Wattch-alpha: A Power Analysis Simulator for Alpha Architecture

CSE548 Computer Architecture Project
Song Li, Zizhen Yao

Abstract

1. Introduction

Traditionally, power dissipation problem was a major concern largely in embedded system and portable computer community. Their goal is design low power chips that work within limited battery life. More recently, however, power issues are also becoming one of the major constraints in high-performance microprocessor design. With fast increasing clock rate and transistors density, it is becoming more difficult to dissipate the heat produced by the chip efficiently. Conventional air-cooling technique is already reaching its limits in today's high-performance microprocessors.

A number of power saving technologies have been proposed and widely deployed in current microprocessor design. Voltage scaling is among the most important ones. Specialized lower-level circuit techniques, conditional clocking, and low swing busses all play a big role in power-efficient computer system. However, none of these techniques are sufficient for chips with fast increasing clock rate and die size. More aggressive approaches have to be taken.

Power analysis tools are needed to evaluate and quantifies the power efficiency of different architecture designs. Lower-level power tools such as PowerMill[1] operate on circuit or Verilog level. While providing pretty good simulation accuracy, they can be applied only at the late stage of architecture design when circuit level details are known. It would be more desirable to estimate the power consumption given the high level description of architecture to avoid the waste of efforts on inappropriate design. Tradeoff between simulation efficiency and accuracy has to be made. Wattch is an architecture-level power analysis tool developed at Princeton University. Their power estimates are based on parameterized power models for different hardware and on resource utilization through cycle-level simulation. It is built on SimpleScalar. Wattch is orders of magnitude faster than low-level simulator. Compared to the published statistics of power consumption for commercial high-end microprocessors, the simulator gives consistent results across different processors, and across difference components of one processor, though the absolute power consumption is not accurate. Therefore, Wattch can be used a tool for comparison of power consumption of different architecture designs, but not for real power consumption estimation. Other similar power analysis tools include SimplePower[3] that simulates bus and memory system, and the PowerAnalyzer[4] on ARM architecture in SimpleScalar toolkits.

In this project, we extend the work in Wattch in the following ways: first, we make Wattch portable to Sim-Alpha simulator, and name it Wattch-Alpha (Wattch for Alpha processor). Sim-Alpha is a superscalar simulator based on alpha 21264 architecture, the state-of-art high-end microprocessor with published architecture design details. Sim-Alpha is claimed as a more realistic, detailed, and hopefully a more accurate simulator compared to SimpleScalar 3.0. In prospect that Sim-Alpha will be a more popular simulation tool in future, we design the power analysis tool based on Sim-Alpha, which is likely to yield more accurate simulation results for alpha-like microprocessors. We compute the simulation results of Wattch and Wattch-Alpha, and show that there are non-trivial differences between the two because of the underline differences in

the architectures/assumptions made by SimpleScalar 3.0 and Sim-Alpha. Then, we apply Wattch-Alpha to evaluate a FIFO-based microarchitecture for issue window design, and we show that this design can effectively reduce the complexity of the circuit, and thus saving power, while preserving good performance.

2. Power Modeling Methodology in Wattch

In CMOS microprocessors, dynamic power consumption is the main source of power consumption. Dynamic power is dissipated when the device output capacitance is charged to VDD through the PMOS device and discharged to VSS through the NMOS device. This power is given by $\boldsymbol{p=cv^2af}$, where c is the load capacitance, v is the supply voltage, and f is the clock frequency. The activity factor a, is between 0 and 1 indicating how often clock ticks lead to switching activity on average. For circuits pre-charge or discharge each cycle, the activity factor is 1. The activity factors of some other critical circuits are measured by running benchmark programs on architecture simulator. For some circuits, it is impossible to measure the activity factor, and base value of .5 is used.

