A Comparative Analysis of Techniques for Executing Branched Instructions

Sanjay Misra, Abraham Ayegba Alfa, Kehinde Douglas Ajagbe, Modupe Odusami and Olusola Abayomi-Alli

1 Introduction

Though in well-defined pipeline stage that is at an appropriate instant, pipelining allows enormous number of instructions to be performed simultaneously [1]. For better performance to be achieved, pipelining is used to overlay the execution of instructions. Instruction-level parallelism (ILP) describes the prospective overlay for different sets of instructions [1]; simply because these instructions can be carried out in parallel [2]. As a result of complication in nature and size of instructions constructs, the circumstances curbing utilizing of ILP in compilers and processing elements keep on increasing. Some of the drawbacks observed include greater part supports only various problems of straight instructions, also prediction analysis is needed to convert branch/conditional instructions to straight instructions, and this usually result in stall of memory/compiler system. One inlet and outlet access in loop body (which is not flexible to allow new instructions scheduling interrupts until the inlet or outlet is detected) in basic block (BB) architecture is available [3]. Among the instructions in basic block architecture, there is primarily tiny or no overlay. By using different architectures and techniques, various forms of parallelism can be exploited.

S. Misra (✉) · M. Odusami · O. Abayomi-Alli
Center of ICT/ICE Research, CUCRID Building, Covenant University, Ota, Nigeria
e-mail: sanjay.misra@covenantuniversity.edu.ng

M. Odusami
e-mail: modupe.odusami@covenantuniversity.edu.ng

O. Abayomi-Alli
e-mail: olusola.abayomi-alli@covenantuniversity.edu.ng

A. A. Alfa · K. D. Ajagbe
Kogi State College of Education, Ankpa, Nigeria
e-mail: abrahamsalfa@gmail.com

K. D. Ajagbe
e-mail: dougajagbe@gmail.com

R. K. Shukla et al. (eds.), *Data, Engineering and Applications*,
https://doi.org/10.1007/978-981-13-6351-1_20

Techniques such as single instruction, multiple data (SIMD), very long instruction Word (VLIW), and the superscalar execution could be used to attain parallel execution at the instruction level. Practical usage of parallelism intrinsic in algorithms with high level can be greatly achieved through multicore architectures [4]. This paper shows the differences between the predication process and the two-way loop process for branched instructions. The rest of the paper is structured as follows: the review of branched instruction execution and execution cycle instructions are presented in Sect. 2. Design methodology and various technique used are discussed in Sect. 3. Results and discussion are presented in Sect. 4. Conclusion of the paper is done in Sect. 5.

2 Review of Branched Instruction Execution

Unrolling loop technique is one of the most utilized techniques for executing branched instruction. In order to exploit parallelism by bypassing branch instructions, many enlarged basic blocks are contained in the unrolling of loop techniques. Enlarging basic block can be referred to as monotonous activities like loops using an algorithm to attain an efficient and small-scale code. Provided the bound variables are defined at compile time and unchanged, however, a loop body can be repeated for a couple of times using this algorithm [4, 5].

Figure 1 depicts the beginning of every iteration for unrolling loop, and no control exists for instruction. Although, there exist more compact in the loop form. In view of stable bound loops making the control instruction by the loop form naturally irrelevant, total iterations are determined at compile time. Unrolling loop appropriated the issue of instruction control. The unrolling loop has the capability to duplicate the body of loop n time with iteration step n incrementation of induction variable by a factor of two for every iteration and unrolling by a factor of two is shown in Fig. 1. For parallel execution, the loop body yields n times longer and higher number of instructions [4–6].

