Functional Unit Selection in Superscalar Microprocessors for Low Power

Pan Yan
National University of Singapore
panyan@nus.edu.sg

Tay Teng Tiow
National University of Singapore
eletaytt@nus.edu.sg

Abstract

A novel technique is introduced to reduce power consumption in Functional Units (FU). Extra slower FU with lower per-execution energy consumption are added into a processor. Thus, through code scheduling, instructions whose results are not immediately referenced after completion are issued to these power-frugal FU. Simulation suggests a prospect of saving 20% to 30% energy consumption in addition instructions while still improving the performance.

1. Introduction

Minimizing power consumption in microprocessors has become a critical design consideration, especially in battery-driven portable devices like PDAs and hand-phones. In such devices, microprocessors are expected not only to deliver high performance but also to consume moderate energy.

Recently, various power reduction techniques targeting on issue queue [1], execution core [2], cache structure [3] and other components inside a processor have been proposed. Apart from them, a most widely used technique is Dynamic Voltage Scaling (DVS) [4], which is based on the observation that maximum performance is not always necessary and by cleverly lowering the performance level here and there, the overall energy consumption can be reduced.

However, none of these available techniques has taken into account the fact that: 1) the design of Functional Units (FU) is always aiming at providing best performance; 2) the results of arithmetic and logic instructions are not always immediately referred to after their completion; 3) slower FUs, typically with a simpler circuit structure, consume significantly less energy than their faster counterparts [5]. Based on these facts, we present, in this paper, a novel power saving technique targeting on utilizing varied FU. We introduce extra slow but power-frugal FU into a processor. Hence, guided by the code scheduling algorithm, we try to expose instructions that could be executed in power-frugal FU while still keeping the overall performance. Simulation shows the potential of issuing 25% to 40% addition instructions to low power integer ALU while slightly improving the execution efficiency.

This technique provides a fine-grain mechanism for lowering performance at an instruction-by-instruction level, which allows instructions of different urgency to be executed at different power price. This technique can be implemented together with other power-saving techniques like DVS [4] and FU assignment [2], so whatever power saving achieved here is an extra gain. What is more, the overall performance is not harmed, thanks to the scheduling algorithm. The charm of this method also lies in its wide applicability and simplicity for practical implementation.

2. Hardware basis for FU selection

Our approach tries to execute instructions at different power cost. To provide such a knob, we intentionally introduce extra slower FU that has exact function of their fast counterparts. A previous work [6] has shown that slower FU are typically schematically simpler, employ fewer transistors for a certain function, and hence have lower per-execution energy. Simulation in [6] has shown that this is true for all kinds of FU, as shown in Table I. The simulation was carried out in Synopsis under 0.35um technology.

The introduced power-frugal FU are associated with additional instructions that will be employed by the scheduling algorithm discussed in the next section.

3. Code Scheduling Algorithm

With the extra power-frugal FU introduced and corresponding power-frugal instructions added, the remaining job is to selectively substitute normal

Table I Per-execution Energy and Data Arrivals

Functions		Fast	Slow	Difference (%)
Adder	Energy (pJ)	56	23	33 (58.9%)
	Latency(ns)	3.77	12.00	8.23
Multiplier	Energy (pJ)	703	394	309 (44.0%)
	Latency(ns)	8.17	27.11	18.94
Divider	Energy (pJ)	1218	1049	169 (13.9%)
	Latency(ns)	30.26	54.68	24.42

instructions for its low power counterpart so that the overall performance is not severely harmed.

3.1. Laxity of result generation

In order not to slow down the overall execution while still using power-frugal FU, the crux is to finish execution right before the result of an instruction is referred to. Conventionally, as FU are designed to provide best performance, many instructions are finished long before necessary. We simulated various benchmark programs on an out-of-order superscalar processor architecture listed in Table II in the SimpleScalar Toolset[6].