Wattch use parameterized modeling techniques to estimate the capacitance for the circuits that make up the processor. For each hardware unit, the capacitance consists of three components: diffusion capacitance, gate capacitance and metal capacitance of wires (not necessarily all three). Wattch classified the main processor units into following four categories

- Array structure: Data and instruction caches, cache tag arrays, register files, register alias table, branch predictor, instruction window, and load/store queue.
  Array structure is parameterized on number of rows, columns and read/write ports.
  The components modeled include decoder, wordline drive, bitline discharge, and output drive. Among them, wordline drive and bitline discharge accounts for bulk of the power consumption. The wordline capacitance is computed as follows:
  $C_{wordline}=C_{diff}(WordlineDriver) + C_{gate}(CellAccess)*NumBitlines + C_{metal}*WordlineLength$
  The Bitline capacitance is computed similarly
  $C_{bitline}= C_{diff}(Precharge) + C_{gate}(CellAccess)*NumWordlines + C_{metal}*BitlineLength$

- Fully Associative Content-Addressable Memories: Instruction window/ reorder buffer wakeup logic, load/store checks, TLB.
  The major components of CAM structures are taglines and matchlines. Again, the parameters are number of rows, columns and ports. The computation formula for the capacitance is
  $C_{tagline}= C_{gate}(CompareEn) *TagSize+ C_{diff}(CompareDriver) + C_{metal}*TaglineLength$
  $C_{matchline}= 2*C_{diff}(CompareEn) *TagSize+ C_{diff}(MatchPreCharge) + C_{diff}(MatchOR)+C_{metal}*MatchlineLength$

- Combinational Logic and wire: Function units, instruction window selection logic, dependency check logic, and result buses.
  ALU power in Wattch is a simple constant [6].
  Instruction window selection modeling [5]
  $C_{instSelection}= (DecodeWidth-1)*DecodeWidth* C_{compare}$
  Result buses modeling[7]:
  $C_{resultBus}=0.5* C_{metal}*NumALU*ALUHeight + C_{metal} * RegfileHeight$

- Clocking: Clock buffers, clock wires and capacitive loads.

Clocking network is the most significant source of power consumption in microprocessor, which can be categorized into three sources:

1. Global Clock Metal lines. Wires that route the clock across the processor. Wattch assumes H-tree network that clock routing paths all over the chip have same length, and have no clock skew.
2. Global Clock Buffers: large transistors used to drive the clock throughout the chip.
3. Clock Loading: both explicit and implicit clock loading are considered. Explicit clock loads include the gate capacitance of nodes directed connected to clock, while implicit clock loads include the load due to pipeline stages.

Conditional Clocking:

To accomplish more power saving for multi-ports hardware units, current CPU designs increasingly use conditional clocking to turnoff parts of a hardware that is not needed on a cycle to cycle basis. Wattch simulates three clocking style:

1. All or nothing clock gating: assume full power consumption independent of number of accesses in one cycle.
2. Linear clock gating: power consumption is in proportional to number of accesses.
3. Linear scale with port or unit usage, plus unused units dissipates 10% of their maximum power.

3. Sim-Wattch Outline

In Wattch, the power models are interfaced with SimpleScalar. SimpleScalar simulates 5 stages out-of-order processor: fetch, decode, issue/execute, writeback and commit. The simulated resources include fetch queue, issue queue, reservation update unit (RUU), etc. It contains 32 integer registers and 32 floating point registers that make up architecture register files. For more accurate estimation of cycle time, Wattch assume three additional pipestages between fetch and issue, and seven cycles of mispredict penalty. Wattch keeps track of which units access on each cycle and how, and power model is applied on accessed units to estimate the power consumption.

Unit Power	Unit Access Counter
Integer ALU	ialu_access;
Function Unit	falu_access
Branch Predictor	bpred_access
Renaming Logic	rename_access
Issue Window (read operands, select ready instructions to FU, wake up pending instructions)	window_preg_access
	window_selection_access
	window_wakeup_access
Load and Store Queue (read operands, wake up pending loads and stores)	lsq_preg_access
	lsq_wakeup_access
Register File	regfile_access
Result bus	resultbus_access
Instruction cache	icache_access
Data Cache	dcache_access
Level 2 Data Cache	Dcache2_access
Clock	

Table 1: Power consumption of hardware structures simulated by Wattch and access counter involved.

4. Wattch-Alpha Design

We notice that Sim-Alpha and SimpleScalar has some architecture level differences. Some of them have great effect on the redesign of Wattch-Alpha based on Wattch. Below we describe some of these differences between Sim-Alpha and SimpleScalar, their impact on power consumption, and how we deal with such differences.