With the view of increasing the capacity for parallelism, unrolling loop expands basic blocks. The loop body is duplicated n times and the counter of the loop changes

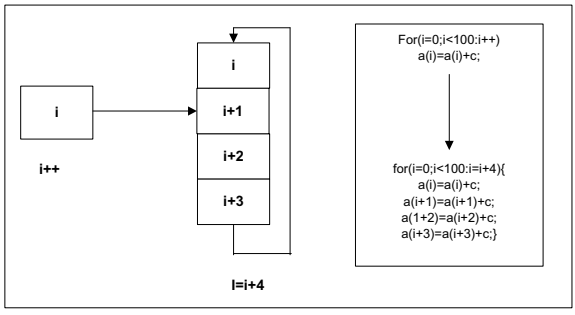

Fig. 1 Unrolling loop architecture and code [6]

by the step n with the unrolling loop. One major drawback of the unrolling loop technique is the difficulty in increasing the code size which does not apply to the outer loop. Considering compiler-time scheduling, the advantage of benefits from parallelism cannot be taken by unrolled adjacent. Out-of-order execution advantage taken by dynamic scheduling make the successive iteration run in parallel alongside the current one.

2.1 Instructions Execution Cycle

Instruction execution cycle is described as the implementation of a single basic block of instructions on compiler sectioned to different operations. Successive instruction address is being maintained by the program counter. Specific instruction to be carried out is preserved by the instruction queue. Fetch, decode, and execute are the three principal steps for carrying out a compiler instruction. If a memory operand is used by the instruction, two additional steps are required, and these are: fetch operand and store output operand [7]. Descriptions of each of the steps are as follows:

Fetch: Instruction is being fetched from the instruction queue by the control unit and there is an increment in instruction pointer (IP).

Decode: To confirm what the instruction does, the function of instruction is decoded the control unit. Signals are forwarded to arithmetic logic unit (ALU) defining the action to be executed as input operands of instruction are being accelerated.

Fetch operands: The control unit indulges the read operation to recollect operand and duplicate it to internal registers as long as the instruction desires an input operand saved on memory. User programs are hidden from internal registers.

Execute: Internal registers and named registers are used as operands as the ALU executes instruction and forward the outcome to memory and/or named registers. ALU updated the status flags by reading the state of processor information.

Store output operand: The advantage of a write operation is used by control unit to save data as long as the location of the output operand is in the memory.

Loop unrolling hardware caching: Hardware made up of components such as a negative branch displacement, stack-based approach, and a comparator (use to detect) are used to classify loop bodies. Information for loop linked is recorded on the loop stack upon loop entry discovery during commit time [8]. The total of successful iterations per visit and for every iteration in-loop branch log is accounted for by this dynamic information [8, 9].

3 Methodology

The design is divided into three stages and these include First stage: Choose the pipelining and two-way loop as the materials for study. Second stage: Conduct branched or conditional construct for each of the techniques is selected in the first phase.

Table 1 Setup of two-way loop technique [1]

Component	Characteristics
ILP	Overlapping/interleave of instructions that support multiple issues
Model	Object/rapid iterative
Program style	Code motion/transformation. Two-way algorithm
GUI	Microsoft Visual Basic programming language
Unrolling loop	Replicating loop bodies into many other independent sub-loops

Third stage: Evaluation of the performance of each technique using utilization, execution rate, and regression coefficients indices.

3.1 A Two-Way Loop Technique Setup

The TWL technique offers a breakdown of the process of making more ILP accessible to basic block instructions in loop structure as shown in Table 1.

3.2 An Algorithm for Two-Way Loop

Various issues/simultaneous instructions executions of branch and straight path of loops is supported by two-way loop algorithm [10]. Three main things are achieved by the TWL algorithm and these include transforms of control dependences into data dependences, increasing in parallelism exploited by layers of instructions in a compiler, and transforms unrolling of loop technique by increasing basic block, respectively, in order to increase the chance of exploiting parallelism, thereby granting several branched instructions to be executed. For a competent and more solid form of the resulting code to be achieved, repeated operations of an algorithm are regularly revealed as a loop. The steps are [10]:

1. Establish conditional branch instructions //over a number of loop unrolling.
2. Step A instruction is converted to predicate defining instructions //instructions that fix a certain degree known as a predicate.
3. Modification of instructions associated to both branch and straight constructs into predicate instructions //according to the degree of the predicate, both of them carried out instruction.
4. Fetch and execute predicated instructions regardless of the degree of their predicate//over a number of loops unrolling.
5. Instructions retirement stage.
6. For predicate degree = TRUE //progress to the next and last pipeline phase.

