View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Crossref

Saravanan *et al. Human-centric Computing and Information Sciences* (2015) 5:2
DOI 10.1186/s13673-014-0016-8

Human-centric Computing and Information Sciences
a SpringerOpen Journal

RESEARCH Open Access

An optimizing pipeline stall reduction algorithm for power and performance on multi-core CPUs

Vijayalakshmi Saravanan[1*], Kothari Dwarkadas Pralhaddas[1], Dwarkadas Pralhaddas Kothari[2] and Isaac Woungang[1]

*Correspondence:
vsaravan@rnet.ryerson.ca
[1]WINCORE Lab, Ryerson University, Toronto, Canada
Full list of author information is available at the end of the article

Abstract

The power-performance trade-off is one of the major considerations in micro-architecture design. Pipelined architecture has brought a radical change in the design to capitalize on the parallel operation of various functional blocks involved in the instruction execution process, which is widely used in all modern processors. Pipeline introduces the instruction level parallelism (ILP) because of the potential overlap of instructions, and it does have drawbacks in the form of hazards, which is a result of data dependencies and resource conflicts. To overcome these hazards, stalls were introduced, which are basically delayed execution of instructions to diffuse the problematic situation. Out-of-order (OOO) execution is a ramification of the stall approach since it executes the instruction in an order governed by the availability of the input data rather than by their original order in the program. This paper presents a new algorithm called Left-Right (LR) for reducing stalls in pipelined processors. This algorithm is built by combining the traditional in-order and the out-of-order (OOO) instruction execution, resulting in the best of both approaches. As instruction input, we take the Tomasulo's algorithm for scheduling out-of-order and the in-order instruction execution and we compare the proposed algorithm's efficiency against both in terms of power-performance gain. Experimental simulations are conducted using Sim-Panalyzer, an instruction level simulator, showing that our proposed algorithm optimizes the power-performance with an effective increase of 30% in terms of energy consumption benefits compared to the Tomasulo's algorithm and 3% compared to the in-order algorithm.

Keywords: Instruction pipeline; Stall reduction; Optimizing algorithm

Introduction

Instruction pipeline is extensively used in modern processors in order to achieve instruction level parallelism in pipelined processor architectures [1]. In a conventional pipelined processor, there are 5- pipe stages, namely FETCH (FE), DECODE (DE), EXECUTE (EXE), MEMORY (MEM) and WRITE-BACK (WB). In the first stage, the instruction is read from the memory, loaded into the register, then the decoding of an instruction takes place in the succeeding stage. In the third stage, the execution of an instruction is carried out and in the fourth stage, the desired value is written into the memory; and finally,

Springer

© 2015 Saravanan et al.; licensee Springer. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

the computed value is written into a register file. For example, in pipelined processors, if there is any dependency between two consecutive instructions, then the instruction in the decode stage will not be valid. The Tomasulo hardware algorithm is used to overcome this situation. Typically, it is a hardware dynamic scheduling algorithm, in which a separate hardware unit (so-called forwarding) is added to manage the sequential instructions that would normally stall (due to certain dependencies) and execute non-sequentially (This is also referred to as out-of-order execution). Due to data forwarding, there is at least a clock cycle delay and the stall is inserted in a pipeline. These no-operation (NOP) or stalls are used to eliminate the hazards in the pipeline. The NOP instructions contribute to the overall dynamic power consumption of a pipelined processor by generating a number of unnecessary transitions. Our main goal is to minimize such stalls which in turn increases the CPU throughput, thus saves the power consumption.
Generally, the time taken by computing devices is determined by the following factors:

- Processor cycle time.
- Number of instructions required to perform certain task.
- Number of cycles required to complete an instruction.

The system performance can be enhanced by reducing one or more of these factors. Pipelining does just that by dividing the workload into various sub units and by assigning a processing time to each unit, thereby reducing the waiting time period which occurs if the sequential execution was adopted. Various approaches can be to increase the pipeline stages, and various strategies can be used to reduce the stalls caused by the pipeline hazards. To solve this hazard, one can use a large and faster buffer to fetch the instructions and perform an out of order execution. Though, this method increases the hardware complexity cost. It also reduces the branch penalty by re-arranging the instructions to fill the stalls due to branching instruction. But, this requires the use of a suitable scheduling algorithm for the instruction [2]. There is an ongoing research on variable pipeline stages, where it is advocated that processor's pipeline stages can be varied within a certain range. In this type of processors, one can vary the workload and power consumption as per our requirement.

Our proposed work on the analysis of stall reduction of pipelined processors is motivated by the following facts: (1) How to identify the power consumption of the instruction execution in a pipelined processor, i.e. does the power consumption of a instruction execution caused by the number of instructions or the type of executions (such as in-order or out-of-order execution) and why? (2) How to balance both the power and performance in instruction execution.