1. Reservation station vs. Physical register architecture.

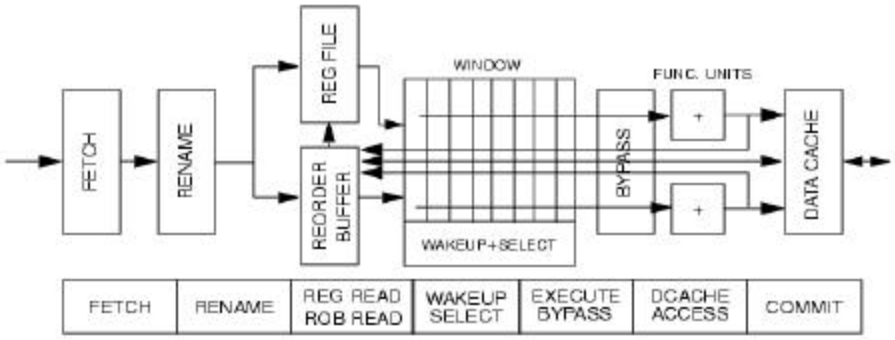

Fig.1. Reservation station architecture (adopted by Intel Pentium Pro, PowerPC)

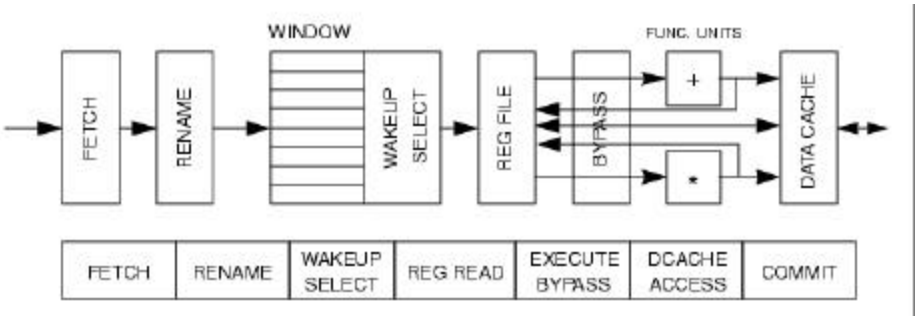

Fig.2. Baseline architecture (adopted by Alpha 21264, MIPS R10000)

SimpleScaler 3.0, thus Wattch is based on Reservation station architecture, while Sim-Alpha, and our Wattch-Alpha uses baseline model. In reservation station model, the reorder buffer holds speculative values and register file holds only committed, non-speculative data. In baseline model, register file holds both committed and speculative data. Another difference is, in the reservation station model, completing instructions broadcast results values to the reservation stations, while in the baseline model, the results are broadcast to issue window.

There are no fundamental differences between the two except that the register file in baseline model is much bigger than the reservation station model. The physical registers can be viewed as reservation stations in some sense. The differences are however, significant to our implementation.

2. Number of pipeline stages.

As we mention above, SimpleScalar has 5 pipeline stages: fetch, decode, issue/execute, writeback, and commit while Sim-Alpha has 6 pipeline stages, with one more slot stage between fetch and map (corresponding to decode stage in SimpleScalar). In slot stage, instructions are statically slotted to either UPPER or LOWER sub-clusters depending on their positions in the

fetch packet and their resource requirement. Since we are not modeling slot logic power consumption, we do not keep it access counter.

3. Fetch unit.

Sim-Alpha fetches the whole block of instructions while SimpleScalar fetches each instruction separately. This will cause significant change of in performance. Sim-Alpha simulates aggressive speculative fetches and execution. All the instructions fetched in a block will be issued and executed unless pipeline is flushed. The actual number of issued instruction could vary significantly compared to number of committed instructions. This will affect the number of accesses to almost all major units in the processors. Therefore power computation based on Sim-Alpha could be very different from the one based on SimpleScalar, even if the power models and parameters used for the major structure units are similar.

4. Line and set prediction.

The Alpha 211264 instruction cache implements two-way associability via line and set prediction technique that combines the speed advantage of a direct-mapped cache with the lower miss ratio of a two-way set-associative cache [8]. Each fetch block of four instructions includes a line and set prediction, which indicates where is the next block of instructions. Sim-Alpha simulates this design while it is not available in SimpleScalar. Line and set prediction is looked up at fetch stage, and updated at slot stage. We do not simulate power consumption of Line and Set prediction at this stage due to time constraint.