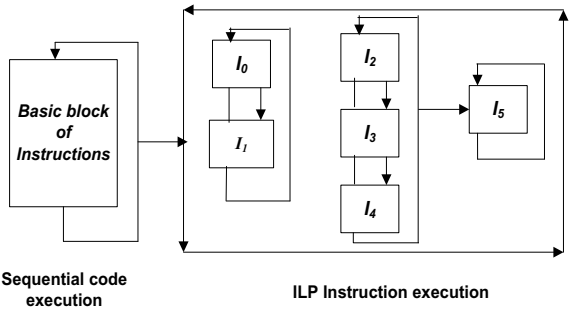

Fig. 2 Framework of two-way loop mechanism [10]

7. For predicate degree = FALSE //nullified: there is no need to write back the results that is produced and hence lost.

Competent supervision of instruction control, conditional instructions, and storage for predicate values of different basic blocks loops are supported by the algorithm [10].

3.3 Framework of Branched Instructions Mechanism

The embedded algorithm in the new design enabled the processing unit of the mechanism, which has fixed characteristic of ILP architecture to control multiple instruction sets, which could be conditional or branch instructions and process them at the shortest CPU cycles time [10]. The result is the capability to handle instructions such as branch and straight instructions, parallelism exploitation among instructions, and the speed of instructions execution as shown in Fig. 2.

3.4 Metrics of Performance

The comparisons of the two branched instruction processing techniques are measured such as the closeness of execution results. ILP impact is measured using the speedup in execution time and is expressed by Eq. (1)

$$ILP\ Speedup = \frac{T0}{T1}. \tag{1}$$

where

T0 previous technique execution time
T1 TWL technique execution time.

Amdahl's Law
The states of Amdahl's law gave the total speedup of a particular component or rate of used up by the system as given by Eq. (2) [7, 11]

$$S = \frac{1}{(1-f) + \frac{f}{k}}. \tag{2}$$

where

S speedup for total system.
F section of work executed by the faster components.
K speedup for new component.

Flynn Benchmark
Execution time = total time required to run program, that is, product development wall clock time and research [12, 13].

performance $= \frac{1}{(execution\ time)} \leq 1$ is overwhelmed by concurrency of execution, ILP, I/O speed, processor speed, program type, looping, instruction paths available, and system workload.

Utilization
Cantrell [11] generates a formula and benchmark to calculate total executed instructions (μ), if T seconds is the mean time of execution, μ is expressed by $\mu = \frac{1}{T}$

$$The\ Utilization\text{: } \rho = \lambda T = \frac{\lambda}{\mu} = 1.$$

When, mean waiting time, Tw $= \infty$, and λ = *rate of issue of instructions*.This implies that parallelism is not available in the program.

4 Results and Discussion

ILP approach experimental execution time versus predicted approach is shown in Table 2.

Table 2 gives details of the implementation of the mean time of executions in ILP mechanism test for 50 students, 10,000 and over instructions set, 4 sub-loops

Table 2 Simulation mean time of executions

Time of execution	Loop fields			
	l1	l2	l3	l4
T0 (sec)	952	374	476	604
T1 (sec)	401	137	143	431

forms executed in parallel and simultaneously with predicted mechanism and ± 0.05 statistical sample error students records.

Total execution time, $\mathrm{Tt} = \sum T0 + \sum T1$
where

T0 time of execution for predicted technique.
T1 time of execution for ILP technique.

$$\sum T0 = 851 + 205 + 337 + 514 = [1] = 1907$$
$$\sum T1 = 502 + 306 + 282 + 291 = [2] = 1381$$
$$\therefore Tt = \sum T0 + \sum T1 = [1] + [2]$$
$$= 1907 + 1381$$
$$= 3288$$

Predicted and ILP techniques percentages of time of execution are given by

$$Percentage\ of T0 = \frac{\sum T0}{\sum Tt} \times \frac{100}{1}$$
$$= \frac{1907}{3288} \times \frac{100}{1} = 0.5799878 \times 100 \quad = 579987 = 58\%$$

and

$$Percentage\ of T1 = \frac{\sum T1}{\sum Tt} \times \frac{100}{1}$$
$$= \frac{1381}{3288} \times \frac{100}{1} = 0.4200122 \times 100 \quad = 4200122 = 42\%$$

