize when the lookups of the BTB and/or the direction predictor can be avoided. As a result, the authors claim 5–6% savings in overall processor power and energy dissipation.

In [BM 02], Baniasadi and Moshovos introduced Branch Predictor Prediction (BPP) as a power–aware branch prediction technique for high performance processors. The proposed predictor reduces the branch prediction power dissipation by selectively turning off and on two of the three tables used in the combined branch predictor. The authors claim significant energy savings by exploting a) the temporal locality amongst branch instructions, and b) the high predictability in their usage of sub–predictors. On average, BPP reduces power dissipation by 28% compared to a non–banked and by 14% compared to a banked conventional combined predictor. This comes with a negligible power degradation.

Hu et.al. proposed the use of the line decay strategies to reduce the leakage power within the branch predictors [HJS+ 02]. All rows of predictor entries that have not been used during a pre–specified interval are assumed to have decayed and are deactivated by turning–off the

power supply to the cells within the row. The more detailed discussion of the decay scheme, as applied to caches, is given in Section 3.1.5. The authors report less than 1% drop in prediction accuracy and 40–60% leakage energy savings in the branch predictor.

Moshovos proposed two techniques to reduce the power consumption of the register alias table [Mosh 02]: 1) The number of read and write ports needed on the register alias table is reduced by exploiting the fact that most instructions do not use the maximum number of source and destination register operands. Additionally, the intra–block dependence detection logic is used to avoid accessing the register alias table for those operands that have a RAW or a WAW dependence with a preceding, simultaneously decoded instruction. 2) The number of checkpoints that are needed to implement aggressive control speculation and rapid recovery from branch mispredictions is reduced. This is done by allowing out–of–order control flow resolution as an alternative to conventional in–order resolution, where the checkpoint corresponding to a branch can not be discarded till this branch itself as well as all preceding branches are resolved. The authors found that the proposed methods can reduce overall register alias table power by 42% with less than 2% average performance penalty.

We proposed two complementary techniques to reduce the energy dissipation within the register alias tables of modern superscalar microprocessors [KEPG 03]. The first technique use the intra–group dependency checking logic already in place to disable the activation of the sense amps within the RAT when the register address to be read is redefined by an earlier co–dispatched instruction. The second technique extend this approach one step further by placing a small number of associatively–addressed latches in front of the RAT to cache a few most recent translations. Again, if the register translation is found in these latches, the activation of the sense amps within the RAT is aborted. Combining the two proposed techniques results in a 30% reduction in the power dissipation of the RAT. The power

savings comes with no performance penalty, little additional complexity and no increase in the processor's cycle time.

3.1.3. Energy and Complexity Reduction in the Issue Logic

One popular approach to power minimization in the issue queue has to do with splitting the issue queue into multiple banks. Buyuktosunoglu et.al. proposed such a design in [BAB+ 02]. They used the two higher–order bits of the tag to determine which bank the instruction needs to be placed into. When a destination tag is broadcast, only one of the four CAM banks is enabled for comparison, thus saving energy.

Several mechanisms for the issue queue energy reduction are discussed by Ghose in [Gh 00]. These include partitioning the monolithic issue queue into several smaller queues based on the instruction type. As a consequence of this partitioning, the number of read and write ports can be reduced without noticeable impact on performance. Further enhancement to the partitioned scheme is to use the customized entry format within each type of the issue queue. A third scheme involves encoding the widths of the operands stored within the issue queue in order to reduce the number of bitlines that need to be driven.

Another popular approach to power minimization in the issue queue is the dynamic resizing of the queue based on the requirements of the application. Several resizing techniques have been proposed along these lines. Specifically, in [BSB+ 00] and [BAS+ 01], Buyuktosunoglu et.al. explored the design of an adaptive issue queue, where the queue entries were grouped into independent modules. The number of modules allocated was varied dynamically to track the ILP. Power savings was achieved by turning off unused modules.

A FIFO–style issue queue that permitted out–of–order issue but avoided the compaction of vacated entries within the valid region of the queue to save power was introduced by

Folegnani and Gonzalez in [FG 01]. The queue was divided into regions and the number of instructions committed from the most–recently allocated issue queue region in FIFO order (called the "youngest region") was used to determine the number of regions within the circular buffer that was allocated for the actual extent of the issue queue. The number of regions allocated was incremented by one periodically to avoid any performance penalty. In addition, at periodic intervals, a region was deactivated to save energy/power if the number of commits from the current youngest region was below a threshold. The complementary technique, proposed in [FG 01], disables the wake–up of ready issue slots as well as the wake–up of the slots that maintain the ready operands. This is accomplished by disabling the precharging of the dynamic pull–down comparators that are associated with the empty slots or slots that hold the ready operands.

Bahar and Manne studied the multi–clustered Alpha 21264 processor with replicated register files [BM 01c]. The dispatch rate was varied between 4, 6 and 8 to allow an unused cluster of execution units (including per–cluster issue queues) to be shut off completely. Significant power savings within the dynamic scheduling components were reported with a minimum reduction in the IPC.

In [BBS+ 00], Brooks et. al. suggested that the use of comparators that dissipate energy on a tag match could significantly reduce the power dissipated in the issue queue. The actual designs of such dissipate–on–match comparators were introduced and evaluated for the use within the issue queue in [KGPK 01, PKE+ 03, EGK+ 02]. Significant energy savings were achieved because in absolute majority of the cases, the tags used for instruction wakeup are mismatching rather than matching. In addition, the design of [EGK+ 02] is slightly faster than the traditional comparator, resulting in a win–win situation.

In [PK+ 03], we propose a series of energy efficient long comparator designs for superscalar microprocessors. Our designs combine the use of 8–bit blocks built using traditional comparators with 8–bit blocks built using dissipate–on–match comparators. We

then evaluate the use of these circuits within the address comparison logic of the load–store queues. We found that for the same delay, the hybrid design consisting of one dissipate–on–match and three traditional 8–bit comparators is the most energy efficient choice for the use within the load–store queue, resulting in 19% energy reduction compared to the use of four traditional 8–bit blocks. The results presented in this paper can be easily extended to the TLBs, highly–associative caches and the BTBs.

Huang et.al. saved energy in the issue queue by using indexing to only enable the comparator at the single instruction to wake up [HRT 02]. The proposed technique exploit the fact that the majority of the produced register values are consumed at most once. In their scheme, when an instruction is inserted in the instruction window, and any of its source registers are not ready, the entry of the producer instruction in the window is identified by looking up the information in the extended RAT. Then, in that issue queue entry, the pointer to the consuming instruction is stored. When the producer instruction completes, it only awakens one consumer, whose location is directly stored within its issue queue entry. In rare cases, the mechanism reverted back to the usual tag broadcasts, when more than one instruction needed to be awakened.

In [EA 02], Ernst and Austin explored the design of the issue queue with reduced number of tag comparators. They capitalized on the observation that most instructions enter the instruction window with at least one of the source operands ready. These instructions were put into specialized entries within the issue queue with only one (or even zero) comparators. The authors of [EA 02] also introduced the last tag speculation mechanism to handle the instructions with multiple unavailable operands. Only the tag of the operand predicted to arrive last was used for the instruction wakeup. The primary advantage of the last–tag scheduler is that more than half of the comparator load on the result tag bus is eliminated.

This results in faster schedulers and reduction of power in the order of 10–20% depending on the number of entries in the issue queue.

The approach of [EA 02] assumes that all instructions following the instruction whose last–tag prediction was misspeculated are rescheduled, irrespective of whether they are actually dependent on the mispredicted instruction. A higher performance can be realized by using selective replays [Hinton+ 01], where only the instructions that are effected by the mispredicted instruction are rescheduled. In the tag elimination scheme of [EA 02], removing tag matching logic prevents early–arriving operands from participating in the dependency propagation needed to implement selective replays. Another mechanism for reducing the capacitance of the wakeup bus, not exhibiting this limitation, was recently proposed in [KL 03] by Kim and Lipasti. The scheme is based on sequential wakeup, where half of the comparators reside on the fast wakeup bus (wakeup of the children occurs in the same cycle when their parent is selected) and the other half of the comparators reside on a slow wakeup bus such that the wakeup of the sources connected to this bus occurs in the next cycle. Just like in the scheme of [EA 02], the last–tag prediction is used to identify which of the two sources is positioned on the fast bus for wakeup. The technique is motivated by the observation that it is rarely the case when both sources operands become ready in the same cycle. That, coupled with the use of the accurate last–tag predictors, results in very low performance degradation and faster issue logic. The effect on power consumption is not discussed in [KL 03], but it is reasonable to expect that the overall power will remain nearly unchanged, since the number of comparators is not reduced.

Some recent efforts have attempted to reduce the complexity of the issue queue. In [PJS 97], Palacharla et.al. proposed a dependence–based microarchitecture. In this scheme, the issue queue is implemented through several FIFOs, so that only the heads of each FIFO need to be examined for issue. Such an approach facilitates a faster clock while exploiting similar levels of parallelism to the traditional design. The FIFOs used to replace the traditional monolithic issue queue essentially store the chain of dependent instructions, so that an

instruction is placed in the same FIFO, where its parent resides. If an instruction begins a new dependency chain, a new FIFO is allocated. Specifically, the following heuristics are used in [PJS 97] to steer the instructions to FIFOs. If an instruction requires the allocation of a new FIFO, but no such FIFO exists, then the dispatching of instruction blocks leading to performance degradation.

The technique for reducing the issue queue complexity as proposed by Canal and Gonzalez in [CG 00] exploits the fact that the majority of operands in a typical application are read at most once. Instructions that have all source operands ready at the time of decoding are dispatched into the ready queue (one such queue is assumed to be in place for each FU), from where they are issued for execution in program order. Instructions, whose operands are not ready at the time of decode, but which were the first consumers of required source register values, are dispatched into a separate data structure called First–use table. From there, upon the availability of both sources, they are eventually moved to the appropriate ready queue. The First–use table is indexed by the physical register identifier of each source, which allows the produced results to be supplied directly to the waiting instructions without any associative tag comparison. Instructions, that are not the first consumers of their source registers are dispatched into a small out–of–order issue queue and traditional issue mechanism is used for those instructions to avoid too much performance degradation. The scheme of [CG 00] reduces the amount of associative lookup in the issue process by limiting it only to the instructions within the small out–of–order queue. This results in significant reduction of the complexity of the control logic, but the effect on power dissipation is not clear, as the additional structures used in [CG 00] certainly have some non–negligible power overhead. The scheme also requires to perform a lookup of the First–use table at the time of dispatch. This will almost inevitably result in the use of a slower clock or the use of multiple dispatch stages, which can effect the overall performance adversely. The scheme was generalized in [CG 01] through the use of N–use table instead of First–use table. The second scheme discussed in [CG 00] attempts to reduce the complexity of associative

wakeup logic through the use of scheduling scheme which exploits the fact the latencies of most instructions are deterministic. When the availability time of all operands is known, the instruction is moved to the issue queue. The issue queue is very simple, it is a circular buffer where for each entry there are as many instruction slots as the issue width and each entry corresponds to one cycle. Thus, the issue queue keeps the instructions in order that they will be executed and separated by a distance that ensures that the dependencies will be obeyed if at every cycle the processor issues the instructions from the head of the issue queue. The instruction location in the issue queue is determined by calculating the maximum of the availability time of the source operands and taking the difference between this value and the current cycle to indicate the displacement with respect to the head pointer. Once the instruction is placed in the issue queue, the time when its destination will be produced is computed and recorded in the register availability table for the use by the subsequent instructions. It is not clear, however, how the resource conflicts are taken into account. A modification to this scheme is introduced in [CG 01], where a separate queue is used to keep the instructions when they are expected to be issued, but the source operands are not yet available. This is a result of predicting the latencies of memory operations assuming that they hit in the cache. If an instruction at the head of the issue queue cannot be issued, because it is dependent on the memory access which has not finished yet, the instruction is moved to the delayed issue queue. Instructions are then issued from both the delayed issue queue and the regular issue queue once the operands are actually ready. Again, the implications of this scheme on the overall power consumption of the issue logic are not discussed.

A way to build large instruction windows without sacrificing delay and increasing complexity was proposed in [LKL+ 02]. The scheme moves the instruction dependent on the long latency operations (e.g. cache misses) out of the conventional, small–size issue queue to a much larger waiting instruction buffer (WIB). These instructions wait in the WIB until they are reawakened by the completing load instructions, at which point they are moved

back to the issue queue and can be considered for issue. Bit–vectors also provided to identify particular load instructions when handling multiple outstanding load misses.

In [BIW+ 02], another mechanism is proposed in the form of hierarchical windows. The new scheduler contains two levels of scheduling windows. The first is a large, slow window and the second is a small, fast window. The slow window provides an ILP–extraction capacity, while the fast window is kept intentionally small to maintain high frequency. A heuristic is built into the scheduler logic, which implicitly identifies latency tolerant instructions. Latency tolerant instructions are issued for the execution from the slow window, while latency critical instructions are issued from the fast window. Each scheduling window has a dedicated execution unit cluster, which simplifies the bypass network. All instructions are first dispatched into the slow scheduling window. A separate logic called mover then removes three oldest non–ready instructions from the slow window and inserts them into the fast window, provided that the latter has sufficient space. A fast window is capable of capturing the data produced by an execution unit in any cluster and designed to support the back–to–back execution of dependent instructions. The requirement of back–to–back to execution is not imposed on the slow window, as it maintains mostly non–critical instructions.

Another scalable issue queue design was introduced in [RBR 02]. The design divides a large queue into small segments that can be clocked at high frequencies. Dynamic dependence–based scheduling is used to promote instructions from a segment to a segment until the instruction reaches the final segment, from where the execution actually commences. The flow of instructions across the segments is governed by a combination of data dependencies and predicted operation latencies.

In [STT 01], Seng et.al. exploit the instruction criticality information to reduce the energy requirement of the issue queue. This is accomplished by observing that the scheduling needs of critical and non–critical instructions are quite different, and thus the

instructions are split into two groups based on their criticality to performance. Splitting the critical and non–critical instructions provides three opportunities for power reduction. First, splitting one queue into two reduces the complexity of the selection logic and also lowers the bitline capacitances. Second, the queue that keeps critical instructions can be further simplified by issuing the instructions from it in–order. Third, the speed of the out–of–order queue can be reduced, because it maintains non–critical instructions. The overhead of the scheme is in the need to maintain the instruction criticality predictor.

In [MS 01], instructions are reordered by a prescheduler so that they enter the issue queue in the data–flow order, i.e. the order of execution assuming unlimited execution resources, taking into account only data dependencies. The instructions wait in a prescheduling buffer until they can enter the issue queue. If the predicted data–flow order is very close to the actual execution order, then the issue queue can be kept very small as it is relieved from the task of buffering instructions that are not ready for execution.

In [OG 98], Onder and Gupta proposed similar scheme to chain the instructions according to the dependencies among them. Their technique builds the data dependence graph dynamically and limits the number of instructions that each instruction can wake up. They assumed that all the execution unit latencies are known and not variable, which simplifies the issue logic.

Stark et.al. discuss how to pipeline the instruction scheduling logic without inhibiting the capability to execute dependent instructions in back–to–back cycles [SBP 00]. They propose to maintain the information about the instruction's grandparents and speculatively wake up an instruction if its grandparents are selected for execution. The complementary technique, a select–free instruction scheduling logic, is proposed by Brown et.al. in [BSP 01]. The scheme essentially enables all ready instructions for issue, capitalizing on the observation that it is very rarely the case that the number of instructions ready for issue in one cycle exceeds the processor's issue width.

3.1.4. Energy and Complexity Reduction in the Register Files

A way of reducing the complexity of the physical register file by using a two–level implementation, along with multiple register banks is described in [BDA 01] for a datapath that integrates physical register file and architectural register file in a structure separate from the ROB (see. Figure 2–11). Their two–level register file design exploits locality of reference using an allocation policy that leaves values that have potential readers in the level one (L1) register file. The rest of the values are transferred into the level two (L2) register file. This cache–like L1 register file has a faster access time, since it has only a small number of entries. An additional structure is necessary to synchronize two register files. Energy savings are achieved through the use of a minimally–ported multi–banked design. The register file is divided into multiple banks such that each bank has only one read and one write port. To handle the limited write bandwidth, the arbitration logic is required before FUs can write their results onto the result bus. Additional registers also have to be provided to buffer results that could not gain the access to the result bus due to the bank conflicts, if, say, more than one instruction writes to the same bank in a cycle. Also, if an instruction has both of its sources in the same bank, then mechanisms are provided for it to read the first operand and save it in the latch in front of the function unit. The selection logic of the issue queue is slightly complicated in this design, as it has to take into account the contention for the ports and the function units.

Similarly, in [CGV+ 00], Cruz et.al. proposed the use of a two–level hierarchical inclusive register file organization (where the second level contains all register values). Results are always written into the second–level register file bank, and optionally, if they are expected to be used, to the first–level bank. The authors proposed several heuristics to pre–allocate those registers that are needed in the near future to the small first–level register file bank. First heuristics was to cache in the smaller bank only those register values that were not read off the bypass network. In this case, results that are read from the bypass

network are only written into the large bank, and the results that are not read off the bypass are written into both banks when they are generated. The second heuristics only writes a produced register value into the smaller bank if the value is needed as a source by an instruction that has all other operands ready.

Various register file partitioning schemes for transport–triggered architectures were studied in [JC 95]. In particular, they studied the various register assignment methods across the multiple register file banks. The simplest methods they considered were vertical (first half of the addresses is mapped to bank 0 and the rest is mapped to bank 1) and horizontal (bank is determined as register address modulo the number of banks) distribution. In addition, they also analyzed several heuristics to improve upon the horizontal distribution. The data independence heuristic maps the variables whose transports may be scheduled in the same cycle to different register file banks. The results showed that distributing 24–entry register file with 4 read and 4 write ports into four partitions, each with 1 read and 1 write port, and applying the above heuristics allows to avoid any significant performance loss due to partitioning.

Replicated register files in a clustered organization have been used in the Alpha 21264 processor [Kessler 99] to reduce the number of ports in each replica and also to reduce delays in the connections in–between a function unit group and its associated register file. Additional logic is needed in this design to maintain the coherence of the copies.

The idea of caching recently produced values for register file energy reduction was used in [HM 00]. At the time of instruction writeback, the FUs write results into a FIFO–style cache called Value Aging Buffer (VAB). The register file, holding both speculative and committed register values, is updated only when entries are evicted from the VAB. In many situations, the step of writing the result into the large multi–ported register file can be avoided, because the physical register may be deallocated during its residency in the VAB. When the required value is not in the VAB, a read of the register file is needed, requiring at

least some read ports for reading the source operands. Unless a sufficient number of register file ports is available or the number of entries in the VAB is sufficiently large, the performance degradation can be considerable due to the sequential access to the VAB and the register file. The authors claim 30% energy reduction in the register file at the cost of 5% average performance loss. Since the recent results may not necessarily have been found in the register file, misprediction handling and interrupt handling with the VAB are somewhat involved; misprediction and interrupt handling required selective lookup of values from the VAB for generating a precise state.

In [BTME 02], Borch et.al introduces a similar scheme with *forwarding buffers* for a multi–clustered Alpha–like datapath. These buffers retains the results for instructions executed in the last nine cycles. Nine stages of forwarding logic are employed to supply these results to dependent instructions. The authors further extend this idea by using per–cluster register file caches (CRC – Clustered Register Cache) to move the register file access out of the critical path and replace it with the faster register cache access. Each CRC only stores operands required by instructions assigned to that cluster and operands needed by a dependent instruction that is unlikely to get these operands by other means. While the primary goal of the work of Borch et.al. is to improve the performance, the complexity of the datapath is certainly increased.

In [ZK 00], a multi–clustered superscalar datapath organization is explored with the explicit goal of reducing the overall energy requirements of the datapath. Here, non–replicated, smaller register files with fewer ports are allocated to each cluster of function units. The cluster to which an instruction is assigned is identified through the use of fairly easy–to–implement heuristics. Additional logic is employed to disambiguate memory references across the clusters and to predict the cluster to which a load/store instruction is to be assigned. Extra logic is also needed to perform accesses to registers in remote clusters. Data caches have to be multi–ported to support the distributed cluster organization efficiently. It is unclear how these additional artifacts/changes and the wire

dissipations in such a decentralized datapath impact the overall power dissipation. There is also a non–negligible area penalty in this scheme. Finally, the multi–cluster scheme of [ZK 00] requires a fair amount of redesign of the datapath.

A scheme for reducing the number of register file ports, for both reads and writes, was recently proposed by Park et.al. in [PPV 02]. For reads, they proposed a bypass hint to reduce register port requirements by avoiding unnecessary register file reads for cases where values are bypassed. Bypass hint is based on the fundamental observation that when a consumer's source operand is woken up in the issue queue by the producer's result broadcast, the consumer often issues before the producer writes back, thus obtaining the value generated by the producer off the bypass network. The scheme uses one extra bit, called the bypass bit, per source operand in the issue queue slot. At the time of instruction issue, the bypass bit indicates whether an operand has been woken up while the instruction was waiting in the issue queue or whether the operand was ready even before the instruction entered the issue queue. The selection logic sees the bypass bits and selects the instructions so that the number of read ports needed do not exceed the number of ports available. Bypass hint mispredictions result in some performance degradation due to the need to stall the instruction issue.

To reduce the number of ports needed for the writebacks, Park et.al. proposed the use of decoupled rename, which separates the tagging of operands for dependencies from the actual register assignments. While the former is still performed during the renaming, the latter is delayed till the time of writeback, which makes it possible to make non–conflicting port assignments across multiple register banks. The virtual tags, used to maintain the dependencies, are stored in the register alias table and are used for wakeup and bypass condition. The physical tags are assigned just before the instruction enters the writeback stage. Instructions in writeback update the physical tag table with their physical tag, so that later consumer instructions know which physical register holds their value. Additional pipeline stage is needed to access the physical tag table, just before the access to the register

file. This introduces additional branch misprediction and load–miss penalty. The use of delayed register allocation is similar in spirit and in implementation to the earlier proposed schemes of [WB 96] and [GGV 98].

The speculative nature of the bypass hint proposed in [PPV 02] requires several additional cycles for bypass misprediction recovery, thus possibly harming the performance. In [TA 03], Tseng and Asanovic introduced an alternative design of the multi–banked register files, where they rely on a more conservative, non–predictive bypass hint to avoid accessing the register file for the operands that are read off the bypass network. In their scheme, the bypass bit is loaded with the result of the wakeup tag match in every cycle. If an instruction is selected in the same cycle that a tag match caused it to wake up, the bypass bit will be set indicating that the value is obtainable from the bypass. If the instruction is not selected, the bypass bit is cleared. The scheme of [TA 03] also avoids complex arbitration logic by using a separate pipeline stage (following the selection and preceding the register file read), which detects bank conflicts and reschedules the instructions that are the victims of the bank conflicts, as well as the prematurely issued dependents of such instructions.

Another way of reducing register file's access delay and power consumption was proposed in [STR 02], where Seznec et.al. introduced Register Write Specialization, forcing distinct groups of execution units to write only to distinct subset of the physical register file, thus limiting the number of write ports on each individual register. They also proposed to combine Register Write Specialization with Register Read Specialization for clustered superscalar processors. This limits the number of read ports on each individual register and simplifies both the wake–up logic and the bypass network. Some extra hardware and/or a few extra pipeline stages are needed for register renaming and also more physical registers are needed in this architecture.

In [SRG 02], Savransky et.al. proposed a technique for reducing the energy of the reorder buffer, which also serves as a repository of physical registers, as implemented in P6

microarchitecture. Their scheme, called Lazy Retirement, delays the copying of the value from the ROB to the architectural register file until the ROB entry is reallocated for a new instruction. In many situations, a new retired instruction invalidates this register before it is needed to be copied. According to the simulation results, more than 75% of all data movement in the course of instruction commitment can be eliminated using this scheme, this reducing the reorder buffer energy. The overhead of the scheme is in the need to maintain additional register mapping table to keep track of the most recent non–speculative location for each architectural register. We defer a more detailed discussion of this work until Chapter 6, where the comprehensive comparison of the scheme of [SRG 02] with the related technique proposed in this thesis is offered.

In [PKEG 03], we propose a technique to effectively remedy some of the principal inefficiencies associated with the datapaths that use separate register file for storing committed register values. Our scheme isolates the short–lived values in a small dedicated register file avoiding the need to write these values into the ROB (or the rename buffers) and later commit them to the architectural register file. With minimal additional hardware support and with no performance degradation, our technique eliminates close to 80% of unnecessary data movements in the course of writebacks and commitments and results in the energy savings of about 20–25% on the ROB or the rename buffers. For considered processor configurations, this roughly translates to about 5% of the overall chip energy savings for the datapath, where the physical registers are implemented as the ROB slots, and to about 2–3% of the overall energy savings if physical registers are maintained in a separate set of rename buffers.

An ISA and microarchitecture for instruction–level distributed processing is proposed in [KS 02] by Kim and Smith. The ISA has hierarchical register files with a small number of accumulators at the top. The instruction stream is divided into chains of dependent instructions, called strands, where intra–strand dependencies are passed through the

accumulator. The general–purpose register file is used for communication between strands and for holding global values that have many customers.

Several schemes for reducing the energy of the register file in a simple 5–stage pipeline are discussed in [Tseng 99]. The first technique, called precise read control, adds an opcode pre–decoder prior to the wordline drivers in the register file to disable the access to the register file for the instructions that do not have register sources. If an instruction only has one register source, only one register file access is enabled. The second technique, called bypass skip, avoids the reading of the register file if the value is obtained through bypassing. This techniques determines the RAW dependency before the register file access stage and disables the read from the register file if it is determined that the value will be bypassed. The additional pipeline stage may be needed if the hazard detection logic has a delay of more than half a cycle. The next method separates the register 0 (which always contains the value 0 in MIPS architecture) from the register file and provides this value through the bypass network, instead of reading it from the register file. This also slightly reduces the size of the register file, and hence, the bitline capacitance. Another method uses the inverted bitlines, capitalizing on the observation that the majority of bits stored in the register file are zeroes. This method reduces the switching activity for the single–ended bitlines used for reading. The last method proposed in [Tseng 99] splits the bitlines into two partitions – one (small) that contains the popular registers and one that contains all other registers. The two segments are isolated from each other by the transmission gate, so that only one segment is accessed, thus reducing the effective switching capacitance of the bitline.

In [KEP+ 03, KPE+ 04], we describe several techniques to reduce the complexity and the power dissipation of the ROB. We introduce the conflict–free ROB distribution scheme, where the conflicts over the use of the write ports are eliminated by allocating a small separate FIFO queue for holding the speculative results for each functional unit. All conflicts over the read ports are eliminated by removing the read ports for reading out the source operand values from the distributed ROB completely and using the combination of

a small set of associatively–addressed retention latches and late result forwarding to supply the results to the waiting instructions in the issue queue. Our designs result in extremely low performance degradation of 1.7% on the average across the simulated SPEC 2000 and significant reduction in the ROB complexity and power dissipation. On the average, the ROB power savings of as high as 49% are realized.

3.1.5. Energy Reduction in Caches

Ghose and Kamble proposed several techniques for dynamic power reduction in caches [GK 99]. First, they proposed the use of multiple line buffers, where the most recently accessed cache sets were buffered to avoid energizing the cache array in the cases where the requested data is located in one of the buffers. The access to the line buffers is overlapped with the normal cache access, thus avoiding any cycle time compromise or making it unnecessary to introduce an additional cycle for the line buffer access. Significant energy savings are reported, because a large percentage of cache accesses can be satisfied from the line buffers. Additionally, the authors discussed the use of bitline segmentation to reduce the switching capacitance of the bitlines and the use of subbanking to only activate the bank that contains the requested data.

Balasubramonian et.al. proposed a technique to monitor the cache and the TLB usage by detecting phase changes using miss rates and branch frequences and tuning the caches and the TLBs so that the performance is not degraded and the minimum number of partitions are activated [BABD 00]. If the phase do not change within an interval, no reconfiguration of the memory hierarchy is needed, otherwise a new appropriate size has to be selected. This is done by trying various configurations and choosing the one that provides the performance level within a threshold compared to the maximum configuration.

Yang et.al. proposed a scheme to dynamically reconfigure the I–cache by adapting the number of bits used for the set selection in [YPF+ 01]. The authors exploited the observation

that there is a large variability in I–cache utilization both within and across the applications. While the memory cells in unused cells are not accessed, they leak current and thus dissipate energy. The technique (called the DRI cache) dynamically estimates and adapts to the required cache size, thus reducing the leakage current. At the circuit–level, the parts of the cache are turned–off from the power supply using a technique called *Gated–Vdd*. While the actual IPC degradations for individual benchmarks are not shown in [YPF+ 01], the drastic reduction of the I–cache size during downsizing can cause substantial performance losses.

A more fine–grain control over the cache lines for leakage reduction was proposed by Kaxiras et.al. in [KHM 01]. Their scheme, called cache decay, reduces cache leakage by invalidating and turning–off cache lines when they hold data that is not likely to be reused in the near future. The cache decay tries to deduce the dead time by turning–off a cache line if a pre–set number of cycles have elapsed since the last access to the line. The authors also study the adaptive variations of the decay, which seeks to improve performance by adaptively varying the decay interval as the program runs.

In [ZTR+ 01], Zhou et.al. proposed the Adaptive Mode Control – the technique to deactivate only the data portion of cache lines, and not the tag portion. By keeping the entire tag store active, hardware can measure the hypothetical cache miss rate if all lines were kept active. By such monitoring, the hardware can control the total percentage of sleep–mode lines to achieve an actual miss rate that closely tracks the hypothetical miss rate.

Significant leakage reduction can also be achieved by putting a cache line into a low–leakage drowsy mode, as suggested by Flautner et.al. in [FKM+ 02]. When in drowsy mode, the information in the cache line is preserved, however the line has to be reinstated to a high–power mode before the contents can be accessed. The circuit technique for implementing drowsy caches is adaptive body–biasing with multi–threshold CMOS (ABB–MTCMOS) [Nii+ 98]. The simple policy is considered, where periodically, regardless of the access patterns, all lines in the cache are put into a drowsy state and a line

is woken up and brought back into a high–power regime only when it is accessed again. This policy requires a single global counter and no per–line statistics.

The *way prediction* technique is proposed to reduce the cache energy by Inoue et.al. in [IIM 99]. Only predicted way of the cache is accessed thus saving energy by not accessing the other ways. The drawback of the technique is some performance degradation due to possible way mispredictions, in which case the regular cache lookup is needed.

Batson and Vijaykumar proposed reactive–associative cache (r–a cache), which provides flexible associativity by placing most blocks in direct–mapped positions and reactively displacing only conflicting blocks to set–associative positions in [BV 01]. The new design uses way prediction to access displaced blocks on the initial probe. To achieve direct–mapped hit times, the r–a cache uses an asymmetric organization in which the data array is organized like a direct–mapped cache and the tag array like a set–associative cache.

A direct–addressed cache as an energy–efficient cache design was proposed by Witchel et.al. in [WLA+ 01]. Direct addressing allows software to access cache data without a hardware cache tag check. These tag–unchecked loads and stores save the energy of a tag check when the compiler can guarantee that an access will be to the same line as the earlier access. The processor state is augmented with some number of direct address registers. These registers are set and used by the software and contain enough information to specify the exact location of a cache line in the cache data RAM as well as a valid bit. Software places values in these registers as an optional side–effect of performing a load or a store. A tag–unchecked load or store instruction specifies a full effective virtual address in addition to a direct register number. If the register is valid, its contents are used to avoid a tag search; if it is invalid, hardware falls back to a full tag search using the entire virtual address.

A history–based tag comparison (HBTC) cache was proposed in [IMK 02] by Inoue et.al. The scheme attempts to re–use tag comparison results for avoiding unnecessary way activation in set–associative caches. The cache records tag comparison results in an

extended BTB and re–uses them for directly selecting only the hit–way which includes the target instruction.

3.1.6. Energy Reduction within the Function Units

In [BM 99], Brooks and Martonosi proposed to package multiple narrow–width operations in a single ALU in the same cycle, thus improving performance. The second technique, which is proposed in [BM 01], use the operand values to gate off portions of the execution units. The scheme exploit the existence of small operand values to reduce the amount of power consumed by the execution units by using aggressive form of clock gating. The scheme disables the upper bits of the ALUs where they are not needed. The results indicate that the energy of the execution units can be reduced by about 54% for integer programs with little additional hardware.

Seng et.al. provided a set of execution units with different power and latency characteristics [STT 01]. Specifically, the instructions predicted as critical for performance, are steered to fast and power–hungry execution units, while the instructions predicted as not critical are steered to power–efficient and slow execution units, thus reducing the overall power.

In [FBH 02], Fields et.al. introduced the notion of *instruction slack*, as a more general measure of instruction criticality. A slack is essentially defined as the number of cycles by which the instruction execution can be delayed without impact on performance. They also designed a slack predictor and used it for steering instructions to various execution units, similar to [STT 01].

3.1.7. Low Power Encoding

Su et.al proposed a technique to reduce switching activity on the address buses [STD 94]. The authors observed that programs often generate consecutive addresses during execution, and the data accesses are also frequently consecutive especially when structured data, such as an array, is accessed. This property is taken advantage of by using the Gray code for the addresses. The code has the advantage that there is only a single transition on the address bus when consecutive addresses are accessed. The reported results show 37% reduction of the switching activity on the average when standard binary encoding is replaced by the Gray encoding. A Gray code encoder must be placed at the transmitting end of the bus, and a decoder is needed at all receiving points.

In [SB 95], Stan and Burleson have proposed an encoding scheme, called bus–invert encoding, that uses redundancy to reduce bus transitions. The scheme adds one line to the bus to indicate if the actual data or its complement is transmitted. If the Hamming distance between the current value and the previous one is less than or equal to n/2 (for n bits), the value is transmitted as such and the value of 0 is transmitted on the extra line. Otherwise, the complement of the value is transmitted and the extra line is set to the value 1. As a result, the average number of transitions per clock cycle is also lowered by less than 25% of the original value, due to the binomial distribution of the distance between consecutive values assuming uniform probability distribution of the values and no correlation between consecutive values.

In [CGS 00], Cruz et.al. proposed to compress data, addresses and instructions by maintaining only significant bytes with two or three extension bits appended to indicate significant byte positions. The extension bits flow down the pipeline to enable only the pipeline operations for the significant bytes. Consequently, the activity and the dynamic power within the register files, the execution units and the caches is significantly reduced.

New extension bit values are generated only when a new cache line is filed from the main memory and when the new data values are produced by the ALU.

3.2. Circuit and Logic Level Techniques

A number of techniques have been designed at the logic, circuit and device levels to reduce power consumption. These include clock gating, the use of half–frequency and half–swing clocks, the use of asynchronous and partially–synchronous logic, and various circuit techniques for reducing the leakage currents in the RAM–based structures.

3.2.1. Clock Gating

The microprocessor clock tree often consumes in excess of 30% of the overall power [Mudge 01, GBJ 98]. Therefore, it is not surprising that the engineers widely rely on the use of clock gating, a technique to turn off clock tree branches to latches and flip–flops whenever they are not used. Until recently, clock gating was considered to be a poor design choice because the clock tree gates can exacerbate clock skew. However, the appearance of more accurate timing analyzers and the use of more flexible design tools have made it possible for developers to produce reliable designs with gated clocks.

Since the usage of the individual circuits varies within and across applications, power reduction can be achieved by AND–ing the clock with a gate–control signal. Clock gating essentially disables the clock to a circuit whenever the circuit is not used, thus avoiding the unnecessary charging and discharging of the unused circuits and saving energy. Specifically, clock gating targets the clock power consumed in pipeline latches and dynamic CMOS circuits used for speed and area advantages over static logic. Figure 3–1(a) shows the schematic of a latch, where C_g is the cumulative gate capacitance connected to the clock. Because the clock switches every cycle, C_g charges and discharges every cycle thus

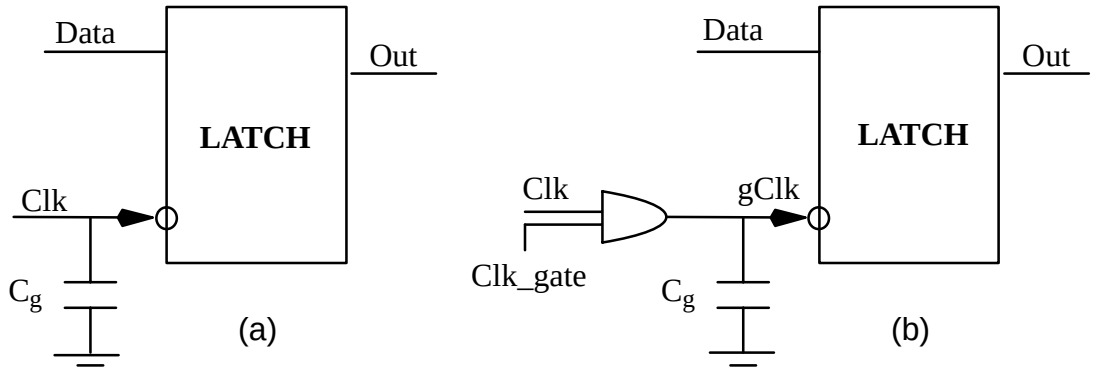


Figure 3–1. Clock gating a latch element

consuming significant amount of dynamic power. Even when the inputs to the latch do not change from one cycle to another, the clock power is still consumed. In Figure 3–1(b), the latch is clock–gated by AND–ing clock input with the control signal. Because the AND gate's capacitance is much smaller than C_g, the overall impact on power is positive.