We notice some limitations of Wattch that may affect its accuracy. In Wattch, branch predictor access counter is updated only when branch predictor buffer is updated, and ignore the branch predictor buffer lookup at fetch or slot stage. Though it is not accurate, we take the same approach due to our lack of knowledge in branch prediction circuit design. Similarly, Wattch also ignore the renaming table lookup at dispatch or map stage. Again, we are not addressing this issue here. We also notice that Sim-Alpha simulates larger instruction set compared to SimpleScalar 3.0, which includes ITOF, FTOI, and two more addressing modes, DISP, RR. In ITOF, the instruction uses integer function unit, while Wattch assumes that the instruction will access floating-point unit. Wattch also assumes that load and store instructions do not access functional units, which is not true. We have fixed this problem.

5. Sim-Alpha simulate more Memory management units, e.g. mshr, victim buffer and bus. We ignore these structures in power consumption computation.

5. Comparison of Wattch and Wattch-Alpha

1. Architecture configuration:

To compare Wattch and Wattch-Alpha, we need to test under the similar architecture configuration, so that the results are comparable. We choose Alpha 21264 microprocessor as targeted simulation environment and use the same set of parameters. However, because Wattch is based on a different architecture, we cannot use exactly the same configuration. Summarization of the configuration we used for Wattch and Sim-Wattch is given in Table 2.

There are some subtle differences between Wattch and Sim-Alpha in addition to the ones we discuss in section 4. In Wattch, load and store instructions share a single load/store queue, while in Sim-Alpha, load instructions and store instructions use different queues. Sim-Alpha can issue up to of 4 integer instructions and 2 floating point instructions at one cycle, while Wattch uses issue width of 4, independent instruction type. Wattch uses a shared issue queue for both integer instructions and floating-point instructions, while Alpha uses separate queues. Wattch-Alpha conforms to Alpha architecture, therefore inherits these differences from Wattch.

	Wattch	Wattch-Alpha
Fetch width	4	4
Issue width	4	4(int), 2(float)
Commit width	11	11
Reorder Buffer size	32	80
Issue window	32	20(int), 15(float)
Load/store queue	64	32 (load) 32 (store)
Register file	32	160
Floating-point ALU	1 adder, 1 multiplier	1 adder, 1 multiplier
Integer ALU	4 adder, 4 multiplier	4 adder, 4 multiplier
L1 Data cache (nsize, bsize, asso)	512, 64, 2	512, 64, 2
Instruction Cache	512, 64, 2	512, 64, 2
Dtlb	1, 64, 128 (fully associative)	1, 64, 128 (fully associative)
Itlb	1, 32, 128 (fully associative)	1, 32, 128 (fully associative)

Table. 2. Configuration for Wattch-Alpha and Wattch.

2. Maximum Power consumption analysis

We first compare the power consumption of each structure when they are accessed based on power model only. Then we use execution-driven simulation to estimate runtime power consumption based on different conditional clocking scheme as we discussed in section 2.

Table.3 summarizes the maximum power consumption of each component on chip. We do not include the L2 Data Cache because it is not defined in Sim-Alpha. Using power model, the estimated power consumption for a 2M direct-mapped L2 cache is about 10 W. We do not include the comparison to real Alpha 21264 power dissipation statistics because our simulator does not model power consumption the memory management units and I/O logic, which contribute more than 10% of total power dissipation. In addition, the published statistics for 21264 [10] are very coarse-grained, and we cannot set up the direct mapping with the components that we simulate. The results are nevertheless quite consistent according to our observation.

	Wattch-Alpha	Wattch
Branch Predictor	1.58118 (2.14%)	1.52119 (2.59%)
Rename Logic	0.753685 (1.02%)	0.311073 (0.53%)
Instruction Window	4.27723 (5.79%)	2.05246 (3.5%)
Load/Store Queue	2.90389 (3.93%)	2.66736 (4.54%)
Register File	7.62154 (10.3%)	1.77124 (3.02%)
Result Bus	7.63545 (10.3%)	4.73024 (8.06%)
Total Clock	19.8863 (26.1%)	16.7163 (28.5%)
Int ALU	9.32026 (12.6%)	9.32026 (15.9%)
FP ALU	7.14052 (9.66%)	7.14052 (12.2%)
Instruction Cache	4.08059 (5.52%)	4.08059 (6.95%)
Itlb_power (W)	0.080420 (0.109%)	0.080420 (0.137%)
Data Cache	8.16117 (11%)	8.16117 (13.9%)
Dtlb_power (W)	0.16084 (0.218%)	0.16084 (0.274%)
Total Power Consumption	75.9282	60.7137

Table 3. Power consumption of each structure computed by Wattch and Wattch-Alpha

The major differences of power consumption estimated between Wattch and Wattch-Alpha present in renaming table, instruction window, register file and result bus. This is due the difference of architecture design as we mentioned in the section 4. Other components have similar or the same power consumption.