Recently, a new trend has been established by multi-threaded and multi-core processors. The demand for these processors are due to the inability of the conventional processor to meet higher performance memory-intensive workloads, which in turn may lead to high cache miss rates. In addition, a conventional processor cannot utilize the pipeline effectively, in the sense that a high percentage of the processor resources are wasted. The state-of-the-art architectural methods and algorithms such as pre-fetching and out-of-order execution are not suitable enough for these types of pipelined processors.

In this paper, an alternative strategic algorithm is proposed, in which the instructions are divided into a number of stages, then sorted and executed simultaneously, thereby increasing the throughput. In other words, our algorithm performs a combination of in-order and out-of-order execution for sequential instructions. Our algorithm is then compared against two traditional benchmark algorithms, namely the in-order algorithm and the Tomasulo algorithm. We have also pointed out that just increasing pipeline stages will not always be beneficial to us.

The paper is organized as follows: Section 'Related work' presentssome related work. In Section 'Proposed algorithm', our proposed algorithm for effective stall reduction in pipeline design on multiprocessors is presented. In Section 'Comparison of LR vs. Tomasulo algorithm', the simulation results are presented. Finally, Section 'Conclusions' concludes our work.

Related work

The length of the pipeline has an impact on the performance of a microprocessor. Two architectural parameters that can affect the optimal pipeline length are the degree of instruction level parallelism and the pipeline stalls [3]. During pipeline stalls, the NOP instructions are executed, which are similar to test instructions. The TIS tests different parts of the processor and detects stuck-at faults [4,5].

Wide Single Instruction, Multiple Thread architectures often require static allocation of thread groups, executed in lockstep. Applications requiring complex control flow often result in low processor efficiency due to the length and quantity of the control paths. Theglobal rendering algorithms are an example. To improve the processor's utilization, a SIMT architecture is introduced, which allows for threads to be created dynamically at runtime [6].

Branch divergence has a significant impact on the performance of GPU programs. Current GPUs feature multiprocessors with SIMT architecture, which create, schedule, and execute the threads in groups (so-called wraps). The threads in a wrap execute the same code path in lockstep, which can potentially lead to a large amount of wasted cycles for a divergent control ow. Techniques to eliminate wasted cycles caused by branch and termination divergence have been proposed in [7]. Two novel software-based optimizations, called iterative delaying and branch distribution were proposed in [8], aiming at reducing the branch divergence.

In order to ensure consistency and performance in scalable multiprocessors, cache coherence is an important factor. It is advocated that hardware protocols are currently better than software protocols but are more costly to implement. Due to improvements on compiler technologies, the focus is now placed more on developing efficient software protocols [9]. For this reason, an algorithm for buffer cache management with pre-fetching was proposed in [10]. The buffer cache contains two units, namely, the main cache unit and the prefetch unit; and blocks are fetched according to the one block lookahead prefetch principle. The processor cycle times are currently much faster than the memory cycle times, and the trend has been for this gap to increase over time. In [11,12], new types of prediction cache were introduced, which combine the features of pre-fetching and victim caching. In [13], an evaluation of the full system performance using several different power/performance sensitive cache configurations was proposed.

In [14,15], a pre-fetch based disk buffer management algorithm (so-called W2R) was proposed. In [16], the instruction buffering as a power saving technique for signal and multimedia processing applications was introduced. In [17], another buffer management technique called dynamic voltage scaling was introduced as one of the most efficient ways to reduce the power consumption because of its quadratic effect. Essentially, the micro-architectural-driven dynamic voltage scaling identifies program regions where the CPU can be slowed down with negligible performance loss. In [18], the run-time behaviour exhibited by common applications with active periods alternated with stall periods due to cache misses, was exploited to reduce the dynamic component of the power consumption using a selective voltage scaling technique.

In [19], a branch prediction technique was proposed for increasing the instructions per cycle. Indeed, a large amount of unnecessary work is usually due to the selection of wrong-path instructions entering the pipeline because of branch mis-prediction. A hardware mechanism called pipeline gating is employed to control the rampant speculation in the pipeline. Based on the Shannon expansion, one can partition a given circuit into two sub-circuits in a way that the number of different outputs of both sub-circuits are reduced, and then encode the output of both sub-circuits to minimize the Hamming distance for transitions with a high switching probability [20].

In [21], file Pre-fetching has been used as an efficient technique for improving the file access performance. In [22], a comprehensive framework that simultaneously evaluates the tradeoffs of energy dissipations of software and hardware such as caches and main memory was presented. As a follow up, in [23], an architecture and a prototype implementation of a single chip, fully programmable Ray Processing Unit (RPU), was presented.

In this paper, our aim is to further reduce the power dissipation by reducing the execution of the stall instruction passes through the pipe stages using our proposed algorithm. Therefore, our algorithm aims at reducing the unnecessary waiting time of the instruction execution and clock cycles, which in turn will maximize the CPU performance and save some amount of energy consumption.