Figure 3–2(a) shows the schematic of a dynamic logic cell. C_g is the effective gate capacitance that is presented as a clock load, and C_L is the capacitive load to the dynamic logic cell. Similar to the latch, C_g also charges and discharges at every cycle. In addition, C_L also consumes power because it charges through the p–device in the precharge phase of the clock and discharges through the n–device or retains the value in the evaluate phase of the clock. Thus, whether C_L consumes power or not depends on both the current input and the previous output. Figure 3–2(b) shows the same dynamic cell with gated clock. If the dynamic logic cell is not used in a cycle, *Clk–gate* signal prevents both C_g and C_L from switching. While clock–gating latches only reduces the clock power, clock–gating dynamic logic reduces not only the clock power due to C_g, but also the dynamic power of the cell due to C_L.

A predictive form of clock gating is the work of [BM 01c], where Bahar and Manne introduced Pipeline Balancing, as discussed in Section 3.1.3. Pipeline balancing relies on some heuristics to predict a program's ILP at the granularity of 256 cycles. If the predicted

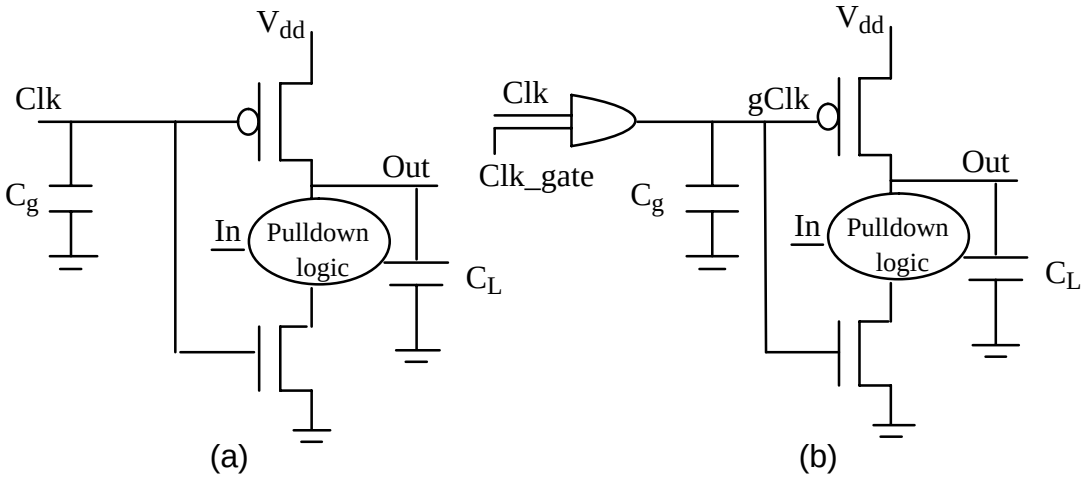


Figure 3–2. Clock gating a dynamic logic gate

number of instructions committed per cycle is lower than the width of the pipeline, then pipeline balancing effectively clock–gates a cluster of pipeline components (including the issue queue, and the register file) during that window.

A technique called Deterministic Clock Gating (DCG) has recently been introduced by Li et.al. in [LBC+ 03]. It is based on the observation that for many of the stages in a modern pipeline, a circuit block's usage in a specific cycle is deterministically known a few cycles in advance. This advanced knowledge can be exploited to clock–gate the unused blocks on a cycle–by–cycle fashion. In particular, DCG clock–gates the execution units, pipeline latches of the back–end stages after issue, L1 D–cache wordline decoders, and result bus drivers. In an out–of–order pipeline whether these blocks will be used is known at the end of the issue stage, based on the type of issuing instruction. There is at least one cycle of register read stage between issue and the stages using the units that can be clock–gated. This cycle is used to gate the clock without impacting the clock speed.

3.2.2. Half–Frequency and Half–Swing Clocks

A half–frequency clock uses both edges of the clock to synchronize the events. Such design is more power efficient than the traditional clock, because the clock runs at half the frequency. However, this comes at a cost of larger and more complex latches plus the clock requirements are more stringent [Mudge 01].

The half–swing clock swings only half of Vdd. It increases the latch design requirements and make it difficult to use in systems with low supply voltages. Lowering clock swing typically results in greater gains than clocking on both edges.

3.2.3. Adiabatic Circuits

Adiabatic circuits represent another approach towards the design of low power electronic components. The approach is based on preserving the charge on the storage nodes instead of dissipating this charge [ASK+ 94]. The rise and fall times of the signals are very carefully chosen so that current flow through resistive elements is minimized. The design uses reversible power supplies and replenishes the charge back to the power supply after evaluation instead of dissipating this charge to the ground. Many preliminary circuit designs are based on adiabatic switching principles [YK 93, AKS 93]. The adiabatic design essentially trades performance (circuit speed) for power, but the energy–delay characteristics of these circuits are much worse than that of the traditional CMOS gates [Ind+ 94].

3.2.4. Asynchronous and Partially Synchronous Design

Another way to reduce the power dissipation in the digital circuits is to use asynchronous logic. Asynchronous design can potentially save significant amount of power, because it does not use the global clock to synchronize the operations and thus the power required by

the clock tree can be saved [McA 92], [Pount 93]. Instead of using the global clock, the circuit components compute the outputs and communicate them to their neighbors by signalling with transitions on the output control lines. The advantage of such handshaking is that the transitions occur only when the results of computations have to be propagated – almost no power is dissipated when the circuits are idle. The requirement to signal the completion of the operation means that additional logic must be used at each register transfer – in some cases, a double–rail implementation, which can increase the amount of logic and wiring. Other drawbacks include testing difficulty and the absence of the design tools. The design is also quite complex and care must be exercised in order to make it robust against the hazards. The AMULET, an asynchronous implementation of the ARM instruction–set architecture, is a project developing a full fledged processor using asynchronous design targeted for the low power embedded market [FDG+ 94, GT+ 98].

Ultimately, asynchronous design does not offer sufficient advantages to warrant a complete switch from synchronous logic. However, asynchronous techniques play an important role in globally asynchronous locally synchronous (GALS) systems [Moore 00, SMB+ 02, SAD+ 02, IM 02]. In these designs, the chip is divided into several coarse–grain clock domains with synchronous communication within the domains and the use of synchronization circuits for communication across the domains.

In [SMB+ 02], the Multiple Clock Domain (MCD) microarchitecture is introduced, where the chip is divided into several clock domains, within which the independent frequency and voltage scaling can be performed. Boundaries in between the domains are chosen to exploit the existing queues, thereby minimizing the interdomain synchronization cost. The independence of each local domain clock implies no global clock skew requirement, permitting potentially higher frequencies within each domain. The main disadvantage of this design is the performance degradation due to the inter–domain synchronization. In [SMB+ 02] the processor is divided into four clock domains – the front end (including the L1 instruction cache), integer units (including the issue queue), floating

point units (including the floating point issue queue) and the load–store units (including the L1 data cache and the L2 cache). The queues, used to decouple the domains, also serve as the synchronization points. The delay associated with crossing a clock domain interface is a function of the synchronization time of the clock arbitration circuit, the ratio of the clock frequencies and the relative phases of the interface clocks. The simulation results, using the offline algorithm, show a reduction in energy–delay product by about 20%.

An online algorithm for scaling the voltage and frequency within each domain of the MCD processor is proposed by Semeraro et.al. in [SAD+ 02]. The algorithm uses processor queue utilization information and operates in two phases (called Attack/Decay algorithm). The queue occupancy is periodically sampled and if the occupancy significantly increases compared to the last interval, then the Attack mode is triggered, which bumps up the frequency of this domain. If no major changes in the queue occupancy are detected, the algorithm decreases the frequency of the domain slightly, in which case the Decay mode is used.

In [IM 02], Iyer and Marculescu analyze the power and performance trade–offs of the GALS design using cycle–accurate simulation. They assumed that the processor is divided into five clock domains, where the front end is subdivided into two domains. The results indicate that going from a synchronous to a GALS design causes a drop in performance, but the elimination of the global clock does not lead to drastic power reductions. The authors conclude that from a power perspective GALS designs are inherently less power efficient than the traditional synchronous architectures. Results also show that for a five–domain GALS processor, the performance drop is in the order of 5–15% and power savings are in the order of 10% compared to traditional design.

3.2.5. Circuit Techniques for Leakage Reduction

Approaches to reducing leakage power at the circuit level can be divided into two categories. Techniques that trade increased circuit delay for reduced leakage current include conventional transistor sizing, the use of lower V_{dd} [Tak+ 98, Usam+ 98], stacked gates [NBD+ 01], longer channels [Mon+ 96] and higher threshold voltages [Wei+ 98]. Techniques for dynamic run–time deactivation of fast transistors include body biasing [Kur+ 98, Mak+ 98, Nii+ 98] and the use of sleep transistors [PYF+ 00]. The former set of techniques is used to reduce leakage on non–critical paths, while the latter is used to reduce leakage on critical paths.

At the expense of additional mask processing steps, it is possible to manufacture transistors with several different threshold voltages on the same die. By using slower, high–threshold transistors on non–critical paths the leakage energy can be reduced significantly without sacrificing performance. Even though most of the transistors are situated on non–critical paths, the achievable savings in leakage energy are limited, because the width of the non–critical transistors has already been reduced and these transistors are stacked into complex gates in order to have low leakage.

After application of the leakage reduction techniques to non–critical paths, the leakage is even more highly concentrated in the critical path devices. One technique for reducing the leakage in such transistors is a dynamic variation of the body bias in order to modulate transistor's threshold voltage. Reverse body biasing sets the p–well voltage higher than V_{dd} and the n–well voltage lower than ground thus increasing threshold voltage because of the body effect. As a result, the leakage current is reduced, and the contents of the memory cell are retained. The drawback is that the technique requires twin or triple well processes, thus increasing manufacturing costs. Because of the large capacitance and distributed resistance of the wells, charging or discharging the well incurs a relatively high delay and dissipates considerable amount of energy. In order to amortize the costs of transitioning into a

low–leakage state, these schemes are only used when a processor enters the sleep mode for at least 0.1 – 100 microseconds [HBH+ 02].

An alternative approach to reducing leakage on the critical paths is to insert a high V_T sleep transistor between V_{dd} and virtual V_{dd}. When turned off, the sleep transistor adds the additional high threshold device in series with the logic transistors to dramatically decrease leakage due to the stacking effect. The disadvantages of the sleep transistors are that they add additional impedance in the power supply network which reduces circuit speed, they require additional area and routing resources for the virtual power supply network, and they can consume significant energy in the course of deactivation [HBH+ 02]. In addition, the contents of the memory cell, which is turned off from the power supply using sleep transistor, are not preserved.

A method for reducing leakage current in caches and register files, called Leakage–Biased Bitlines, was proposed by Heo et.al. in [HBH+ 02]. For caches, a key observation is that the leakage current from each bitline into the cell depends on the stored value on that side of the cell; there is effectively no leakage if the bitline is at the same value as that stored in the cell. Rather than forcing a zero sleep values into the read bitlines of inactive subbanks, the proposed technique just lets the bitlines float by turning off the high–threshold precharging n–devices. The leakage currents from the bitcells automatically bias the bitline to a mid–rail voltage that minimizes the bitline leakage current [HBH+ 02]. Similar technique is applied to the register files.

3.3. Compiler Techniques

The energy–oriented compiler techniques mainly attempt to reduce the number of cycles to execute a given program, reduce the energy strength of the instructions or reorder the instructions to reduce switching activity [TMW 94]. As expected, standard compiler optimizations like loop unrolling and software pipelining are also beneficial from the energy

standpoint since they reduce the number of cycles needed to execute a program. Some of these optimizations can be done by traditional code optimizers, with the performance metrics substituted by the energy costs.

The problem of register allocation for reducing switching activity is targeted by the works of [ZHC 99, Geb 97, CP 95]. However, the energy–optimized and performance–optimized generators produce nearly identical code, so it is difficult to gauge the advantages of the energy–optimized code over cycle count–optimized programs [ZHC 99].

Compiler techniques for reducing the memory energy include mechanisms for improving the locality of memory accesses, minimizing the number of memory accesses, reducing the total memory area and making the effective use of the memory bandwidth. Techniques for improving locality of accesses include linear loop transformations (loop permutation, loop skewing, loop reversal and loop scaling), iteration space tiling, multi–loop transformations (loop fusion/fission and inter–procedural analysis) and data transformation. Number of memory accesses can be minimized by performing scalar replacement to enable effective use of registers. This, however, increases register pressure and requires the use of more sophisticated register allocation/deallocation algorithms. Memory area can be reduced by reusing the same memory space, relying on both inter–variable and intra–variable analysis [Bose+ 01].

Techniques for minimizing the code space include mechanisms for address assignment to variables and effective use of auto–increment/decrement addressing modes. Code compression techniques extract the common code sequences and place them in a directory. Instances of these sequences are then replaced by mini–subroutine calls.

Register re–labeling is a post–compilation technique that exploits corresponding fields in consecutive instructions to reduce switching activity of the instruction buses and register files. The compiler constructs a register transition graph and either records all consecutive

transitions between all possible pairs of registers or uses profiling in the form of sample traces. It then determines the paths that contain edges with high transition counts and relabels the registers to minimize the switching activity. Registers with large transition counts are labeled using minimum Hamming Distance.

3.4. Operating System Techniques

There are two approaches to voltage scaling using the operating system. The first approach provides an interface that allows the operating system to directly set the voltage – often by simply writing to a register. The application uses these operating system functions to schedule its voltage requirements [PBB 00]. The second approach uses a similar interface, but the operating system automatically detects when to decrease the voltage when application is executed. An advantage of automatic detection is that applications do not require modifications to perform voltage scheduling. A disadvantage is in the difficulty of determining when it is best to scale down the voltage.

In [GCG 00], resource usages are controlled indirectly through pipeline gating and dispatch mode variations by letting the Operating System dictate the IPC requirements of the application. An industry standard, Advanced Configuration and Power Interface (ACPI) defining an open interface used by the OS to control power/energy consumption is also emerging [ACPI 99]. The standard evolves existing collection of BIOS code into a well specified power management and configuration system. The goal of ACPI is to enable all PCs to implement motherboard configuration and power management functions, using appropriate cost/function tradeoffs. As a result of the unification of power management algorithms in the OS, the opportunities for miscoordination will be reduced and the reliability will be enhanced.

3.5. Power–Aware Instruction Sets

Energy–efficient instruction set architecture have been proposed by *Bunda et.al.* in [BF+ 95]. The key observation is that energy–efficiency can be achieved using smaller program encoding and thus the instruction sets can have some impact on the total power dissipation, as it impacts the number of executed instructions as well as the density of the compiled code. Through analytical models, Bunda et.al. explore the relative energy efficiency of two instruction encodings for the same microarchitecture. They showed that a 16–bit instruction encoding for a RISC–style microprocessor is more energy–efficient than traditional 32–bit format. This fact is also reflected in the design of some low power microprocessors. For example, the Hitachi SH family [Fleet 94] of embedded controllers uses a 16–bit ISA in the interests of low power consumption.

In [AHK+ 02], Asanovic et.al. proposed energy–exposed hardware–software interfaces to give software more fine–grain control over energy–consuming microarchitectural operations. First, they introduced software restart markers to make temporary processor state visible to software without complicating exception handling. Software restart markers reduce energy by allowing the compiler to annotate at which points it requires precise exception behavior. The compiler explicitly divides the instruction stream into restartable regions and after handling a trap, the OS resumes execution at the beginning of the restart region for the associated instruction. A conventional architecture with precise exceptions is equivalent to placing every instruction into its own restart region. Second, *Asanovic et.al.* implemented exposed bypass latches to allow the compiler to eliminate register file traffic by directly targeting the processor bypass latches. In effect, when using the bypass latches, software turns off the register fetch and write back stages of the pipeline, and thereby removes the microarchitectural energy overhead.

Chapter 4

Power Estimation and Simulation Methodology

It has been recognized by the processor design community that power dissipation is a first–class architectural design constraint not only for portable computers and mobile communication devices, but also for high–performance superscalar microprocessors [Mudge 01]. A fair amount of research efforts has been directed towards reduction of power dissipation in this high–end systems. Proposed solutions include both microarchitectural and circuit–level techniques.

Several power estimation tools for processors have been designed, including Wattch [BTM 00], Simplepower [VKI+ 00] and TEM^{2}P^{2}EST [DLC+ 00] to name a few. Simplepower is used only for simple 5–stage scalar pipelines and only models the execution of integer instructions; it is not applicable to superscalar processors. The major drawback of other tools is their reliance on the Simplescalar simulator [BA 97], which lumps many critical datapath artifacts like the issue queue (IQ, also known as an “dispatch buffer”), the reorder buffer (ROB) and physical register files (PRFs) into a unified structure called Register Update Unit (RUU), quite unlike the real implementations, where the number of entries and the number of ports in all these structures are quite disparate. Consequently, power dissipation can not be estimated accurately on a per–component basis. Considering

that in many cases, these components collectively contribute to more than half of the overall power dissipation of the chip [FG 01], it is necessary to have facilities that allow the design space of these components and their interactions be modeled as accurately as possible. This is exactly one of the areas where AccuPower provides a more accurate simulation/power estimation framework than existing tools. Specifically, the main features of the AccuPower toolset are as follows:

- Detailed cycle–level simulation of all major datapath components and interconnections that mimic the actual hardware implementation, including separate and realistic implementations of the IQ, register files, ROB, load–store queues and forwarding mechanisms.
- Detailed and accurate simulations of the on–chip cache hierarchy (including multiple levels of on–chip caches, interconnections, arbitration and chip–level I/O traffic).
- Built–in models for four major variants of superscalar datapaths in wide use.
- Well–instrumented facilities for collecting datapath statistics of relevance to both power and performance at the level of bits, bytes (for data and instruction flows) within logic blocks and subsystem–level components and the entire processor.
- Implementations of cutting–edge techniques for power/ energy reduction at the microarchitectural level, logic level and circuit level, as well as techniques based on clock gating, voltage and frequency scaling to facilitate the exploration of the design space.
- Use of energy/power dissipation coefficients for energy dissipating events within datapath components derived from SPICE measurements of actual layouts of these components. These coefficients are used in conjunction with transition counts obtained from the microarchitectural simulation component of AccuPower to accurately estimate the power/energy dissipations. Coefficients for leakage dissipations are also provided.

We believe that short of an actual implementation, AccuPower's power estimation strategy is as accurate as it gets. As part of an ongoing effort, we are also in the process of validating the energy dissipation coefficients of critical datapath components from actual subscale implementations through MOSIS. A release of the AccuPower toolset, including the VLSI layouts and SPICE–measured data is planned for in the near future. The planned release will also incorporate models for multi–clustered datapaths; this feature is not implemented in AccuPower at this point.

4.1. Superscalar Datapaths Modeled by AccuPower

Four variations of superscalar datapaths are predominantly in use in modern implementations as briefly examined in Chapter 2. They mainly differ in two respects: how the physical registers are implemented and when the readout of the source physical registers occurs. The four datapaths that are modeled by AccuPower are:

1) **Datapath A: the datapath with dispatch–bound operand reads.** This is the datapath where operand reads occur at dispatch time. Examples of processors using this datapath style are the Intel Pentium Pro, Pentium II, IBM Power PC 604, 620 and the HAL SPARC 64 [MR 9X].

2) **Datapath B: the datapath with issue–bound operand reads.** This is the datapath where operand reads occur at issue time. Examples of processors using this datapath style are the MIPS 10000, 12000, the IBM Power 3, the HP PA 8000, 8500, and the DEC 21264 [Kessler 99] and Intel Pentium 4 [Hinton+ 01].

3) **Datapath C: the datapath with physical register files and reorder buffer integration.** This is the datapath where physical register files are integrated into reorder buffer. Examples of processors using this datapath style are the Intel Pentium II and Pentium III [Intel 99].

4) **Datapath D: the datapath with physical register files and architectural register files integration.** This is the datapath where physical register files are integrated into architectural register files. Examples of processors using this datapath style are the MIPS R10000, DEC 21264 [Kessler 99] and Intel Pentium 4 [Hinton+ 01].

The details of these datapaths can be found in Chapter 2 where we examine the supercalar datapaths, in detail.

4.2. Implementation

The AccuPower tool consists of the three components: a microarchitectural simulator, which is a greatly modified version of the Simplescalar; the VLSI layouts for major datapath components and caches in a 0.18u process; and power coefficients obtained from the SPICE simulations. Figure 4–1 summarizes the overall power/energy estimation methodology used in AccuPower.

4.2.1. Microarchitectural Simulators

To support the superscalar datapath configurations discussed in previous section, we significantly modified the Simplescalar simulator [BA 97] and implemented separate structures for the issue queue, the reorder buffer, the rename table, the physical register files and the architectural register files. Different versions of the simulator were designed – one for each datapath configuration discussed above. We also accurately modeled the pipeline structures and various interconnections within the datapath, namely the dispatch buses, issue buses, result buses and commit buses. In a typical superscalar processor, multiple sets of such buses are needed to sustain the dispatch/issue/commit rate. Traffic on each such bus and read/write activity within the register files implementing the datapath storage components are separately monitored and analyzed as discussed later.

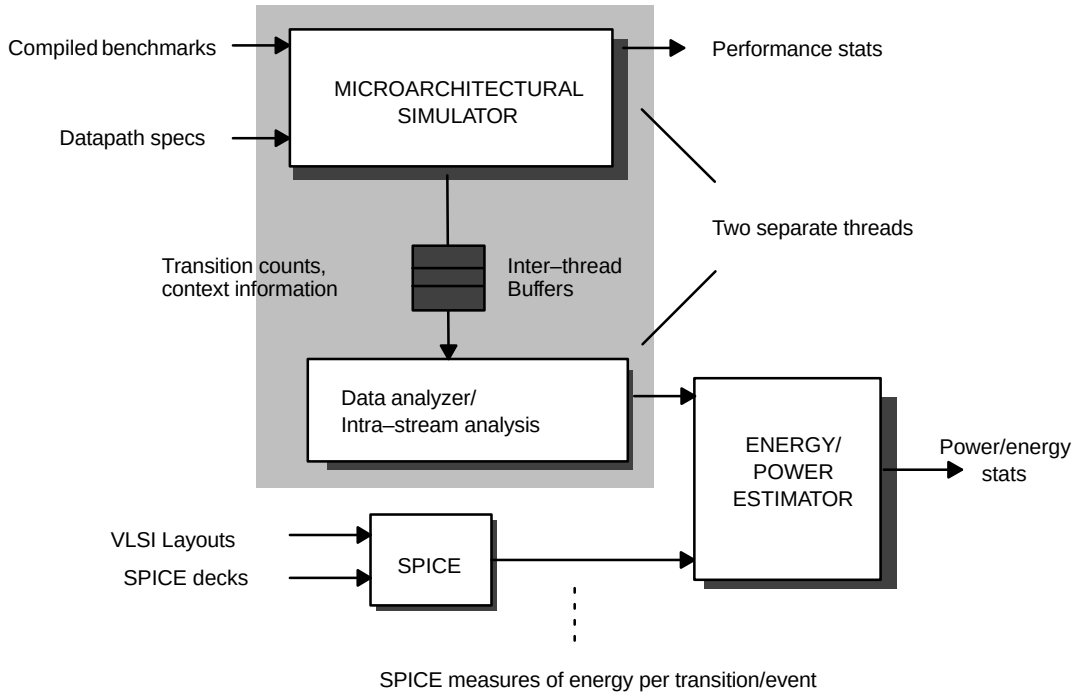


Figure 4–1. Power Estimation Methodology

The extent of our modifications to the Simplescalar can be gauged by the fact that barely 10% of the original code was retained. The *sim–outorder.c* file was completely rewritten to support a true cycle–by–cycle out–of–order instruction execution. This is in contrast to the original Simplescalar code, where instructions are actually executed in–order at the dispatch stage and the effects of out–of–order execution are achieved through the convoluted manipulations with the RUU. Significant modifications have also been incorporated into the cache simulator, as discussed below. Specifically, the changes that were made to the Simplescalar to simulate the realistic superscalar pipelines are as follows:

(i) We split the monolithic cache access stage as used in Simplescalar into two stages to mimic the real–world situation where cache accesses – even L1 cache accesses – are typically performed in multiple cycles and provided a support for pipelined cache.

(ii) We modeled the contention for the bus between L1 and L2 caches. This is important because the L1 caches are typically split with separate caches for instructions and data.

L2 cache, however, is typically unified. Situation when both I1 miss and D1 miss occur in the same cycle are certainly possible and they require proper modelling to arbitrate for the access to the L2 cache.

(iii) Along similar lines, we modelled the contention for the off–chip interconnection from the L2 cache to the DRAM modules.

(iv) The decode stage of the Simplescalar datapath, where instruction dispatching and register renaming is performed, was split into two stages, as it is unrealistic to perform the fairly complicated operations of dispatching, register renaming and source register readout in a single cycle.

(v) We also assumed realistic delays on the interconnections, noting that it takes a full cycle to distribute the result produced by one of the FUs to the waiting operands in the issue queue.

To summarize, we attempted to design a simulator that would closely mimic the actual microarchitecture and hardware implementations of real CPUs on a cycle–by–cycle basis. The focus of the AccuPower tool is to facilitate the exploration of the design space of superscalar processors and gauge the impact of well–used and cutting–edge techniques for saving power and/or energy. The tool also supports the exploration of circuit–level techniques and the more standard power reduction techniques like voltage and frequency scaling.

4.2.2. VLSI Layouts

For estimating the energy/power for the key datapath components using AccuPower, the transition counts and event sequences gleaned from the microarchitectural simulator were used, along with the energy dissipations for each type of event, as measured from the actual VLSI layouts using SPICE. CMOS layouts for the on–chip caches, issue queue, physical register files, architectural register files, reorder buffer and load/store queue in a 0.18 micron

6 metal layer CMOS process (TSMC) were used for the key datapath components to get an accurate idea of the energy dissipations for each type of transition.

The register files that implement the issue queue and reorder buffer were carefully laid out to optimize the dimensions and allow the use of a 2 GHz clock. A V_{dd} of 1.8 volts is assumed for all the measurements. (The value of the clock rate was determined by the cache cycle time, as determined by the cache layouts for a two–stage pipelined cache in the same technology.) In particular, these register files feature differential sensing, limited bit line driving and pulsed word line driving to save power. Augmentations to the register file structures for the issue queue (mainly in the form of comparators with each of the 3 source operand fields and the four result/tag buses) were also fully implemented; a pulldown comparator was used for associative data forwarding to entries within the issue queue and the device sizes were carefully optimized to strike a balance between the response speed and the energy dissipations. For each energy dissipating event, SPICE measurements were used to determine the energy dissipated. These measurements are used in conjunction with the transitions counted by the hardware–level, cycle–by–cycle simulator to estimate dynamic energy/power accurately. Actual layout data was also used for estimating the leakage power of the layouts in the smaller feature sizes.

4.2.3. Speeding Up the Execution – Multithreading

The instrumentation needed to determine the bit level activities within data flow paths and data storages (both explicit and implicit) and log all major switching activities slows down the simulation drastically. To get reasonable overall simulation performance with all the instrumentation in place, we resorted to the use of multithreading. Specifically, we use a separate thread for the data stream analysis, as shown in Figure 4–1. The two–threaded implementation is run on SMPs to get an acceptable level of simulation speed – one that approached close to that of the original Simplescalar without the heavy instrumentation. The

data acquired from basic instrumentation within the main simulation thread is buffered and fed into a separate thread where it was further analyzed.

With a single thread implementing all of the simulation, instrumentation and analysis, the overall simulation speed was reduced by as much as 40% compared to the original Simplescalar simulation without any modification and instrumentation. With both threads in place as shown in Figure 4–1, and with the use of inter–thread buffers of an optimized size, the overall simulation time achieved was often significantly better on a SMP compared to the single–threaded implementation. The performance of the dual–threaded version was also acceptably close to that of the original Simplescalar simulator without any of the enhancements and the instrumentation.

Public release of AccuPower is planned in the very near future. Validations of the SPICE–measured data from MOSIS–supported implementations of some critical datapath components are also planned as part of an ongoing effort.

4.3. The Use of AccuPower

AccuPower can be used to obtain realistic measurements in a variety of ways. These include:

1) The raw data collection. This falls into two categories:

a) The tool monitors the bit–level datapath activity on the interconnections, dedicated transfer links, and read/write activity of the register files that implement the datapath storage components. We also provide the data analyzer (implemented as a separate thread for performance reasons, as discussed above) to examine the collected data streams on the presence of zero–bytes. Presence of zero bytes has been exploited to reduce the switching activity and hence power dissipations in caches [VZA 00] and in other datapath components [GPK+ 00]. In addition, the data analyzer estimates the

percentage of bits that did not change their values since the previous value had been driven on the same link. Considerable power savings can be achieved on the datapath interconnections by not driving such bits. [GPK+ 00]. To further reduce the number of bits driven on the interconnections, such bit–slice invariance can be used on top of zero–byte encoding.

b) In AccuPower, the occupancies of individual datapath resources, as measured by the number of valid entries, are monitored and recorded. This capability is currently implemented for the issue queue, the reorder buffer, and the load/store queue. Support for caches will be added in the near future. Figure 4–2 shows the occupancies of the

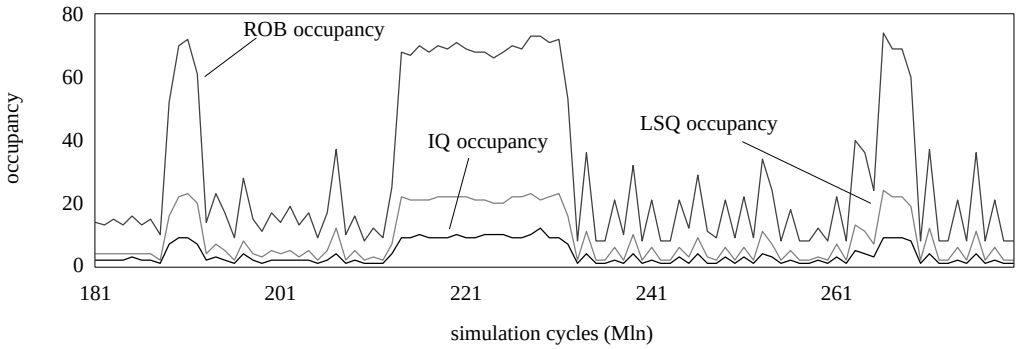


Figure 4–2. The occupancies of the IQ, the ROB, and the LSQ

issue queue, the reorder buffer, and the load/store queue obtained from the execution of *fpppp* benchmark on AccuPower simulator. Resource occupancies have been recorded at every cycle and the averages have been taken for every 1 million of simulated cycles. This occupancy statistics can be used to drive the decision to dynamically resize the resources, in case the occupancy sampling shows that the resource is currently overcommitted. Correlations among the resource occupancies can be further exploited to reduce the overhead of the control logic needed to implement the dynamic resizing. Such dynamic resizing of multiple resources has been done in [PKG 01b] with very small impact on the performance. Achieved power savings and performance degradation can be measured using AccuPower for all SPEC benchmarks for a variety

of resizing parameters, as discussed in [PKG 01b]. Representative results for power savings within the issue queue, obtained from AccuPower, are shown in Figure 4–3.

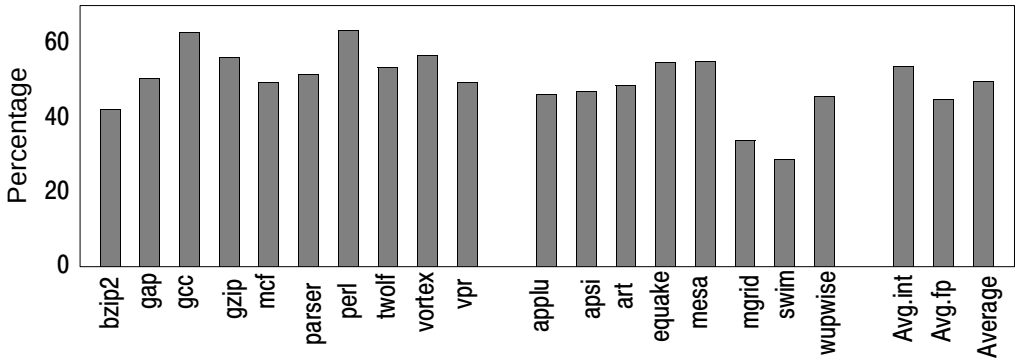


Figure 4–3. Savings in energy per cycle within the issue queue with the use of dynamic resizing and bitline segmentation

2) **Accurate datapath component–level power estimation.** Here, we use transition counts obtained from the architectural simulation in conjunction with the capacitance coefficients obtained from SPICE simulations.

3) **Exploration of various power reduction techniques.** These include the use of zero–encoding [VZA 00], dynamic resource allocation using occupancy sampling [PKG 01b] and the use of partitioned datapath components [ZK 00].

4) **Exploration of alternative circuit–level techniques.** These currently include the use of fast dissipate–on–match comparators for associative lookup within the issue queue, the reorder buffer, the load/store queue and the TLB, bit–line segmentation within register files and zero–byte encoding [KGPK 01, PKE+ 03].

5) **Explorations of alternative datapath architectures.** As mentioned above, four datapath configurations are currently supported.

Chapter 5

Techniques for Energy–Efficient Issue Queue

We examine the use of three relatively independent techniques for reducing the power dissipations in the issue queue. To set the right context, it is useful to examine the major power dissipation components within the IQ. Figure 5–1 shows the total issue queue energy breakdown across the three components. The distribution is shown for a datapath with dispatch–bound operand reads, that is, wide issue queue is in use. The numbers represent the averages over the SPEC 2000 integer and floating–point benchmarks for the base case

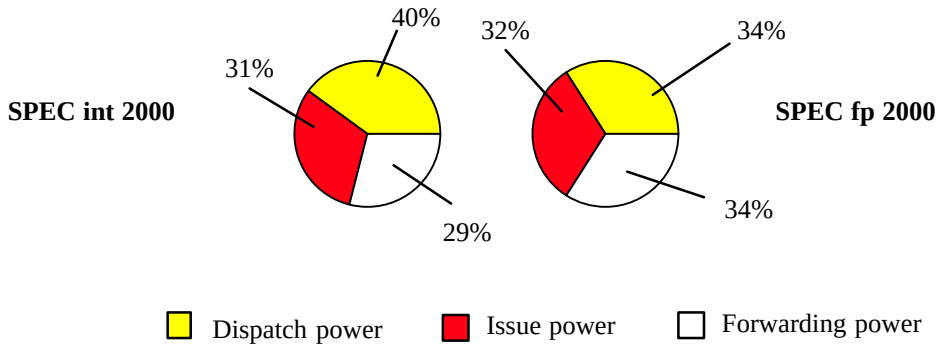


Figure 5–1. Energy dissipation components of the traditional IQ for a datapath with dispatch–bound operand reads (% of total)

IQ, which is a traditional design using comparators that dissipate energy on a tag mismatch. Forwarding power represents a relatively higher percentage of the total power dissipation for floating point benchmarks, because more tag comparisons are performed in an average

cycle for two reasons. First, more tags are broadcast because floating point benchmarks exhibit higher ILP than integer benchmarks and second, more slots in the issue queue await for results in the course of execution of floating point benchmarks. Simulation of SPEC 2000 benchmarks within our experimental framework shows that on the average about 28 slots are waiting for the results during the execution of integer benchmarks and 47 slots are waiting during the execution of floating point benchmarks (for this graph, we assumed that only the comparators associated with the invalid sources of valid entries are activated; in the result section we also consider the case when all comparators are active). All such slots employ the traditional pulldown comparators (that dissipate energy on tag mismatches) in current implementations of IQs [PJS 97].