3. Run-time Power Analysis.

Conditional clocking strategy that turns off unused units/ports is now widely applied in modern microprocessor like Alpha 21264. The power consumption is then heavily dependent on types of application. In the table below, we summaries the average power consumption of a set of benchmark programs tested on Wattch-Alpha for different condition clocking options as we defined in section2.

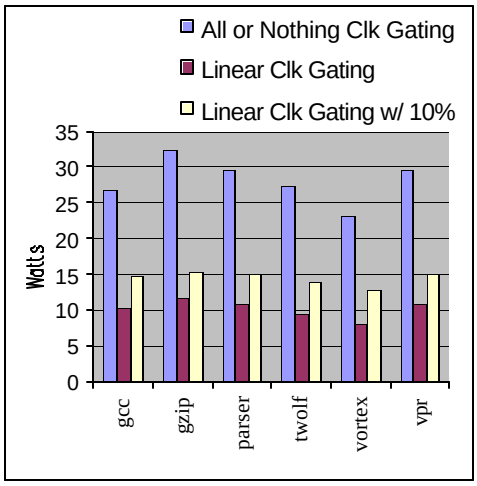

Fig.3 Power consumption with Conditional Clocking

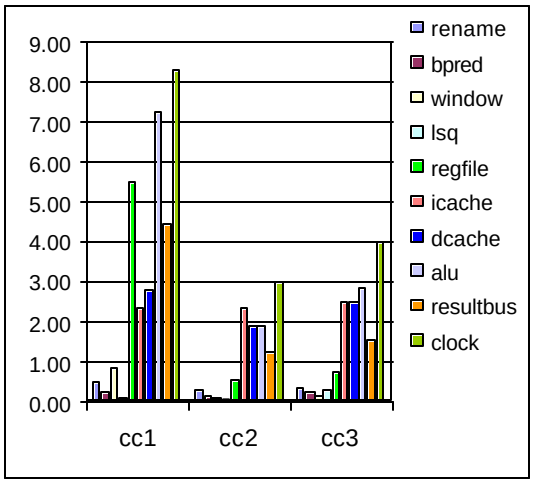

Fig.4 Power consumption breakdown run on gcc
(cc1: All or nothing, cc2: linear clk, cc3: linear clk w/ 10%)

As shown in Fig.3, using conditional clocking can significantly reduce power consumption. Even with option "all nor nothing clock gating", the power dissipation is less than 35 W in all cases, which is less than half than the maximal power consumption. It suggests that most components are idle most of the time. Conditional scaling is therefore, a power saving technique with great potential. In Fig.4 we show the power consumption for each component with conditional clocking scheme. Clock as always, is the top power consumer. Multi-ports structures, like register file, ALU, result bus consumes much less power in linear clocking scheme, which implies that conditional clocking technique should be applied on these structures.

As we have showed in the power simulation above, associative structures are the major contributors to the power consumption on chip. Therefore, a lot of research on power saving techniques focuses on the issue of reducing associability. One of such research that looks particularly interesting to us is FIFO-based micro architecture [7,9]. This technique has been applied to design of issue window: instead of using fully associative wake up logic and expensive selection logic on all instructions in the issue queue, a set of FIFO queues are used, each contains a set of dependent instructions. Only the instructions on head of the queue can be issued, and the result bus only needs to broadcast/wakeup the instructions on the heads of the queue. This will significantly reduce the circuit complexity and power consumption. The idea behind the FIFO-based microarchitecture is to exploit the natural dependence among the instructions. The key point is, dependent instructions cannot be issued in parallel. Limited instruction-level parallelism bounds the performances of superscalar machine, it also signifies the waste of excessive associative structures one the parallelism that will never present in practice. In the next section,

we first give a brief introduction of the FIFO-based microarchitecture, and discuss the power modeling of the FIFO-queue microarchitecture. Then, we propose a scheduling heuristics for instructions-queue allocation and analyze its performance. We have implemented the FIFO-based architecture in Wattch-Alpha. However, the current version is not stable with observed abnormal behaviors for NOP instructions. Instead of presenting the unreliable results we produced, we cited the results in [7] for reader's reference.