Proposed algorithm

Performance improvement and power reduction are two major issues that are faced by computer architects, and various methods and algorithms have been proposed to optimize the performance and power. To add on these, other methodsrely on reducing the pipeline hazards and increasing the efficiency of a processor. Processors have evolved into two main categories, namely, in-order execution and out-of-order execution.

In-order execution: In this method, instructions are fetched, executed and completed in a compiler generated order. Instructions are scheduled statically, and if one stall occurs, the rest all are stalled. The Alpha and Intel Atom processors in-order models have been implemented with peak performance. However, complex designs are required for the integration of peak capacity and increase in clock frequency.

```
example of in-order instruction execution:
lw $3, 100($4) in execution, cache miss
add $2, $3, $4 waits until the miss is satisfied
sub $5, $6, $7 waits for the add
```

Out-of-order execution: In the previous method, data dependencies and latencies in functional units can cause reduction in the performance of the processor. In order to overcome this issue, we have uses an out-of-order (OOO) method which is the traditional way to increase the efficiency of pipelined processors by maximizing the instruction issued by every cycle [24]. But, this technique is very costly in terms of its implementation. Most of the high level processors (such as DEC and HP) execute the instructions in out-of-order. In this method, the instructions are fetched in a compiler generated order and the execution of the instruction takes place in pipeline as one which is not dependent on the current instruction, i.e. independent instructions are executed in some other order. The instructions are dynamically scheduled and the completion of instruction may be in in-order or out-of-order.

```
Example of out-of-order instruction execution:
lw $3, 100($4) in execution, cache miss
sub $5, $6, $7 can execute during the cache miss
add $2, $3, $4 waits until the miss is satisfied
```

Tomasulo's algorithm: As a dynamic scheduling algorithm, it uses a hardware-based mechanism to remove the stalls at the runtime. It allows sequential instructions that would normally be stalled due to certain dependencies to execute non-sequentially (out-of-order execution). It also utilizes the concept of register renaming, and resolves Write-after-Write (WAW), Read-after-Write (RAW) and Write-after-Read (WAR) computer architecture hazards by register renaming, which allows the continual issuing of instructions. This algorithm also uses a common data bus (CDB) on which the computed values are broadcasted to all the reservation stations that may need it. This allows for improved parallel execution of instructions, which may otherwise stall. The Tomasulo algorithm was chosen because its order of instruction execution is nearly equivalent to that of our proposed algorithm. Both algorithms are scheduled statically at the micro-architecture level.

Proposed algorithm (LR(Left-Right)): The above mentioned methods and algorithms have their own merits and demerits while executing an instruction in a pipelined processors. Instead of using some other methods to reduce the power consumption, we have

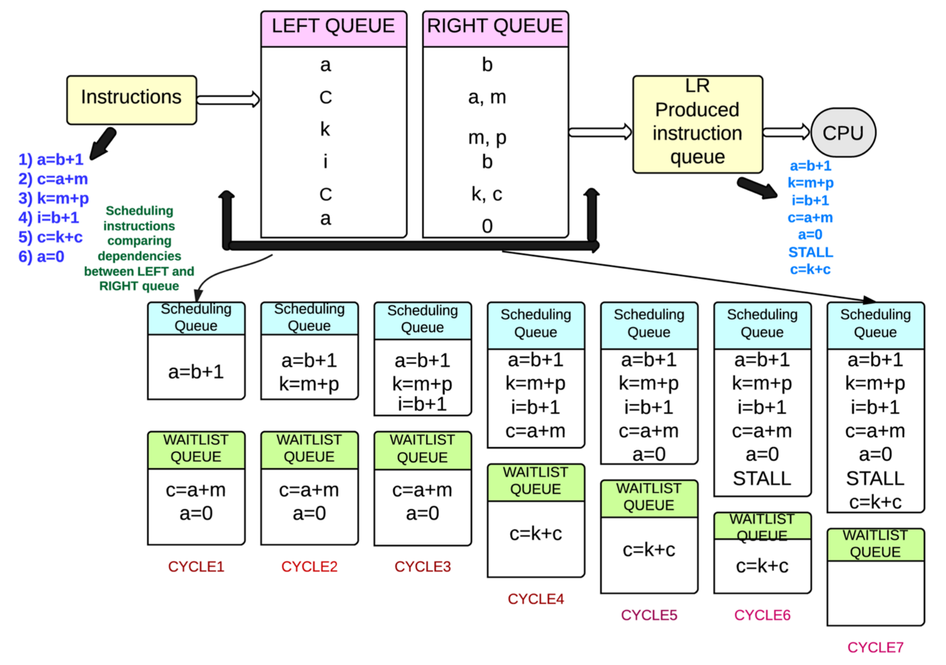

Figure 1 Execution of LR (Left-Right) Algorithm.

proposed an algorithm which performs the stall reduction in a Left-Right (LR) manner, in sequential instruction execution as shown in Figure 1. Our algorithm introduces a hybrid order of instruction execution in order to reduce the power dissipationl. More precisely, it executes the instructions serially as in-order execution until a stall condition is encountered, and thereafter, it uses of concept of out-of-order execution to replace the stall with an independent instruction. Thus, LR increases the throughput by executing independent instructions while the lengthy instructions are still executed in other functional units or the registers are involved in an ongoing process. LR also prevents the hazards that might occur during the instruction execution. The instructions are scheduled statically at compile time as shown in Figure 2. In our proposed approach, if a buffer in presence can hold a certain number of sequential instructions, our algorithm will generate a sequence in which the instructions should be executed to reduce the number of stalls while maximizing the throughput of a processor. It is assumed that all the instructions are in the form of op-code source destination format.