We added probes in the simulator to track the issue time of each instruction. For simplicity, we only analyzed addition instructions and estimated, for each iteration, the time gap between the issue of each addition and the issue of the first downstream instruction that uses its result. The percentage of

Table II Processor Configuration

Functional Units	4 integer ALU *Latency = 1 cycle*		
	2 slow integer ALU *Latency = 2 cycles*		
	1 integer Multiplier 1 integer Divider		
$Width_{Decode}$	4/cycle	Fetch Speed	4 per cycle
$Width_{Issue}$	4/cycle	Issue Mode	Out-of-order
$Width_{Commit}$	4/cycle	Branch Pred	Bimod

addition instructions are listed as "Addition%" in Table III. It can be seen they are predominant in every benchmark. The percentage of executed addition instructions that have a gap time of more than two cycles at run-time are listed, as "Laxity%". The abundance of execution laxity is obvious, ranging from 25% to 35% of all executed addition instructions at run-time. In code scheduling, our aim is thus to substitute these instructions for their power-frugal counterparts. What is more, we also try to re-order the codes so as to increase this power saving potential.

Table III Simulation Statistics

Bench-mark	Coverage	Addition%	Laxity%
go	78.7%	37.9%	26.3%
ijpeg	72.4%	36.2%	26.2%
gcc	69.7%	31.5%	25.1%
bzip	79.5%	39.2%	32.0%
gzip	68.1%	35.8%	34.6%
mcf	76.2%	30.3%	28.7%
parser	75.2%	32.2%	28.1%
vpr	67.7%	33.1%	25.7%

3.2. Overview and definitions

We designed an all-in-one algorithm which both tries to expose more instructions for power-frugal execution and to make the FU selection. The objective of the scheduling is first to minimize the execution time and second to pick out instructions that are likely to have enough laxity in result generation. The algorithm works on each separate Basic Block (BB). The general BB scheduling without FU selection, which is NP-complete as discussed in [7], is a special case of our program. Thus our problem is also NP-complete. To prevent super large BB from severely slowing down the scheduling of object codes, we chop BB larger than 40 instructions into smaller pieces. Such size limit does not noticeably harm the scheduling room but effectively reduces scheduling time.

To maintain the function of each BB, the Instruction Dependence Table (IDT), which tracks the relationship between each pair of instructions, is built. This is done through a single-pass scanning, which compares the input and output registers of each instruction. The IDT hence serves as the constraint for the valid positions for each instruction in the BB. An initially empty pipeline is assumed for each BB. The scheduling is carried out on a cycle-by-cycle basis, trying to fit as many instructions with all-ready input registers into a certain cycle slot as the issue width allows, emulating the issue logic of out-of-order processors. Instructions in a solution must preserve their original relative order as in the original BB. To efficiently describe the algorithm, we have the following definitions.

Definition: Dominance. Instruction ***i*** is said to dominate ***j*** if ***j*** is Read-After-Write (RAW) dependent on ***i*** according to IDT.

Definition: Semi-Dominance. Instruction ***i*** is said to semi-dominate ***j*** if ***j*** is Write-After-Write (WAW) or Write-After-Read (WAR) dependent on ***i*** in IDT.

Definition: Live Instruction. An instruction ***i*** is said to be a live instruction for cycle **n**, if ***i*** is scheduled in a previous cycle and has not finished execution.

Definition: Quasi-Ready Instruction. An instruction ***i*** is said to be a quasi-ready instruction for cycle **n** if ***i*** is not dominated by any un-scheduled or live instruction for cycle n.

Definition: Ready Instruction. An instruction ***i*** is said to be a ready instruction for cycle **n** if ***i*** is a quasi-ready instruction and also is not semi-dominated by any quasi-ready instruction for cycle **n**.

Definition: Instruction Equivalence. Two instructions ***i*** and ***j*** are said to be *equivalent* if the *all* the following conditions are met at the same time:

1. For any other instruction ***k***, either ***k*** dominates both ***i*** and ***j*** or ***k*** dominates neither.
2. For any other instruction ***k***, either both ***i*** and ***j*** dominates ***k*** or neither dominates ***k***.
3. The FU related to ***i*** and ***j*** are the same.

Definition: Group. Equivalent instructions are said to be of a same *group*.

3.3. Scheduling Algorithm

For each cycle slot, IDT is first scanned to build a pool of all the quasi-ready instructions. Then, all the possible valid combinations of instructions to be filled in the current cycle slot are generated. These combinations of instructions are called solutions for a specific cycle slot. A valid solution is one that meets either of the following two conditions:

1. Consist of all ready instructions
2. Consist of some quasi-ready instructions and all the un-scheduled instructions that semi-dominate these quasi-ready instructions.