The performances (P_0) of predicted technique and (P_1) of ILP are given by

$$P0 = \frac{1}{Te0} = \frac{1}{1907} = 5.2438 \times 10^{-4}(Approx.)$$
$$P1 = \frac{1}{Te1} = \frac{1}{1381} = 7.2411 \times 10^{-4}(Approx.)$$

Units are cycle per second (CPS). The performance rate of (*P1*) improved over (*P0*) due to lesser execution time of the ILP technique. The speedup (*n*) is computed from Eq. 1 by

$$n = \frac{\mathrm{T0}}{\mathrm{T1}} = \frac{1907}{1381} = 1.381(Approx.)$$

This signifies that the ILP mechanism is 1.381 times faster than predicted mechanism.

Comparisons: To estimate the strength and direction of a linear relationship between the outcomes of two approaches for executing branched instructions constructs, linear correlation coefficient (r) statistical quantity is used as expressed in Eq. 3:

$$r = \frac{n\sum T0.T1 - (\sum T0)(\sum T1)}{\sqrt{n(\sum T0^2) - (\sum T0)^2}.\sqrt{n(\sum T1^2) - (\sum T1)^2}}. \tag{3}$$

where n is numbers loop iterations = 4,

$$\sum T0 = 2406,$$
$$\sum T1 = 882,$$
$$\sum T0.T1 = 622462,$$
$$(\sum T0)^2 = 1637572,$$
$$(\sum T1)^2 = 240420,$$
$$n\sum T0.T1 - \Big(\sum T0\Big)\Big(\sum T1\Big) == 4 * 622462) - (2406 * 882) = [3] = 367756$$
$$\sqrt{n\Big(\sum T0^2\Big) - \Big(\sum T0\Big)^2}.\sqrt{n\Big(\sum T1^2\Big) - \Big(\sum T1\Big)^2} = [4]$$
$$= -2480389922$$

$$r = \frac{[3]}{[4]} = \frac{367756}{-2480389922} - 0.000148265$$

The value of linear correlation coefficient (r) is (0.000148265), which indicates a negative value tending toward zero. Again, the directions of execution times are inverse of the other (that is, as T0 increases, T1 decreases in long term). The slope of fitness is in the negative region of the graph; and the correlation is random, nonlinear, or weak between the branched instruction execution times of pipeline and TWL techniques.

Similarly, to determine the proportion of the variance (or changes) of predictable variable (T0) from unpredictable variable (T1) for execution times, coefficient of determination (r^2) was chosen. This is the ratio of explained variance to total variance given by Eq. 4.

$$r^2 = \left[\frac{n \sum T0.T1 - (\sum T0)(\sum T1)}{\sqrt{n(\sum T0^2) - (\sum T0)^2}.\sqrt{n(\sum T1^2) - (\sum T1)^2}} \right]^2. \quad (4)$$

where

$$r = -0.000148265,$$
$$r^2 = \left[\frac{[3]}{[4]} \right]^2 = (-0.000148265)^2 = 0.0000000219826$$

The value of the coefficient of determination (r^2) is 0.0000000219826 (or 2.919826×10^8). This result shows that approximately one-quarter of the area is covered by the line of regression in scatter plot. Therefore, the regression line passes exactly at most one data point in the scatter plot to reveal that the regression line was further away from the points and less explainable. The implication is that ILP technique used lesser execution times as compared to the predicted technique because, it increased numbers of iterations, multiple issues, parallel execution of both conditional and straight instructions constructs, and availability of overlaps in loops processes.

Users estimate ILP from execution time simply as overall time needed to run a program loop. According to Flynn's benchmark, the numbers of instructions revolved per cycle is equal or less than 1 (P1 $\leq$ 1). That signifies that for improved technique with execution time of 1381; recall that: $P_1 = \frac{1}{time\ of\ execution}$

$$P_1 = \frac{1}{Te1} = \frac{1}{1381} = 7.2411 \times 10^{-4} (Approx.)$$

P1 $\leq$ 1 satisfies Flynn's benchmark, i.e., the user will perceive computer system as high-performance system due to parallelism, number of loop pipelines, and I/O speed.