To reduce the energy spent in data forwarding directly, we explore the use of comparators that dissipate energy predominantly on a tag match. We then explore the use of zero–byte encoding [Gh 00, BM 99, CGS 00, GPK+ 00] to reduce the number of bitlines activated during dispatching, forwarding and issuing. Finally, we add bit–line segmentation to reduce energy dissipations in the bitlines that are driven. The overall energy saving realized for the IQ by using all of these techniques in combination is between 56% to 59%, and this is realized with an acceptable increase in the cycle time of the processor and silicon real estate for the IQ.

5.1. Using Energy–Efficient Comparators in the IQ

The typical comparator circuitry used for associative matching in an IQ is a dynamic pulldown comparator or a 8–transistor associative bitcell. This is depicted in Figure 5–2. Such comparators have a fast response time to allow matching and the resulting updates to be completed within the cycle time of the processor. All of these comparators dissipate energy on a *mismatch* in any bit position. A significant amount of energy is thus wasted in comparisons that do not locate matching entries, while little energy (in the form of

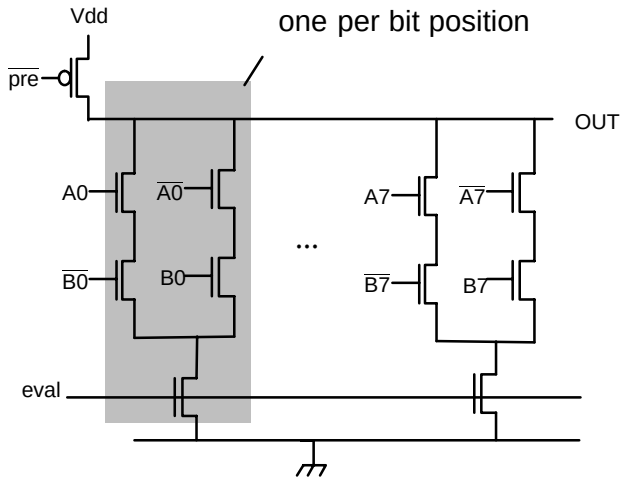


Figure 5–2. Traditional pull–down comparator

precharging) is spent in locating matching entries. In a typical cycle, only a small percentage of the slots waiting for the result actually match the destination address of the forwarded data. As an example, the data collected for the simulated execution of SPEC 2000 benchmarks on our system indicates that about 36 operand slots out of the 128 that we have in our 64–entry issue queue are actually waiting for results (i.e., the comparators for these slot are enabled for comparison). Out of these, only 1 to 4 comparators produce a match per cycle on the average. This is clearly an energy–inefficient situation, as more energy is dissipated due to mismatches (compared to the number of matches) with the use of traditional comparators that dissipate energy on a mismatch. Similar observations are valid for reorder buffers which double as physical registers (as used in Pentium III, for example), where associative matching is used to locate the most recently established entries for instruction destinations. We propose to remedy this by designing and using comparators that (predominantly) dissipate energy on a match.

In theory, one can design CMOS comparators to dissipate energy dynamically only on a full match but these designs require a large number of transistors and/or switch slowly. Instead of choosing such a design, we opted for a comparator that dissipates minimum energy on (infrequent) partial mismatches in the comparand values and dissipates an

acceptable amount of energy on a full match. This comparator consciously takes into account the characteristics of the distribution of the physical register addresses that are compared in a typical IQ to minimize energy dissipations for partial matches.

Number of bits matching —>	% of total cases			
	2 LSBs	4 LSBs	6 LSBs	All 8 bits
Avg. SPECint 2000	26.7	8.8	4.9	4.6
Avg. SPECfp 2000	25.7	7.5	3.1	2.4
Avg. all SPEC 2000	26.2	8.2	4.1	3.6

LSB = least significant bits

Table 5–1. Issue Queue comparator statistics

Table 5–1 shows how comparand values are distributed and the extent of partial matches in the values of two adjacent bits in the comparands, averaged over the simulated execution of SPEC 2000 integer and floating point benchmarks, as well as the average over both the integer and floating point benchmarks. The lower order 4 bits of both comparands are equal in roughly 8.2% of the cases, while the lower order 6 bits match 4.1% of the time. A full match occurs 3.6% of the time on the average. Equivalently, a mismatch occurs in the lower order 4 bits of the comparator 91.8% of the time. The behavior depicted in Table 5–1 is a consequence of the localization of dependencies in typical source code. This causes physical registers from localized regions to be allocated as the destination of instructions involved in the dependency (the addresses of these physical registers are compared within the IQ to satisfy data dependencies). Consequently, the likelihood of a match in the higher order bits of the register addresses (i.e., the comparands) is higher. Our comparator design directly exploits this fact by limiting dynamic energy dissipation due to partial matches to less than 8.2% of cases when the lower order 4 and 6 bits match; no energy dissipation occurs in the more frequent cases of the higher order bits matching. The overall energy dissipation due to partial mismatches is thus greatly reduced. Energy dissipation occurs, of course, on a full

match but this dissipation is smaller than that of comparable traditional comparators (that pulls down a precharged line only on a mismatch in one or more bit positions.)

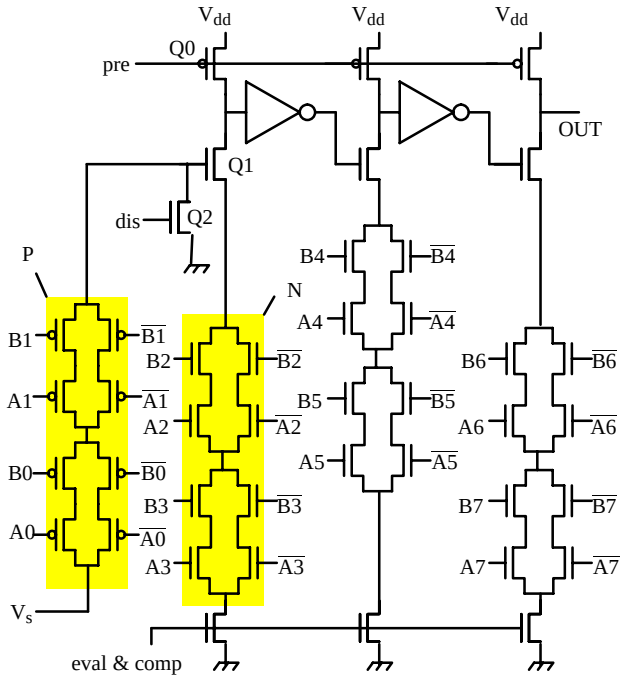


Figure 5–3. The proposed comparator dissipate–on–match comparator

Figure 5–3 depicts our proposed comparator for comparing two 8–bit comparands, A7A6..A0 and B7B6..B0, where 0 is the least significant bit. The basic circuit is a three–stage domino logic, where the first stage detects a match of the lower 4 bits of the comparands. The following two stages do not dissipate any energy when the lower order 4 bits do not match. The precharging signal is cut off during the evaluation phase (when eval is active) and an evaluation signal is applied to each stage of the domino logic only when the comparison is enabled (comp is high). The series structure P composed of 8 p–transistors passes on a high voltage level (V_s, where V_s is < V_{dd} but higher than lower threshold for a logic 1 level) to the gate of the n–transistor Q1 only when the two lower order bits of the comparands match. The series structure N turns on when the next pair of lower order bits of the comparands (A3A2 and B3B2) match. When comparison is enabled, the output of the first stage (driving an inverter) goes low during the evaluation phase only when all lower

order 4 bits of the comparands match. Till such a match occurs, no dynamic energy is dissipated in the other stages of the comparator. Transistor Q2 is needed to prevent Q1 from turning on due to the presence of charge left over on its gate from a prior partial match of the two lower order bits. The charge moved from the gate of Q1 by Q2 is dissipated to ground only when there is a subsequent match in bits 3 and 2 (which turns the structure N on.) This effectively reduces dissipations in the case when only the two lower order bits match; this dissipation could be further minimized by keeping V_s slightly lower than V_{dd}, but we did not explore this option because of the implementation difficulties. As in any series structure of n–transistors that pull down a precharged line, the W/L ratios of the n–devices go up progressively from the top to the bottom (ground). The p–transistors in the structure P, the precharging transistors and the inverters are sized to get an acceptable response time on a match.

The comparator delay component that lies on the critical path in a typical cycle is the evaluation delay. The precharge time can be used to provide and stabilize the new input data, such that the inputs are ready when the evaluation signal rises. For example, if the comparators are used within the issue queue, then the source operand tags (which are the data inputs to the comparators) can be driven across while the comparator is precharged. Therefore, the evaluation time defines the delay of both comparators for all practical purposes. For the proposed comparator, a separate discharge signal (*dis* in Figure 5–3) is used to discharge the gate of the n–transistor Q1 to prevent false matches. Notice, that the discharge signal is not on the critical path, since the inputs can still be changing when discharge signal is high. This may lead to a momentary short–circuit energy dissipation, but this energy is small and was fully accounted for in our SPICE measurements. The evaluation delay of the comparator of Figure 5–3 in a full match condition is comprised of five non–overlapping components which are listed below.

T1 – time needed to discharge the input of the first inverter

T2 – delay of the first inverter

T3 – time needed to discharge the input of the second inverter

T4 – delay of the second inverter

T5 – time needed to discharge the output.

To compare the evaluation delay of the proposed comparator with that of the traditional design, we first optimized the traditional comparator circuitry by minimizing its delay through careful resizing of the devices. The evaluation delay is highest when the comparands mismatch in only one bit position, since the matchline has to be discharged through a single stack of n–devices. This is the situation that we considered in our experiments. As it turns out, there is an optimal width of n–devices that results in the minimum evaluation delay. As the width of n–devices increases, the output node discharges faster, but the output capacitance also increases, resulting in slower discharging if the transistor widths are increased beyond a certain point. The minimum evaluation delay determined by the SPICE simulations was measured as 121 ps.

The evaluation delay of the proposed comparator increased by 150 ps. compared to the traditional design. The breakdown across the five delay components is as follows: T1=126 ps, T2=37 ps, T3=41 ps, T4=39 ps, T5=28 ps. Notice that T3 and T5 are significantly lower than T1 because the N–blocks discharge simultaneously during the first 126 ps. The time T1 also includes the propagation delay of the high voltage level Vs through the P–block to the gate of n–transistor Q1 (when bits 0 and 1 match). Most of this delay is overlapped with the discharging of the source of Q1 through the first N–block (when bits 2 and 3 match). Figure 5–4 shows the waveforms obtained from the SPICE simulations (using Cadence toolkit) of the proposed comparator. The waveforms are shown for the situation of a full match in the comparands. Figure 5–4(a) depicts the voltage at the output node of the comparator as a function of time. As shown, the output discharges after 271 ps, which it should in a full match situation. Figure 5–4(b) shows the voltage at the output of the second

inverter, Figure 5–4(c) depicts the voltage at the input of the second inverter, Figure 4–4(d) shows the voltage at the output of the first inverter, Figure 5–4(e) shows the voltage at the input of the first inverter and, finally, Figure 5–4(f) depicts the evaluation signal.

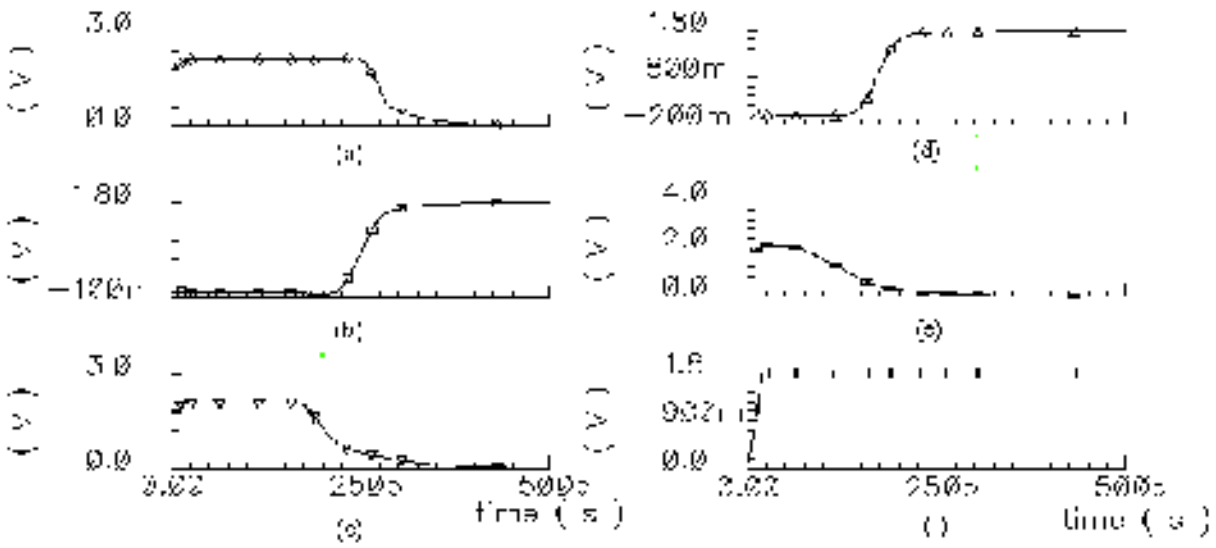


Figure 5–4. SPICE waveforms of the proposed comparator.

For the cycle time of 566 ps (which was the read access time of the multi–ported issue queue used for the datapath with issue–bound operand reads in our simulations), the increase in comparator evaluation delay represents about 26% overhead. The read cycle of the issue queue used for the datapath with dispatch–bound operand reads is 658 ps, and the increase in cycle time due to the use of the new comparator is about 23%. We discuss the timing considerations in more detail in Section 5.1.4. An alternative dissipate–on–match comparator circuitry which is actually slightly faster than the traditional comparator was introduced in our recent work in [EGK+ 02].

The area overhead of the proposed comparator is 28% (642 um^2 vs. 500 um^2 for the traditional design). To conclude the discussion of the proposed comparator, Tables 5–2 and 5–3 shows the energy dissipations of the traditional and the proposed comparators for various matching patterns in the comparands. Combined with the performance simulation

results summarized in Table 5–1 and the relevant percentages for the traditional comparator (not shown here), these numbers were used to estimate the energy savings that can be achieved with the use of the proposed comparator.

Number of matching bits in the comparands	Energy of the traditional comparator (fJ)
0	709.2
1	671.0
2	631.7
3	593.1
4	551.3
5	513.2
6	470.8
7	432.7
8	7.7

Table 5–2. Energy dissipations of the traditional comparator for various matching patterns

Number of matching leading bits in the comparands	Energy of the new comparator (fJ)
none of the bits	8.2
2 least significant bits	142.5
4 least significant bits	473.8
6 least significant bits	792.0
full match	889.1

Table 5–3. Energy dissipations of the new comparator for various matching patterns

5.2. Using Zero–Byte Encoding

A study of data streams within superscalar datapaths as reported in [Gh 00] shows that significant number of bytes are all zeros within operands on most of the flow paths (dispatch stage to IQ, IQ to function units, function units to destinations and forwarding buses etc.). On the average, in the course of simulated execution of SPEC 2000 benchmarks on cycle accurate and true register level simulator, about half of the byte fields within operands are all zeros. This is really a consequence of using small literal values, either as operands or as address offsets, byte–level operations, operations that use masks to isolate bits etc. Considerable energy savings are possible when bytes containing zero are not transferred, stored or operated on explicitly. Related research in the past for caches [VZA 00], function units [BM 99] and *scalar* pipelines have made the same observation. We extend this work to superscalar datapaths, where additional datapath artifacts to support out–of–order execution can benefit from the presence of zero–valued bytes. The issue queue is an example of just such an artifact.

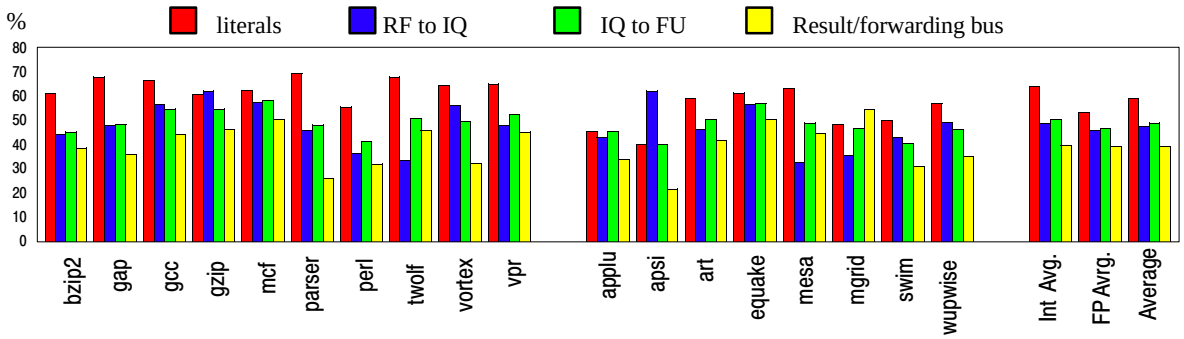


Figure 5–5. Percentage of zero bytes on various paths within the superscalar processor

Figure 5–5 shows the percentage of bytes with all zeroes on various paths within the simulated superscalar processor. On the average across all benchmarks, about 59% of bytes in the literals that are written to the IQ directly from the instruction register are the bytes

containing all zeroes. Within the data items being read from the register file to the IQ, 47% of all bytes are the bytes that contain all zeroes. On the issue path, 49% of all bytes are zero–bytes. On this path, the data stream represents a mixture of literals (with high percentage of zero bytes) and the regular data items (with lower percentage of zero bytes), and therefore the resulting percentage of zero bytes on this path is higher than on the path from the register file but lower than the percentage of zero bytes in literals. Finally, about 40% of all bytes travelling on the result/forwarding busses are zero bytes. The statistics of Figure 5–5 was collected only for the bytes with significant data. For example, if a 32–bit data piece is transferred, we only consider the 4 bytes that constitute these 32 bits. For a 64–bit piece of data we analyzed all 8 bytes.

By not writing zero bytes into the IQ at the time of dispatch, energy savings result as fewer bitlines need to be driven. Similarly, further savings are achieved during issue by not reading out implied zero valued bytes. This can be done by storing an additional bit with each byte that indicates if the associated byte contains all zeros or not. The contents of this zero indicator bit can be used to disable the word select strobe from going to the gates of the pass transistors. By controlling the sensitivity of the sense amps, we can also ensure that sense amp transitions are not made when the voltage difference on differential bitlines is below a threshold (as would be the case for the precharged bitline pairs associated with the bitcells whose readouts are disabled as described above.) Zero–valued bytes do not have to be driven on the forwarding buses that run through the IQ – this is another source of energy savings that result from zero–byte encoding.

Figure 5–6 depicts the circuitry needed to derive the zero indicator bit (ZIB) for a byte; such an encoder can be implemented within function units, decode/dispatch stage of the pipeline or within output latches/drivers. Two n–devices (*not shown*) are also used to discharge any charge accumulated on the gates on the two pulldown n–transistors to prevent false discharges of the output due to prior zero indications. The logic used to disable the readout of bytes with all zeroes is shown in Figure 5–7. P–transistor Q4 and n–transistor Q5

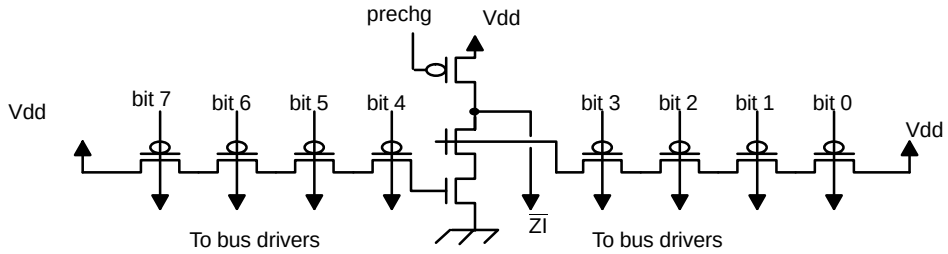


Figure 5–6. Encoding logic for all zero bytes

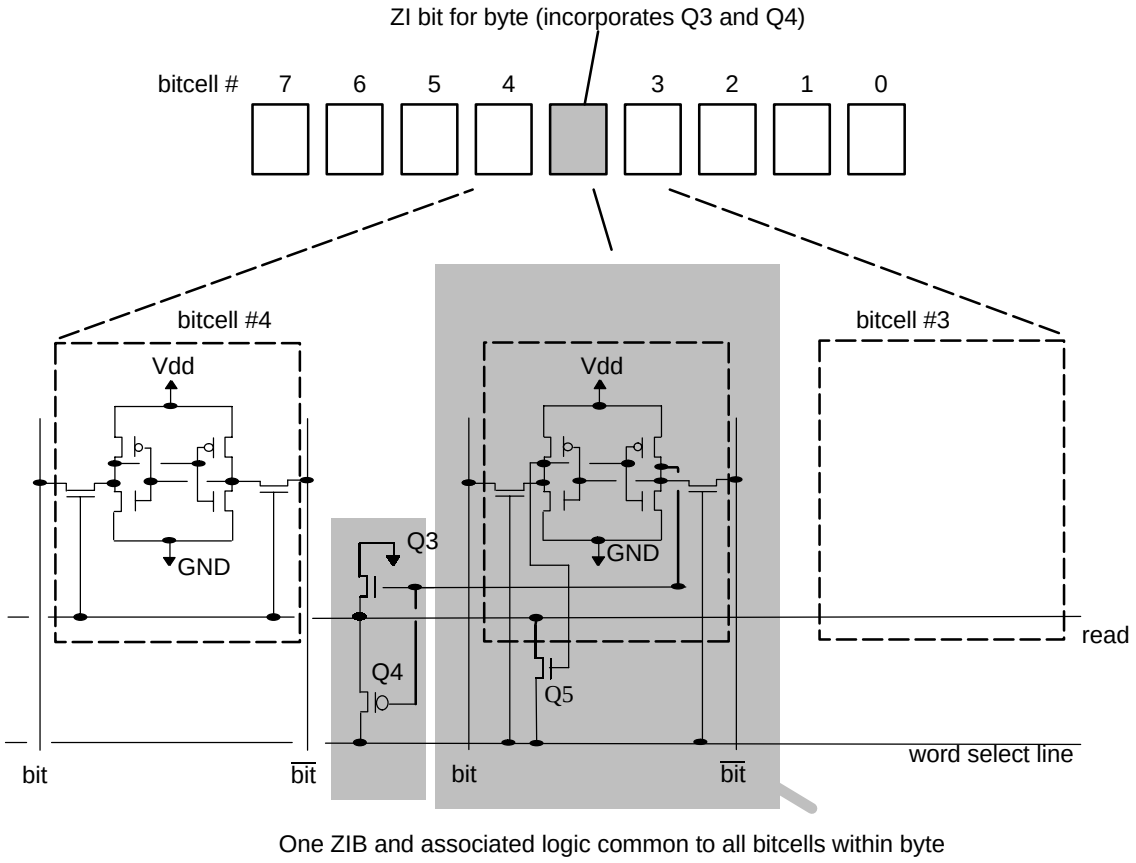


Figure 5–7. Bit–storage enhancements for avoiding the reading of all–zero bytes (for a single port)

are used to connect the read line of the corresponding byte to the word select line if the value stored in the ZIB is one. Otherwise, the local read line is grounded and the byte is not read out. The n–transistor Q5 in parallel with the p–transistor Q4 is needed to ensure that the local

word select line for a byte is properly and timely discharged at the end of the read cycle. Effectively, the p–transistor Q4 and the n–transistor Q5 formed a complementary switch to guarantee fast assertion and deassertion of the local word select line of a non–zero byte. To further limit the cycle time increase due to the use of ZIB logic, we used such pair of p and n transistors for every 4 bits, therefore requiring the use of 2 n and 2 p transistors for each byte. This increased the area of the issue queue and the device count, but resulted in significantly faster operation of the circuit. In our layouts, we also matched the height of a bitcell incorporating zero–byte encoding logic to that of the traditional bitcell by using the additional metal layer (metal 4 in this case). Thus, only the width of the issue queue increased as a result of using zero–byte encoding logic.

5.3. Area and Timing Considerations

The price paid for energy savings within the IQ through the use of zero–byte encoding is in the form of an increase in the area of the IQ by about 21% for the datapath with dispatch–bound operand reads and 17% for the datapath with issue–bound operand reads. The former number is higher, because more bytes are being zero–encoded if the operand results are stored in the issue queue. Therefore, the relative area increase is higher in that case. There is also some increase in the IQ access time, which we will now quantify. In the baseline issue queue, the read access can be performed in 566 ps (from the appearance of the address at the input to the driver of the decoder to the appearance of the data at the output of the sense amp) for the issue–bound datapath. The use of ZI bit to enable the corresponding byte for readout increases this delay to 616 ps, which represents just less than 9% increase in cycle time. For the dispatch–bound datapath, the data can be read from the baseline issue queue in 658 ps In the design with zero encoding logic, the delay increases to 740 ps, thus stretching the cycle by about 12%.

Notice that the additional delays introduced by the comparators and the zero–byte encoding logic are not cumulative. While comparators are used in the wakeup stage, the zero–byte encoding logic is used during the select cycle in the course of reading the selected instructions out of the issue queue. Since wakeup and select operations are spread out over two pipeline stages in high–frequency designs, the increase in pipeline cycle time is the maximum of the two overheads. If the wakeup cycle determines the critical path, then the comparator design proposed in this paper is not an attractive choice and other solutions, such as the ones proposed in [EGK+ 02] should be used instead. However, if a small slack exists in a stage where comparators are used (the wakeup stage), then the comparator of Figure 5–3 can be used without any effect on the cycle time. For example, consider the use of the proposed comparator in conjunction with zero–byte encoding logic for a dispatch–bound datapath. The zero–encoding logic increases the duration of the select cycle by about 80 ps while the new comparator increases the wakeup cycle by about 150 ps. So, if a slack of as little as 70 ps exists in the wakeup stage, the increase in the processor's cycle time will be bounded by the delays of zero–encoding logic and not by the delays of the comparator.

5.4. Using Bitline Segmentation in the IQ

Bitline segmentation is just one of the many techniques used to reduce power dissipation in RAMs [Itoh, ISN 95, EF 95]. In this section, we show how to incorporate bitline segmentation into our low power issue queue design.

As mentioned earlier, the IQ is essentially a register file with additional associative logic for data forwarding. Writes to the IQ, which occur at the time of dispatch, use the normal logic for a register file to set up the IQ entry for dispatched instructions. For each instruction dispatched in a cycle, a write port is needed. The only difference from a normal register file is that the word being written to from each write port is selected associatively (an associative search is needed to locate free entries, i.e., words within the IQ), instead of being selected

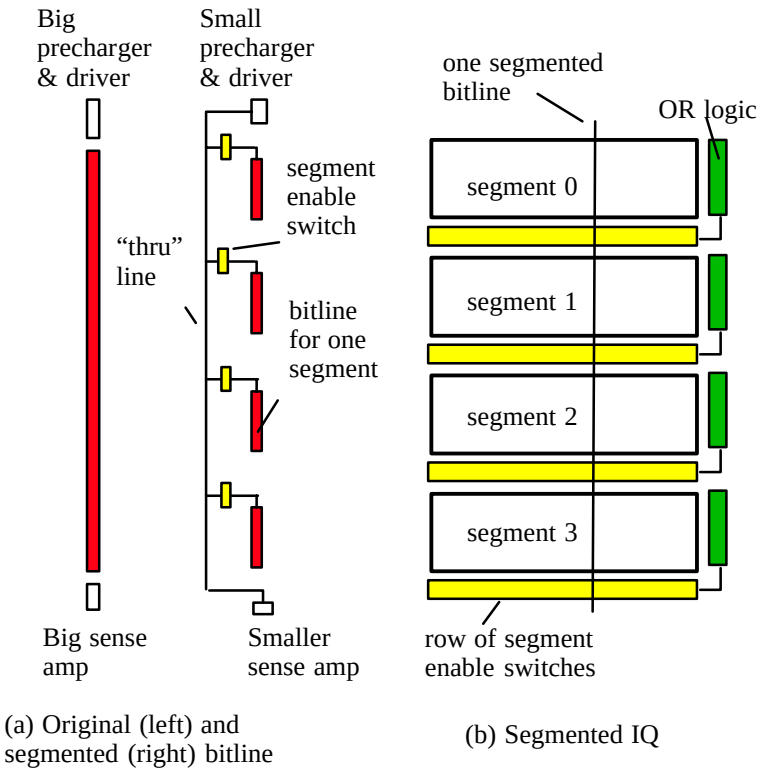


Figure 5–8. Bitline segmented IQ

through an explicit address. At the time of instruction issue, instructions ready for execution are read out from the IQ through independent read ports. Other than the use of the wakeup and arbitration logic to select ready entries, this readout is identical to what happens when a normal register file is read out. Sense amps, similar to those used in RAMs are needed to sense the data being read out as the bitlines are fairly long. As in a multiported RAM or register file, the bitlines in the IQ are a significant source of energy dissipation in the course of instruction dispatching (writes) and instruction issuing (reads). The bitlines associated with each read and write port present a high capacitive load, which consists of a component that varies linearly with the number of rows in the IQ. This component is due to the wire capacitance of the bitlines and the diffusion capacitance of the pass transistors that connect bitcells to the bitlines.

The capacitive loading presented by the bitlines in the IQ can be reduced through the use of bitline segmentation. The entire IQ is viewed as a linear array of segments for this purpose, with consecutive bitcell rows making up a segment. Figure 5–8(a) is useful in understanding how bitline segmentation reduces the capacitive loading encountered during reads and writes from the bitline. As an example, consider an IQ with 64 rows which has been restructured into 4 segments. Each segment will then consist of 16 consecutive rows. The original bitline, shown in the left of Figure 5–8(a) is loaded by the diffusion capacitance of 64 pass devices, the diffusion capacitances of the precharging and equlibrator devices, sense amp input gate capacitances and the diffusion capacitances of tri–stated devices used to drive the bitlines (during a write). In addition, there is the wire capacitance of the bitline itself. In the segmented version, the bitline is split into four segments; each segment of the bitline covers a column of the bitcells of the rows within a segment. As a result, the capacitive loading on each segment is lowered: each segment is connected to only 16 pass devices and the wire length of the bitline segment is one fourth of the original bitline.

To read a bitcell within a segment, the bitline for that segment has to be connected to the precharger and sense amp; for a write, the bitline segment has to be connected to the tri–state enable devices for the bitline driver. This is accomplished by running a “through” wire across all of the segments (typically in a different metal layer, right over the segmented bitlines), which is connected to the prechargers, sense amp and tri–state drivers as in the non–segmented design. A switch is then used to connect the segment in question to this “through” line. For the IQ, the segment switch is turned on by OR–ing the associative read or write enable flags for the port (associated with the bitline) for all the rows in that segment. As the effective capacitive loading on the through line and the connected bitline segment is smaller than the capacitive loading of the unsegmented bitline, lower energy dissipation occurs during reads or writes. The extent of energy savings depends on the relative contribution of the energy dissipated in the bitlines to the total energy expended within the issue queue. On the down side, additional energy dissipation occurs in the control logic for

the segment enable switch, the energy needed to drive the switches for all of the columns and the loading imposed on the through line by the diffusion capacitances of the complementary segment enabling switches. By carefully choosing the size of a segment, the overall energy dissipations can be minimized – making the size of a segment too small can actually increase the overall energy consumption, while making the size too large defeats the purpose of segmentation. An optimal segment size of 8 rows was discovered for the 64–entry IQ described here.

Figure 5–8(b) depicts a segmented IQ and shows that there is some increase in the overall area of the IQ due to the use of a row of segment enable switches with each segment. For the 0.18 micron CMOS layouts used here, the overall growth of the layout area of the 64–entry IQ when it was segmented into 8 segments (8 rows per segment) was only about 5%.

5.5. Evaluation Methodology

To evaluate the power savings achieved in the IQ by the use of the proposed mechanisms, we used the AccuPower toolset. The widely–used Simplescalar simulator [BA 97] was significantly modified (the code for dispatch, issue, writeback and commit steps was written from scratch) to implement *true hardware level, cycle–by–cycle* simulation models for such datapath components as the issue queue, the reorder buffer, physical register files, architectural register files and the rename table. Configuration of the simulated system is shown in Table 5–4. The execution of the SPEC 2000 benchmarks was simulated (each benchmark was run for 200 million instructions after a 200 million instructions startup phase). Benchmarks were compiled using the Simplescalar GCC compiler that generates code in the portable ISA (PISA) format. Specified optimization levels and reference inputs were used for all the simulated benchmarks.

Parameter	Configuration
Machine width	4–wide fetch, 4–wide issue, 4–wide commit
Window size	64 entry issue queue, 100 entry reorder buffer, 32 entry load/store queue, 128 Int phys. reg, 128 FP phys.reg.
L1 I–cache	32 KB, direct–mapped, 32 byte line, 2 cycles hit time
L1 D–cache	32 KB, 4–way set–associative, 32 byte line, 2 cycles hit time
L2 Cache combined	512 KB, 4–way set–associative, 64 byte line, 4 cycles hit time.
BTB	1024 entry, 2–way set–associative
Memory	128 bit wide, 12 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), 4–way set–associative, 30 cycles miss latency
Function Units and Latency (total/issue)	4 Int Add (1/1), 1 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 1FP Mult (4/1) / Div (12/12) / Sqrt (24/24)

Table 5–4. Architectural configuration of a simulated 4–way superscalar processor

Detailed accounting for detecting zero bytes in operands and lower level transition counting was implemented by a separate thread. Transition counts for reads, writes, associative addressing, FU arbitration, tag matching, data latching and other notable events were recorded separately. Transition counts and other data gleaned from the simulator were then fed into a power estimation program that used dissipation energies measured using SPICE for actual 0.18 micron layouts of key datapath components. (The process used was a 0.18 micron 6 metal layer CMOS process, TSMC). Our power estimation program generated power measures in picojoules for major energy dissipating events within the IQ for each benchmark in the SPEC 2000 suite individually as well as for the averages of the integer and floating point benchmarks and total averages.

5.6. Results and Discussions

In this Section we estimate the power savings that can be achieved within the issue queue if the three techniques described in this study are used. The extent of achievable power

savings, as well as the relative contribution of each technique to the total power reduction depends on the issue queue design style and whether all comparators are activated during the tag matching operation or only those comparators that are associated with the slots awaiting a result are activated. We now analyze the four possible combinations in detail. We begin with the issue queue of a datapath with dispatch–bound operand reads.

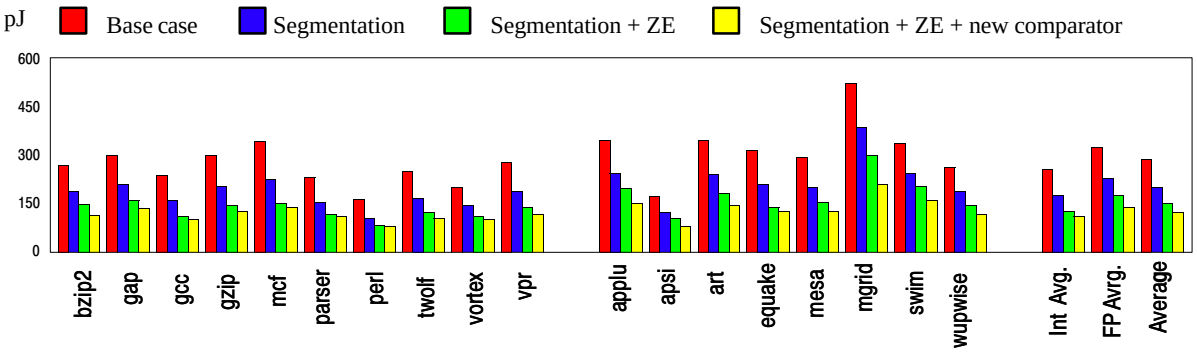


Figure 5–9. Total power dissipation within IQ using bitline segmentation, power efficient comparators and zero–byte encoding for the datapath with dispatch–bound register reads. Comparisons are only performed for the issue queue slots that wait for a result.