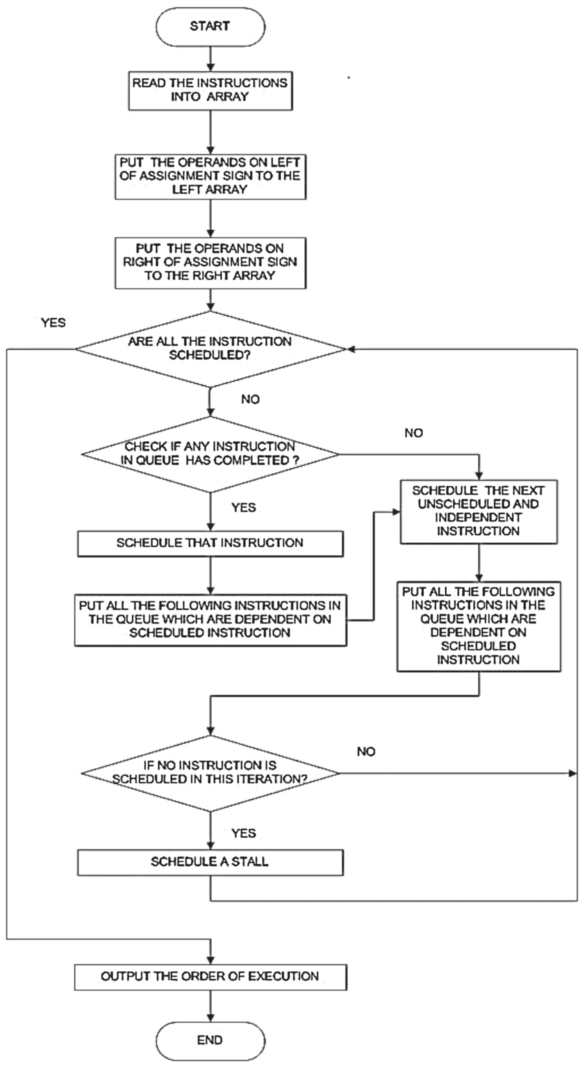
START
READ THE INSTRUCTIONS INTO ARRAY
PUT THE OPERANDS ON LEFT OF ASSIGNMENT SIGN TO THE LEFT ARRAY
PUT THE OPERANDS ON RIGHT OF ASSIGNMENT SIGN TO THE RIGHT ARRAY
YES
ARE ALL THE INSTRUCTION SCHEDULED?
NO
CHECK IF ANY INSTRUCTION IN QUEUE HAS COMPLETED ?
NO
YES
SCHEDULE THAT INSTRUCTION
PUT ALL THE FOLLOWING INSTRUCTIONS IN THE QUEUE WHICH ARE DEPENDENT ON SCHEDULED INSTRUCTION
SCHEDULE THE NEXT UNSCHEDULED AND INDEPENDENT INSTRUCTION
PUT ALL THE FOLLOWING INSTRUCTIONS IN THE QUEUE WHICH ARE DEPENDENT ON SCHEDULED INSTRUCTION
IF NO INSTRUCTION IS SCHEDULED IN THIS ITERATION?
NO
YES
SCHEDULE A STALL
OUTPUT THE ORDER OF EXECUTION
END

Figure 2 Flow chart of Proposed Algorithm.

Comparison of LR vs. Tomasulo algorithm

In this section, the performance and power gain of the LR and the Tomasulo algorithms are compared.

Simulation and power-performance evaluation

As our baseline configuration, we use an Intel core i5 dual core processor with 2.40GHZ clock frequency, and 64-bit operating system. We also use the Sim-Panalyzer simulator [25]. The LR, in-order, and Tomasulo algorithms are developed as C programs. These C programs were compiled using arm-linux-gcc in order to obtain the object files for each of them, on an ARM microprocessor model.

At the early stage of the processor design, various levels of simulators can be used to estimate the power and performance such as transistor level, system level, instruction level, and micro-architecture level simulators. In transistor level simulators, one can estimate the voltage and current behaviour over time. This type of simulators are used for integrated circuit design, and not suitable for large programs. On the other hand, micro-architecture level simulators provide the power estimation across cycles and these are used in modern processors. Our work is similar to this kind of simulator because our objective is to evaluate the power-performance behaviour of a micro-architecture level design abstraction. Though, a literature survey suggests several power estimation tools such as CACTI, WATTCH [26], and we have choose the Sim-Panalyzer [25] since it provides an accurate power modelling by taking into account both the leakage and dynamic power dissipation.