With the generated valid solutions, a trimming step is taken to remove all the equivalent solutions to improve the efficiency of this scheduling algorithm. Two *solutions* are said to be equivalent if they have an equal number of equivalent instructions in each group.

The remaining solutions are expanded with regard to FU selection. That is, if a solution has **n** FU related instructions and each of them has **m** possible versions of FU of different latency, the solution will be expanded to m^n solutions, each of which represents a different FU selection combination for the FU related instructions. Redundant expanded solutions (for example, supposing a solution to be expanded containing two instructions A & B from a same *group* that has 2 versions of FU associated, then one of [A_1, B_2] and [A_2, B_1] is redundant as they are identical for scheduling.) are again trimmed.

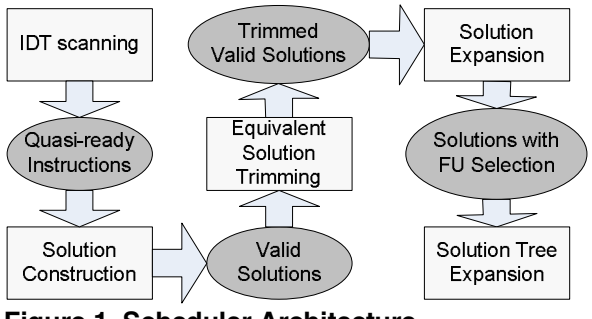

Figure 1. Scheduler Architecture

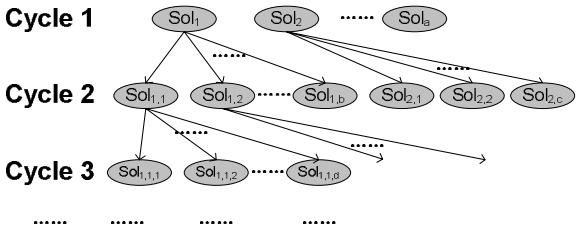

Figure 2. Solution Tree

Now all the possible instructions to be issued in the current cycle have been generated. The architecture of the scheduler for one cycle is represented in Figure 1.

Each of these solutions will be tried for the current cycle and the live instruction list will be updated accordingly before the next cycle is triggered. Hence a solution tree, illustrated in Figure 2, is explored

When all the instructions in the BB have been scheduled, the trace from the cycle 1 level solution node to the last cycle level solution node represents a unique instruction order and FU selection. The total number of cycles represents the estimated execution duration of the BB. An optimal schedule of the BB is the one having the lowest accumulated power cost among those with the shortest estimated execution duration. Obviously execution efficiency is the priority.

4. Simulation Results

We applied our scheduling algorithm on seven SPEC benchmark programs compiled for the PISA architecture [6]. For simplicity, only addition instructions were analyzed. Scheduled codes were executed on a processor model as described in Table I. Statistics are listed in Table IV. Due to the BB size limit and solution tree trimming , scheduling time for the benchmark programs are all within 10 minutes on a Pentium 4 3GHz desktop PC.

Table IV Scheduled Statistics

Bench-marks	Static Tag-rate	Dynamic Tag-rate	Baseline IPC	Unsched. IPC	Sched. IPC
go	30.1%	27.9%	0.9516	0.9497	0.9537
ijpeg	33.7%	29.5%	1.3701	1.3744	1.3809
gcc	24.9%	25.4%	1.0825	1.0811	1.0859
bzip	26.6%	37.1%	1.4200	1.4149	1.4246
gzip	24.2%	39.6%	1.4762	1.4738	1.4829
mcf	24.5%	30.3%	0.6721	0.6729	0.6728
parser	26.9%	29.5%	0.9892	0.9887	0.9955
vpr	31.8%	27.5%	1.1703	1.1623	1.1707

The scheduling algorithm re-orders instructions so as to allow more laxity for result generation. As a result, approximately 25% to 33% addition instructions were statically identified as candidates for power-frugal execution in the object codes, as shown in column 2 in Table IV. This static tag-rate, however, is not representative of power reduction because instructions are not evenly executed throughout the object code. As the power consumption is calculated by the actual FU energy dissipated at run time, the dynamic tag-rate, which is generated by another round of simulation of the modified object codes, should be used for estimating the overall power saving. This dynamic tag-rate can be compared with the "Laxity%" in Table III, as illustrated by the figure below.