Figure 5–9 shows the total energy reduction within the issue queue if all three techniques discussed in this paper are used. Here, we assumed that forwarding comparisons are enabled only for IQ entries that are awaiting a result; comparisons are not done for unallocated entries or allocated entries that have already been forwarded the result. Bitline segmentation results in 30% of energy reduction on the average, the bulk of the savings coming from reduced bitline dissipations during writes to the issue queue at the time of instruction dispatching. Specifically, the dispatch power is reduced by more than 45%. Effects of the bitline segmentation are much less noticeable during reads from the issue queue (as that component of energy dissipation is dominated by the energy of sense amps). As the dispatch energy is a sizable portion of the total energy in this design, the overall energy savings due to the use of bitline segmentation are quite significant. Zero–byte encoding by itself saves about 28% of the issue queue energy – this is not shown in the graph. The energy expended during

instruction dispatching is reduced by 26%, during instruction issuing by 38% and the fact that fewer forwarding data lines need to be driven also reduces the forwarding energy by about 18%. To summarize, bitline segmentation and zero–byte encoding are equally effective at reducing the issue queue energy. The combination of zero–byte encoding and bitline segmentation results in 48% energy reduction in the issue queue. Notice that the combined energy reduction is a little smaller than the sum of the individual reductions, because zero–byte encoding is applied to the bitlines with the reduced capacitive load (due to the use of segmentation). Finally, the last bar of Figure 5–9 shows the energy reductions is the issue queue possible when all three techniques are used. The new comparator is 77% more energy–efficient on the average than the traditional comparator. However, in this configuration the forwarding energy is dominated by the energy of driving the tag and the data lines across the issue queue and not by the energy dissipated within the comparators. The energy savings in the forwarding component amount to 26% if the proposed comparators are used. In terms of total issue queue energy, the savings due to the use of the new comparators amount to only 9%. The combined effect of all three techniques is 56% of energy reduction in the issue queue – this is shown in the last bar of Figure 5–9.

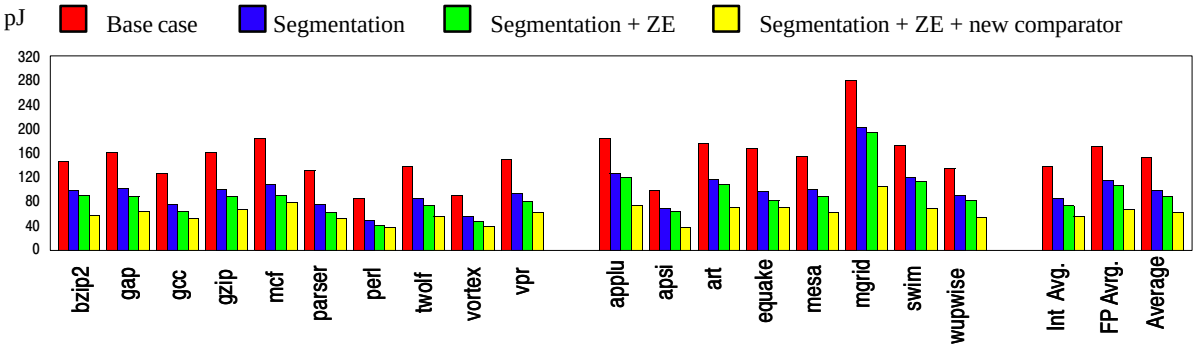


Figure 5–10. Total power dissipation within IQ using bitline segmentation, power efficient comparators and zero–byte encoding for the datapath with issue–bound register reads. Comparisons are only performed for the issue queue slots that wait for a result.

Next, we show similar results for the issue queue used in datapaths with issue–bound operand reads. Figure 5–10 depicts the power savings achievable within the issue queue, assuming that only useful tag comparisons are performed. Here, the bitline segmentation results in 36% of energy savings. The zero–byte encoding results in 12% energy reduction, since the encodable operands are no longer held in the issue queue. The proposed comparator reduces the forwarding energy by 66% and the overall energy of the issue queue by 16%. The combined effect of the three techniques is 59% reduction in the issue queue energy dissipations. If we compare the results of Figure 5–9 and Figure 5–10, the relative contribution of each technique varies, but the overall achievable energy savings are more than 55% for both designs.

To summarize, the importance of each discussed power reduction technique depends on the datapath design style (dispatch–bound register reads vs. issue–bound register reads). The combined power savings in the issue queue range between 56% and 59% for various design alternatives.

5.7. Related Work

The problem of power reduction in dynamically scheduled superscalar processors in general, and in the issue queues in particular, has received a considerable attention in recent literature. One approach to power minimization in the issue queues has to do with dynamic resizing of the queue based on the requirements of the application. Several resizing techniques have been proposed along these lines. Specifically, in [BSB+ 00], Buyuktosunoglu explored the design of an adaptive issue queue, where the queue entries were grouped into independent modules. The number of modules allocated was varied dynamically to track the ILP; power savings was achieved by turning off unused modules. In [FG 01], Folegnani and Gonzalez introduced a FIFO issue queue that permitted out–of–order issue but avoided the compaction of vacated entries within the valid region of

the queue to save power. The queue was divided into regions and the number of instructions committed from the most–recently allocated issue queue region in FIFO order (called the "youngest region") was used to determine the number of regions within the circular buffer that was allocated for the actual extent of the issue queue. To avoid a performance hit, the number of regions allocated was incremented by one periodically; in–between, also at periodic intervals, a region was deactivated to save energy/power if the number of commits from the current youngest region was below a threshold. In [PKG 01b], we introduced a resizing mechanism, where the IQ occupancy (measured as the number of valid entries within the IQ) was used as an indication of resource needs to drive the issue queue resizing, in conjunction with similar resizing of the reorder buffer and the load/store queue.

In [BM 01c], Bahar and Manne studied the multi–clustered Compaq 21264 processor with replicated register files. The dispatch rate was varied between 4, 6 and 8 to allow an unused cluster of function units (including per–cluster issue queues) to be shut off completely. The dispatch rate changes were triggered by the crossing of thresholds associated with the floating point and overall IPC, requiring dispatch monitoring on a cycle–by–cycle basis. Significant power savings within the dynamic scheduling components were achieved with a minimum reduction of the IPC.

In [BBS+ 00], Brooks et. al. suggested that the use of comparators that dissipate energy on a tag match could significantly reduce the power dissipated in the issue queue. The actual designs of the comparators and the extent of achievable power savings were, however, not presented anywhere. In [KGPK 01, PKE+ 03], we proposed a design of a dissipate–on–match comparator and quantified the power savings realized by using such comparators within the issue queue. Alternative comparator designs were also proposed in our recent work in [EGK+ 02].

In [PK+ 03], we propose a series of energy efficient long comparator designs for superscalar microprocessors. Our designs combine the use of 8–bit blocks built using

traditional comparators with 8–bit blocks built using dissipate–on–match comparators. We then evaluate the use of these circuits within the address comparison logic of the load–store queues. We found that for the same delay, the hybrid design consisting of one dissipate–on–match and three traditional 8–bit comparators is the most energy efficient choice for the use within the load–store queue, resulting in 19% energy reduction compared to the use of four traditional 8–bit blocks. The results presented in this paper can be easily extended to the TLBs, highly–associative caches and the BTBs.

Two recent efforts [PJS 97, CG 00] have attempted to reduce the complexity of the issue queue. The implications of this reduced complexity at the cost of additional peripheral logic, on the overall power dissipation is not addressed. In [PJS 97], Palacharla et. al. proposed an implementation of the issue queue through several FIFOs, so that only the heads of each FIFO need to be examined for issue. This significantly reduced the number of instructions that need to be considered for issue every cycle.

The technique for reducing the issue queue complexity as proposed by Canal and Gonzalez in [CG 00] exploits the fact that the majority of operands in a typical application are read at most once. Instructions that had all source operands ready at the time of decode were dispatched into the ready queue (one such queue was assumed for each FU), from where they were issued for execution in program order. Instructions, whose operands were not ready at the time of decode, but which were the first consumers of required source register values, were dispatched into a separate data structure called First–use table, from where, upon the availability of both sources, they were eventually moved to the appropriate ready queue. The First–use table was indexed by the physical register identifier of each source (instruction could end up in two entries of the First–use table if both sources were not available at the time of decode and the instruction was the first consumer of both sources), which allowed the produced results to be supplied directly to the waiting instructions without

any associative tag comparison. Instructions, that were not the first consumers of their source registers were dispatched into a small out–of–order issue queue and traditional issue mechanism (including the associative tag comparison) was used for those instructions to keep performance degradation to a minimum. The scheme of [CG 00] reduces the amount of associative lookup in the issue process by limiting it only to the instructions within the small out–of–order queue. This results in significant reduction of the complexity of the control logic, but the effect on power dissipation is not clear, as the additional structures used in [CG 00] certainly have some power overhead. Canal and Gonzalez's scheme also requires to perform a lookup of the First–use table at the time of dispatch. This will almost inevitably result in the use of a slower clock or the use of multiple dispatch stages, which can effect the overall performance adversely.

A related effort is the work of [HRT 02], where Huang et.al. proposed to save energy in the issue queue by using indexing to only enable the comparator at the single instruction to wake up. In rare cases, the mechanism reverted back to the usual tag broadcasts, when more than one instruction needed to be awakened.

In [EA 02], Ernst and Austin explored the design of the issue queue with reduced number of tag comparators. They capitalized on the observation that most instructions enter the instruction window with at least one of the source operands ready. These instructions were put into specialized entries within the issue queue with only one (or even zero) comparators. The authors of [EA 02] also introduced the last tag speculation mechanism to handle the instructions with multiple unavailable operands. The combination of the two techniques reduced the number of tag comparators used by 75% compared to the traditional issue queue designs.

5.8. Conclusions

We studied three relatively independent techniques to reduce the energy dissipation in the instruction issue queues of modern superscalar processors. First, we proposed the use of comparators in forwarding/tag matching logic that dissipate the energy mainly on the tag matches. Second, we considered the use of zero–byte encoding to reduce the number of bitlines that have to be driven during instruction dispatch and issue as well as during forwarding of the results to the waiting instructions in the IQ. Third, we evaluated power reduction achieved by the segmentation of the bitlines within the IQ. Combined, these three mechanisms reduce the power dissipated by the instruction issue queue in superscalar processors by 56% to 59% on the average across all simulated SPEC 2000 benchmarks depending on the datapath design style.

The IQ power reductions are achieved with little compromise of the cycle time. Specifically, zero–byte encoding logic adds 9% to the cycle time of the datapath with issue–bound operand reads and 12% to the cycle time of the dispatch–bound datapath. The delay of the new comparator roughly doubled compared to the delay of the traditional comparator and therefore the practical use of the proposed comparator is determined by the amount of slack (if any) present in the wakeup stage of the pipeline. If the wakeup cycle determines the processor's cycle time, then the alternative dissipate–on–match comparator designs that we proposed in [EGK+ 02] are better solutions. It is very likely that the issue queue area increase when all three proposed techniques are used will not be cumulative. In the worst case, assuming cumulative effect of our techniques on the issue queue area, the total area of the issue queue increases by about 25%. Our ongoing studies also show that the use of all of the techniques that reduce the IQ power can also be used to achieve reductions of a similar scale in other datapath artifacts that use associative addressing (such as the reorder buffer [PKG 02b] and load/store queues). As the power dissipated in instruction dispatching, issuing, forwarding and retirement can often be as much as half of the total chip

power dissipation, the use of the new comparators, zero–byte encoding and segmentation offers substantial promise in reducing the overall power requirements of contemporary superscalar processors.

Chapter 6

Complexity–Effective Reorder Buffer Design

The typical implementation of a ROB is in the form of a multi–ported register file (RF). In a W–way superscalar machine where the physical registers are integrated within the ROB, the ROB has the following ports:

- At least 2*W read ports for reading source operands for each of the W instructions dispatched/issued per cycle, assuming that each instruction can have up to 2 source operands.
- At least W write ports to allow up to W FUs to write their result into the ROB slots acting as physical registers in one cycle.
- At least W read ports to allow up to W results to be retired into the ARF per cycle.
- At least W write ports for establishing the PRF entries for co–dispatched instructions.

The actual number of ports can double if a pair of registers are used to hold double–width operands. Alternatively, double–width operands can be handled by using a pair of register files in parallel, each with the number of ports as given above – this approach is wasteful when double and single width operands are intermixed – as is typical. Complicated allocation and deallocation schemes have to be used if such wastages are to be avoided. At low clock frequencies, the number of ports needed on the ROB can be reduced by multiplexing the ports – this, alas, is a luxury not permissible in the designs that push the

performance envelope. As the number of ports increases, the area of the RF grows roughly quadratically with the number of ports. The growth in area is also accompanied by a commensurate increase in the access time stemming from the use of longer bit lines and word lines, bigger sense amps and prechargers.

The large number of ports on the ROB results in two forms of penalties that are of significance in modern designs. The first penalty is in the form of the large access delay that can place the ROB access on the critical path; the second is in the form of higher energy/power dissipations within the highly multi–ported RFs implementing the ROB.

Our first approach to the ROB complexity minimization is based on the observation that in a typical superscalar datapath, almost all of the source operand values are obtained either through data forwarding of recently generated results or from the reads of committed register values. Only a small percentage of operands, which are not generated recently, need to be read from the ROB; delaying such reads have little or no impact on the overall performance. This allows the 2*W ports for reading source operands at the time of dispatch or issue to be completely eliminated with small impact on performance. This scheme was described in our ICS'02 paper [KPG 02]. We extend that work in the following ways:

1) We extend the approach of [KPG 02] to further reduce the overall complexity, power and delay of the ROB by implementing a centralized ROB as a distributed structure. Each distributed component of the ROB, assigned to a single FU, has a single write port for committing the result as opposed to W ports that are needed in a centralized implementation. Unlike recently proposed approaches that simplify a multi–ported RF by decomposing it into identical components, our ROB components are not homogeneous in size: the size of each ROB component is tailored to the assigned FU, allowing its delays and dissipations to be tuned down further. Furthermore, such distribution eliminates all conflicts over the write ports, which are inherent in most previously proposed designs that used multi–banked organization.

2) We demonstrate that the port reduction scheme proposed in [KPG 02] can be naturally combined with multi–banked ROB. The net effect is the conflict–free ROB distribution scheme, where conflicts over the write ports are eliminated by the nature of the ROB distribution and conflicts over the read ports are eliminated by simply removing the read ports from the ROB. As far as we know, this is the first scheme that uses multibanking for energy reduction and yet avoids all port conflicts and associated performance degradations.

3) We study the effect of distributing the retention latches across the functional units, thus achieving higher performance than the scheme of [KPG 02].

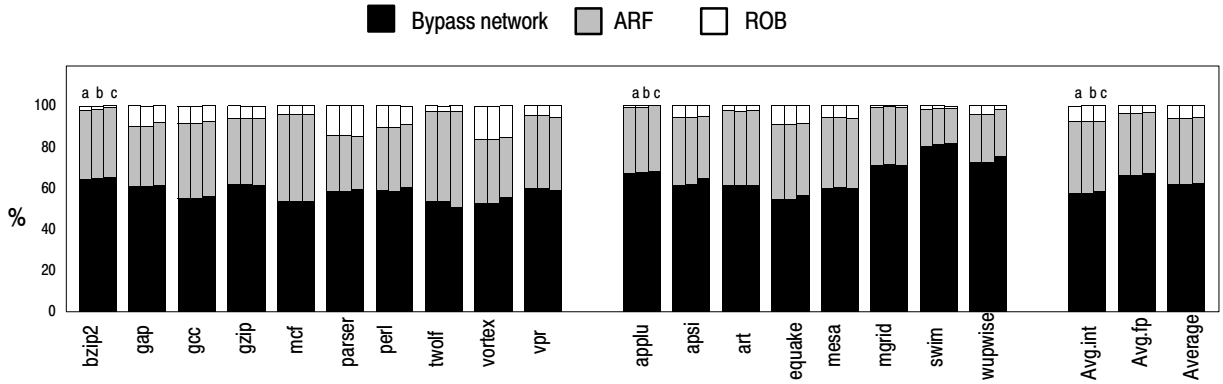
4) We optimize the scheme by filtering short–lived results. This results in better utilization of retention latches. First, the hit ratio to retention latches increases, considerably. Second, the reduction in the size of the retention latches becomes feasible. More detailed discussion will be presented in the following sections.

5) Finally, we evaluate the impact of our mechanisms on the datapath that maintains the non–committed register values in the set of rename buffers, instead of keeping them directly in the ROB slots.

The approaches presented in this study for reducing the ROB complexity have some close parallels with other recent work on reducing the complexity of register files and reducing the register file access latencies, particularly the ones described in [BDA 01, BTME 02, HM 00]. Our techniques are applicable to processors that read the register values at the time of instruction dispatching and store these values in the issue queue entry allocated for the instruction. The non–committed result values can be stored either within the ROB slots directly or in the separate rename buffers. In this study we provide the results for both of these schemes.

This work was primarily motivated by observing the sources of the operands for dispatched instructions. A detailed simulation of the SPEC 2000 benchmarks [Sp 00] on the

baseline processor of Figure 6–1 reveals an interesting fact – only a small fraction of the source operand reads require the values to come from the ROB. Figure 6–1 depicts the percentage of operand reads that are satisfied through forwarding, from reads of the ARF or from reads of the ROB, for three different configurations. The three configurations differ in the number of ROB entries and the issue widths. As seen from Figure 6–1, for a 4–way machine with 72–entry ROB, more than 60% of the source operands arrive through forwarding network. Sizable percentage of operands (about 32%) comes from the ARF. These are mostly the values of stack pointer, frame pointer and base address registers for the memory instructions. Irrespective of processor configuration, the percentage of cases when source operand reads are satisfied from the ROB represents only a small percentage of all source operands. For example, for a 4–way processor with a 72–entry ROB, only 0.5% through 16% of the source operands are obtained from the ROB with the average of 5.9% across all simulated benchmarks. Note, that the results change slightly between the three considered configurations in a non–intuitive manner because of dynamics of instruction processing. The rather small percentage of reads of source operand values from the ROB suggests that the number of read ports for reading source operands from the ROB can be possibly eliminated altogether without any noticeable impact on performance.



(a) 72–entry ROB, 4–way processor (b) 96–entry ROB, 4–way processor (c) 128–entry ROB, 6–way processor

Figure 6–1. The origin of source operands in the baseline superscalar processor

Completely eliminating the read ports for reading source operand values is fundamentally different from reducing the number of ports in two ways. First, the elimination of ports removes the ROB access from the critical path. Of course, other means of supplying the source operand values to the dispatched instructions must be provided, as discussed later. In contrast, even if a single port for reading out the source operands is retained, the two–cycle ROB access time (which cannot be reduced because the ROB still remains heavily–ported due to the ports needed for writeback, commitment and entry setup) still resides on the critical path. Second, complete elimination of ports avoids design complications associated with managing the register file (ROB) with reduced number of ports, as documented in Borch et.al. [BTME 02]. A desirable side–effect of eliminating the source read ports on the ROB is that we only read the source values that are committed from the ROB, thereby eliminating some – but not all – spurious computations using results that are generated on the mispredicted path(s).

Even if the source operand read ports of a ROB are eliminated completely, in a W–way machine, W (or more) write ports are needed to write the results from FUs into the ROB slots and W read ports are needed to commit/retire results. The number of these ports can be reduced drastically by distributing the ROB into components, with one component assigned to a single FU. This drastic distribution can help in the following ways:

- The size of each component can be tailored to the FU to which it is assigned.
- Since a FU can only write at most a single result per cycle, a single write port on each component will suffice. Furthermore, this avoids the need to use any logic used for arbitrating access to shared ports and FU output buffers for temporarily holding results produced by FUs that fail in such arbitrations, as in a centralized implementation or in a minimally ported multi–banked implementation.
- A single read port can be used on each component to commit result from the ROB component to the ARF. A simple round robin style instruction allocation within an

identical set of FUs (such as a pool of integer units) can be used to guarantee that all co–issued instructions will be retired from different ROB components in a common cycle. Where one or a few FUs of the required type exists, there is little or no choice in the allocation of an instruction to FUs. Very rarely, and only if mutiple consecutive instructions of the same type (for example, MUL) are allocated to the same ROBC, will we have a situation where two or more results to be retired in a common cycle come from the same ROB component. A single read port on each of the ROB components is thus likely not to be an impediment in retiring instructions. We present detailed results supporting this claim in Section 6.6.

In the next subsections, we describe the datapath variations that reduce the ROB complexity as described in this section. Our experimental results also show how the expectations described above turn out to be true – significant complexity reductions are achieved with only 1.7% degradation of the IPC on a 4–way machine on the average across SPEC 2000 benchmarks [Sp 00].

6.1. Eliminating Source Operand Read Ports on the Baseline ROB

Figure 6–2 depicts a datapath that exploits the observation made from Figure 6–1 to completely eliminate the read ports used for reading source operands on the ROB in the baseline design of Figure 2–10.

Results are written into the ROB and simultaneously forwarded to dispatched, waiting instructions on the result/status forwarding bus. Till the result is committed (and written into the ARF) it is not accessible to any instruction that was dispatched since the result was written into the ROB. Since it is not possible for dispatched instructions to repeatedly check for the appearance of a result in the ARF (and then read it from the ARF), we need to supply

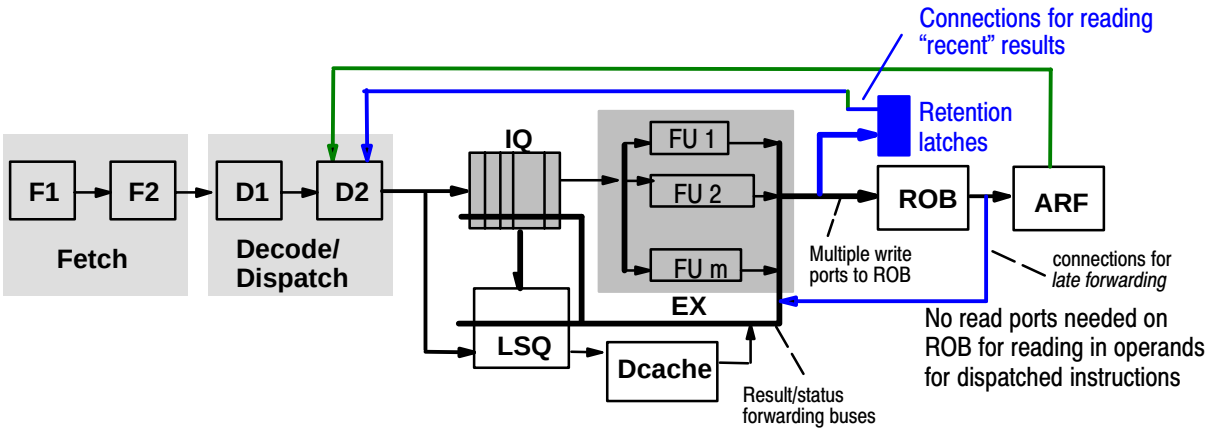


Figure 6–2. Superscalar datapath with the simplified ROB and retention latches

the operand value to instructions that were dispatched in the duration between the writing of the result into the ROB and the cycle just prior to its commitment to the ARF. As shown in Figure 6–2, we do this by simply forwarding the value (again) on the forwarding buses at the time it is committed. We call this *late forwarding*. Similar mechanism was used in [SV 87], where the bus from the Register Update Unit to the register file was monitored to resolve data dependencies. We apply this concept in the context of modern superscalar out–of–order processor.

Regular forwarding performed at the time of writeback is always given priority over the late forwarding. This avoids the stalling of a FU pipeline and does not require any changes to the issue logic. Committment process is stalled if a forwarding bus can not be obtained for a result that needs to be forwarded for the second time. We will see shortly that there is no need to forward each and every result from the ROB at the time of commitment. Neither do we need to increase the number of forwarding paths compared to the baseline case, as the utilizations of such paths are quite low.

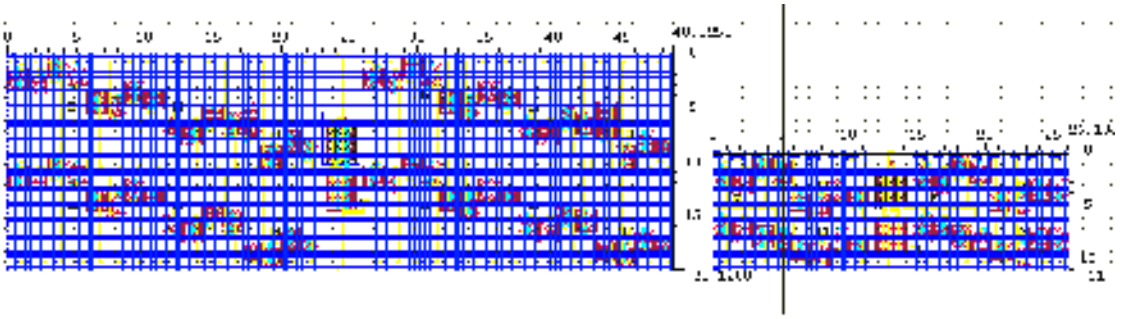
Figure 6–3 depicts 32 and 16–ported ROB SRAM bitcells side by side to show the complexity reduction that is possible with our scheme. The area reduction in bitcell area is as much as 71%. The total area reduction in ROB is around 45% (It is not as much as 71%

since some parts of the ROB still have full–fledged ports.) Energy dissipation is also reduced because of the following facts:

a) Bitlines and wordlines become shorter,

b) Capacitive loading is reduced, and

c) Fewer decoders, drivers and sense amps are used.



a) 32–ported SRAM bitcell **b) 16–ported SRAM bitcell**

Figure 6–3. Comparison of ROB bitcells (0.18 u, TSMC)

6.1.1. Reducing the Performance Degradation

The elimination of the ROB read ports for reading out source operand values results in "disappearance" of these values for several cycles. Specifically, the value is initially available through forwarding network in the writeback cycle. As operand read ports are eliminated in the ROB, the value is not available again until it commits from the ROB to the ARF, at which point the late forwarding of the same value occurs. In between these two times, instructions requiring the value as a source cannot be issued, resulting in a performance penalty. We avoid this penalty in most cases, as seen from the results presented later, by using a small set of latches – called **retention latches** – to cache recently written results. A result written by a FU will thus write to the retention latch and to the ROB entry of the corresponding instruction in parallel.

Each retention latch is capable of holding a single–precision result, so double precision results would need two latches. Instructions dispatched since the writing of a result to the ROB can still access the value as long as they can get it from the retention latches. In case the lookup for source operand fails to find it within the retention latches, the operand value will be eventually obtained through late forwarding of the same value when it is written to the ARF from the ROB.

6.1.2. Retention Latch Management Strategies

We study three variations of managing the retention latches. The first one is a simple FIFO scheme, where the retention latch array is used as a shifter. The writing of a result into a retention latch (or a pair of retention latches) causes the contents of one (or two) earliest written latch(es) – at the tail end of the latch array – to be shifted out and lost. The new results are written into the vacated entries at the head of the latch array. Even though the number of latches in the shifter array is small compared to the ROB size, it is possible, albeit highly unlikely, that duplicate entries keyed by a common ROB index will co–exist within the retention latches. This can, for example, happen in an extreme scenario when an instruction with a destination register is allocated to an ROB entry, and there is no other instruction with destination register (only stores and branches are encountered) until the same ROB entry is again reused for allocations. In such case, the contents of retention latches are not shifted and we end up having two retention latches keyed with the same physical register id. Branch mispredictions can also cause a ROB slot to be reallocated and two entries keyed with the same ROB index to co–exist within the retention latches. To avoid any ambiguity in finding the correct entry in such situations, we design the matching logic to return the most recently–written result in the retention latches. This is relatively easy to achieve, since the most recently–written value is always the closest to the head end of the latch array. The latch array is designed as a multi–ported structure to support simultaneous writes and reads. To

support the reading and writing of up to W double precision or W single precision results from/to the retention latches, they are implemented to support 4*W associative addressing ports (requiring 4*W comparators per latch) and to support 2*W write ports.

The second variation of the retention latch structure is one where the latch contents are managed in a true LRU fashion. A true LRU scheme can be implemented since the number of latches is small (8 to 16). The LRU management policy allows only recently–used result values – and ones that are likely to be used again – to be retained within these latches. The LRU management policy is offering a better performance on the average across the simulated benchmarks. As in the case of FIFO latches, it is conceivable, that two different entries, keyed with the same ROB index can reside within the retention latches. This happens when a result value continues to be frequently used from the retention latches and when the ROB slot is committed and eventually allocated to another instruction that establishes a duplicate entry in the retention latches. In contrast to the FIFO latches, the logic needed for selecting the most recently–produced value is more complicated, because it can be positioned anywhere in the latch array. For this reason, we avoid the need for any disambiguating logic by deleting the entry for a ROB slot from the retention latches when its contents are committed to the ARF. We do this by associatively addressing the retention latches at the time of committing and marking the matching entry, if any, as deallocated. Branch mispredictions can also introduce duplicate entries in the retention latches, keyed with a common ROB index. We avoid this by invalidating the existing entries for ROB indices that are to be flushed because of the misprediction. This later invalidation is accomplished in a single cycle by adding a “branch tag” to the instructions and to the retention latch keys to invalidate retention latch entries that match the tag of the mispredicted branch. A simpler solution is to flush the entire set of retention latches in the case of branch misprediction. This alternative degrades performance to a very little extent.

The third conceivable option is to use retention latches with random replacement policy. While this scheme is inferior in performance to both FIFO and LRU latches (as we demonstrate in Section 6.6.1), none of the design complications associated with LRU latches is eliminated: the corresponding entries still need to be flushed from the latches at the time of result commitment and some form of flushing is needed on branch mispredictions.

6.1.3. Optimization of RLs Through Selective Operand Caching

One inefficiency of the scheme proposed so far stems from the fact that all generated results, irrespective of whether they could be potentially read from the retention latches (RLs), are written into the latches unconditionally. Because of the small size of the RL array, this inevitably leads to the evictions of some useful values from the RLs. As a consequence, the array of RLs is not utilized efficiently and performance loss is still noticeable because many source operands can only be obtained through the late forwarding, thus delaying a sizable number of instructions. If, however, one could somehow identify the values which are never going to be read after the cycle of their generation and avoid writing of these values into the RLs, then the overall performance could be improved significantly. In this section, we propose exactly such a mechanism. Our technique leverages the notion of short–lived operands – values, targeting architectural registers which are renamed by the time the instruction producing the value reaches the writeback stage.

Short–lived values do not have to be written into the RLs, because all instructions that require this value as a source will receive it through forwarding. Since the destination register is renamed by the time of instruction writeback, all dependent instructions are already in the scheduling window and the produced value is supplied to these instructions in the course of normal forwarding. Since the instruction queue entries have the data slots in addition to the result tags, the produced values are latched in the queue entries associated with the dependent instructions. Consequently, the values of short–lived operands will never

be sought from the ROB or from the RLs. It is thus safe to avoid writing such short–lived values into the RLs, thus preserving the space within the RL array for the results that could actually be read. The short–lived values still, however, have to be written into the ROB to maintain the value for possible recovery in the cases of branch mispredictions, exceptions or interrupts. In the rest of this section, we describe the hardware needed for the identification of short–lived values and quantify the percentage of short–lived results generated across the simulated execution of the SPEC 2000 benchmarks.

We first outline how the register renaming is implemented in processors that use the ROB slots to implement repositories for non–committed results. The ROB slot addresses are directly used to track data dependencies by mapping the destination logical register of every instruction to the ROB slot allocated for this instruction, and maintaining this mapping in the *Register Alias Table (RAT)* until either the value is committed to the ARF, or the logical register is renamed. If an instruction commits and its destination logical register is not renamed, the mapping in the RAT changes to reflect the new location of the value within the ARF. Each RAT entry uses a single bit (the “location bit”) to distinguish the locations of the register instances. If the value of this bit is set for a given architectural register, then the most recent reincarnation of this register is in the ROB, otherwise it is in the ARF. When a value is committed and no further renaming of the architectural register targeted by this value occurred, the location bit in the RAT is reset. With this in mind, we now describe the mechanism for identifying short–lived values.

6.1.3.1. Detecting Short–Lived Results

The extensions needed in the datapath for the identification of short–lived operands are very simple, they are similar to what is used in [MRH+ 02] for early register deallocation and in [BDA 01] for moving the data between the two levels of the hierarchical register file. We maintain the bit vector, called *Renamed*, with one bit for each ROB entry. An instruction that renames destination architectural register X sets the *Renamed* bit of the ROB entry

corresponding to the previous mapping of X, if that mapping points to a ROB slot. If, however, the previous mapping was to the architectural register itself – i.e., to a committed value, no action is needed because the instruction that produced the previous value of X had already committed. These two cases are easily distinguished by examining the location bit in the RAT. *Renamed* bits are cleared when corresponding ROB entries are deallocated. At the end of the last execution cycle, each instruction producing a value checks the *Renamed* bit associated with its ROB entry. If the bit is set, then the value to be generated is identified as short–lived.

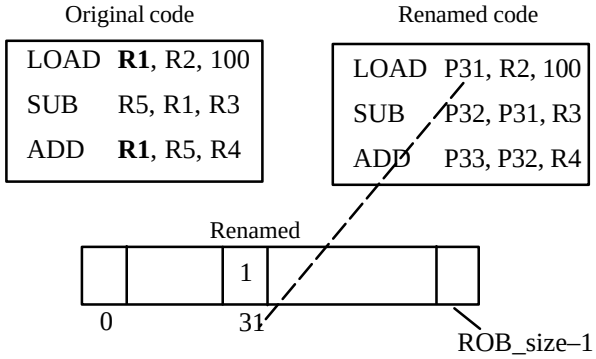


Figure 6–4. Identifying short–lived values

Figure 6–4 shows an example of identifying a short–lived value. The instruction ADD renames the destination register (R1) previously written by the LOAD instruction. Assume that the ROB entries numbered 31 and 33 are assigned to the instructions LOAD and ADD respectively. When the ADD is dispatched, it sets the *Renamed* bit corresponding to the ROB entry 31 (the previous mapping of R1), thus indicating that the value produced by the LOAD *could be* short–lived. When the LOAD reaches the writeback stage, it examines *Renamed[31]* bit and identifies the value it produces as short–lived. Notice, however, that the indication put by the ADD is just a hint, not necessarily implying that the value produced by the LOAD will be identified as short–lived. For example, if the LOAD had already passed the writeback stage by the time the ADD set the value of *Renamed[31]* bit, then the LOAD

would have not seen the update performed by the ADD and the value produced by the LOAD would not be identified as short–lived.

At the time of writeback, only the values that were not identified as short–lived are inserted into the RLs. To understand why such separation of values is useful, it is instructive to examine the results presented in Figure 6–5, which shows the percentage of short–lived result values for realistic workloads. Across the SPEC 2000 benchmarks, about 87% of all produced values were identified as short–lived on the average in our simulations, ranging from 97% for *bzip2* to 71% for *perl*.

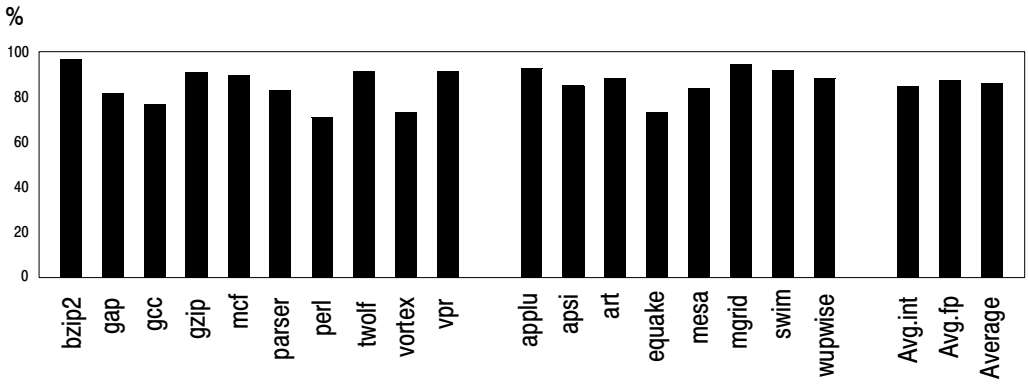


Figure 6–5. Percentage of short–lived values

The advantages of selective result caching in the RLs are two–fold. First, the performance improves as the RLs are used more efficiently. Alternatively, fewer RLs can be used to maintain the same performance. Second, the design is even more energy–efficient as fewer writes to the RLs take place. In addition, as the percentage of variables not identified as short–lived is small, fewer write ports to the RLs are needed to sustain performance. We now describe the main steps involved in the resulting datapath.

6.1.3.2. Instruction Renaming and Dispatching

A group of logically consecutive instructions is decoded and the corresponding RAT entries are updated. The *Renamed* bit vector is updated as described earlier in this section. For each dispatched instruction, the source operands are handled as follows:

If the source operand is in the ARF (as indicated in the "location bit" within the RAT), it is read out from the ARF and stored in the issue queue entry of the instruction.