The actual instruction execution of our proposed algorithm against existing ones is shown in Algorithms 1 and 2. In the LR algorithm, an instruction is executed serially in-order until a stall occurs, and thereafter the out-of-order execution technique comes to play to replace the stall with an independent instruction stage. Therefore, in most cases, our proposed algorithm takes less cycle of operation and less cycle time compared to existing algorithms as shwon in algorithm [2]. The comparison of our proposed algorithm against the Tomasulo algorithm and the in-orderalgorithm is shown in Table 1. The next section focusses on the power-performance efficiency of our proposed algorithm.

Algorithm 1 Pseudo code of the proposed Left-Right (LR) algorithm.

```
1: Read the instruction into an array;
2: Separate the left operands and right operands of assignment sign into left array and
   right array respectively;
3: Check if all the instructions have been scheduled. If yes goto step 7 else go to 4a.
4: 4a: Check if the instruction in the queue has completed two stalls. If yes go to
   step 4b else go to step 5a; 4b: Schedule that instruction and send all the dependent
   instruction into the queue. Go to step 5a.
5: 5a: Schedule the next independent and unscheduled instruction; 5b: Send all the
   dependent instruction into the queue. Go to step 6.
6: If no instruction was scheduled in the last iteration then schedule a stall and go to
   step 3.
7: Output the order of execution.
```

Algorithm 2 Actual Instruction execution of proposed algorithm LR(Left-Right) vs. In-order, Tomasulo

```
1: Instructions: 1) a=b + 1; 2) c=a + m; 3) k=m + p; 4) i=b + 1; 5) c=k + c; 6) a=0;
2: LEFT Operand: a, c, k, i, c, a
3: RIGHT Operand: [b,1] [a,m] [m,p] [b,1] [k,c] [0,0]
4: Cycle Operation: 1 2 3 4 5 6
5: First Step: Schedule 1st instruction. And put 2nd and 6th instruction in queue. Then,
   schedule a 3rd instruction and 4th instruction.
6: Then, 2nd instruction from queue, schedule 6th instruction, WAIT for a cycle then
   schedule 5th instruction.
7: This will give rise to the following sequence for proposed algorithm (LR): 1 3 4 2 6
   Stall 5
8: In-order: 1 Stall 2 3 4 Stall 5 6
9: Tomasulo: 1 3 4 2 6 Stall 5. Though, tomasulo takes same cycle time as LR, due to
   hardware unit high power dissipation to perform the same operation than LR.
```

Performance evaluation

In general, computer architects use simulation as a primary tool to evaluate the computer's performance. In this setting, instructions per cycle (IPC) represents a performance metric that can be considered, and it is well-known [27] that an increase in IPC generally yields a good performance of the system. The use of instructions per cycle (IPC) to analyze the performance of a system is challenged at least for the multi-threaded workloads running on multiprocessors. In [27], it was reported that work-related metrics (e.g. time per transaction) are the most accurate and reliable way to estimate the multiprocessor workload performance. We have also proved that our algorithm produces less IPC compared to that generated by the Tomasulo algorithm (see Figure 3(b)). According to this result, work-related metrics such as *time per program* and *time per workloads* are the most accurate and reliable methods to calculate the performance of the system. The time

Table 1 Comparison of algorithms

In-order execution	Tomasulo's algorithm	Proposed algorithm (LR)
Static-scheduling	Hardware dynamic-scheduling	Static-scheduling
Compiler tries to reorder the instructions during the compilation time in order to reduce the pipeline stalls	The dynamic scheduling of the hardware tries to rearrange the instructions during run-time to reduce the pipeline stalls	Compilation time instruction execution
Uses less hardware	More hardware unit added	Use more powerful algorithmic techniques (sorting)
Sequential-order	Register-renaming is used to reduce the stall	Sorting takes place first, then execution of an instruction
Bottom-up approach	Re-ordering of CPU instructions	Hybrid order of an in-order and OOO
For ex: char x; //read x, starts on cycle 1 & completes on cycle 2; int a= 10 + 20; // assignment to a, starts on cycle 3 & completes on cycle 4; print char x; // starts on cycle 5 & completes on cycle 6;	char x; // read x, starts on cycle 1 & completes on cycle 2; int a= 10 + 20; // assignment to a, starts on cycle 2 & completes on cycle 3; print char x; // starts on cycle 3 & completes on cycle 4;	char x; // read x, starts on cycle 1 & completes on cycle 2; int a= 10 + 20; // assignment to a, starts on cycle 2 & completes on cycle 3; print char x; // starts on cycle 3 & completes on cycle 4; Due to hardware unit, more power dissipation