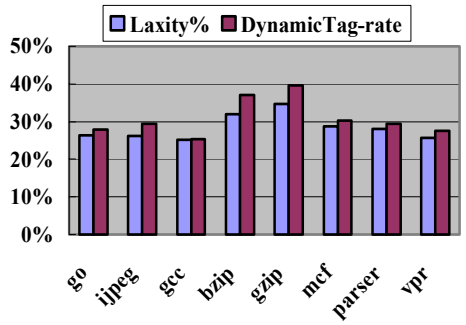

Figure 3. Power-frugal Execution Percentage

The "Laxity%" is the actual result generation laxity of each *original* object code at run time. Through instruction re-ordering, our scheduling algorithm exposed more candidates for low-power execution than the original and effectively achieves 101% to 116% of the power saving potential. Assuming the adder energy profile in Table I, around 20% to 30% of dynamic power saving can be achieved in integer adders.

While saving the dynamic power, our algorithm does not compromise the execution performance of the object code, but rather slightly improves it. Instruction Per Cycle (IPC) is used as the index for execution performance.

In our algorithm, there are two contradictory forces that affect the IPC of the scheduled codes. On the one hand, the algorithm makes "shortest execution duration" its first priority and hence tends to improve the IPC of the scheduled code by re-ordering. On the other hand, the BB-based algorithm inevitably incurs some inaccuracy in estimating the execution laxity of each instruction and thus may cause IPC degradation. To split the two effects, we generated another set of object codes that only contains FU selection according to the simulation statistic. In these "un-scheduled" object codes, addition instructions that have result generation laxity of more than 2 cycles for over 90% of

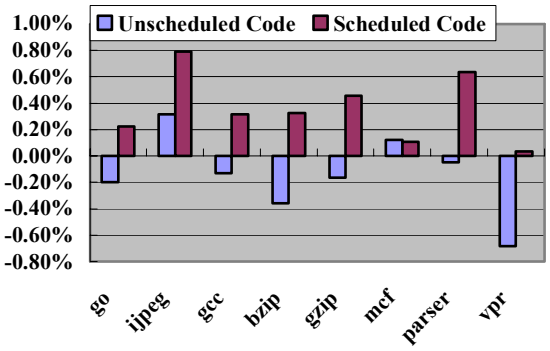

Figure 4. Normalized IPC Difference

execution iterations were tagged for low-power execution. Simulation results show that the IPC of many of these unscheduled codes slightly degraded (less than 0.8%) as expected. The exception of ijpeg and mcf are possibly because the extra FU included relieved resource conflicts. Such performance degradation is already trivial. However, with our algorithm, the scheduled codes all enjoy improved IPC, demonstrating that the FU selection inaccuracy is satisfactorily counteracted by the performance gain caused by code order scheduling.

5. Conclusion

In this paper, we presented a novel processor power reduction technique by executing instructions of different urgency at different power cost. Through static object code scheduling, instructions with enough execution laxity are exposed, substituted for power-frugal counterparts and hence executed in low power FU. Simulation shows a prospect of saving 20% to 30% energy in integer adders while still slightly improving IPC.

6. References

[1] T. M. Jones, *et al*, "Software Directed Issue Queue Power Reduction", *Proc. of HPCA-11*, 2005.

[2] S. Haga, *et al*, "Dynamic Functional Unit Assignment for Low Power", *Proceedings of DATE'03*, 2003.

[3] W. Zhang, *et al*, "Compiler-Directed Instruction Cache Leakage Optimization", *MICRO-35,* 2002

[4] T.D. Burd, *et al*, "A dynamic voltage scaled microprocessor system", *Solid-State Circuits, IEEE Journal of*, Nov. 2000

[5] T.T. Tay *et al*, "Hw/Sw Co-Design for Low Power Arithmetic and Logic Units", *Int'l Jnrl. of Software Eng. and Knowledge Eng.* 15(2): pg. 335-342, 2005.

[6] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 3.0. *Technical report, Computer Sciences Department, University of Wisconsin-Madison*, 1999.

[7] P. Faraboschi, et al, “Instruction Scheduling for Instruction Level Parallel Processors”, *Proc. of The IEEE, Vol. 89,, No. 11*, November 2001.