If the result has not been committed, as evident from the setting of the location bit for the RAT entry for source, an attempt is made to read the operand from the RL array associatively using the ROB slot corresponding to the source operand as the key. In parallel, the bit indicating if the ROB entry for the source holds a valid result ("result valid" bit) is read out from the ROB. (For faster lookup, the vector of result valid bits and the vector of "late forwarding" can be implemented as individual smaller, faster arrays.) Three cases arise in the course of the associative lookup of the RLs:

Case 1: If the source value is in one of the RLs, it is read out and stored in the issue queue entry for the instruction.

Case 2: If the value was not found in the RL array but the result was already produced and written to the ROB ("result valid" bit for the ROB entry was set), the corresponding tag field for the operand within the issue queue entry for the instruction is set to the ROB index read out from the RAT. The late forwarding bit of the ROB entry is also set in this case. This allows the waiting instruction to read out the result when it is committed from the ROB in the course of *late* forwarding.

Case 3: If the value was not found in the RLs and it is yet to be produced, as evident from the setting of the "result valid" bit of the ROB entry, the corresponding tag field for the operand within the issue queue entry for the instruction is set to the ROB index read out from

the RAT. In this case, the source operand will be delivered to the waiting instruction in the course of *normal* forwarding.

To complete the dispatching process, the index of the bit in the *Renamed* vector that was altered by the dispatched instruction, is saved in the ROB entry for the instruction. This allows the state of the *Renamed* vector to be restored on branch mispredictions.

6.1.3.3. Instruction Completion and Commitment

During the last execution cycle, the *Renamed* bit associated with the destination ROB slot is examined. If it is set, the result is simply written to the ROB entry in the writeback cycle. If the *Renamed* bit is not set, the result is additionally written to a pre–identified victim entry in the RL.

The only additional step needed during instruction commitment has to do with the late forwarding mechanism. If the late forwarding bit is set, a reservation is requested for a forwarding bus. The commitment process stalls until such a bus is available. When the forwarding bus becomes available, the result being committed is forwarded – for a second time – as it is written to the ARF.

In Section 6.6, we quantify the performance gains and the additional energy savings achievable through the exploitation of short–lived values.

6.1.4. Reducing Forwarding Bus Contention

To avoid the late forwarding of committed values unnecessarily, a single bit flag, *late–forwarding–needed*, is maintained in each ROB entry. This bit flag is cleared when the ROB entry for an instruction is set up. If at dispatch/issue time a required source operand was sought from the ROB (as indicated by the rename table), the *late–forwarding–needed* bit in its ROB entry is set, even if the actual result was obtained from the retention latches. At the time of committing this entry into the ARF, a late forwarding is needed only when

the *late–forwarding–needed* flag of the entry is set. Our results show that on the average, less than 4% of the ROB entries that are committed require late forwarding when a 8–entry FIFO retention latch set with two read ports are used on a 4–way machine. This is again consistent with the results of Figure 6–1 – only a small percentage of sources are sought from the ROB and thus very few ROB entries are marked with *late–forwarding–needed* flag. This is the real reason why we do not need to have additional forwarding buses to handle late forwarding – the existing set of W forwarding buses in a W–way machine can handle normal and late forwarding simultaneously, without impacting the IPC in any noticeable way. This optimization requires 4*W additional 1–bit wide ports to the simplified ROB.

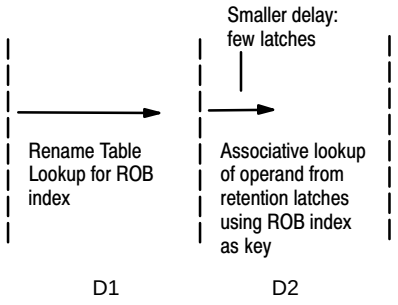


Figure 6–6. Assumed timing for the low–Complexity ROB scheme

The small size of the retention latch set and the small number of read ports allow source operands to be read with little delay, removing the otherwise slow ROB from the critical path. Figure 6–6 shows the resulting timing of operand access using the retention latches. The ROB itself is faster and less energy consuming in this case than the baseline ROB, because of the complete elimination of all source read ports on the ROB.

6.2. Fully Distributed ROB

Figure 6–7 depicts a datapath that implements a centralized ROB for a datapath as shown in Figure 2–10 in a distributed manner that has a distributed component assigned to each FU. The only centralized ROB component is a FIFO structure that maintains pointers to the

entries in the distributed ROBs. This centralized structure has W write ports for establishing the entries in dispatch (program) order and W read ports for reading the pointers to the entries established in the distributed ROB components (ROBCs). (An alternative is to implement this structure as a (wider) register file with a single read port and a single write port, with each entry capable of holding W pointers into the ROBCs.) Each ROBC can be implemented as a register file, where a FIFO list is maintained. The pointer in the centralized ROB entry is thus a pair of numbers: a FU id and an offset within the associated ROBC. The ROBCs associated with FUs 1 through n in Figure 6–7 are shown as register files RF_1 through RF_n.

The process of instruction retirement now requires an indirection through the centralized ROB by following the pointers to the ROBC entries that have to be committed. This indirection delay is accommodated by using an extra commit stage in the pipeline that has a negligible impact on the overall IPC – less than 0.5% on the average across the simulated benchmarks.

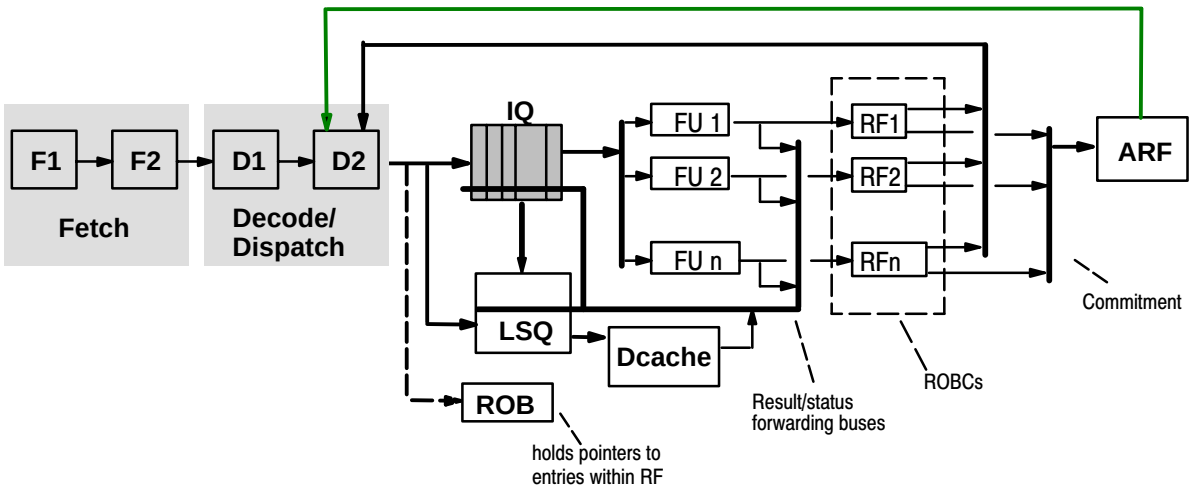


Figure 6–7. Superscalar datapath with completely distributed physical registers: one RF per function unit with one write port and two read ports per RF

As shown in Figure 6–7, each ROBC has a single write port that allows the result produced by the associated FU to be written without the need for any arbitration. Each

ROBC has two read ports – one read port for reading source operands and another for committing results. The dispatch of an instruction is delayed if both of its operands come from the same ROBC. Likewise, if more instructions require their sources to come from the same ROBC, the dispatch of those instructions is delayed. In a similar way, the instruction retirement is also blocked, if the results to be committed reside in the same ROBC. According to our simulations, the percentage of cycles when committment blocks in this fashion is only 5.5% on the average across all the executed benchmarks, resulting in only 0.1% IPC drop on the average. The size of each ROBC shown in Figure 6–7 can be tuned to match the characteristics of its associated FU's workload.

The latency of the branch misprediction recovery does not increase with the introduction of the ROB distribution. To reconstruct the correct state of the ROBCs, the tail pointer of each ROBC will have to be updated on a misprediction, along with the main ROB tail pointer. This can be done, for example, by checkpointing the tail pointers of all individual ROBCs for each branch. A cheaper alternative is to walk down the ROB starting from the mispredicted branch and flushing the values from the ROBCs one by one.

6.3. Using Retention Latches with a Distributed ROB

The distributed ROB implementation shown in Figure 6–7 can be further simplified in complexity through the incorporation of a set of retention latches – the read ports used for reading operands from the ROBCs, RF_1 through RF_n, can now be completely eliminated. Even more importantly, the arbitration for the ROBC read ports is avoided. As a result, the area of the ROB is reduced as much as 70%. The resulting datapath is shown in Figure 6–8.

Dispatched instructions are allocated to the appropriate type of FUs. Where multiple instances of the required type of FU exist, instructions are allocated within the identical instances in a round robin fashion. This distributes the load on the ROBCs and also avoids delays as mentioned earlier. When a FU j completes, it writes the relevant entry within its

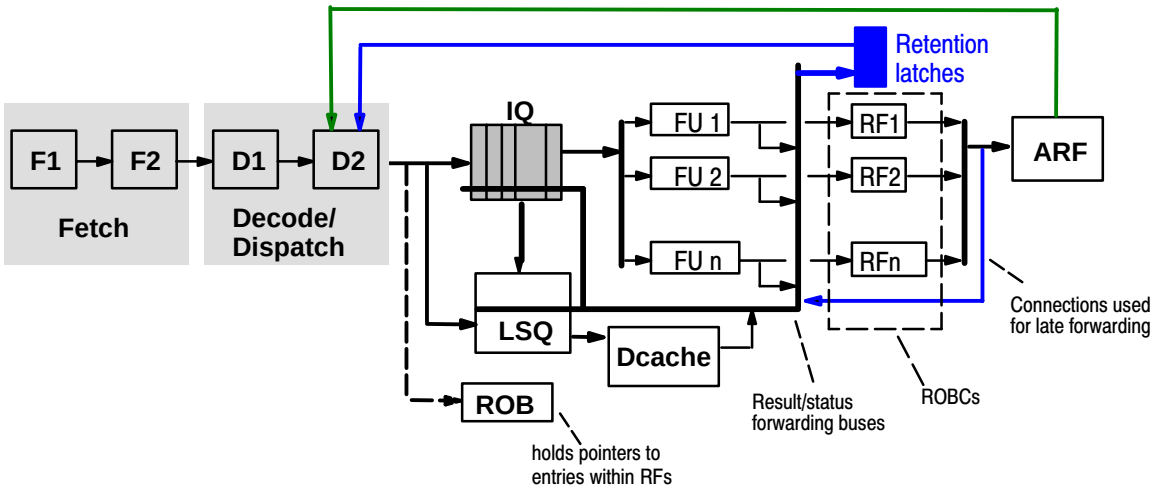


Figure 6–8. Superscalar datapath with the distributed ROB and a centralized set of retention latches

ROBC, RF j, using the dedicated write port. Simultaneously, the result (or the tag – the FU–index pair, in this case) is forwarded on the result/status bus. Late forwarding uses these same forwarding connections when results are committed from the ROBCs (i.e., RF_1 through RF_n in Figure 6–8) to the ARF using the only read port on these components. As in the case of a centralized ROB without source read ports and retention latches, a common set of W forwarding buses is able to handle normal and late forwarding in a W–way machine.

Needless to say, when the FUs compete for the shared forwarding connections for writing results, output latches are provided in each FU to temporarily buffer results till they can be written. We assume the presence of one such latch at the output of each FU in all of the datapaths shown thus far.

6.4. Further Optimizations

In an effort to further optimize the design of the ROBCs, we considered limiting the out–of–order execution in such a way, that instructions assigned to the same FU execute in

order. This guarantees, that the results are written into each individual ROBC strictly in program order. If that is the case, the design of the ROBCs could be further significantly simplified by avoiding the use of explicit decoders and use shift registers instead. Commit logic could be also simplified in this case. Unfortunately, the IPC drop of such optimized design turned out to be unacceptably high – around 20% on the average across all simulated benchmarks. Therefore, we did not consider this optimization further.

6.5. Evaluation Framework and Methodology

The widely–used Simplescalar simulator [BA 97] was significantly modified (the code for dispatch, issue, writeback and commit steps was written from scratch) to implement realistic models for such datapath components as the ROB (integrating a physical register file), the issue queue, and the rename table. The studied configuration of superscalar processor is shown in Table 6–1. We assumed the pipeline with two stages for instruction fetch, two stages for decode/rename/dispatch, and one stage each for issue, execution, writeback and commit. Of course, instructions with multi–cycle execution latencies require more than one cycle to execute, as detailed in Table 6–1. Branch mispredictions are detected right after they occur – at the end of the execution stage. We assume that the first instruction along the correct path is fetched one cycle after the detection of the misprediction.

We simulated the execution of 10 integer SPEC 2000 benchmarks *(bzip2, gap, gcc, gzip, mcf, parser, perlbmk, twolf, vortex* and *vpr*) and 8 floating point SPEC 2000 benchmarks (*applu, apsi, art, equake, mesa, mgrid, swim* and *wupwise*). Benchmarks were compiled using the Simplescalar GCC compiler that generates code in the portable ISA (PISA) format. Reference inputs were used for all the simulated benchmarks. For SPEC 2000 benchmarks, the results from the simulation of the first 1 billion instructions were discarded and the results from the execution of the following 200 million instructions were used.

Parameter	Configuration
Machine width	4–wide fetch, 4–wide issue, 4–wide commit
Window size	32 entry issue queue, 96 entry ROB, 32 entry load/store queue
Function Units and Latency (total/issue)	4 Int Add (1/1), 1 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 1FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
L1 I–cache	32 KB, 2–way set–associative, 32 byte line, 2 cycles hit time
L1 D–cache	32 KB, 4–way set–associative, 32 byte line, 2 cycles hit time
L2 Cache combined	512 KB, 4–way set–associative, 128 byte line, 4 cycles hit time
BTB	1024 entry, 4–way set–associative
Branch Predictor	Combined with 1K entry Gshare, 10 bit global history, 4K entry bimodal, 1K entry selector
Memory	128 bit wide, 60 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), fully associative, 30 cycles miss latency

Table 6–1. Architectural configuration of simulated processor

For estimating the energy/power of the ROB, the ROBC and the retention latches, the event counts gleaned from the simulator were used, along with the energy dissipations, as measured from the actual VLSI layouts using SPICE. CMOS layouts for the ROB, the ROBC, and the retention latches in a 0.18 micron 6 metal layer CMOS process (TSMC) were used to get an accurate idea of the energy dissipations for each type of transition. The register file that implements the ROB was carefully designed to optimize the dimensions and allow the use of a 2 GHz clock. A V_{dd} of 1.8 volts was used for all the measurements.

6.6. Results and Discussions

We begin by showing the effects of eliminating the ROB read ports for reading out source operand values on the performance. Figure 6–9 shows the IPCs of a 4–way machine with 96–entry ROB. Results are shown for four configurations: three baseline cases and the machine without any ROB read ports. (In the rest of the chapter by saying “no ROB read ports” we mean “no ROB read ports for reading out source operand values”. Of course, ROB

read ports needed for instruction commitment are still retained). The first bar of Figure 6–9 shows IPCs of the idealized baseline model with a single cycle ROB access time. We call this configuration *Base 1*. The second bar shows IPCs of a more realistic baseline machine with two–cycle ROB access and a full bypass network; this is referred to as *Base 2*. The third bar shows IPCs of a machine that does not implement a full bypass and only maintains the last level of bypassing logic, effectively extending the latencies of functional units by one cycle. Finally, the fourth bar shows the IPCs of the proposed machine with no ROB ports. We assume that write accesses to the ROB still take two cycles.

Compared to the idealized baseline machine *Base 1*, the configuration with no ROB read ports performs 9.6% worse in terms of IPCs on the average. (We computed the average performance drop/gain by computing the drops/gains within individual benchmarks and taking their average). Across the individual benchmarks, the largest performance drop is observed for *swim* (36.7%), *parser* (16.9%) and *equake* (13%). There are many reasons why the performance drop is not necessarily proportional to the percentage of sources that are read from the ROB in the baseline model. Performance is dependent on the number of cycles that an instruction, whose destination was sought from the ROB as a source, spends in the ROB from the time of its writeback till the time of its commitment. This directly effects the duration of "holes" that are created by eliminating the read ports from the ROB. Another important factor is the criticality of these sources for the performance of the rest of the pipeline.

Some benchmarks experience almost no performance degradation, such as *mgrid* (1.05%) and *bzip2* (1.3%). What is remarkable about these results, is that even compared to the idealized base case, the total elimination of the ROB read ports for reading out the source operand values results in only less than 10% performance degradation on the average. This data supports the basic tenet of this paper, which is that the performance loss is quite limited even if the capability for reading out the sources from the ROB is not present. A 10% performance drop is the absolute worst case and, as we show later, the use of retention latches

significantly improves performance and, combined with faster access time of the results, actually provides a better–performing architecture in some cases.

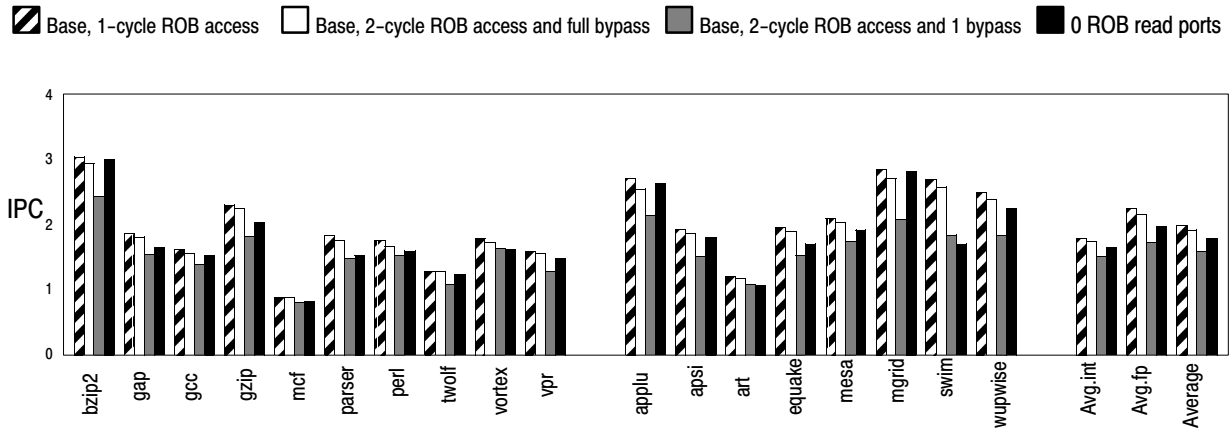


Figure 6–9. IPCs of baseline configurations and configuration without ROB read ports for reading out source operand values

Compared to the baseline model with 2–cycle ROB access and full bypass, the configuration with zero ROB ports results in 6.5% drop in IPCs on the average. The drop is smaller than in the previous case because of the extra cycle that is added to the front end of the base case pipeline, thus increasing the branch misprediction penalties. A few benchmarks (*applu, bzip2* and *mgrid*) actually showed small performance improvements.

Not surprisingly, the system with no ROB read ports performed significantly better than the baseline model with only the last level of bypassing – performance gains are in excess of 11% on the average across all benchmarks. Only three benchmarks (*vortex, art* and *swim*) still exhibited some performance degradation.

6.6.1. Evaluation of the Centralized ROB Organization with RLs

6.6.1.1. Performance

Table 6–2 shows the performance improvements achieved by using retention latches. The first two columns of Table 6–2 show the IPCs of two optimistic baseline configurations

– *Base 1* and *Base 2*. We do not consider the baseline model with partial bypass in the rest of the paper, because of its poor performance. Results in the first two columns are similar to those shown in Figure 6–9 and they are given here only for convenience. The next column shows the IPCs for the configuration with zero ROB read ports – again, these results were already graphed in Figure 6–9. The following three sets of columns show the performance of the architecture that uses retention latches managed as a FIFO, retention latches with LRU replacement policy and retention latches with random replacement policy, respectively. The "x–y" notation specifies the number of retention latches (x) and the number of read ports (y) to these latches in each case. The number of write ports to the retention latches was assumed to be eight in all simulations to support simultaneous writeback of four double–precision values. Results of Table 6–2 were obtained by simulating a 4–way machine with 96–entry ROB.

As seen from the table, even eight single–ported retention latches managed as a FIFO reduce the performance penalty to about 4.1% on the average compared to the baseline configuration *Base 1*. The most dramatic improvement is observed for *swim*: 36.7% performance drop is reduced to 2.2%. With the exception of *equake* (11.7%), the IPC drop of all benchmarks was measured to be below 10% in this case. Compared to the baseline model *Base 2*, the average IPC drop is only 0.9%, and several benchmarks (*bzip2, gcc, applu* and *mgrid*) actually have performance improvement of up to 5%. Fully–ported set of eight FIFO retention latches brings the performance drop down to 3% on the average compared to *Base 1*, and results in a small performance gain of about 0.2% compared to *Base 2*. As seen from this example and other results presented in Table 6–2, the number of ports to the retention latches have only marginal impact on the IPCs for the majority of the simulated benchmarks.

	Base 1	Base 2	0 ROB read ports	Retention Latches with FIFO replacement					Retention Latches with LRU replacement					Retention Latches with random replacement	
				8–1	8–2	8–16	16–2	16–16	8–1	8–2	8–16	16–2	16–16	8–16	16–16
bzip	3.05	2.95	3.01	3.03	3.03	3.03	3.03	3.03	3.03	3.04	3.04	3.04	3.04	3.03	3.03
gap	1.87	1.82	1.65	1.80	1.81	1.82	1.84	1.84	1.85	1.85	1.87	1.87	1.87	1.74	1.76
gcc	1.62	1.56	1.53	1.60	1.60	1.61	1.61	1.62	1.61	1.61	1.62	1.62	1.62	1.59	1.61
gzip	2.31	2.25	2.04	2.25	2.26	2.26	2.29	2.29	2.28	2.30	2.30	2.30	2.31	2.25	2.28
mcf	0.88	0.88	0.82	0.83	0.84	0.84	0.86	0.87	0.86	0.86	0.87	0.88	0.88	0.86	0.86
parser	1.84	1.76	1.53	1.68	1.71	1.72	1.79	1.80	1.72	1.79	1.80	1.82	1.83	1.66	1.71
perl	1.76	1.67	1.60	1.65	1.66	1.66	1.70	1.71	1.70	1.72	1.72	1.74	1.75	1.63	1.65
twolf	1.29	1.29	1.24	1.27	1.27	1.27	1.28	1.28	1.27	1.29	1.29	1.29	1.29	1.27	1.28
vortex	1.80	1.73	1.62	1.72	1.73	1.75	1.76	1.79	1.75	1.76	1.77	1.78	1.79	1.75	1.77
vpr	1.59	1.56	1.49	1.53	1.55	1.56	1.56	1.57	1.54	1.57	1.58	1.58	1.58	1.54	1.56
applu	2.71	2.56	2.64	2.66	2.66	2.66	2.67	2.67	2.66	2.66	2.66	2.68	2.68	2.66	2.67
apsi	1.94	1.88	1.81	1.88	1.89	1.90	1.90	1.91	1.89	1.89	1.90	1.92	1.93	1.89	1.89
art	1.21	1.18	1.07	1.15	1.16	1.16	1.18	1.18	1.16	1.16	1.16	1.18	1.18	1.12	1.15
equake	1.97	1.90	1.71	1.75	1.88	1.88	1.88	1.88	1.74	1.84	1.95	1.84	1.97	1.66	1.73
mesa	2.10	2.04	1.93	2.04	2.04	2.05	2.06	2.06	2.06	2.06	2.06	2.06	2.08	2.03	2.05
mgrid	2.85	2.71	2.82	2.85	2.85	2.85	2.85	2.85	2.85	2.85	2.85	2.85	2.85	2.84	2.84
swim	2.70	2.59	1.71	2.64	2.64	2.64	2.65	2.66	2.65	2.69	2.69	2.69	2.69	2.62	2.65
wupwise	2.50	2.39	2.26	2.26	2.26	2.26	2.26	2.26	2.26	2.50	2.50	2.50	2.50	2.26	2.29
Int avg.	1.80	1.75	1.65	1.74	1.75	1.75	1.77	1.78	1.76	1.78	1.79	1.79	1.80	1.73	1.75
FP avg.	2.25	2.16	1.99	2.15	2.17	2.18	2.18	2.18	2.16	2.21	2.22	2.21	2.24	2.14	2.16
Average	2.00	1.93	1.80	1.92	1.94	1.94	1.95	1.96	1.94	1.97	1.98	1.98	1.99	1.91	1.93

Table 6–2. IPCs of various ROB configurations

Performance can be further improved by increasing the number of latches, although increasing the number of latches beyond 16 can make it problematic to perform the associative lookup of the latches and read the source operand values (in the case of a match) in one cycle. The use of sixteen 16–ported FIFO latches results in 1.9% performance loss compared to *Base 1* and 1.4% performance gain compared to *Base 2*. In this configuration, *wupwise* experienced the largest performance drop among the individual benchmarks – 9.6% compared to *Base 1* and 5.4% compared to *Base 2*.

Further performance improvement is achieved by using the set of retention latches with LRU replacement policy. The motivation here is that the same value stored in one of the latches can be used more than once. According to [CGV+ 00], around 10–15% of the sources are consumed more than once, the fact corroborated by our experiments. One obvious example is the value of a base register used by neighboring load or store instructions or the use of values of the stack pointer and the frame pointer. Table 6–2 shows the performance of the system with 8 and 16 LRU retention latches with various number of ports. The average performance loss of the system with 8 single–ported LRU latches is 3.1% compared to *Base 1*. Compared to *Base 2*, there is a performance gain of 0.1% on the average. Example of *equake* shows that the LRU scheme does not necessarily outperform FIFO scheme, especially if the number of ports to the latches is limited. This is because some of the values that are consumed only once are retained in the latches by LRU policy at the expense of other potentially usable sources that would have otherwise been kept in the latches if FIFO strategy was used. In general, FIFO latches require larger sizes to cope with the capacity misses whereas eight LRU latches are sufficient to hold frequently used operands in most of the cases. The configuration with sixteen 16–ported LRU latches comes as close as 0.4% in performance to *Base 1* configuration and improves the performance by about 3% on the average with respect to baseline configuration *Base 2*. This provides a measurable improvement over the performance of sixteen 16–ported FIFO latches.

The last two columns of Table 6–2 show the performance of a configuration that uses retention latches with random replacement policy. Eight fully–ported retention latches with random replacement policy perform 3.5% worse than eight fully–ported LRU latches and 1.6% worse than eight fully–ported FIFO latches. For sixteen fully–ported latches, the random replacement strategy performs 3% worse than LRU and 1.6% worse than FIFO. This makes the use of retention latches with random replacement an unattractive choice.

We also studied the performance effects of simpler retention latch management in the cases of branch mispredictions for LRU latches, where the contents of the entire set of retention latches are flushed on every misprediction instead of selective invalidation of retention latch entries. Figure 6–10 shows the results for eight fully–ported LRU latches, comparing the organizations, where the contents of the retention latches keeping the results of the instructions executed on a mispredicted path are flushed and the organization with complete flushing of retention latches on every branch misprediction. Recall, that if a complete retention latch flushing is not implemented in the case of LRU latches, it is necessary to flush the contents of the latches selectively. The performance drop due to the complete flushing of the latches is 1.5% on the average. The largest drop among individual benchmarks was observed for *equake* (10.8%) and *gap* (4.8%). Notice that this optimization is not applicable to FIFO latches, because the logic needed to select the most recently–produced result corresponding to an ROB index is still needed and thus no additional logic is required to support branch misprediction recovery if FIFO latches are used.

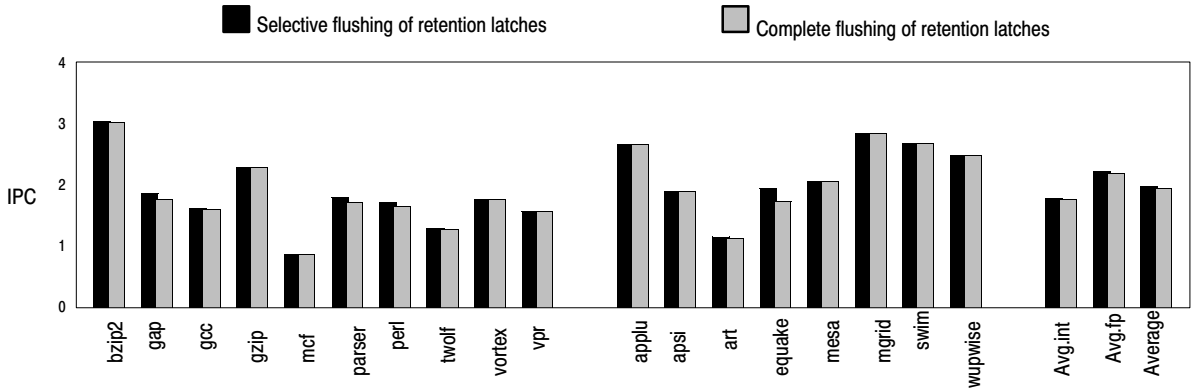


Figure 6–10. Effects of simplified retention latch management in the case of branch mispredictions (LRU latches)

Finally, we evaluated the impact of reducing the number of ROB ports and using the retention latches on the performance of a more aggressive, 6–way superscalar CPU. For the

sake of brevity, we only compare the performance of a 6–way machine against the idealized base case – *Base 1*. The configuration with zero ROB read ports performs 9.8% worse than *Base 1* model. The largest IPC drop was observed for *equake* (24.6%) and *parser* (16%). Notice that the performance of *swim* only decreased by about 11% compared to *Base 1* configuration in contrast to more than 36% drop observed for 4–way machine. The reason for the smaller drop is the larger window size assumed for a 6–way machine that makes stalls at the time of dispatch less frequent if either issue queue or the ROB is full. Simulations of a 6–way machine with the window size identical to that of a 4–way machine showed performance drop in *swim* of about 34% – almost as high as what is observed for a 4–way machine. Indeed, one of the reasons for the performance drop due to the elimination of read ports is the potential ROB saturation, because the oldest in–flight instruction cannot get one of its sources for a large number of cycles. This inevitably causes the ROB to become full and consequently results in pipeline stalls. As instruction window size increases, this problem is somewhat alleviated.

We also studied one FIFO and one LRU configuration for the 6–way machine. In both configurations we used twelve fully–ported retention latches. The average performance degradation with the use of FIFO retention latches was measured as 3.2% with the largest drop for *wupwise* (10.4%) and *equake* (6.6%). The average performance drop with the use of LRU retention latches was measured as 1% with the maximum drop of 5.2% observed for *art* and 2.4% for *apsi*. These results are almost identical to what was achieved for a 4–way processor with eight fully–ported retention latches.

6.6.1.2. Power and Complexity

Figure 6–11 shows the percentage of ROB power savings for the simulated benchmarks for various ROB configurations studied in this study. These results are for a 4–way machine with 96–entry ROB. On the average across all benchmarks, the power savings compared to the fully–ported ROB are 30% for the simplified ROB with no read ports, 23.4% for eight

2–ported FIFO latches, 22.2% for eight 2–ported LRU latches, 21.1% for sixteen 2–ported FIFO latches, and 20.2% for sixteen 2–ported LRU latches. Results are consistent across integer and floating–point benchmarks. A large power reduction for *swim* in the case with no read ports is explained by the large IPC drop which reduces the number of instructions accessing the ROB per cycle. Power dissipation within the LRU latches can be reduced by the use of low–power comparators that dissipates energy predominantly and only on full matches as suggested in [BBS+ 00] and implemented in [KGPK 01, PKE+ 03], but we avoided the evaluation of such optimization to avoid "coloring" our current results.

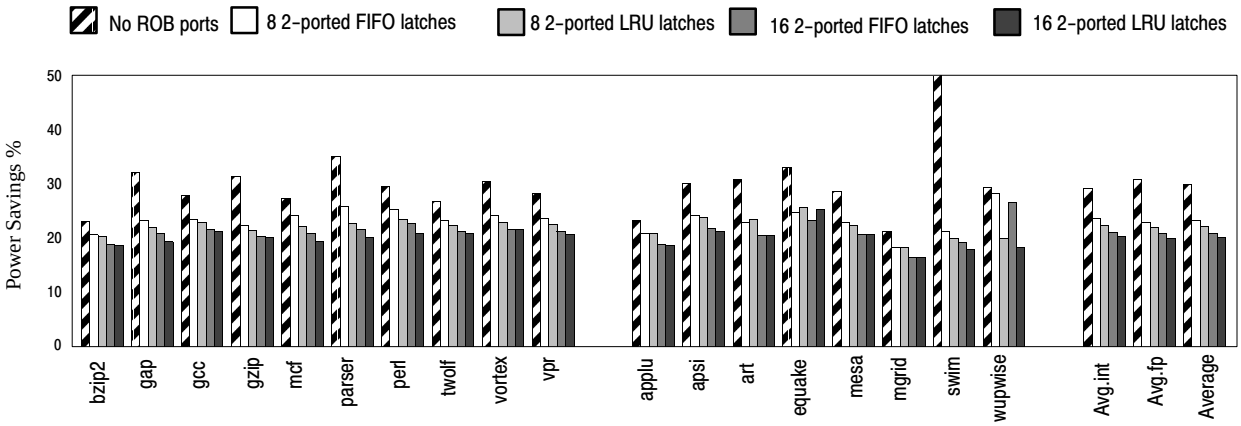


Figure 6–11. Power savings within the ROB

Power savings are lower when the retention latches are used because of two factors: first, retention latches themselves dissipate some extra power and second, performance increase as a result of using retention latches also leads to higher power dissipation. Nevertheless, noticeable overall power savings are still retained. In terms of total chip power dissipation (recalling that the ROB is about 27% of it), savings are on the order of 6–8% depending on the number of retention latches used.

To get an approximate idea of the reduction in the complexity of the ROB, we estimated the device counts of the baseline ROBs and the simplified ROBs and their retention latches. The device counts for the ROB include the bit arrays, the decoders, the prechargers the sense

amps. All devices in the retention latches are counted. For a 96–entry ROB of a 4–way superscalar processor, the device count savings from getting rid of the ROB read ports are in the range of 23% to 26% depending on the number of retention latches used, the number of ports to these latches and the replacement policy used. ROB area reduction due to the elimination of read ports for reading out the source operand values is in the order of 45%, taking into account the area occupied by the retention latches.

6.6.2. Evaluation of the Distributed ROB Organization

In order to sufficiently size each of the ROBCs, we first recorded the maximum and the average number of ROBC entries that are simultaneously in use (allocated to hold the results of in–flight instructions) by each functional unit, provided that the number of entries in the ROBCs is unlimited and the instruction window is constrained by the sizes of the ROB and the issue queue. The results of these experiments with unlimited number of the ROBC entries are shown in Table 6–3. As seen from these results, even the maximum demands on the number of the ROBC entries are quite modest across the simulated benchmarks. Considerable variations in the number of used ROBC entries are observed for the MUL FUs, both integer and floating point. The ROBCs for integer and floating point ADD and LOAD FUs are utilized fairly uniformly across the benchmarks. The ROBCs for integer and floating point ADD and LOAD FUs are utilized fairly uniformly across the benchmarks. Notice that the results are presented for each individual ROBC, thus the total number of the ROBC entries across all integer ADD units is four times the number of entries shown in the second column of the table (for 4–way machine with 4 integer ADD units).