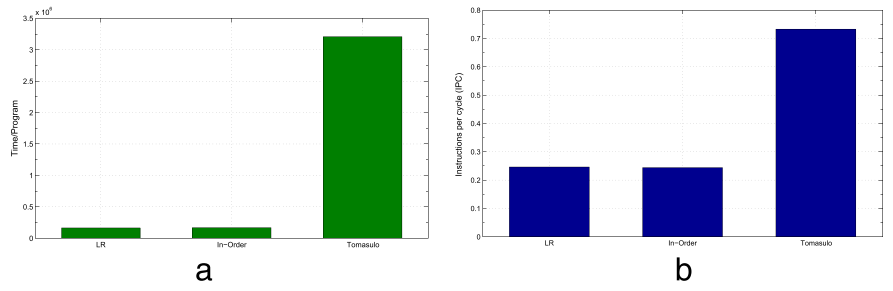

Figure 3 Performance comparison of LR, In-order vs. Tomasulo algorithm. a) Time/Program, **b)** Instructions per cycle.

per program is calculated as shown in Eq. (1) and Eq. (2). We use time per program and IPC as performance metrics.

$$Time/Program = Instructions/Program \times Cycles/Instruction \times Time/Cycle, \qquad (1)$$

$$Time/Program = CP \times CPI \times IPP$$

where TP- time per program, CP- clock period, CPI- Cycles per instruction and IPP- instructions executed per program. We have executed our program on the same machine, therefore the clock period will be the same. Hence, Eq. (1) becomes,

$$Throughput(\%) = 100 - \left[(CPI \times IPP)_{LR}/(CPI \times IPP)_{In-order}\right] \times 100 = 98 \qquad (2)$$

$$Throughput(\%) = 100 - [(CPI \times IPP)_{LR}/(CPI \times IPP)_{Tomasulo}] \times 100 = 95 \qquad (3)$$

By using Eq. 1, we calculated the time per program for the proposed LR algorithm, and its efficiency iscompared against the traditional in-order and Tomasulo's algorithm as shown in Table 2.

Power consumption evaluation

In simulation-based power evaluation methods, the system is integrated with various components such ALU, level-1 I-cache, D-cache, irf (register files), and clock. The energy consumption of a program is estimated as the sum of all these components as shown in Table 3 and the mean power dissipation results from Sim-Panalyzer for the same experiment are shown in Table 4.

Result analysis and discussions

To analyse the efficiency of our proposed algorithm, we have simulated both algorithms on the Sim-Panalyzer and obtained the average power dissipation of the ALU, level-1 instruction (il1) and data (dl1) caches as well as the internal register file (irf) and the clock

Table 2 Performance estimation of LR vs. in-order

Metrics	LR	In-order	Tomasulo
Instructions per cycle (IPC)	0.2462	0.2442	0.7327
Clocks per instruction (CPI)	4.061	4.09	1.3648
Simulation speed (inst/sec)	785	606	98021.9
Total number of instructions executed	40516	41004	2350639
Time/Program	164535.476	167706.36	3207982.549

Table 3 Average power dissipation for LR, in-order vs. Tomasulo's algorithm

Component	Simulation parameters	LR	In-order	Tomasulo
ALU	**alu avg power # Avg power for alu**	**0.0001**	**0.0001**	**0.0003**
dl1	dl1.avgswitching #dl1 avg in switching power dissipation	0.0060	0.0079	0.0172
	dl1.avginternal #dl1 avg internal power dissipation	0.0192	0.0199	0.0431
	dl1.avgleakage #dl1 avg leakage power dissipation	0.0029	0.0029	0.0029
	dl1.avgpdissipation #dl1 avg power dissipation	0.223	0.2208	0.2459
il1	il1.avgswitching #il1 avg in switching power dissipation	0.0343	0.0338	0.0950
	il1.avginternal #il1 avg internal power dissipation	0.0861	0.0849	0.2384
	il1.avgleakage #il1 avg leakage power dissipation	0.0029	0.0029	0.0029
	il1.avgpdissipation #il1 avg power dissipation	0.4274	0.4292	0.4525
irf	irf.avgswitching # irf avg in switching power dissipation	0.0063	0.0066	0.0162
	irf.avginternal #irf avg internal power dissipation	0.0085	0.0089	0.0218
	irf.avgleakage #irf avg leakage power dissipation	0.0001	0.0001	0.0001
	irf.avgpdissipation #irf avg power dissipation	0.041	0.0414	0.0407
Clock	clock.avgleakage #clock avg leakage power dissipation	191.13	191.13	191.13

power dissipation. We have plot the energy consumptions of the different components of both algorithms as shown in Figure 4.

It can be observed that with Tomasulo's algorithm, the absolute power dissipation differs significantly between LR and in-order. In terms of ALU power dissipation, it can be observed there is not much improvement in power-performance. But on comparing the results for dl1 and il1, it can be noticed that there is a significant difference in power dissipation in the level-1 data and instruction caches. In both dl1 and il1, the average switching power dissipation (resp. the average internal power dissipation) show up to 60% less power dissipation in LR than in the Tomasulo algorithm. Also, the power dissipation generated by LR is 2.5% less compared to that generated by the Tomasulo algorithm.