6. FIFO-based Microarchitecture

The FIFO-queue microarchitecture is shown in Fig 3. Instructions are steering to FIFO queues after map stage. At map stage, each instruction checks whether its source operands are dependent on other live instructions. If yes, the instruction should be put to the same FIFO queue as the instruction that it is dependent on. In order to store the dependence information between the instructions, for each architecture registers that contains non committed value, we need to keep queue tag of the instruction that produce the result for the registers in a table. This table is referred as SCR_FIFO table in [7]. For example, if an instruction that need operand *Ra*, and *Ra* is the destination of instruction that is in queue *q*, which can be found in SRC_FIFO table, then the instruction should also be put into queue *q*.

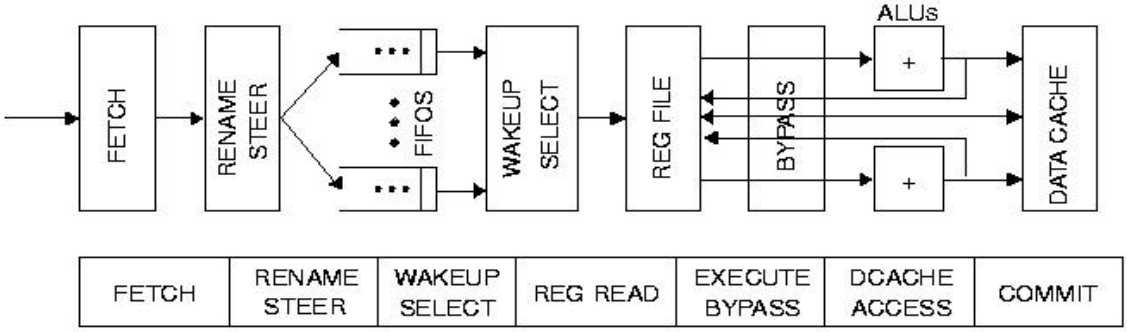

Fig.3. FIFO Queue Microarchitecture.

To implement FIFO-based microarchitecture, we need two additional structures: SCR_FIFO table and FIFO queues. At this point, we ignore the power consumption of steering logic, which allocate instructions to queues, whose complexity is heavily relied on the complexity of the scheduling algorithm it uses. The issue window now contains only the head of the queues. Both SCR_FIFO table and FIFO queues are implemented as RAM. SCR_FIFO is a RAM indexed by architecture register address, and each entry contains a queue tag of logN bits where N is the number of queue. The RAM for FIFO queue has queue length number of entries, and the number of bits in each entry is equal to the data width of the system. Each FIFO queue has only one port for read, and one port for write, compared to the 2*issue-width number of read ports, and issue-width of write port in a fully associative queue. Fully associative issue window can be treated as a special case of FIFO_based architecture in which queue length is one. In this case, SCR_FIFO and FIFO queues are not needed. In the power analysis we conduct below, we change the queue number (or queue length) while fixing the total number of entries in the issue window, i.e., keeping the product of number of queues and queue length constant, and compute the power consumption in each case. The results are given in Fig. 4.

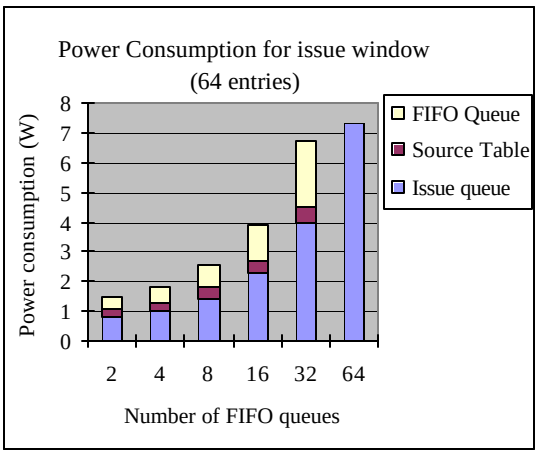

Fig 4. Power consumption for FIFO-based architecture (64 entries)