FU type	Integer Add	Int. Mul/Div	FP Add	FP Mul/Div	Load
bzip2	18 / 10.21	0 / 0.00	0 / 0.00	0 / 0.00	27 / 9.93
gap	18 / 3.39	8 / 0.03	0 / 0.00	0 / 0.00	27 / 8.42
gcc	20 / 2.54	5 / 0.04	1 / 0.00	2 / 0.00	30 / 5.32
gzip	19 / 6.03	2 / 0.00	0 / 0.00	0 / 0.00	30 / 6.09
mcf	13 / 2.25	1 / 0.00	0 / 0.00	0 / 0.00	28 / 14.95
parser	13 / 3.99	3 / 0.23	0 / 0.00	0 / 0.00	28 / 9.37
perl	17 / 2.70	6 / 0.25	3 / 0.10	10 / 0.00	29 / 8.06
twolf	15 / 3.74	4 / 0.31	3 / 0.02	5 / 0.06	29 / 12.32
vortex	13 / 2.48	4 / 0.07	0 / 0.00	0 / 0.00	28 / 5.98
vpr	15 / 6.39	2 / 0.02	4 / 0.26	11 / 0.35	27 / 13.06
applu	16 / 3.70	0 / 0.00	2 / 0.40	5 / 1.59	18 / 5.14
apsi	12 / 3.12	8 / 1.14	8 / 1.28	15 / 2.55	26 / 6.53
art	11 / 4.51	0 / 0.00	8 / 0.47	15 / 1.30	26 / 8.19
equake	12 / 5.73	4 / 0.05	0 / 0.00	0 / 0.00	25 / 7.00
mesa	11 / 2.91	9 / 0.50	6 / 0.77	11 / 1.05	26 / 7.38
mgrid	10 / 4.12	8 / 4.42	2 / 0.66	2 / 0.42	27 / 12.40
swim	18 / 9.57	0 / 0.00	1 / 0.58	1 / 0.63	14 / 7.91
wupwise	12 / 5.25	0 / 0.00	3 / 0.58	4 / 1.18	22 / 5.75
Int avg.	16.10 / 4.37	3.50 / 0.10	1.10 / 0.04	2.80 / 0.04	28.30 / 9.35
FP avg.	12.75 / 4.86	3.63 / 0.76	3.75 / 0.59	6.63 / 1.09	23.00 / 7.54
Average	14.61 / 4.59	3.56 / 0.39	2.28 / 0.28	4.50 / 0.51	25.94 / 8.54

Table 6–3. Maximum/Average number of entries used within each ROBC for 96–entry ROB

The averages from Table 6–3 were then used to determine the number of entries in each ROBC in the following manner. Based on the results of Table 6–3, the initial values for the sizes of each ROBC were derived from the averages presented in Table 6–3 by taking the corresponding averages from the last row of Table 6–3 and rounding them to the nearest power of two. In an attempt to keep the sizes of the ROBCs to the minimum and ensure that the total size of the ROBCs does not increase the size of the centralized ROB, the exception was made for the ROBCs of the integer ADD units, whose initial sizes were set as 8 entries each. Specifically, 8 entries were allocated for each of the ROBCs dedicated to integer ADD units, 4 entries were allocated for the ROBC of integer MUL/DIV unit, 4 entries were used

for the ROBCs of floating–point ADD units, 4 entries were used for floating point MUL/DIV unit and 16 entries were allocated for the ROBC serving the LOAD functional unit to keep the result values coming out of the data cache. In the subsequent discussions, the notation "x_y_z_u_v" will be used to define the number of entries used in various ROBCs, where x, y, z, u, and v denote the number of entries allocated to the ROBCs for each of the integer ADD units, integer MUL unit, floating point ADD units, floating point MUL unit and the LOAD unit respectively. Using this notation, the initial configuration of the ROBCs is described as "8_4_4_4_16".

To gauge how reasonable was our choice for sizing the various ROBCs, we measured the percentage of cycles when instruction dispatching blocked because of the absence of a free ROBC entry for the dispatched instruction. Round–robin allocation style was used for the instructions requiring the same type of FU. Results are presented in Table 6–4. Notice that in both Tables 6–3 and 6–4, a single column is used to represent four integer ADD units and four floating point ADD units. This is because the presented statistics is identical for all four individual functional units due of the round–robin allocation of instructions.

Table 6–4 indicates that the number of latches allocated for integer ADD, integer MUL/DIV, floating–point MUL/DIV and LOAD functional units could be increased to reduce the number of dispatch stalls. As a consequence of these stalls and the conflicts over the read ports of the ROBCs, the average performance across all benchmarks decreases by 4.4% in terms of committed IPCs, as shown in Figure 6–12.

To improve the performance, consistent with the observations from Table 6–4 and Figure 6–12, we increased the sizes of all ROBCs with the exception of the ROBCs dedicated to floating–point ADD units. Specifically, the ROBC configuration "12_6_4_6_20" was considered. The average performance degradation across all simulated benchmarks decreases to only 1.75% in that case. The highest performance drop was observed in *bzip2* (4.96%) followed by *swim* (4.88%) and *mesa* (4.44%).

FU type	Integer Add	Int. Mul/Div	FP Add	FP Mul/Div	Load
bzip2	18 / 10.21	0 / 0.00	0 / 0.00	0 / 0.00	27 / 9.93
gap	18 / 3.39	8 / 0.03	0 / 0.00	0 / 0.00	27 / 8.42
gcc	20 / 2.54	5 / 0.04	1 / 0.00	2 / 0.00	30 / 5.32
gzip	19 / 6.03	2 / 0.00	0 / 0.00	0 / 0.00	30 / 6.09
mcf	13 / 2.25	1 / 0.00	0 / 0.00	0 / 0.00	28 / 14.95
parser	13 / 3.99	3 / 0.23	0 / 0.00	0 / 0.00	28 / 9.37
perl	17 / 2.70	6 / 0.25	3 / 0.10	10 / 0.00	29 / 8.06
twolf	15 / 3.74	4 / 0.31	3 / 0.02	5 / 0.06	29 / 12.32
vortex	13 / 2.48	4 / 0.07	0 / 0.00	0 / 0.00	28 / 5.98
vpr	15 / 6.39	2 / 0.02	4 / 0.26	11 / 0.35	27 / 13.06
applu	16 / 3.70	0 / 0.00	2 / 0.40	5 / 1.59	18 / 5.14
apsi	12 / 3.12	8 / 1.14	8 / 1.28	15 / 2.55	26 / 6.53
art	11 / 4.51	0 / 0.00	8 / 0.47	15 / 1.30	26 / 8.19
equake	12 / 5.73	4 / 0.05	0 / 0.00	0 / 0.00	25 / 7.00
mesa	11 / 2.91	9 / 0.50	6 / 0.77	11 / 1.05	26 / 7.38
mgrid	10 / 4.12	8 / 4.42	2 / 0.66	2 / 0.42	27 / 12.40
swim	18 / 9.57	0 / 0.00	1 / 0.58	1 / 0.63	14 / 7.91
wupwise	12 / 5.25	0 / 0.00	3 / 0.58	4 / 1.18	22 / 5.75
Int avg.	16.10 / 4.37	3.50 / 0.10	1.10 / 0.04	2.80 / 0.04	28.30 / 9.35
FP avg.	12.75 / 4.86	3.63 / 0.76	3.75 / 0.59	6.63 / 1.09	23.00 / 7.54
Average	14.61 / 4.59	3.56 / 0.39	2.28 / 0.28	4.50 / 0.51	25.94 / 8.54

Table 6–4. Percentage of cycles when dispatch blocks for 8_4_4_4_16 configuration of ROBCs

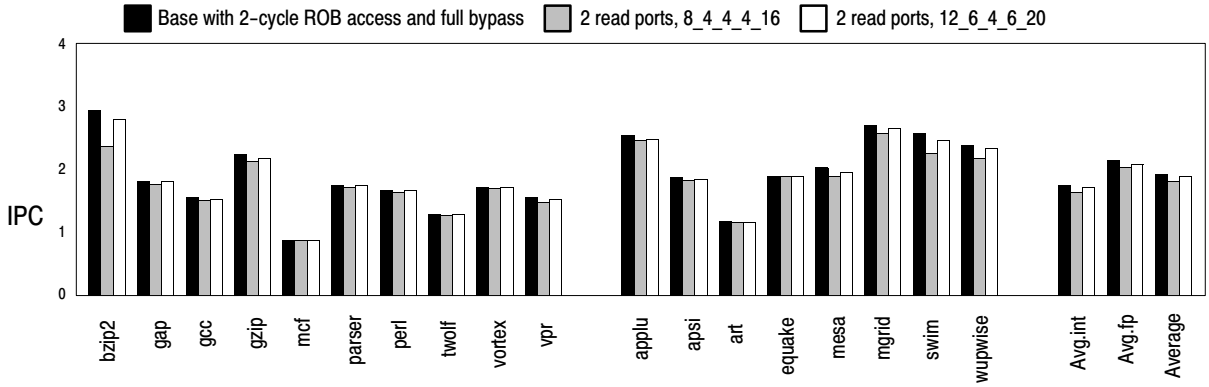


Figure 6–12. IPCs of 8_4_4_4_16 and 12_6_4_6_20 configurations of the ROBCs compared to the baseline model. 2 read ports are used for each ROBC – one port is used for source operand reads and the other port is used for commits. Access to the baseline ROB is assumed to take 2 CPU cycles. Full bypass network is assumed.

6.6.3. Evaluation of the Distributed ROB Organization with RLs

We now study the effects of integrating the set of retention latches into the datapath with ROBCs. To minimize the design complexity, the simplest configuration of eight 2–ported FIFO retention latches was used in our experiments. Recall that by themselves, this combination of retention latches only degrades the performance of the baseline datapath by about 0.2% on the average. Figure 6–13 shows the IPCs of the processor that uses this small set of retention latches in conjunction with "12_6_4_6_20" ROBCs. The average

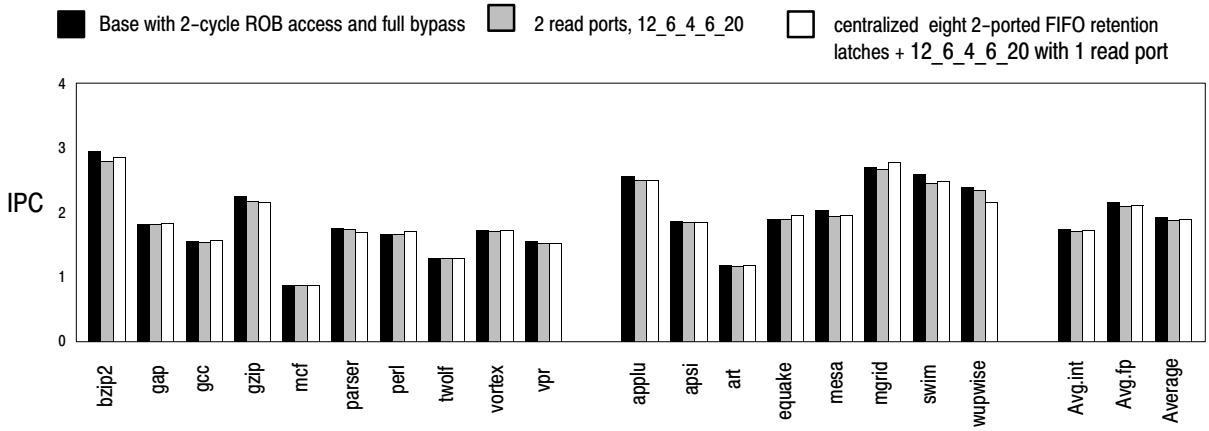


Figure 6–13. IPC of the "12_6_4_6_20" ROBC configuration with retention latches compared to baseline configuration and the ROBC configuration without retention latches

performance drop is reduced to 1.1% and each of the ROBCs only has a single port that is still needed for commitment. In this configuration, the retention latches enable the processor to perform faster in some cases. As a result, *gap*, *gcc*, *perl*, *twolf*, *vortex*, *equake* and *mgrid* benchmarks perform faster than the baseline configuration. On the other hand, a significant performance drop is observed when the retention latches are not effectively used. In this case, since the reorder buffer no longer provides the source operand values, a large percentage of capacity misses in the retention latches result in long time periods in which the source operands are not available to the waiting instructions. *parser* and *wupwise* are two example benchmarks with this behavior.

6.6.4. Evaluation of Selective Operand Caching in the RLs

Figure 6–14 shows the IPCs of various schemes that use the retention latches. Results are shown for five configurations. The first bar shows the IPCs of the baseline model. The second bar shows IPCs of the original scheme presented in [18], where each and every result is written into the retention latches (RL). 8 RLs managed as a FIFO are assumed to be in place for this configuration. The average IPC difference between these two schemes is 1.6%. The highest IPC drop is observed for *wupwise* with 7.4% followed by *mcf* with 2.9%. The

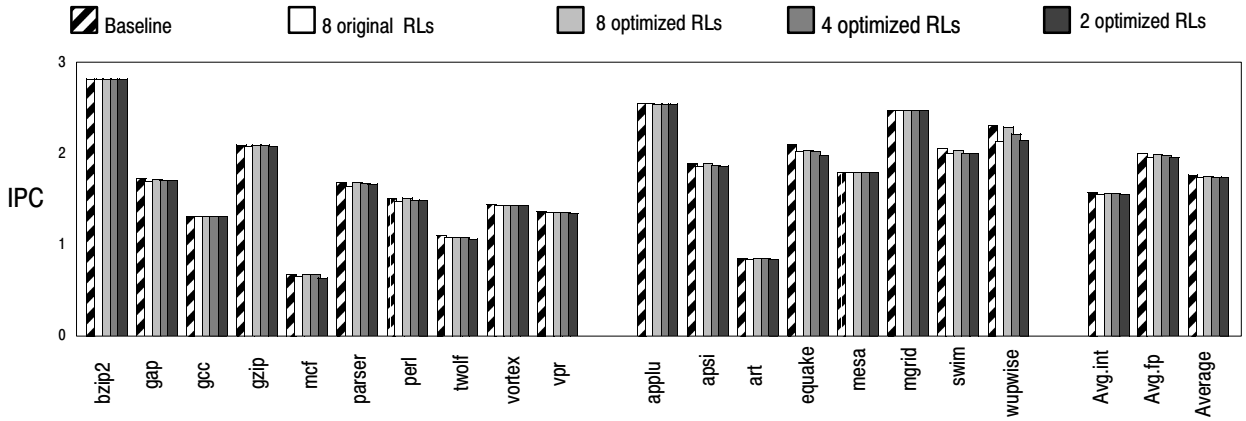


Figure 6–14. IPCs of various schemes with retention latches

last three bars show IPCs for the schemes, where the short–lived result values are not written into the RLs. The bars correspond to the use of 2, 4 and 8 RLs respectively, each managed as a FIFO. Remarkably, as few as 2 FIFO RLs keeping only the results that are not identified as short–lived practically match the performance of 8 FIFO RLs which keep all results. The IPC drop in the scheme that uses 2 FIFO RLs is 1.8% on the average compared to the baseline case. The difference between the IPCs represented by the second and the third bars in Figure 6–13 is only 0.2%. Further reduction in complexity in the RL array is possible by limiting the number of write ports on the optimized RLs, as we only need two write ports to buffer two results. The use of 4 and 8 RLs optimized not to cache the short–lived results achieves the IPCs within 0.9% and 0.4% of the baseline IPC respectively. The use of the optimized

RL management is especially beneficial for certain benchmarks, such as *wupwise.* In the original scheme of [KPG 02], even the allocation of 32 RLs resulted in the performance degradation of more than 7% for *wupwise*. In contrast, the use of 8 RLs managed in an optimized fashion reduces the performance loss to 0.4% compared to the baseline model. We assume that RLs are managed as a FIFO in all considered schemes. *We gave the benefit of the doubt to the baseline model and assumed that all accesses to the baseline ROB take a single cycle. If the access to the ROB is pipelined across 2 or more cycles, our techniques actually result in a performance gain!*

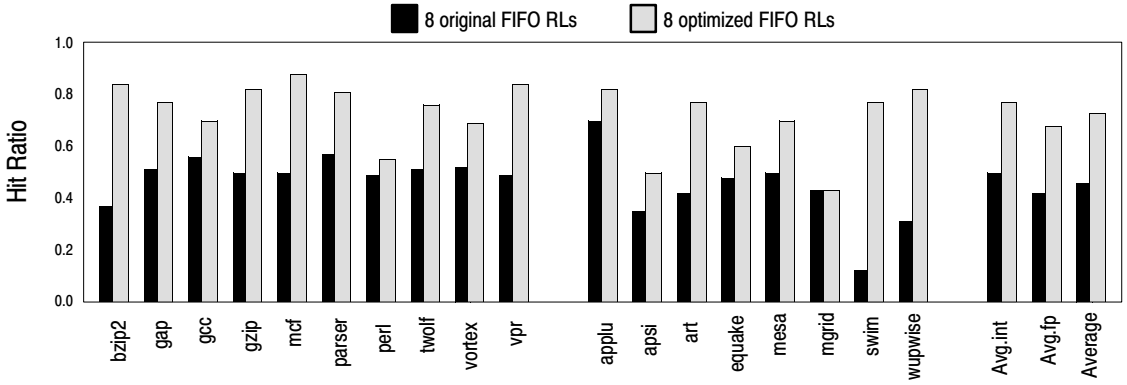


Figure 6–15. Hit rates in the retention latches

Figure 6–15 shows the percentage of cases when the search for the data in the RLs results in a hit. As seen from Figure 6–15, the average hit rate in the RLs increases from 46% to 73% if short–lived values are not cached. Across the individual benchmarks, the hit rate of *swim* improves by about 6.5 times (from 12% to 77%), and the hit rate of *wupwise* improves by almost 3 times (from 31% to 82%). The relatively low hit rate to the RLs in the original scheme is a result of the suboptimal use of the RLs, as most of the entries hold the short–lived results.

6.6.5. Implications on ROB Power Dissipations

Figure 6–16 shows the power savings achieved within the ROB by using the techniques proposed here. The power was calculated by using the actual layouts of all considered components in conjunction with the microarchitectural–level statistics. The configuration with just the retention latches results in 23% of ROB power savings on the average. The ROB distribution results in more than 51% reduction in energy dissipations within the ROB. The savings are attributed to the use of much shorter bitlines and word–select lines and the absence of sense amps and prechargers for the small ROBCs. If the retention latches are integrated into the datapath with the ROBCs, the power dissipation is actually slightly increases (47% power savings) for two reasons. First, the retention latches introduce the extra source of power dissipation, and second, the performance increases with the use of the retention latches, resulting is slightly higher energy dissipation per cycle. But complexity is still reduced, because the read ports and connections from all ROBCs are replaced by a small centralized buffer with just a few ports.

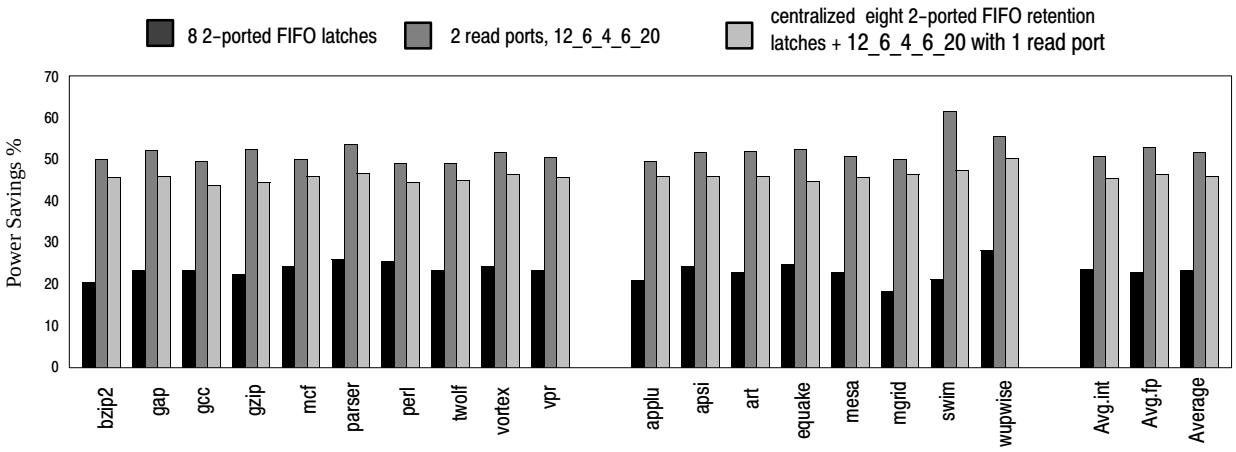


Figure 6–16. Power savings within the ROB

Figure 6–17 shows the power savings achieved by applying our techniques to a datapath, where the non–commited register values are stored within the rename buffers, implemented

separately from the ROB. Such a datapath has the advantage from the power standpoint, because fewer rename buffers than the ROB entries can be used, as some instructions, notable branches and stores, do not have a destination register. We simulated the system with a 96–entry ROB and 64 rename buffers. The use of eight 2–ported FIFO retention latches save 20% of the rename buffers power. The distribution of the rename buffers in the form of the ROBCs results in more than 42% reduction in power dissipations within the rename buffers. Finally, the combined savings of the two techniques is 37%. These savings are somewhat lower compared to those of Figure 6–16, as expected. This is because the design with the rename buffers is inherently more energy efficient. In any case, our designs result in non–trivial reduction of the speculative register storage in datapaths, where the committed register values are stored in a separately maintained register file.

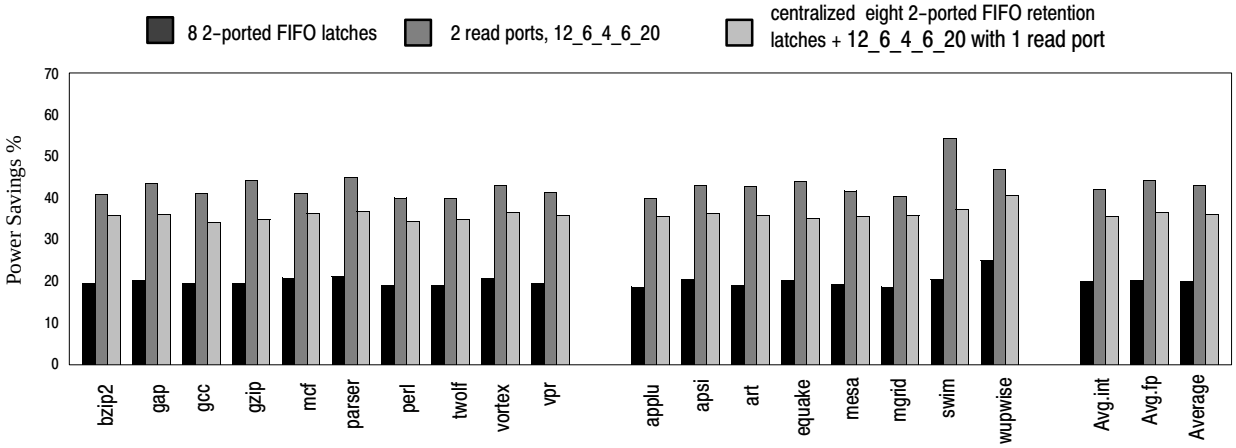


Figure 6–17. Power savings within the rename buffers

Figure 6–18 shows the energy reduction achieved by the use of optimized retention latches, as well as by the use of the original scheme of [KPG 02]. The scheme with 2 optimized RLs results in 26% ROB energy reduction compared to about 23% energy savings achieved with the use of 8 retention latches managed as proposed in [KPG 02]. Recall that these two schemes have comparable loss in performance – about 1.6%.

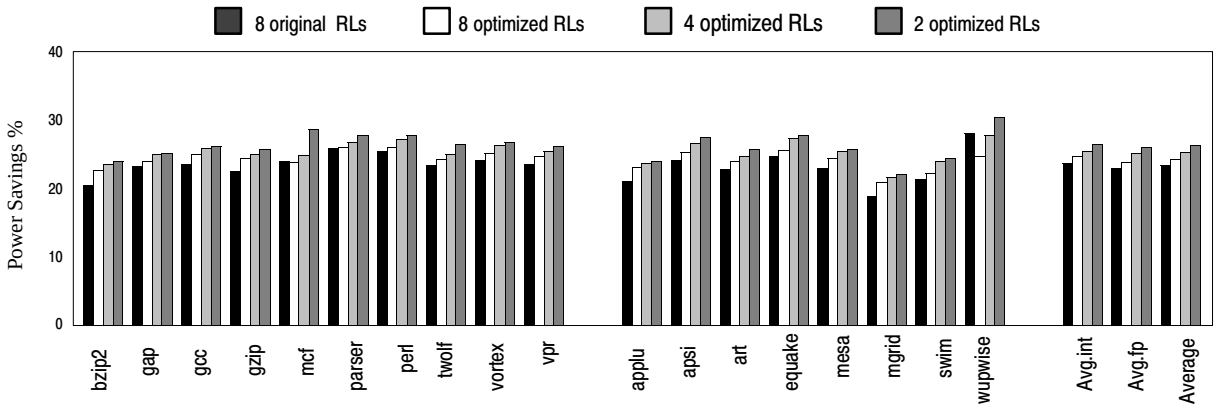


Figure 6–18. Power savings with optimized RLs.

6.7. Related Work

Lozano and Gao [LG 95] observed that about 90% of the variables in a datapath are short–lived, in the sense that they are exclusively consumed during their residency in the ROB. At first glance, this seems to contradict the results presented in Figure 6–1. While Lozano and Gao consider the statistics from the perspective of the destination registers by keeping track of how many instances of the destination registers are short–lived, our analysis of Figure 6–1 are presented from the perspective of source registers. Each destination, considered by Lozano and Gao, can be used as a source by several instructions – this is typically the case with most registers that are accessed from the ARF. This explains why the percentage of reads from the ARF reported here is so much higher than the percentage of long–lived variables as reported by Lozano and Gao. These are essentially different statistics.

The idea of using the retention latches in the context of the ROB is similar in philosophy to forwarding buffer described by Borch et.al. in [BTME 02] for a multi–clustered Alpha–like datapath. Our solution and Borch et.al.'s solution both essentially extend the

existing forwarding network to increase the number of cycles for which source operands are available without accessing the register file. A set of forwarding buffers retains the results for instructions executed in the last nine cycles. Nine stages of forwarding logic are employed to supply these results to dependent instructions. Borch et.al. further extend this idea by using per–cluster register file caches (CRC – Clustered Register Cache) to move the register file access out of the critical path and replace it with the faster register cache access. Each CRC only stores operands required by instructions assigned to that cluster and operands needed by a dependent instruction that is unlikely to get these operands by other means. While the primary goal of the work of Borch et.al. is to improve the performance, the complexity of the datapath is certainly increased. The effects of the CRCs on power dissipation are not reported, although it is reasonable to assume that the overall effect on power is positive, because the power–hungry accesses to a large register file are often replaced with the more energy–efficient accesses to the CRCs.

There is a growing body of work that targets the reduction of register file ports. Alternative register file organizations have been explored primarily for reducing the access time (which goes up with the number of ports and registers), particularly in wire–delay dominated circuits [BTME 02, CGV+ 00, LVA 95, WB 96]. Replicated [Kessler 99] and distributed [CPG 00, FCJ+ 97] register files in a clustered organization have been used to reduce the number of ports in each partition and also to reduce delays in the connections in–between a function unit group and its associated register file. While replicated register files [Kessler 99] or multi–banked register files with dissimilar banks (as proposed in [CGV+ 00], organized as a two–level structure – cached RF – or as a single–level structure, with dissimilar components) are used to reduce the register file complexity, additional logic is needed to maintain coherence in the copies or to manage/implement the allocation/deallocation of registers in the dissimilar banks.

While replicated register files [Kessler 99] or multi–banked register files with dissimilar banks (as proposed in [CGV+ 00], organized as a two–level structure – cached RF – or as a single–level structure, with dissimilar components) are used to reduce the register file complexity, additional logic is needed to maintain coherence in the copies or to manage/implement the allocation/deallocation of registers in the dissimilar banks.

A way of reducing the complexity of the physical register file by using a two–level implementation, along with multiple register banks is described by Balasubramonian et.al. in [BDA 01] for a datapath that integrates physical register file and ARF in a structure separate from the ROB. The complexity and power reduction comes from the use of banks with a single read port and a single write port in each bank, thus resulting in a minimally–ported design. The two–level structure allows the first level (banked) register file to be kept small (and fast), with the higher speed compensating for IPC drops arising from limiting the number of ports. There are some noticeable differences between the distributed ROB scheme with retention latches and the minimally ported multi–banked register file design of [BDA 01]. First, we use components (the analog of banks) that are sized to match the characteristics of the specific FUs instead of identical banks. Second, in the design of [BDA 01], FUs that target a common bank have to compete for access to the bank, as well as for access to the forwarding paths. Two independent arbitration mechanisms that co–operate together in selecting FUs that get to update the banks and forward are needed; using just one arbitration mechanism may lead to lower commit rates, underutilized forwarding buses or bank contention. The first arbitration needed for writing into a ROBC is unnecessary in our design. Neither do we have the complexity of dealing with two different types of arbitration. In our case, the forwarding bus arbitration mechanism decides when the result gets forwarded, simultaneously with the write into the ROBC using the dedicated connection. A third difference of our scheme with the multi–banked design of [BDA 01] has to do with the allocation of a register file and the allocation of a ROBC entry. We use a simple FIFO queue implemented within each ROBC to perform allocations,

deallocations and single cycle rollbacks on misprediction or interrupts by simply updating the tail pointer within the ROBCs and the ROB. The design of [BDA 01] has to rely on a more general non–FIFO allocation scheme, since each bank holds both committed and non–committed values; consequently, rollback on an interrupt is complicated – the authors of [BDA 01] suggest that rollback has to be performed serially from the head of the ROB, requiring the rename table to be restored and to resort to a precise state following an interrupt or misprediction.

Additional work attempting a reduction in the complexity of the register file is also described in [BDA 01], including solutions used in VLIW datapaths and novel transport–triggered CPUs.

The idea of caching recently produced values was used in [HM 00] (hereafter called “the VAB scheme”). At the time of instruction writeback, FUs write results into a cache called Value Aging Buffer (VAB). The register file, holding both speculative and committed register values, is updated only when entries were evicted from the VAB. Furthermore, when the required value is not in the VAB, a read of the register file is needed, requiring at least some read ports for reading the source operands. Unless a sufficient number of register file ports is available or the number of entries in the VAB is sufficiently large, the performance degradation can be considerable. In addition, in the VAB scheme, the multi–cycle register file access is still an intimate part of the issue process. In contrast to this, if the required value is not found in the retention latches or in the ARF in our scheme, we still do not read the ROB, thereby eliminating any read ports on the ROB for reading source operands. By separating the ARF from the ROB, we satisfy a large percentage of the dependencies from forwarding, the retention latches, and the ARF, requiring only a small percentage of sources to be obtained from the ROB. Deferring these reads does not have any significant impact on performance. Further, in our scheme, the register file (ROB) access does not form a component of a critical path.

Some other differences between the VAB scheme and the proposed technique are as follows. Since the recent results may not necessarily have been found in the register file, misprediction handling and interrupt handling with the VAB were somewhat involved; misprediction and interrupt handling required selective lookup of values from the VAB for generating a precise state. In our scheme, results are written *to both* the retention latches and the ROB; we simply clear the entire set of the retention latches or invalidate their contents selectively on a branch misprediction. In fact, as shown in Figure 6–9, clearing all of the retention latches on a misprediction, which is much more simpler to implement, produces almost the same level of performance as the selective clearing scheme. Unlike the VAB scheme, there is no need to copy anything from the retention latches into the ROB. On a miss to the VAB, the necessary accesses to the large (integrated) register file can dissipate considerable energy. Such dissipations are eliminated in our scheme; we only access the relatively small ARF and the retention latches.

In a recent study, Savransky, Ronen and Gonzalez [SRG 02] proposed a mechanism to avoid useless commits in the datapath that uses the ROB slots to implement physical registers. Their scheme delays the copy from the ROB to the architectural register file until the ROB slot is reused for a subsequent instruction. In many cases, the register represented by this slot is invalidated by a newly retired instruction before it is needed to be copied. Such a scheme avoids about 75% of commits, thus saving energy.

In [PKEG 03], we propose a technique to effectively remedy some of the principal inefficiencies associated with the datapaths that use separate register file for storing committed register values. Our scheme isolates the short–lived values in a small dedicated register file avoiding the need to write these values into the ROB (or the rename buffers) and later commit them to the architectural register file. With minimal additional hardware support and with no performance degradation, our technique eliminates close to 80% of unnecessary data movements in the course of writebacks and commitments and results in the energy savings of about 20–25% on the ROB or the rename buffers. For considered

processor configurations, this roughly translates to about 5% of the overall chip energy savings for the datapath, where the physical registers are implemented as the ROB slots, and to about 2–3% of the overall energy savings if physical registers are maintained in a separate set of rename buffers.

In [PKG 02b], we studied several alternative techniques for the ROB power reduction. Those include dynamic ROB resizing, the use of–energy comparators within the associatively–addressed ROBs and the use of zero–byte encoding logic. All techniques described in [PKG 02b] can be use in conjunction with the schemes described in this study.

6.8. Conclusions

This study described several techniques to reduce the complexity and the power dissipation of the ROB. First, we proposed a scheme to eliminate the ports needed for reading the source operand values for dispatched instructions. Our approach for eliminating the source read operand ports on the ROB capitalizes on the observation that only about 5% or so of the source operand reads take place from the ROB. Any performance loss due to the use of the simplified ROB structure was compensated for by using a set of retention latches to cache a few recently–written values and allowing the bulk of the source operand reads to be satisfied from those latches, from forwarding and from reads of the architectural registers. Our technique also removes the multi–cycle ROB access for source operands read from the critical path and substitutes it with the faster access to the retention latches.

Second, we introduced the conflict–free ROB distribution scheme, where the conflicts over the use of the write ports are eliminated by allocating a small separate FIFO queue for holding the speculative results for each functional unit. All conflicts over the read ports are eliminated by removing the read ports for reading out the source operand values from the distributed ROB completely and using the combination of a small set of

associatively–addressed retention latches and late result forwarding to supply the results to the waiting instructions in the issue queue.

Our designs result in extremely low performance degradation of 1.7% on the average across the simulated SPEC 2000 benchmarks and significant reduction in the ROB complexity and power dissipation. On the average, the ROB power savings of as high as 51% are realized. This includes the extra power dissipated within the retention latches. Taking into account that the ROB, as used in the considered datapath style, can dissipate as much as 27% of the total chip energy [FG 01], our techniques result in about 13% of total CPU power reduction with only 1.7% average drop in performance in the worst case.

Chapter 7

Energy–Efficient Register Renaming

7.1. Motivation

One on–chip structure with a high power density is the Register Alias Table (RAT). RAT maintains the register address translations needed for handling the true data dependencies. True data dependencies are handled by assigning a new physical register for every new result that is produced into a register. The RAT maintains information to locate the most recent instance of an architectural register. Register renaming is a technique used in all modern superscalar processors to cope with false data dependencies by assigning a new physical register to each produced result. The mappings between the logical and the physical registers are maintained in the RAT, so that each instruction can identify its source *physical* registers by performing the RAT lookups indexed by the addresses of the source *logical* (architectural) registers. The read and write accesses to the RAT, as well as the actions needed for checkpointing and the state restoration, result in a significant amount of power dissipated in the RAT. For example, about 4% of the overall processor's power is dissipated in the rename unit of the Pentium Pro [MKG 98]. Even higher percentage of the overall power – 14% – is attributed to the RAT in the global power analysis performed in [FG 01]. When coupled with the relatively small area occupied by the RAT, this creates a hot spot, where

the power density is significantly higher than in some other datapath components, such as the on–chip caches.

In this study, we propose mechanisms to reduce the RAT power and the power density by exploiting the fundamental observation that most of the generated register values are used by the instructions in close proximity to the instruction producing a value. Specifically, we propose two methods to reduce the RAT power dissipation. The first method disables the RAT readouts if it is determined that the required source operand value is produced by the instruction that is dispatched in the same cycle. The second technique eliminates some of the remaining RAT read accesses even if the requested source value is produced by an instruction dispatched in an earlier cycle. This is done by buffering a small number of recent register address translations in a set of external latches and satisfying some RAT lookup requests from these latches.

The rest of the chapter is organized as follows. To motivate our work, we describe the register alias table complexities and sources of associated energy dissipations in Section 7.2. We present the details of our energy reduction techniques in Sections 7.3 and 7.4. Our simulation methodology is described in Section 7.5 and we present and discuss the simulation results in Section 7.6. Section 7.7 reviews the related work and we offer our concluding remarks in Section 7.8.

7.2. The RAT Complexity

For this study, we used RISC–type ISA, where instructions may have at most two source registers and one destination register. We further assumed that the RAT is implemented as a multi–ported register file, where the number of entries is equal to the number of architectural general–purpose registers in the ISA. The RAT is indexed by the architectural register address to permit a direct lookup. The width of each RAT entry is equal to the number of bits in a physical register address. An alternative design is to have the number

of entries in the RAT equal to the number of physical registers, such that each RAT entry stores the logical register corresponding to a given physical register and a single bit to indicate if the entry corresponds to the most recent instance of the architectural register. In this scheme, as implemented in the Alpha 21264 [Kessler 99], the RAT lookup is performed by doing the associative search using the logical register address as the key. We did not consider this variation of the RAT in this study, because it is inherently less energy–efficient than the direct–lookup implementation, due to the large dissipations that occur during frequent associative lookups. One way to address this problem is to use a recently proposed dissipate–on–match comparator (see. Section 5.1) in the associative logic within the RAT, but such an evaluation is beyond the scope of this work.

The number of ports needed on the RAT in a W–way superscalar machine is quite significant. Specifically, 2*W read ports are needed to translate the source register addresses from logical to physical identifiers and W write ports are needed to establish W entries for the destinations. In addition, before the destination register mapping is updated in the RAT, the old value has to be checkpointed in the Reorder Buffer for possible branch misprediction recovery. Consequently, W read ports are needed for that purposes, bringing the total port requirements on the RAT to 3*W read ports and W write ports.