From Eq. 2 and Table 2, it can be concluded that LR performs 95% better than Tomasulo. Figure 3a and 3b, Table 3 depict the IPC & time per program of both algorithms. The instruction execution efficiency of the Tomasulo algorithm is 1% and the data efficiency is

Table 4 Total power dissipation of LR, In-order vs. Tomasulo algorithm

Component	LR algorithm	In-order algorithm	Tomasulo	(%) Improvement (Tomasulo)
ALU	0.0001	0.0001	0.0003	33.3
il1	0.5507	0.5508	0.3091	21.1
dl1	0.2527	0.2515	0.7888	33.5
irf	0.0559	0.057	0.0788	33.5

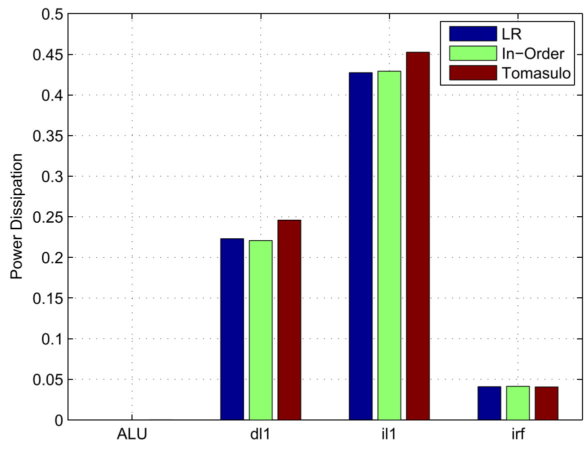

Figure 4 Comparison of power dissipation LR, In-order vs. Tomasulo's algorithm.

2% higher than that of the LR algorithm. In terms of overall power-performance benefits, our proposed LR algorithm outperforms the Tomasulo algorithm. In our experiment, it was also observed that the fraction of clock power dissipation is almost the same for both algorithms. This significant increase of clock power in Sim-Panalyzer is mostly due to the fact that it is dependent on the dynamic power consumption.

Discussions

With this simulator, we are able to obtain power-performance of various below mentioned components , and compared our results.

1. **ALU:** As shown in Table 1, the average power dissipation in ALU indicates the usage of ALU during the simulation of both algorithms. The maximum power dissipation is noticed to be equal for both the algorithm but the average power dissipation in LR shows a 66% improvement against Tomasulo. This simply states that for the processing LR requires less computation as compared to Tomasulo to order the instructions.
2. **DL1:** dl1 represents the level -1 data caches and in the experiment, LR shows an overall improvement of 21% as compared to Tomasulo in the average power dissipation in dl1. This exemplifies that the usage of cache and the cache hit ratio is improved in LR then in the Tomasulo.
3. **IL1:** il1 represents the level -1 instruction caches and in the experiment LR shows an overall improvement of 33% as compared to Tomasulo in the average power dissipation in il1. Hence, we can deduce that the cache used for holding instructions performs better while processing for LR as compared to Tomasulo.
4. **IRF:** IRF indicates the usage of the internal register file. The average power dissipation in LR is 33% less than Tomasulo's. IRF indicates that the register's usage is less in LR and hence the power consumption is less than the Tomasulo.

Overal, thel improvement in average power dissipation of LR comes out to be 30% better than that of the Tomasulo algorithm. Also, a slight performance-power improvement of LR against in-order is also achieved. Hence, it can be concluded that our algorithm increases the throughput of the pipelined processor by reducing the stalls, and the power-performance of our algorithm is better than that of the Tomasulo's algorithm.

Conclusions

We have presented a stall reduction algorithm for optimizing the power-performance in pipelined processors. Our algorithmic technique is based on the hybrid order of instruction execution and it operates at a higher level of abstraction than more commonly used hardware level algorithms in instruction level power-performance estimation do. Simulation results have been conducted to validate the effectiveness of our proposed algorithm, revealing the following findings: (1) Our proposed algorithm is able to optimize the stall reductions during instruction scheduling in the pipelined processor; (2) It can also help preventing the data hazards that might occur; (3) Compared to the Tomasulo algorithm chosen as benchmark, it can achieve up to 30% of power and 95% of performance improvement on simulation in a pipelined processor; (4) the performance-power exhibited by in-order execution are relatively low compared to that performed by our algorithm; and (5) Our algorithm is statically scheduled, and it performs better in terms of power and performance than the existing stall reduction algorithm. As futre work, the proposed algorithm can further be enhanced by using more advanced sorting techniques, for instance, techniques that can help overlapping the instructions, making them more data dependable.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
All authors read and approved the final manuscript.