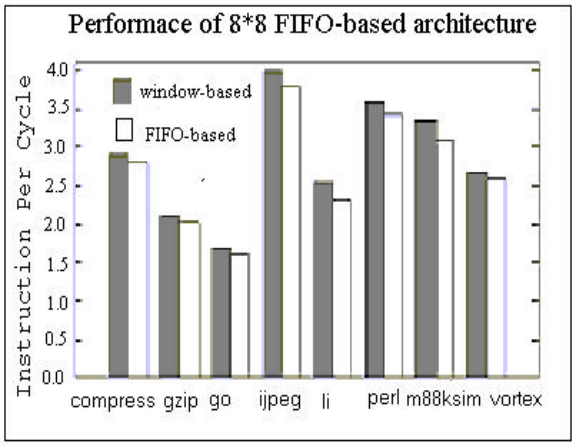

Fig 5. Performance for FIFO-based architecture (cited from [7])

As we observe from Fig. 4, the power consumption increases significantly as the number of queues increases. All three components, SCR_FIFO, FIFO queues and issue queues have increasing power consumption as the number of queues increase. For SCR_FIFO, the power increases as Log (Queue_size), each look like linear in Fig. 4 since the queue number grow exponentially. The power for FIFO queue increases at the ratio less than 2, because as the queue length decrease, the power consumption for each queue decreases. The power for issue queue grows proportional to number of FIFO queues.

Now let's turn to the performance of FIFO-based queue. The overall performance of a FIFO-based microarchitecture is highly dependent on the amount of ILP that can be extracted relative to the conventional microarchitecture. If the number of queues is smaller than the available ILP, then performance will be compromised. The performance is also highly dependent on the strategy to allocate instructions to queues. Following is a simple heuristics that we use:

> If both operands of an instruction are ready, then assign it to an empty queue if there is one. If there are no empty queues, assign it to the shortest queue.
>
> If the first operand is unavailable, assign the instruction to the same queue as the instruction that produces the operand. If the queue is full, check the second operand.
>
> If the second operand is unavailable, assign the instruction to the same queue as the instruction that produces its second operand. If the queue is full, select a nearby queue.

This heuristics works well if each instruction is dependent on the instruction right in front of it in the queue, and instructions belong to different queues have no dependence. However, instruction dependence graph is in general, a DAG. It is hard to partition instructions into queues to minimize the parallelism within a queue while maximize the parallelism among the queues. We present two cases, as shown in Figure 3, in which FIFO queue design degrade performance. In the first case, instruction 2, 3, 4, 5 are dependent on instruction 1, while they are independent of each other. They are steered into the same queue, waiting for the results from instruction 1. Once the first instruction is finished, instructions 2, 3, 4, 5 can be executed in parallel, theoretically. However, because they are in a same FIFO queue, they are executed sequentially, even if there are empty queues available, resulting in lower ILP. In case 2, instruction 5 is put in the same FIFO queue as instruction 1. When instruction 1 is finished, instruction 5 becomes the head of queue. However, instruction 5 is still waiting for its second operands, blocking instruction 6. It is an inherent problem due to the inconsistency of FIFO-based model and DAG. The scheduling problem of this type is NP-hard problem, and there is no easy fix. Moreover, scheduling algorithm for this problem should use as less circuit as possible, because our

motivation is to reduce the complexity of the system. Based on above reasons, we decide to use the above simple heuristic despite of its defects. To avoid the situations discussed in Fig, we limit the length of the queue, which partly alleviate the problem. The hope is, in practice, when the number of queues available is about the same or higher level of ILP in the program, the above situations is unlikely to happen.

At this point, our implementation of FIFO-based architecture in Wattch still contains