The energy dissipations take place in the RAT in the course of the following events:

1) Obtaining physical register addresses of the sources: This is in the form of reads from the register file implementing the RAT.

2) Checkpointing the old mapping of the destination register. This, again, is in the form of reading the register file that implements the RAT. This read is necessary in machines that support speculative execution and read out the old mapping of the destination register, which is then saved into the reorder buffer. If the instruction that overwrites the entry is later discovered to be on the mispredicted path, the old mapping saved within the reorder buffer is used to restore the state of the RAT.

3) Writing to the RAT for establishing the new mapping for the destination register. If the dispatched instruction has a destination register, a new physical register is assigned for that destination register, and the RAT entry for the destination architectural register is updated with the address of the allocated physical register by this write. If there is no physical register available for the destination register at the time of dispatch, the dispatch stalls.

7.3. Reducing the RAT Power by Exploiting the Intra–Group Dependencies

In a superscalar machine, there may be sequential dependencies among the group of instructions that are co–dispatched within a cycle. To take care of such dependencies, register renaming must logically create the same effect as renaming the instructions individually and sequentially in program order. A sequential implementation of register renaming will be too expensive and will dictate the use of a slower clock. To avoid this, the accesses to the RAT and dependencies are handled as follows:

Step 1. The following substeps are performed in parallel:

(a) RAT reads for the sources of each of the co–dispatched instructions are performed in parallel, assuming that no dependencies exist among the instructions.

(b) New physical registers are allocated for the destination registers of all of the co–dispatched instructions.

(c) Data dependencies among the instructions are noted, using a set of comparators. The address of each destination register in a group of instructions is compared against the sources of all following instructions in the group and if a match occurs, the dependency is detected.

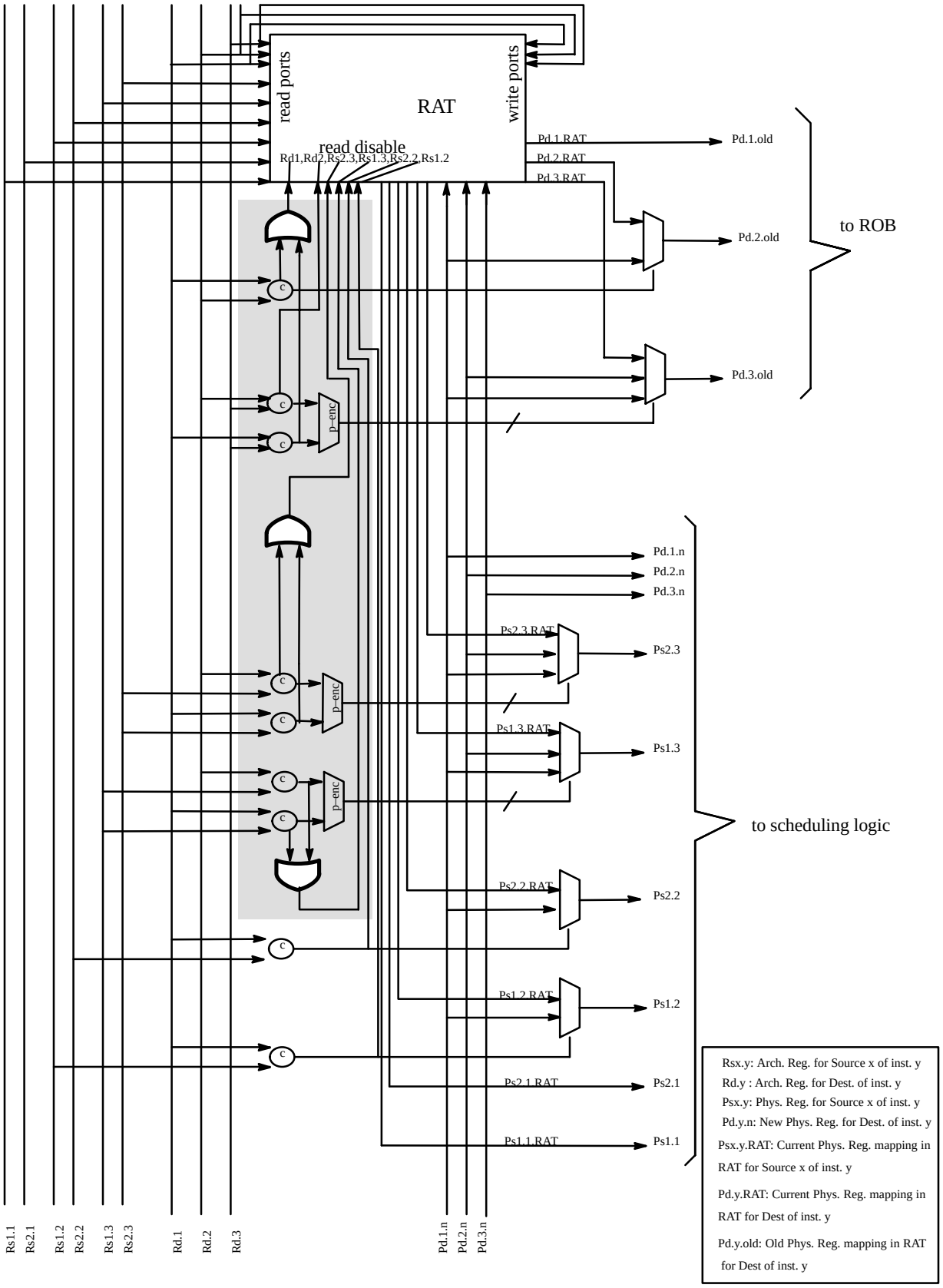


Figure 7–1. Proposed renaming Logic for a 3–way Processor

Step 2. If a data dependency is detected among a pair of instructions, the source physical register for the dependent instruction, as read out from the RAT, is replaced with the allocated destination register address of the instruction producing the source to preserve the true dependencies. After resolving dependencies as described, the RAT is updated with the addresses of the new destinations.

The above steps, including the concurrent substeps within Step 1, avoid a bottleneck that would otherwise result from performing RAT lookup and dependency handling in a strictly sequential manner among the co–dispatched instructions. The price paid in this approach is in the form of redundant accesses to the RAT for the mappings of source registers that are renamed because of dependencies within the group of co–dispatched instructions. If a dependency is not detected, the source mapping obtained from the RAT is used, otherwise it is tossed out. The gray–box in Figure 7–1 shows additional circuitry for these changes in the register renaming logic for a 3–way superscalar processor.

A considerable amount of energy is expended within the multi–ported register file that implements the RAT because of the concurrent accesses to it within a single cycle. It is thus useful to consider techniques for reducing the energy dissipation within the RAT. Our first proposed solution disables parts of the RAT read accesses if the inter–group data dependency is noted, as described by Step 1(c). The outputs of the comparators, corresponding to a given source, are NOR–ed and the output of the NOR gate is used as the enabling signal for the sensing logic on the bitlines used for reading the physical address mapping of this source from the RAT. For example, consider three instructions – I1, I2 and I3 (in program order) that are renamed in the same cycle. Consider, for example, the second source of instruction I3. It is compared for possible intra–group dependencies against the destination of I1 and the destination of I2. If one of these comparators indicates a match (the output of the comparator stays precharged at the logical “1”), the output of the NOR–gate becomes zero and the sense amps used for reading the bitlines of the second source of instruction I3 are not activated, thus avoiding the energy dissipation in the course of sensing. Measurable power savings can

be realized, because sense amps attribute to a large fraction of the overall read energy. Because the reading of the bitlines in the RAT is conditionally enabled by the outcome of the comparisons performed within the group of instructions, in the rest of the chapter we abbreviate this technique as GDBR (Group Dependency Based Reading).

Figure 7–2 shows the percentage of RAT lookups for the source registers that can be aborted if the GDBR is used for a 4–way and a 6–way processor. Results are presented for the simulated execution of a subset of SPEC 2000 benchmarks, including both integer and floating point codes. Detailed processor configurations are described in Section 7.5. On the average across all benchmarks around 31% of the RAT accesses can be aborted for a 4–way processor and around 41% for a 6–way processor. The latter number is higher, because more instructions are dispatched in the same cycle, thus increasing the possibility that the most recent definition of a source register is within the same instruction group. Notice, finally, that our technique does not prolong the cycle time, because the output of the NOR gate is driven to the sense amp before the wordline driver completes the driving of the word line. Our simulations of the actual RAT layouts in 0.18 micron 6–metal layer TSMC process (more details are given in Section 7.5) show that the decoder delay is about 150 ps, the delay of the wordline driver is about 100 ps, the bitline delay to create the small voltage difference across the bitline pair is 60 ps and at that point sense amp is activated. The comparator delay is about 120 ps and the delay of the three–input NOR–gate is 60 ps Consequently, the signal that controls the activation of the sense amp is available 180 ps after the beginning of the cycle, while the sense amp is normally activated after 310 ps are elapsed since the beginning of the cycle. These numbers show that the sense amp control signal is available well in advance of when it needs to be used, leaving enough time to route the signal to the sense amps, if need be. These delays were obtained using highly–optimized hand–crafted layouts of the RAT assuming 32 architectural registers and 4–way wide dispatch/renaming. Therefore, the GDBR can be applied without any increase in cycle time.

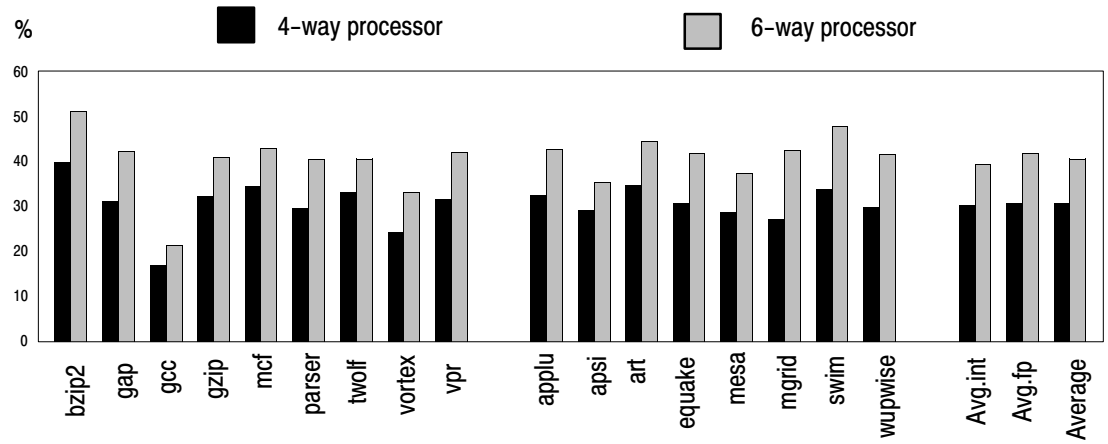


Figure 7–2. The percentage of source operands that are produced by the instructions co–dispatched in the same cycle.

7.4. Reducing the RAT Power by Buffering Recent Translations

Our simulations of the SPEC 2000 benchmarks show that the dependent instructions are usually very close in proximity to each other. If the register needed as a source is not defined by an earlier instruction dispatched in the same cycle, then it is likely defined by an instruction dispatched one or at most a few cycles earlier. We exploit this behavior by caching recently updated RAT entries in a small number of associatively–addressed latches, external to the RAT. The RAT access for a source register now proceeds as follows:

1) Start accessing the RAT and at the same time address the external latches (ELs) to see if the desired entry is located in one of the external latches.

2) If a matching entry is found, discontinue the access from the RAT.

As long as the overhead of accessing the ELs is less that the energy spent in accessing the RAT before the RAT accessing is aborted, this technique will result in an overall energy saving.

Figure 7–3 depicts a RAT with multiple ELs (4 in this case). The hardware augmentations to the basic register renaming structure are as follows. First, we need four latches to hold the address of each of the four most recently accessed architectural registers. Second, four comparators are used to compare the address of the architectural register, whose lookup is the RAT is being performed against the register addresses located in the four ELs.

The access steps for a RAT with multiple ELs are as follows:

Phase 1:

– Precharge the RAT for a read access

– Start the decoding of the register number

– Simultaneously, compare the register number with the register numbers stored in the latches

Phase 2:

– If a match occurs to an EL (latch hit), abort the readout from the RAT. Otherwise (on a latch miss), proceed with the regular RAT access.

Notice that on a read miss the data is not brought from the RAT array into the latches. This is so for the following reason. It was noticed by several researchers [CGV+ 00] that most register values (about 85%) are consumed at most once. Our simulation results show similar high percentages. If this is the case, then bringing the data that was once read from the RAT into the ELs will only result in polluting the latches with unusable data in most situations. In addition, extra energy dissipation occurs if such data movement is needed. Because of this, we only record the translation in the ELs when the physical register is allocated. In other words, the update of the RAT and the update of the ELs proceed in parallel, where the latch entry that is replaced is selected randomly to simplify the design.

Notice, that as a consequence of this policy, there is no need to write the translation information back to the RAT once an entry is evicted from the ELs.

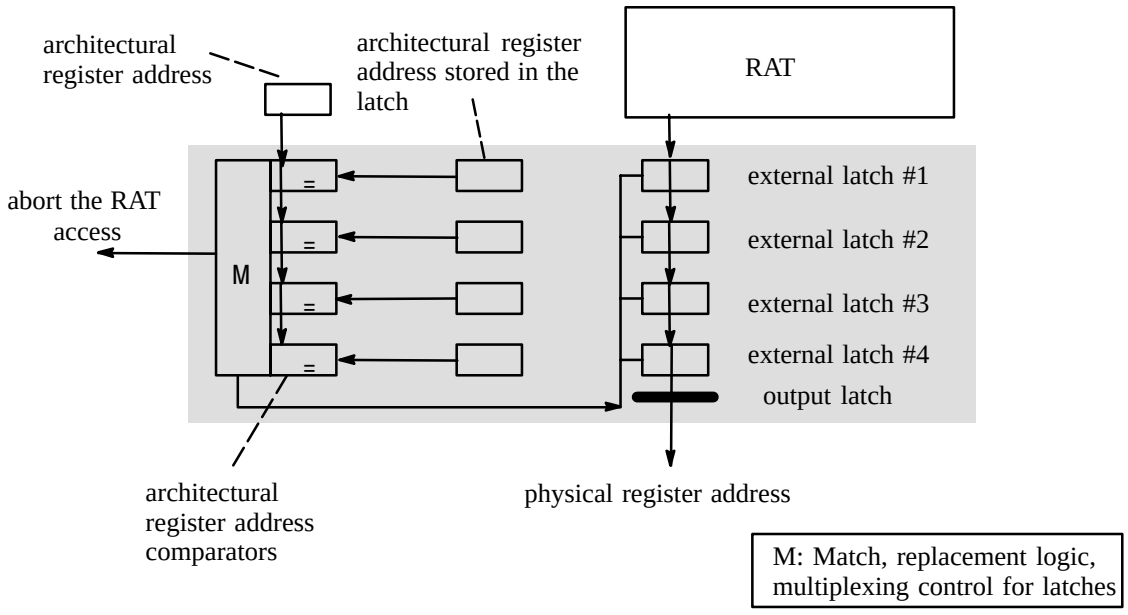


Figure 7–3. Renaming Logic with Four External Latches

It is critical to limit the energy spent in associatively addressing the ELs. To accomplish this, we make use of comparators that dissipate energy on a match (for example, as introduced in [EGK+ 02]), instead of traditional CAM cells or comparators that dissipate energy on a mismatch. The use of these comparators actually helps in two ways. First, as at most one of the ELs will have the matching entry, energy is dissipated within at most one comparator in the associative latch array at any time. Second, the comparators of [EGK+ 02] actually have a faster response time than the traditional comparators or CAM cell. The difference in timing between the comparator of [EGK+ 02] and the traditional comparator is, however, very small so even the traditional pull–down comparator can be used in our scheme, albeit with higher power dissipated in the latches. For the same reason as in GDBR scheme, the detection of a match in the ELs is completed well in advance of the normal sense amp activation instant. Therefore, the energy dissipation in the sense amps can be avoided without any cycle time sacrifice. These savings exceed that spent in locating a matching

entry within a latch array consisting of four ELs, as shown in the result section. Notice also, that the use of dissipate–on–match comparators within the intra–group dependency checking logic is not justified because of higher percentage of match situations. Our detailed analysis, using microarchitectural data about the bit patterns of the comparands indicate that the use of traditional comparators is a more energy–efficient approach for the use in the intra–block dependency checking logic [PKE+ 04].

Parameter	Configuration
Machine width	4–wide fetch, 4–wide issue, 4–wide commit
Window size	32 entry issue queue, 96 entry ROB (integrating physical regisrer file), 32 entry load/store queue
Function Units and Latency (total/ issue)	4 Int Add (1/1), 1 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 1FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
L1 I–cache	32 KB, 2–way set–associative, 32 byte line, 2 cycles hit time
L1 D–cache	32 KB, 4–way set–associative, 32 byte line, 2 cycles hit time
L2 Cache combined	512 KB, 4–way set–associative, 128 byte line, 4 cycles hit time
BTB	4096 entry, 4–way set–associative
Branch Predictor	Combined with 1K entry Gshare, 10 bit global history, 4K entry bimodal, 1K entry selector
Memory	128 bit wide, 60 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), fully associative, 30 cycles miss latency

Table 7–1. Architectural configuration of a simulated 4–way superscalar processor.

7.5. Simulation Methodology

To evaluate the energy impact of the proposed techniques, we designed and used the AccuPower toolsuite [PKG 02a]. The widely–used Simplescalar simulator [BA 97] was significantly modified to implement *true hardware level, cycle–by–cycle* simulation models for realistic superscalar processor. The main difference from the out–of–the–box Simplescalar is that we split the Register Update Unit (RUU) into the issue queue, the reorder buffer and the physical register file. It is important, because in real processors the number of entries in all these structures, as well as the number of ports to them are quite disparate.

The configuration of a 4–way superscalar processor are shown in Table 7–1. For simulating 6–way machine, we increased the window size, and the cache dimensions proportionately.

We simulated the execution of 9 integer *(bzip2, gap, gcc, gzip, mcf, parser, twolf, vortex* and *vpr*) and 8 floating point (*applu, apsi, art, equake, mesa, mgrid, swim* and *wupwise*) benchmarks from SPEC 2000 suite. Benchmarks were compiled using the Simplescalar GCC compiler that generates code in the portable ISA (PISA) format. Reference inputs were used for all the simulated benchmarks. The results from the simulation of the first 2 billion instructions were discarded and the results from the execution of the following 200 million instructions were used for all benchmarks.

For estimating the energy/power for the key datapath components, the event counts gleaned from the simulator were used, along with the energy dissipations, as measured from the actual VLSI layouts using SPICE. CMOS layouts for the RAT in a 0.18 micron 6 metal layer CMOS process (TSMC) were used to get an accurate idea of the energy dissipations for each type of transition. A 2 GHz clock and a V_{dd} of 1.8 volts were used for all the measurements.

7.6. Results and Discussions

Figure 7–4 shows the average number of accesses for each RAT separately for each simulated SPEC 2000 benchmark. As expected, floating–point register traffic for integer benchmarks is almost 0%. The exceptions are twolf and vpr benchmarks with very low floating–point traffic. The average is 4.2 accesses per cycle for the integer RAT and 0.6 accesses per cycle for the floating–point RAT. These percentages are identical for RAT line buffers, since RAT line buffers are accessed in parallel with RATs.

Figure 7–5 shows the hit ratio to the ELs. Separate results are presented for integer and floating point registers, with the use of 4 and 8 ELs. With the use of 4 ELs, the average hit

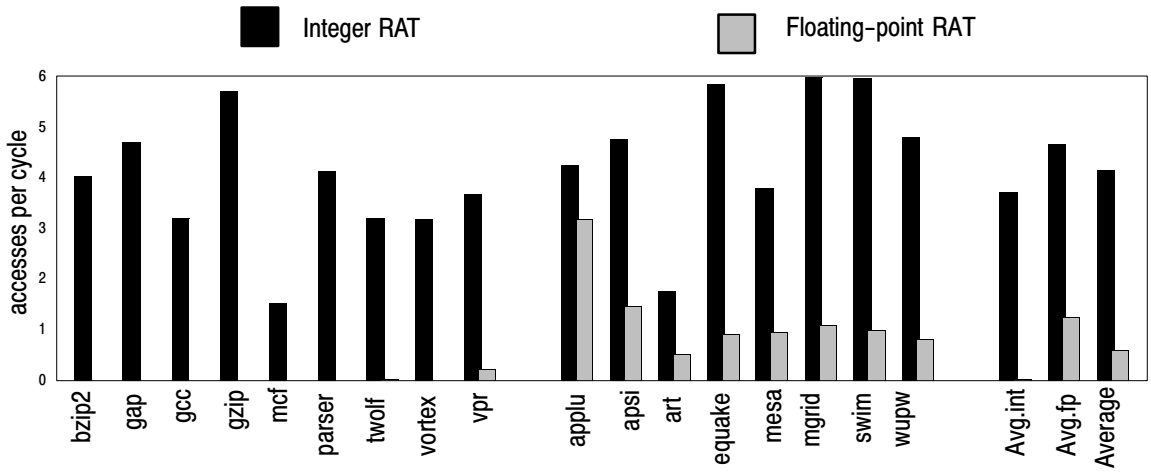


Figure 7–4. The average number of accesses to integer and floating–point RAT

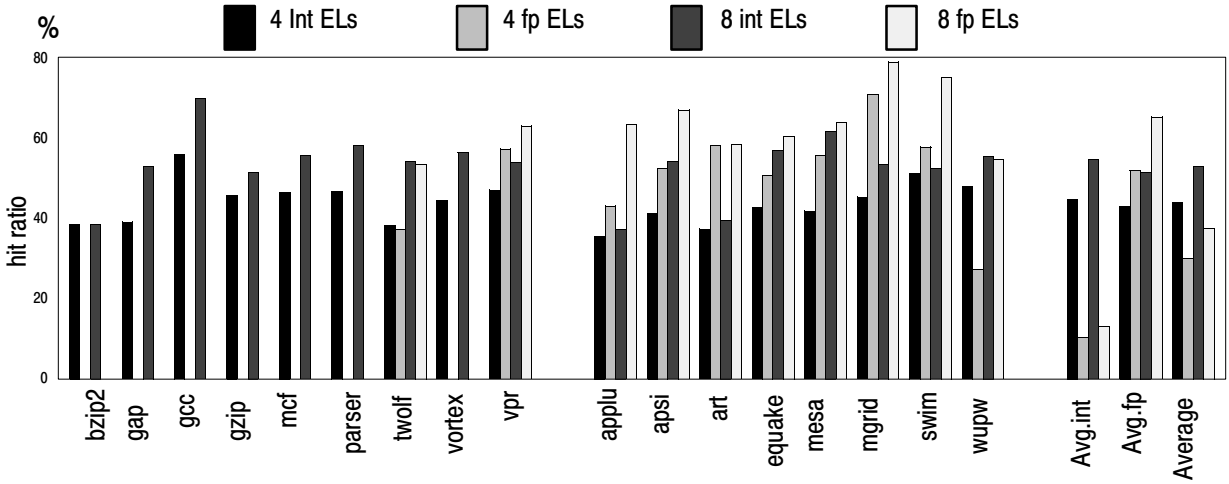


Figure 7–5. The hit ratio to integer and floating–point external latches (ELs)

ratio is about 44% for the integer ELs, and about 30% for the floating–point ELs. Adding four more entries to the ELs increases the hit ratios only slightly (about 53% for integer, and about 38% for floating–point ELs) because, again, the use of most register values is very close in proximity to the definitions of those registers. On the other hand, the higher number of ELs incurs increase in complexity and power.

Figure 7–6 shows the energy reduction achievable by applying the proposed techniques. The combined results for integer and floating point RAT logic are presented. The first bar shows the energy dissipation of the baseline RAT. The second and third bars show the energy impact of adding four and eight ELs, respectively. The last two bars show the energy reduction in the RAT id ELs and used in conjunction with GDBR. Average energy savings with the use of four and eight ELs are around 15% and 19%, respectively. The combination of both techniques results in about 27% energy savings on the average for four ELs and 30% energy reduction for eight ELs. Our analysis also indicate that the use of eight ELs is the optimal configuration, because the overall energy increases if the number of ELs goes beyond eight, due to the additional complexity and power of managing the ELs. At the same time, the percentage of hits stays nearly unchanged due to the reason described earlier.

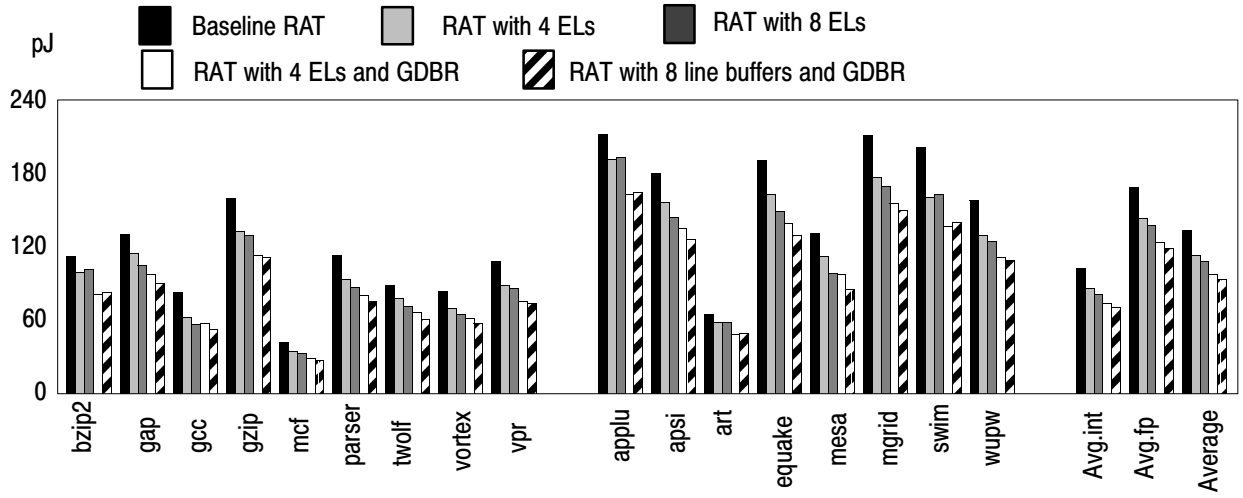


Figure 7–6. Energy of the baseline and proposed RAT designs

7.7. Related Work

A way to reduce the power consumption of the register alias table was proposed by Moshovos in [Mosh 02]. The proposed optimizations reduce power of the renaming unit in two ways. First, the number of read and write ports needed on the register alias table is

reduced. This is done by exploiting the fact that most instructions do not use the maximum number of source and destination register operands. Additionally, the intra–block dependence detection logic is used to avoid accessing the register alias table for those operands that have a RAW or a WAW dependence with a preceding, simultaneously decoded instruction. The technique of [Mosh 02] by only steering the sources that actually need to be translated through the RAT lookup to the RAT ports. The source addresses that are not used (for example, some instructions have only one source) or those produced by the instruction in the same block, do not have to be translated through the RAT array. All sources that need to be translated using the RAT access are first identified and then they are steered to the available RAT ports. If a port is not available, the renaming blocks. This incurs inherent performance degradation in terms of IPCs and, in addition, stretches the cycle time, because the filtering of source addresses and the RAT accesses are done serially. The payback is, of course, in significant reduction on the number of ports on the RAT. Simulation results show that for an aggressive 8–way superscalar machine it is possible to reduce the number of read ports from 24 to 12 and the number of write ports from 8 to 6 with a performance penalty of only 0.5% on the average across the SPEC 2000 benchmarks. The second optimization reduces the number of checkpoints that are needed to implement aggressive control speculation and rapid recovery from branch mispredictions. This is done by allowing out–of–order control flow resolution as an alternative to conventional in–order resolution, where the checkpoint corresponding to a branch can not be discarded till this branch itself as well as all preceding branches are resolved.

In [LL 00], Liu and Lu suggested using the hierarchical RAT. A small, first–level RAT is used to hold the mappings of the most recent renamed registers. Instructions access this small RAT first and only on a miss access the large full–blown second–level RAT. Because the process is serialized, the performance degradation is unavoidable as at least one extra cycle is needed in the front–end of the pipeline, thus increasing branch misprediction penalty.

In contrast to these techniques, our proposed mechanisms do not have any performance penalty, nor do they increase the cycle time.

7.8. Conclusions

In this study, we proposed two complementary techniques to reduce the energy dissipation within the register alias tables. The first technique uses the intra–group dependency checking logic already in place to disable the activation of sense amps within the RAT if the register address to be read is redefined by an earlier instruction dispatched in the same group. The second technique extends this approach one step further by placing a small number of associatively–addressed latches in front of the RAT to cache a few most recent translations. Again, if the register translation is found in these latches, the activation of sense amps within the RAT is aborted. The combined effect of the two proposed techniques is the energy reduction in the RAT of about 30%. This comes with no performance penalty, little additional complexity and no increase in the cycle time.

Chapter 8

Conclusions and Future Work

8.1. Summary of Contributions

Modern superscalar microprocessors exploit instruction–level parallelism (ILP) in the sequential codes by utilizing dynamic instruction scheduling, register renaming and speculative execution techniques. The power dissipation in these microprocessors will increase to alarming levels enough to melt the chip itself if it is not cooled adequately. The issue of adequately dissipating the heat generated will be the most formidable challange. For most devices targeted towards the consumer market, the change in price point due to extra cost of the cooling system can directly affect the success of the product in the market. Environment related laws and standards such as the EPA Energy Star also demand reduction in the power consumed by computer systems with the ultimate goal of reducing environmental pollution. Reducing the power dissipated in system with external power sources is also ultimately advantageous to the users because it reduces the operating expenses. Therefore, the power becomes a first–class, universal design constraint in the domain of high performance systems [Mudge 01]. Additionally, the performance of these microprocessors is directly proportional to the product – *Instructions_Per_Cycle* x *Clock_Frequency. Instructions_Per_Cycle* or IPC measures the amount of instruction level parallelism extracted by the dynamic scheduling of the microarchitecture and *Clock_Frequency* is the speed at which the microarchitecture clocked. Complex hardware

(larger structures, additional logic, etc.) helps to improve the IPC factor by extracting higher levels of instruction level parallelism. However, complex hardware employed to achieve higher IPC values can potentially slow the clock and hence, nullify the improvements in performance. In this dissertation, we have proposed several microarchitectural and circuit–level techniques for power and complexity reduction within superscalar microprocessors. All of our proposed techniques are technology independent and can be used in conjunction with other orthogonal techniques for saving power, such as voltage and frequency scaling and additional circuit–level techniques.

First, we studied three relatively independent techniques to reduce the energy dissipation in the instruction issue queues of modern superscalar processors:

1) We proposed the use of comparators in forwarding/tag matching logic that dissipate the energy mainly on the tag matches. This new comparator is measured to be around 70% more energy–efficient on the average than the traditional comparator. In terms of total issue queue energy, the savings due to the use of the new comparators amount to more than 10%.

2) We considered the use of zero–byte encoding to reduce the number of bitlines that have to be driven during instruction dispatch and issue as well as during forwarding of the results to the waiting instructions in the IQ. Zero–byte encoding by itself saves about 28% of the issue queue energy when the dispatch–bound issue queue is in place. On the other hand, energy savings with this technique amount to less than 10% when issue–bound datapath is utilized, since the encodable operands are no longer kept in the issue queue with this datapath style.

3) Finally, we evaluated power reduction achieved by the segmentation of the bitlines within the IQ. This technique results in more than 30% of energy savings in both issue–bound and dispath–bound datapath styles.

Proposed Technique	Impact of proposed technique		Competing Techniques
	Power	Complexity	
Bitline Segmentation	30–36% savings	IQ area increase as much as 5%	• Adaptive issue queue [BSB+ 00, FG 01, PKG 01b, BM 01c] • Energy–Efficient Comparators in IQ [BBS+ 00, KGPK 01, EGK+ 02, PKE+ 03] • IQ through several FIFOs [PJS 97] • A low–complexity issue logic using First/n–use table and distance issue/ deterministic latency schemes [CG 00, CG 01] • Reduced number of tag comparators [EA 02]
Zero–byte Encoding	12–28% savings	IQ area increase as much as 17–21%	
Energy–Efficient Comparators	9–16% savings	very small IQ area increase	
Combined	56–59% savings	IQ area increase as much as 25%	

Table 8–1. Summary of our proposed tecnhiques/their impact on Issue Queue and competing techniques

Table 8–1 shows the summary of our study on Issue Queue, in detail. Combined, these three mechanisms reduce the power dissipated by the instruction issue queue in superscalar processors by 56% to 59% on the average across all simulated SPEC 2000 benchmarks depending on the datapath design style. The IQ power reductions are achieved with little compromise of the cycle time. Specifically, zero–byte encoding logic adds 9% to the cycle time of the datapath with issue–bound operand reads and 12% to the cycle time of the dispatch–bound datapath. The delay of the new comparator roughly doubled compared to the delay of the traditional comparator and therefore the practical use of the proposed comparator is determined by the amount of slack (if any) present in the wakeup stage of the pipeline. If the wakeup cycle determines the processor's cycle time, then the alternative dissipate–on–match comparator designs that we proposed in [EGK+ 02] are better solutions.

It is very likely that the issue queue area increase when all three proposed techniques are used will not be cumulative. In the worst case, assuming cumulative effect of our techniques on the issue queue area, the total area of the issue queue increases by about 25%. Our ongoing studies also show that the use of all of the techniques that reduce the IQ power can also be used to achieve reductions of a similar scale in other datapath artifacts that use associative addressing (such as the reorder buffer [PKG 02b] and load/store queues). As the power dissipated in instruction dispatching, issuing, forwarding and retirement can often be as much as half of the total chip power dissipation, the use of the new comparators, zero–byte encoding and segmentation offers substantial promise in reducing the overall power requirements of contemporary superscalar processors.

Secondly, we proposed several techniques to reduce the complexity and the power dissipation of the ROB. These techniques are as follows:

1) We proposed a scheme to eliminate the ports needed for reading the source operand values for dispatched instructions. Our approach for eliminating the source read operand ports on the ROB capitalizes on the observation that only about 5% or so of the source operand reads take place from the ROB. Any performance loss due to the use of the simplified ROB structure was compensated for by using a set of retention latches to cache a few recently–written values and allowing the bulk of the source operand reads to be satisfied from those latches, from forwarding and from reads of the architectural registers. Our technique also removes the multi–cycle ROB access for source operands read from the critical path and substitutes it with the faster access to the retention latches.

2) We also introduced the conflict–free ROB distribution scheme, where the conflicts over the use of the write ports are eliminated by allocating a small separate FIFO queue for holding the speculative results for each functional unit. All conflicts over the read ports are eliminated by removing the read ports for reading out the source operand values from the distributed ROB completely and using the combination of a small set of

associatively–addressed retention latches and late result forwarding to supply the results to the waiting instructions in the issue queue.

Proposed Technique	Impact of proposed technique		Competing Techniques
	Power	Complexity	
Elimination of source read ports + Retention Latches	23–26% savings	• ROB area reduction as much as 45% • No arbitration logic for source read ports • Less complex bypass network	• Cluster Register Cache [BTME 02] • Replicated Register Files, Multi–banked register files [Kessler 99, CGV+ 00, BDA 01] • Value Aging Buffer [HM 00] • Lazy Retirement [SRG 02] • Value Aging Buffer [HM 00] • Isolation of short–lived values [PKEG 03] • Zero–byte encoding, energy–efficient comparators and dynamic resizing [PKG 02b]
Distributed Reorder Buffer	51% savings	• ROB area reduction as much as 70% • Additional arbitration logic for source read ports • Bypass network complexity for reading source operand values	
Combined	47% savings	• More than 70% ROB area reduction • No arbitration logic for source read ports • Less complex bypass network	

Table 8–2. Summary of our proposed tecnhiques/their impact on Reorder Buffer and competing techniques

Our designs result in extremely low performance degradation of 1.7% on the average across the simulated SPEC 2000 benchmarks and significant reduction in the ROB complexity and power dissipation. On the average, the ROB power savings of as high as 51% are realized. This includes the extra power dissipated within the retention latches. Taking into account that the ROB, as used in the considered datapath style, can dissipate as much as 27% of the total chip energy [FG 01], our techniques result in about 13% of total

CPU power reduction with only 1.7% average drop in performance in the worst case. Our design reduced the ROB complexity by eliminating all the read ports for reading source operand values and distributing the ROB structure into small register files. As a result, the area of the ROB is reduced by more than 70% in a P6 style datapath. Table 8–2 summarizes our study on the Reorder Buffer and lists some of the competing work on this area.