Author details
[1]WINCORE Lab, Ryerson University, Toronto, Canada. [2]I.I.T. Delhi, India.

Received: 11 February 2014 Accepted: 28 August 2014
Published online: 29 January 2015

References

1. Kogge PM (1981) The Architecture of pipelined computers. McGraw-Hill advanced computer science series, Hemisphere, Washington, New York, Paris, Includes index
2. Johnson WM (1989) Super-scalar processor design. Technical report
3. Hartstein A, Puzak TR (2002) The optimum pipeline depth for a microprocessor. SIGARCH Comput Archit News 30(2):7–13
4. Shamshiri S, Esmaeilzadeh H, Navabi Z (2005) Instruction-level test methodology for cpu core self-testing. ACM Trans Des Autom Electron Syst 10(4):673–689
5. Patterson DA, Hennessy JL (2006) In praise of computer architecture: a quantitative approach. Number 704. Morgan Kaufmann
6. Steffen M, Zambreno J (2010) Improving simt efficiency of global rendering algorithms with architectural support for dynamic micro-kernels. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO '43, IEEE Computer Society, Washington, DC, USA. pp 237–248
7. Frey S, Reina G, Ertl T (2012) Simt microscheduling: Reducing thread stalling in divergent iterative algorithms. In: Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing PDP '12. IEEE Computer Society, Washington, DC, USA. pp 399–406
8. Han TD, Abdelrahman TS (2011) Reducing branch divergence in gpu programs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4, ACM, New York, NY, USA. pp 3:1–3:8
9. Lawrence R (1998) A survey of cache coherence mechanisms in shared memory multiprocessors
10. Chaudhary MK, Kumar M, Rai M, Dwivedi RK (2011) Article: A Modified Algorithm for Buffer Cache Management. Int J Comput Appl 12(12):47–49
11. Bennett JE, Flynn MJ (1996) Reducing Cache Miss Rates Using Prediction Caches. Technical report
12. Schnberg S, Mehnert F, Hamann C-J, Hamann Clj, Reuther L, Hrtig H (1998) Performance and Bus Transfer Influences. In: In First Workshop on PC-Based Syatem Performance and Analysis

13. Bahar RI, Albera G, Manne S (1998) Power and performance tradeoffs using various caching strategies. In: Proceedings of the 1998 international symposium on Low power electronics and design, ISLPED '98, ACM, New York, NY, USA. pp 64–69
14. Jeon HS, Noh SH (1998) A database disk buffer management algorithm based on prefetching. In: Proceedings of the seventh international conference on Information and knowledge management, CIKM '98, ACM, New York, NY, USA. pp 167–174
15. Johnson T, Shasha D (1994) 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In: Proceedings of the 20th International Conference on Very Large Data Bases. VLDB '94, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. pp 439–450
16. Bajwa RS, Hiraki M, Kojima H, Gorny DJ, Nitta K, Shridhar A, Seki K, Sasaki K (1997) Instruction buffering to reduce power in processors for signal processing. IEEE Trans. Very Large Scale Integr Syst 5(4):417–424
17. Hsu C-H, Kremer U (2003) The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. SIGPLAN Not 38(5):38–48
18. Marculescu D (2000) On the Use of Microarchitecture-Driven Dynamic Voltage Scaling
19. Manne S, Klauser A, Grunwald D (1998) Pipeline gating: speculation control for energy reduction. In: Proceedings of the 25th annual international symposium on Computer architecture. ISCA '98, IEEE Computer Society, Washington, DC, USA. pp 132–141
20. Ruan S-J, Tsai K-L, Naroska E, Lai F (2005) Bipartitioning and encoding in low-power pipelined circuits. ACM Trans Des Autom Electron Syst 10(1):24–32
21. Lei H, Duchamp D (1997) An Analytical Approach to File Prefetching. In: In Proceedings of the USENIX 1997 Annual Technical Conference. pp 275–288
22. Li Y, Henkel Jrg, Jrghenkel Y (1998) A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems
23. Woop S, Schmittler J, Slusallek P (2005) Rpu: a programmable ray processing unit for realtime ray tracing. In: ACM SIGGRAPH 2005 Papers. SIGGRAPH '05. ACM, New York, NY, USA. pp 434–444
24. Johnson M, William M (1989) Super-Scalar Processor Design. Technical report
25. Whitham J (2013) Simple scalar/ARM VirtualBox Appliance. Website. http://www.jwhitham.org/simplescalar
26. Brooks D, Tiwari V, Martonosi M (2000) Wattch: a framework for architectural-level power analysis and optimizations. SIGARCH Comput Archit News 28(2):83–94
27. Alameldeen AR, Wood DA (2006) Ipc considered harmful for multiprocessor workloads. IEEE Micro 26(4):8–17

Submit your manuscript to a SpringerOpen journal and benefit from:

► Convenient online submission
► Rigorous peer review
► Immediate publication on acceptance
► Open access: articles freely available online
► High visibility within the field
► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com