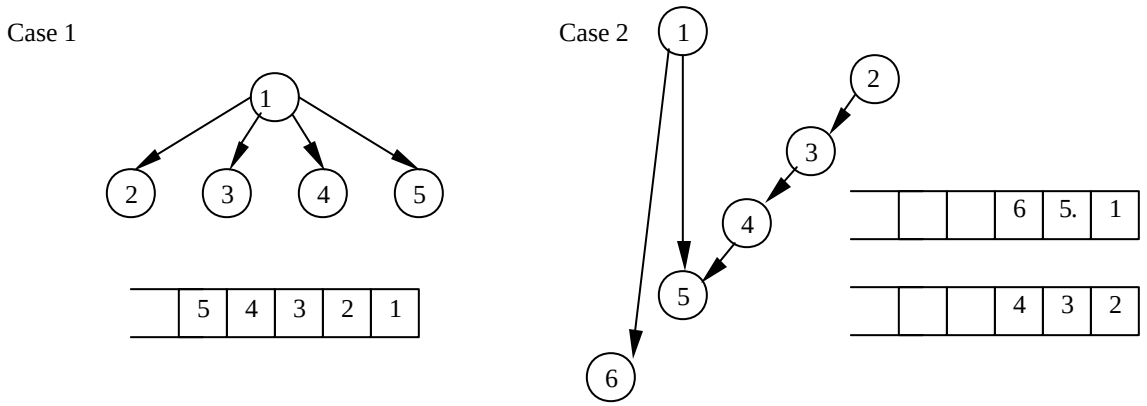

Figure 6: Two cases in which FIFO-based architecture has poor performance

bugs caused by NOP instructions. Within the time constraint, we cannot produce reliable experimental results. Fig.5 presents the results given in [7]. Compared to fully associatively issue window of the equal entries, the FIFO-based architecture with 8 FIFO queues, each with 8 entries has cycle count numbers within 5% for five of the seven benchmarks, and the worst performance degradation is 8.7%.

Discussion

In this project, we have successfully developed Wattch-Alpha, version of Wattch that is portable to Sim-Alpha with justified modification. The experimental data show that the simulated power dissipation of Wattch-Alpha compared to Wattch fall within the expected scope, considering their architecture level differences. Compared with published Alpha 21264 data, the estimated power consumption of Wattch-Alpha is 75 W, which is pretty close to 72 W in [10]. Power model in Wattch provides a general framework for power analysis and we find it a convenient tool to analysis the power consumption of a new architecture design FIFO-based architecture.

The FIFO-based issue window design we studied in this project provides some hint of reducing power consumption without hampering the processor's performance too much. There's a considerable amount of energy spent on exploiting IPC, and since the marginal utility is decreasing with the energy spent on finding IPC, there will be a point, which varies from architecture to architecture, that the energy spent on looking for more IPC won't pay back as much as it should be. Either higher IPC needs to be found, or the energy should be saved for better usage.

Conditional clocking is shown to be an effective way of saving energy. Current conditional clocking technology applies at functional unit level. With the increasing number of homogeneous architectures emerging, like Computational Cache and Nano-fabric architecture, conditional clocking can be used at a much finer level, i.e. certain block of fine-grained units can be turned off when not in use. The function of turning units on or off can be implemented by

hardware. It can also be implemented by software, which hides the detail of the architecture from upper software, including operating system and the application.

Reference

[1]. Mentor Graphics Corporation, 1999

[2]. Synopsys Corporation. Powermill Data Sheet, 1999

[3]. N. Vijaykrishnam, M. Kandemir, M.J.Irwin, H,Y.Kim, and W.Ye. *Energy-driven integraded hardware-software optimizations using SimplePower.* In Proc. The International Symposium on Computer Architecture, Vancouver, British Columbia, June 2000. 20.

[4]. The SimpleScalar-Arm Power Modeling Project. http://www.eecs.umich.edu/~tnm/power/

[5]. B. Bishop, T. Kelliher, and M. Irwin. *The Design of a Register Renaming Unit.* In Proc. Of Great Lakes Symposium on VLSI, 1999.

[6]. M. Borah, R.Owens, and M.Irwin. *Transistpr sizing for low power CMOS circuits.* IEEE Transactions on Computer Aided Design of Integrated Circuits and System.s 15(6):665-71, 1996.

[7]. S. Palacharla, N. Jouppi, and J. Smith. *Quantifying the Complexity of SuperScalar Processors.* In Proc. Of the 24th Int'l Symp. On Computer Architecture, 1997

[8]. R. E. Kesseler, E.J. McLellan, and D.A. Webb. *The Alpha 21264 Microprocessor Architecture.* Compaq Computer Corporation.

[9] S. Palacharla, N.P. Jouppi, and J.E. Smith, *Complexity Effective Superscalar Processors*, in Proc of the 24th. Int. Symp. on Comp. Architecture, 1997, pp 1-13.

[10]M. K. Gowan, L. L. Biro, D. B. Jackson, *Power Considerations in the Design of the Alpha 21264 Microprocessor* 35th Design Automation Conference, June 1998.