We also proposed two complementary techniques to reduce the energy dissipation within the register alias tables:

1) The first technique uses the intra–group dependency checking logic already in place to disable the activation of sense amps within the RAT if the register address to be read is redefined by an earlier instruction dispatched in the same group.

2) The second technique extends this approach one step further by placing a small number of associatively–addressed latches in front of the RAT to cache a few most recent translations. Again, if the register translation is found in these latches, the activation of sense amps within the RAT is aborted.

Proposed Technique	**Impact of proposed technique**		**Competing Techniques**
	Power	**Complexity**	
External Latches	15–19% savings	Additional area requirements for external latches	• RAT port reduction and checkpoint reduction [Mosh 02] • hierarchical RAT [LL 00]
Group Dependency Based Reading	10–12% savings	No cycle time increase, slight area increase in RAT	
Combined	27–30% savings	Slight area increase in RAT and additional area requirements for external latches	

Table 8–3. Summary of our proposed tecnhiques/their impact on Register Alias Table and competing techniques

The combined effect of the two proposed techniques is the energy reduction in the RAT of about 30%. This comes with no performance penalty, little additional complexity and no

increase in the cycle time. Table 8–3 summarizes our proposed techniques and their impact on power and complexity of the Register Alias Table.

Finally, we presented AccuPower – an accurate simulation tool for estimating power dissipation within various flavors of modern superscalar datapaths on a per–component basis. AccuPower consists of microarchitectural simulator in the form of significantly redesigned version of the Simplescalar simulator, the actual VLSI layouts for the key components of a superscalar processor, parameterized dissipation models and a power/energy estimator module. AccuPower can be used to estimate power savings achieved by several proposed microarchitectural and circuit–level techniques and also to study the impact of microarchitectural innovations in general. A variety of superscalar datapath models are built into AccuPower to facilitate such studies.

Bibliography

[ACPI 99] Advanced Configuration and Power Interface Specification (Intel, Microsoft, Toshiba), 1999.

[Alb 99] Albonesi, D., “Selective cache ways: On–demand cache resource allocation”, *in Proceedings of the 32nd International Symposium on Microarchitecture (MICRO)*, Heifa, 1999.

[AGG 03] Aragon, J., Gonzalez, J., Gonzalez, A., “Power–Aware Control Speculation Through Selective Throttling”, in Proceedings of the *9th International Symposium on High–Performance Computer Architecture (HPCA)*, February 2003.

[AHK+ 02] Asanovic, K., Hampton, M., Krashinsky, R., Witchel, E., “Energy–Exposed Instruction Sets”, in *Power–Aware Computing*, ed. by R. Graybill and R. Melhem, Kluwer Academic Publishing, 2002.

[AKS 93] Athlas, W., Koller, J., Svensson, L., “An Energy–Efficient CMOS Line Driver Using Adiabatic Switching”, *Technical Report ACMOS–TR–2, Information Sciences Institute USCLA*, July 1993.

[ASK 94] Athas, W., Svensson, L., Koller, J., Tzarzanis, N., Chou, E., “Low–Power Digital Systems Based on Adiabatic–Switching Principles”, *IEEE Transactions on Very Large Scale Integration Systems*, vol.2, No 4, December 1994, pp. 398–407.

[BA 97] Burger, D., and Austin, T. M., “The SimpleScalar tool set: Version 2.0”, Tech. Report, Dept. of CS, Univ. of Wisconsin–Madison, June 1997 and documentation for all SimpleScalar releases (through version 3.0).

[BABD 00] Balasubramonian, R., Albonesi, D., Buyuktosunoglu, A., and Dwarkadas, S., “Memory Hierarchy Reconfiguration for Energy and Performance in General–Purpose Processor Architectures”, *in Proceedings of the 33rd International Symposium on Microarchitecture (MICRO)*, 2000.

[BBB+ 02] Bose, P., Brooks, D., Buyuktosunoglu, A., et.al. “Early–Stage Definition of LPX: A Low–Power Issue–Execute Processor Prototype”, in Proceedings of HPCA Workshop on Power–Aware Computer Systems, 2002.

[BAB+ 02] Buyuktosunoglu, A., Albonesi, D., Bose, P., Cook, P., Schuster, S., “Trade–offs in Power–Efficient Issue Queue Design”, in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2002.

[BAS+ 01] Buyuktosunoglu, A., Albonesi, D., Schuster, S., Brooks, D., Bose, P., Cook, P., “A Circuit Level Implementation of an Adaptive Issue Queue for Power–Aware Microprocessors”, *in Proceedings of Great Lakes Symposium on VLSI Design*, 2001.

[BDA 01] Balasubramonian, R., Dwarkadas, S., Albonesi, D., “Reducing the Complexity of the Register File in Dynamic Superscalar Processor”, in *Proceedings of the 34th International Symposium on Microarchitecture* (MICRO), 2001, pp. 237–248.

[BF+ 95] Bunda, J., Fussell, D. et.al., “Energy–Efficient Instruction Set Architecture for CMOS Microprocessors”, in *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, 1995, pp. 298–304.

[Bha 96] Bhandarkar, D., “Alpha Implementations and Architecture – Complete Reference and Guide”, Digital Press, 1996.

[Borkar 99] Borkar, C., “Design Challenges for Technology Scaling”, *IEEE Micro,* Vol.19, N4, July–August 1999, pp. 23–29.

[BBS+ 00] Brooks, D.M., Bose, P., Schuster, S.E. et al, “Power–Aware Microarchitecture: Design and Modeling Challenges for Next–Generation Microprocessors”, *IEEE Micro* Magazine, 20(6), Nov./Dec. 2000, pp. 26–43.

[BIW+ 02] Brekelbaum, E., Il, J.R., Wilkerson, C., Black, B., “Hierarchical Scheduling Windows”, in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, Istanbul, Turkey, 2002.

[BKA+ 03] Buyuktosunoglu, A., Karkhanis, T., Albonesi, D., Bose, P., “Energy Efficient Co–Adaptive Instruction Fetch and Issue”, in *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, 2003.

[BM 02] Baniasadi, A., and Moshovos, A., “Branch Predictor Prediction: A Power–Aware Branch Predictor for High–Performance Processors”, in *Proceedings of the International Conference on Computer Design (ICCD)*, 2002, pp. 458–461.

[BM 01a] Brooks, D., Martonosi, M., “Dynamic Thermal Management for High–Performance Microprocessors,” in Proceedings of the 7*th International Symposium on High–Performance Computer Architecture (HPCA)*, 2001.

[BM 01b] Baniasadi, A., and Moshovos, A., “Instruction Flow–Based Front–end Throttling for Power–Aware High–Performance Processors”, *in Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2001, pp. 16–21.

[BM 01c] Bahar, I., Manne, S., “Power and Energy Reduction Via Pipeline Balancing”, *in Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2001, pp.218–229.

[BM 99] Brooks, D. and Martonosi, M., “Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance”, in *Proceedings of the 5th International Symposium on High Performance Architecture (HPCA*), 1999.

[Bose+ 01] Bose, P., Brooks, D., Irwin, M., Kandemir, M., Martonosi, M., Vijaykrishnan, N., “Power–Efficient Design: Modeling and Optimizations”, *tutotial notes, the International Symposium on Computer Architecture (ISCA)*, Goteborg, Sweden, July 2001.

[BSB+ 00] Buyuktosunoglu, A., Schuster, S., Brooks, D., Bose, P., Cook, P. and Albonesi, D., “An Adaptive Issue Queue for Reduced Power at High Performance”, *Workshop on Power–Aware Computer Systems*, held in conjunction wth ASPLOS, November 2000.

[BTM 00] Brooks, D., Tiwari, V., Martonosi, M., “Wattch: A Framework for Architectural–Level Power Analysis and Optimizations”, *in Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, 2000.

[BSP 01] Brown, M., Stark, J., Patt, Y., “Select–Free Instruction Scheduling Logic”, in *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, 2001.

[BTME 02] Borch, E., Tune, E., Manne, S., Emer, J., “Loose Loops Sink Chips”, in *Proceedings of the 8th International Conference on High Performance Computer Architecture*, (HPCA), February 2002.

[Burd 01] Burd, T., “ Energy–Efficient Processor System Design”, Ph.D. thesis, University of California at Berkeley, 2001.

[BV 01] Batson, B., Vijaykumar, T., “Reactive–Associative Caches”, in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[Cai 99] Cai, G., “Architectural Level Power/Performance Optimization and Dynamic Power Estimation”, *in Proc. of the Cool–Chips tutorial*. An Industrial Perspective on Low Power Processor Design in conjunction with MICRO–32, 1999.

[CG 00] Canal, R., Gonzalez, A., “A Low–Complexity Issue Logic”, in *Proceedings of the International Conference on Supercomputing (ICS)*, 2000.

[CG 01] Canal, R., Gonzalez, A., “Reducing the Complexity of the Issue Logic”, in *Proceedings of the International Conference on Supercomputing (ICS)*, 2001.

[CGS 00] Canal R., Gonzales A., and Smith J., “Very Low Power Pipelines using Significance Compression”, in *Proceedimngs of the 33rd International Symposium on Microarchitecture (MICRO)*, 2000.

[CGV+ 00] Cruz, J–L., Gonzalez, A., Valero, M., Topham, N., ”Multiple–Banked Register File Architecture”, in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, 2000, pp. 316–325.

[CP 95] Chang, J., Pedram, M., “Register Allocation and Binding for Low Power”, in *Proceedings of Design Automation Conference (DAC)*, 1995, pp. 29–35.

[CPG 00] Canal, R., Parserisa, J.M., Gonzalez, A., “Dynamic Cluster Assignment Mechanisms”, in Proceedings of HPCA, 2000.

[DB 99] De, V., and Borkar, S., “Technology and Design Challenges for Low Power and High–Performance”, Keynote Presentation, in *Proceedings of the International. Symposium on Low Power Electronics and Design*, 1999, pp. 163–168.

[DBB+ 02] Dropsho, S., Buyuktosunoglu, A., Balasubramonian, R., et.al. “Integrating Adaptive On–chip Structures for Reduced Dynamic Power”, in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002.

[DLC+ 00] Dhodapkar, A., Lim, C., Cai, G., Daasch R, “TEM^{2}P^{2}EST: A Thermal Enabled Multi–Model Power/Performance ESTimator”, in PACS Workshop, held in conjunction with ASPLOS, 2000.

[Dief 99] Diefendorff, K., “Athlon Outruns Pentium III”, Microprocessor Report, vol. 13, N 11, 1999.

[DKA+ 02] Dropsho, S., Kursun, V., Albonesi, D., Dwarkadas, S., Friedman, E., “Managing Static Leakage Energy in Microprocessor Functional Units”, in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, 2002.

[DLCD 00] Dhodapkar, A., Lim, C., Cai, G., Daasch R, “TEM^{2}P^{2}EST: A Thermal Enabled Multi–Model Power/Performance ESTimator”, in *Workshop on Power–Aware Computer Systems, held in conjunction with ASPLOS*, 2000.

[DS 02] Dhodapkar, A., Smith, J., “Managing Multi–Configuration Hardware via Dynamic Working Set Analysis”, in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[EA 02] Ernst, D., Austin, T., “Efficient Dynamic Scheduling Through Tag Elimination”, in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[EAB+ 02] Emer, J., Ahuja, P., Borch, E et.al. “ASIM: A Performance Model Framework”, in *IEEE Compute*r, February 2002, pp. 68–76.

[EF 95] R. J. Evans and P. D. Franzon., “Energy Consumption Modeling and Optimization for SRAMs”, *IEEE Journal of Solid–State Circuits*, Vol. 30, No. 5, pp. 571–579, 1995.

[Emer 01] Emer, J., “EV8: The post–ultimate alpha”, Keynote talk at *the International Conference on Parallel Architectural and Compilation Techniques*, September 2001.

[EGK+ 02] Ergin, O., Ghose, K., Kucuk, G., Ponomarev, D., ”A Circuit–Level Implementation of Fast, Energy–Efficient CMOS Comparators for High–Performance Microprocessors”, in *Proceedings of International Conference on Computer Design* (ICCD’02), Freiburg, Germany, 2002, pp. 118–121.

[FBH 02] Fields, B., Bodik, R., Hill, M., “Slack: Maximizing Performance Under echnological Constraints”, in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[FCJ+ 97] Farkas, K., Chow, P., Jouppi, N., Vranesic, Z., “The Multicluster Architecture: Reducing Cycle Time Through Partitioning”, in Proceedings of Int’l. Symp. on Microarchitecture, 1997.

[FDG+ 94] Furber, S., Day, P., Garside, J., Paver, N., Wods, J., “AMULET1: A Micropipelined ARM”, in *Proceedings of COMPCON*, 1994.

[FG 01] Folegnani, D., Gonzalez, A., “Energy–Effective Issue Logic”, *in Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, 2001, pp.230–239.

[FKM+ 02] Flautner, K., Kim, N., Martin, S., Blaauw, D., Mudge, T., “Drowsy Caches: Simple Techniques for Reducing Leakage Power”, in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[Fleet 94] Fleet, P., “The SH Microprocessor: 16–bit Fixed Length Instruction Set Provides Better Power and Die Size”, in IEEE 1994, pp. 486–488.

[FS 92] Franklin, M., Sohi, G., “Register Traffic Analysis for Streamlining Inter–Operation Communication in Fine–Grain Parallel Processors”, in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1992.

[GBC+ 01] Gunther, S., Binns, F., Carmean, D., Hall, J., “Managing the Impact of Increasing Microprocessor Power Consumption”, *Intel Technology Journal*, Q1, 2001.

[GBJ 98] Gowan, M., Biro, L., Jackson, D., “Power Considerations in the Design of the Alpha 21264 Microprocessor”, in *Proceedings of the 35th Design Automation Conference (DAC)*, 1998.

[GCG 00] Ghiasi, S., Casmira, J., and Grunwald, D., “ Using IPC variation in workloads with externally specified rates to reduce power consumption”, *in Proceedings of the Workshop on Complexity–Effective Design*, June 2000.

[Geb 97] Gebotys, C., “Low Energy Memory and Register Allocation Using Network Flow”, in *Proceedings of Design Automation Conference* (DAC), 1997.

[Gh 00] Ghose, K., “Reducing Energy Requirements for Instruction Issue and Dispatch in Superscalar Microprocessors,” in *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2000, pp. 231–234.

[GH 96] Gonzalez, R., and Horowitz, M., “Energy dissipation in general purpose microprocessors”, *IEEE Journal of Solid State Circuits*, 31(9): September 1996, pp 1277–1284.

[GGV 98] Gonzalez, A., Gonzalez, J., Valero, M., “Virtual–Physical Registers”, in *Proceedings of the 4th International Symposium on High–Performance Computer Architecture (HPCA)*, 1998.

[GK 99] Ghose, K., and Kamble, M. B., “Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit–line segmentation”, in *Proceedings of International Symposium on Low Power Electronics and Design*, August 1999, pp. 70–75.

[GPK+ 00] Ghose, K., Ponomarev, D., Kucuk, G., Flinders, A., Kogge, P., and Toomarian N.,“Exploiting Bit–slice Inactivities for Reducing Energy Requirements of Superscalar Processors,” in *Proceedings of Kool Chips Workshop, held in conjunction with MICRO–33*, 2000.

[GT+ 98] Garside, J., Temple, S., et.al., “The AMULET2 Cache System”, in *Power Driven Microarchitecture Workshop, held in conjunction with ISCA’98*, 1998.

[Gwe 94] Gwennap, L., “PA–8000 Combines Complexity and Speed”, Microprocessor Report, vol 8., N 15, 1994.

[HBH+ 02] Heo, S., Barr, K., Hampton, M., Asanovic, K., “Dynamic Fine–Grain Leakage Reduction Using Leakage–Biased Bitlines”, in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[HRT 02] Huang, M., Renau, J. and Torrellas, J. “Energy–Efficient Hybrid Wakeup Logic”, in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2002.

[HRYT 00] Huang, M., Renau, J., Yoo, S–M. and Torellas, J., “A Framework for Dynamic Energy Efficiency and Temperature Management”, *in Proceedings of the 33rd International Symposium on Microarchitecture (MICRO)*, 2000.

[Hinton+ 01] Hinton, G., et.al., “The Microarchitecture of the Pentium 4 Processor”, Intel Technology Journal, Q1, 2001.

[HCP 01] Hsieh, C–T., Chen, L–S, and Pedram, M., “Microprocessor Power Analysis by Labeled Simulation”, in *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, 2001.

[HJS+ 02] Hu, Z., Juang, P., Skadron, K., Clark, D., Martonosi, M., “Applying Decay Strategies to Branch Predictors for Leakage Energy Savings”, in *Proceedings of the International Conference on Computer Design (ICCD)*, 2002.

[HM 00] Hu, Z. and Martonosi, M., “Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics”, in Workshop on Complexity–Effective Design, 2000.

[HRT 03] Huang, M., Renau, J., Torrellas, J., “Positional Adaptation of Processors: Application to Energy Reduction”, in *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, 2003.

[IIM 99] Inoue, K., Ishihara, T., Murakami, K., “Way–prediction Seet–associative Cache for High Performance and Low Energy Consumption”, in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 1999, pp. 273–275.

[Ind+ 94] Indermaur, T. et.al. “Evaluation of Charge Recovery Circuits and Adiabatic Switching for Low Power CMOS Design”, in *Proceedings of the International Symposium on Low Power Electronics*, 1994.

[Intel 99] Intel Corporation, “The Intel Architecture Software Developers Manual”, 1999.

[IM 00] Iyer, A. and Marculescu, D., “Run–time Scaling of Microarchitecture Resources in a Processor for Energy Savings”, *in Proceedings of Kool Chips Workshop*, held in conjunction with MICRO–33, December 2000.

[IM 02] Iyer, A. and Marculescu, D., “Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors”, in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[IMM 02] Inoue, K., Moshnyaga, V., Murakami, K., “A Low Energy Set–Associative I–cache with Extended BTB”, in *Proceedings of the International Conference on Computer Design (ICCD)*, 2002, pp. 187–192.

[Itoh] K. Itoh, “Low Power Memory Design,” in Low Power Design Methodologies, ed. by J. Rabaey and M. Pedram, Kluwer Academic Pub., pp. 201–251.

[ISN 95] K. Itoh, K. Sasaki, and Y. Nakagome, “Trends in Low–Power RAM Circuit Technologies,” in *Proceedings IEEE*, vol. 83, No. 4, pp. 524–543, April 1995.

[JBM 03] Joseph, R., Brooks, D., Martonosi, M., “Control Techniques to Eliminate Voltage Emergencies in High Performance Processors”, in *Proceedings of the 9th International Symposium on High–Performance Computer Architecture*, February 2003.

[JC 95] Janssen, J and Coporaal, H., “Partitioned Register File for TTAs”, in *Proceedings of the 28th International Symposium on Microarchitecture (MICRO)*, 1995, pp. 303–312.

[Kessler 99] Kessler, R.E., ”The Alpha 21264 Microprocessor”, *IEEE Micro*, 19(2) (March 1999), pp. 24–36.

[KEP+ 03] Kucuk, G., Ergin, E., Ponomarev, D., Ghose, K., “Distributed Reorder Buffer Schemes for Low Power”, in *Proceedings of the 21st International Conference on Computer Design (ICCD)*, 2003.

[KEPG 03] Kucuk, G., Ergin, E., Ponomarev, D., Ghose, K., “Energy Efficient Register Renaming”, in *Proceedings of the 13th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2003.

[KGPK 01] Kucuk, G., Ghose, K., Ponomarev, D., Kogge, P., “Energy Efficient Instruction Dispatch Buffer Design for Superscalar Processors”, in *Proceedings of the International Symposium on Low–Power Electronics and Design (ISLPED)*, 2001, pp.237–242.

[KHM 01] Kaxiras, S., Hu, Z. and Martonosi, M., “Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power”, *in Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, 2001, pp.240–251.

[KL 03] Kim, I., Lipasti, M., “Half–Price Architecture”, in *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, 2003.

[KN 02] Keshavarzi, A., Narendra, S., “Impact of Process Variation and Leakage on Future CMOS Circuits”, *tutorial notes*, in *12th ACM Great Lakes Symposium on VLSI (GLSVLSI’02)*, April 2002.

[KPEG 03] Kucuk, G., Ponomarev, D., Ergin, O., Ghose, K., “Reducing Reorder Buffer Complexity Through Selective Operand Caching”, *in Proceedings of International Symposium on Low Power Electronics and Design (ISLPED’03)*, Seoul, Korea, August 2003.

[KPE+ 04] Kucuk, G., Ponomarev, D., Ergin, O., Ghose, K., “Complexity–Effective Reorder Buffer Designs for Superscalar Processors”, *IEEE Transactions on Computers*, 2004.

[KPG 02] Kucuk, G., Ponomarev, D., Ghose, K., “Low–Complexity Reorder Buffer Architecture”, in *Proceedings of the 16th International Conference on Supercomputing (ICS)*, 2002, pp. 57–66.

[KSB 02] Karkhanis, T., Smith, J., Bose, P., “Saving Energy with Just–In–Time Instruction Delivery”, in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2002.

[Kur+ 98] Kuroda, T., et.al., “Variable Supply–Voltage Scheme for Low–Power High–Speed CMOS Digital Design”, *IEEE Journal of Solid State Circuits*, vol. 33, No 3, March 1998, pp. 454–462.

[LBC+ 03] Li, H., Bhunia, S., Chen, Y., Vijaykumar, T.N. and Roy, K., “Deterministic Clock Gating for Microprocessor Power Reduction”, in *Proceedings of the 9th International Symposium on High–Performance Computer Architecture (HPCA)*, February 2003.

[Leib 00] Leibson, S., “XScale (StrongArm–2) Muscles In”, Microprocessor Report, 14(9), pp. 7–12, September, 2000.

[LG 95] Lozano, G. and Gao, G., “Exploiting Short–Lived Variables in Superscalar Processors”, *in Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1995, pp. 292–302.

[LKL+ 02] Lebeck, A., Koppanalil, J., Li, T., Patwardhan, J., Rotenberg, E., “A Large, Fast Instruction Window for Tolerating Cache Misses”, in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[LL 00] Liu, T., Lu, S., “Performance Improvement with Circuit Level Speculation”, in Proceedings of the 33rd International Symposium on Microarchitecture, 2000.

[LVA 95] Llosa, J. et al, ”Non–consistent Dual Register Files to Reduce Register Pressure”, in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 1995, pp. 22–31.

[Mak+ 98] Makino, H. et.al., “An Auto–backgate–controlled MT–CMOS Circuit”, in *Proceedings of the International Symposium on VLSI Circuits*, 1998, pp.42–43.

[McA 93] McAuley, A., “Dynamic Asynchronous Logic for High–Speed CMOS Systems”, *IEEE Journal of Solid–State Circuits*, Vol. 27, No 3, March 1992, pp. 382–388.

[MKG 98] Manne, S., Klauser, A., Grunwald, D., “Pipeline Gating: Speculation Control for Energy Reduction”, *in Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, 1998, pp. 132–141.

[Mon+ 96] Montanero, J. et.al., “A 160–MHz, 32–b, o.5–W CMOS RISC Microprocessor”, *IEEE Journal of Solid State Circuit*s, vol. 31, N 11, November, 1996, pp. 1703–1714.

[Moore 00] Moore, S., et.al., “Self–Calibrating Clocks for Globally Asynchronous Locally Synchronous Systems”, in *Proceedings of the International Conference on Computer Design (ICCD)*, 2000.

[Mosh 02] Moshovos, A., “Power–Aware Register Renaming”, Technical Report, University of Toronto, August, 2002.

[MPV 93] Moudgill, M., Pingali, K., Vassiliadis, S., “Register Renaming and Dynamic Speculation: An Alternative Approach”, in International Symposium on Microarchitecture, 1993, pp.202–213.

[MRH+ 02] Martinez, J., Renau, J., Huang, M., Prvulovich, M., Torrellas, J., “Cherry: Checkpointed Early Resource Recycling in Out–of–order Microprocessors”, in Proceedings of the 35th International Symposium on Microarchitecture, 2002.

[MR 9X] Microprocessor Report, various issues, 1996–1999.

[MR 02] Microprocessor Report, August 2002, p.35.

[MS 01] Michaud, P., Seznec, A., “Data–Flow Prescheduling for Large Instruction Windows in Out–of–Order Processors”, in *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA),* 2001.

[Mosh 02] Moshovos, A., “Power–Aware Register Renaming”, Computer Engineering Group Technical Report, University of Toronto, August 2002.

[Mudge 01] Mudge, T., “Power: A First–Class Architectural Design Constraint”, *IEEE Computer*, April 2001, pp. 52–58.

[Nii+ 98] Nii, K., et.al. “A Low Power SRAM Using Auto–Backgate–Controlled MT–CMOS”, in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED),* 1998, pp. 293–298.

[NBD+ 01] Narendra, S., Borkar, S., De, V., Antoniadis, D., Chandrakasan, A., “Scaling of Stack Effect and its Application for Leakage Reduction”, in Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), 2001, pp. 195–200.

[OG 98] Onder, S., Gupta, R., “Superscalar Execution with Dynamic Data Forwarding”, in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT),* 1998, pp.130–135.

[PAV 01] Powell, M., Agarwal, A., Vijaykumar, T., Falsafi, B., Kaushik, R., “Reducing Set–Associative Cache Energy via Way–Prediction and Selective Direct–Mapping”, in *Proceedings of the 34th International Symposium on Microarchitecture (MICRO),* 2001.

[PBB 00] Pering, T., Burd, T., Brodersen, R., “Voltage Scheduling in lpARM Microprocessor System”, in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2000.

[PJS 96] Palacharla, S., Jouppi, N. P. and Smith, J. E., “Quantifying the complexity of superscalar processors”, Technical report CS–TR–96–1308, Dept. of CS, Univ. of Wisconsin, 1996.

[PJS 97] Palacharla, S., Jouppi, N, and Smith, J., “Complexity–Effective Superscalar Processors”, in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997, pp. 1–13.

[Pol 99] Pollack, F., “New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies”, Keynote Presentation, *32nd International Symposium on Microarchitecture*, November 1999.

[PK+ 03] Ponomarev, D., Kucuk, G., Ergin, O., Ghose, K., “Power Efficient Comparators for Long Arguments in Superscalar Processors”, *in Proceedings of International Symposium on Low Power Electronics and Design (ISLPED’03)*, Seoul, Korea, August 2003.

[PKG 01a] Ponomarev, D., Kucuk, G., Ghose, K., “Dynamic Allocation of Datapath Resources for Low Power”, *in Proceedings of Workshop on Complexity–Effective Design*, held in conjunction with ISCA–28, June 2001.

[PKG 01b] Ponomarev, D., Kucuk, G., Ghose, K., “Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources”, in *Proceedings of the 34th International Symposium on Microarchitecture* (MICRO–34), December 2001, pp. 90–101

[PKG 01c] Ponomarev, D., Kucuk, G., Ghose, K., “Power Reduction in Superscalar Datapaths Through Dynamic Bit–Slice Activation”, *International Workshop on Innovative Architecture (IWIA)*, 2001, pp. 16–24.

[PKG 02a] Ponomarev, D., Kucuk, G., Ghose K., “AccuPower: An Accurate Power Estimation Tool for Superscalar Microprocessors”, *5th Design and Test in Europe Conference* (DATE’02), pp. 124–129.

[PKG 02b] Ponomarev, D., Kucuk, G., Ghose, K., “Energy–Efficient Design of the Reorder Buffer”, *12th International Workshop on Power and Timing Modeling, Optimizations and Simulation (PATMOS’02)*, Seville, Spain, September 2002, pp. 189–199.

[PKE+ 03] Ponomarev, D., Kucuk, G., Ergin, O., Ghose, K. and Kogge, P., “Energy–Efficient Issue Queue Design”, *IEEE Transactions on Very Large Scale Integration Systems*, Vol.11, No.5, October 2003, pp. 789–800.

[PKE+ 04] Ponomarev, D., Kucuk, G., Ergin, O., Ghose, K., “Energy–Efficient Comparators for Superscalar Datapaths”, *IEEE Transactions on Very Large Scale Integration Systems*, 2004.

[PKEG 03] Ponomarev, D., Kucuk, G., Ergin, O., Ghose, K., “Reducing Datapath Energy Through the Isolation of Short–Lived Operands”, *12th International Conference on Parallel Architectures and Compilation (PACT’03)*, New Orleans, LA, September 2003.

[Pop 91] Popescu V., et.al. “The Metaflow architecture”, *IEEE Micro*, June 1991, pp. 10–13, 63–73.

[Pount 93] Pountain, D., “Computing Without Clocks”, *Byte*, January 1993, pp. 145–150.

[PPV 02] Park, Il., Powell, M., Vijaykumar, T., “Reducing Register Ports for Higher Speed and Lower Energy”, in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, 2002.

[PSZ+ 02] Parikh, D., Skadron, K., Zhang, Y., Barcella, M., Stan, M., “Power Issues Related to Branch Prediction”, in *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA)*, 2002.

[PYF+ 00] Powell, M., Yang, S., Falsafi, B., Roy, K., Vijaykumar, T., “Gated–Vdd: A circuit Technique to Reduce Leakage in Deep–Submicron Cache Memories”, in Proceedings of the International Symposium on Low Power Electronics and Design, 2000.

[RBR 02] Raasch, S., Binkert, N., Reinhardt, S., “A Scalable Instruction Queue Design Using Dependence Chains”, in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[Ronen 02] Ronen, R., “Power – The Next Frontier: a Microarchitecture Perscpective”, keynote speech at the Workshop on Power Aware Computer Systems (PACS), held in conjunction with HPCA’02, Boston, February, 2002.

[SAD+ 02] Semeraro, G., Albonesi, D., Dropsho, S., Magklis, G., Dwarkadas, S., Scott, M., “Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture”, in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, 2002.

[SB 95] Stan, M., Burleson, W., “Bus–Invert Method Uses Coding for Low–Power I/O”, IEEE Transactions on Very Large Scale Integration Systems, 1995, Vol. 3, No 1, pp. 49–58.

[SBP 00] Stark, J., Brown, M., Patt, Y., “On Pipelining Dynamic Instruction Scheduling Logic”, in *Proceedings of the 33rd International Symposium on Microarchitecture (MICRO),* 2000.

[SC 99] Sherwood, T. and Calder, B., “Time Varying Behavior of Programs”, Tech. Report No. CS99–630, Dept. of Computer Science and Engg., UCSD, August 1999.

[SDC 94] Song, S.P., Denman, M., Chang, J., “The PowerPC 604 Microprocessor”, IEEE Micro, 14(5), pp.8–17, October 1994.

[Sla 94] Slater, M., “AMD’s K5 Designed to Outrun Pentium”, Microprocessor Report, vol.8, N 14, 1994.

[Small 94] Small, C., “Shrinking Devices Put a Squeeze on System Packaging”, EDN, Vol, 39, N4, pp. 41–46, Feb.17, 1994.

[SMB+ 02] Semeraro, G., Magklis, G., Balasubramonian, R., Albonesi, D., Dwarkadas, S., Scott, M., “Energy–Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling”, in *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA),* 2002.

[SP 85] Smith, J. and Pleszkun, A., “Implementation of Precise Interrupts in Pipelined Processors”, in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp.36–44, 1985.

[Sp 00] Standard Performance Evaluation Corporation, “Spec2000”, 2000. http://www.spec.org

[SRG 02] Savransky, G, Ronen, R., Gonzalez, A., “Lazy Retirement: A Power Aware Register Management Mechanism”, in Workshop on Complexity–Effective Design, 2002.

[STD 94] Su, C., Tsui, C., Despain, A., “Saving Power in the Control Path of Embedded Processors”, *IEEE Design and Test of Computers*, Vol.11, No 4, 1994, pp. 24–30.

[STR 02] Seznec, A., Toullec, E., Rochecouste, O., “Register Write Specialization Register Read Specialization: A Path to Complexity–Effective Wide–Issue Superscalar Processors”, in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO),* November 2002.

[STT 01] Seng, J., Tune, E., Tullsen, D., “Reducing Power with Dynamic Critical Path Information”, in *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, December 2001, pp. 114–123.

[SV 87] Sohi, G., Vajapeyam, S., “Instruction Issue Logic for High–Performance, Interruptable Pipelined Processors”, in Proceedings of ISCA, 1987.

[TA 03] Tseng, J., Asanovic, K., “Banked Multiported Register Files for High Frequency Superscalar Microprocessors”, in *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, 2003.

[Tak+ 98] Takahasi, M. et.al., “A 60–mw MPEG4 Video Codec Using Clustered Voltage Scaling With Variable Supply–Voltage Scheme”, *IEEE Journal of Solid State Circuits*, Vol. 33, No 11, pp.1772–1778, November, 1998.

[Tiwari 98] Tiwari, V. et al, “Reducing power in high–performance microprocessors”, *in Proceedings of the 35th Design Automation Conference (DAC)*, 1998.

[Tseng 99] Tseng, J., “Energy–Efficient Register File Design”, Master’s Thesis, Massachussets Instutute of Technology, December, 1999.

[TLD+ 01] Tune, E., Liang, D., Tullsen, D., Calder, B., “Dynamic Prediction of Critical Path Instructions”, in *Proceedings of the 7th Internatuonal Symposium on High Performance Computer Architecture (HPCA)*, 2001.

[TMW 94] Tiwary, V., Malik, S., Wolfe, A., “Compilation Techniques for Low Energy: an Overview”, in *Proceedings of the Internationsl Symposium on Low Power Electronics*, 1994.

[Usam+ 98] Usami, K., et.al. “Automated Low–Power Technique Exploiting Multiple Supply Voltages Applied to a Media Processor”, *IEEE Journal of Solid State Circuits*, vol.33, No 3, March 1998, pp. 463–471.

[VKI+ 00] Vijaykrishnan, N., Kandemir, M., Irwin, M.J. et al, “Energy–Driven Integrated Hardware–Software Optimizations Using SimplePower”, in *Proceedings of the 27th International. Symposium on Computer Architecture (ISCA)*, pp.95–106.

[VZA 00] Villa, L., Zhang, M. and Asanovic, K., “Dynamic Zero Compression for Cache Energy Reduction”, in *Proceedings of the 33rd International Symposium on Microarchitecture (MICRO)*, December. 2000.

[Wall 91] Wall, D.W., “Limits on Instruction Level Parallelism”, *in Proceedings of ASPLOS*, November 1991.

[WB 96] Wallace, S., Bagherzadeh, N., “A Scalable Register File Architecture for Dynamically Scheduled Processors”, in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1996.

[Wei+ 98] Wei, L. et.al., “Design and Optimization of Low Voltage High Performance Dual Threshold CMOS Circuits”, in *Proceedings of the Design Automation Conference (DAC)*, 1998.

[WJ 94] Wilton, S., and Jouppi, N., “An Enhanced Access and Cycle Time Model for On–Chip Caches”, WRL Research Report 93–5, DEC WRL, 1994.

[WLA+ 01] Witchel, E., Larsen, S., Ananian, S., Asanovic, K., “Direct Addressed Caches for Reduced Power Consumption”, in *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, December 2001, pp. 124–132.

[WM 99] Wilcox, K., Manne, S., “Alpha processors: A History of Power Issues and a Look to the Future”, *in Cool–Chips Tutorial*, held in conjunction with MICRO–32, November 1999.

[Yeager 96] Yeager, K., “The MIPS R10000 Superscalar Microprocessor”, IEEE Micro, Vol. 16, No 2, April, 1996.

[YK 93] Younis, S., Knight, T., “Practical Implementation of Charge Recovery Asymptotically Zero Power CMOS”, in *Proceedings of the Symposium on Integrated Systems*, 1993.

[YPF+ 01] Yang, S–H. Powell, M. D., Falsafi, B., Roy, K. and Vijaykumar, T.N., “An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep–Submicron High–Performance I–caches”, *in Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA)*, 2001.

[ZHC 99] Zhang, Y., Hu, X., Chen, D., “Global Register Allocation for Minimizing Energy Consumption”, in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 1999.

[ZK 00] Zyuban, V. and Kogge, P., “Optimization of High–Performance Superscalar Architectures for Energy Efficiency”, in *Proceedings of the International Symposium on Low–Power Electronics and Design (ISLPED)*, 2000, pp. 84–89.

[ZTR+ 01] Zhou, H., Toburen, M., Rotenberg, E., Conte, T., “Adaptive Mode Control: A Static–Power–Efficient Cache Design”, in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniaues (PACT)*, 2001.