The total work done per unit time is termed *Capacity Index* and can also be perceived as the throughput. The total of requests completed per second is the quantities measured. Utilization is the total instructions issued/total completed per sec.

If average time of execution is *Tt* seconds, the total instructions executed (μ) is given by

$$\mu = \frac{1}{T0} = 0.00052438 \times 3600 \times 24 = 0.3020451$$
$$\mu = \frac{1}{T1} = 0.00072411 \times 3600 \times 24 = 62.563104$$

and $\lambda = no\ of\ loops = 4$

From Eq. 3, utilization for:
Predicted technique, $\rho = \lambda T0$

$$= \frac{\lambda}{\mu} = \frac{4}{0.3020451} = 13.243056$$

and ILP, $\rho = \lambda T1$

$$= \frac{\lambda}{\mu} = \frac{4}{62.563104} = 0.06393545$$

This implies that ρ for ILP technique is lower than 1 satisfying the Cantrell's benchmark, which means several parallel processes and instructions can be executed in a day.

5 Conclusion

The approach revealed that ILP exploits branched executions better than predicted method on the basis of the time of execution, rate of execution, and utilization indexes. ILP offers increased speed of execution in programs, enabling capabilities of processing elements/compiler to point several instructions concurrently.

More so, ILP converts basic block of instructions to more dependent and independent instructions, performs parallel executions unlike the predicted approach (pipeline). ILP makes compiler/processing elements to overlap (or interleave) execution, replicates the original loops (loop iterations) loop copies, thereby encouraging parallel execution of several loop bodies at the same time and several issues of instructions.

This paper recommends ILP approach as the most preferable choice for speedup and parallel execution of instructions from either instruction constructs as compared to pipeline approach. Comparing several techniques with ILP could be considered as future work.

Acknowledgements We acknowledge the support and sponsorship provided by Covenant University through the Centre for Research, Innovation and Discovery (CUCRID).

References

1. Hennessy, J., Patterson, D.A.: Computer Architecture, 4th edn, pp. 2–104. Morgan Kaufmann Publishers Elsevier, San Francisco (2007)
2. Smith, J.E., Weiss, J.: PowerPC 601 and Alpha 21064: A tale of two RISCs. J. Comput. IEEE Press **27**(6), 46–58 (1994)

3. Jack, W.D., Sanjay, J.: Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. Unpublished M.Sc. thesis of Department of Computer Science, Thornton Hall, University of Virginia, Charlottesville, pp. 1−8 (1995)
4. Pepijn, W.: Simdization transformation strategies - polyhedral transformations and cost estimation. Unpublished M.Sc thesis, Department of Computer/Electrical Engineering, Delft University of Technology, Netherlands, pp. 1−77 (2012)
5. Vijay, S.P., Sarita, A.: Code transformations to improve memory parallelism. In: 32nd Annual ACM/IEEE International Symposium on Microarchitecture, pp. 147 − 155. IEEE Computer Society, Haifa (1999)
6. Pozzi, L.: Compilation techniques for exploiting instruction level parallelism, a survey. Department of Electrical and Information, University of Milan, Italy Technical Report 20133, pp. 1−31 (2010)
7. Parthasarathy, K.A.: Performance measures of superscalar processor. Int. J. Eng. Technol. IJET Publications, UK **1**(3), 164–168 (2011)
8. Kaeli, D., Rosano, R.A.: Exposing instruction level parallelism in presence of loops. J. Comput. Syst. **8**(1), 74–85 (2004)
9. Marcos, R.D.A., David, R.K.: Runtime predictability of loops. In: 4th Annual IEEE International Workshop on Workload Characterization, I.C., Ed., Texas, pp. 91−98 (2001)
10. Misra, S., Alfa, A.A., Adewale, O.S., Akogbe, A.M., Olaniyi, M.O.: A two-way loop algorithm for exploiting instruction-level parallelism in memory system. In: Beniamino, M., Sanjay, M., Ana, M., Rocha, A.C. (eds.) ICCSA 2014, LNCS, vol. 8583, pp. 255–264. Springer, Heidelbreg (2014)
11. Cantrell, C.D.: Computer system performance measurement. Lecture CE/EE 4304, The University of Texas, Dallas (2012). http://www.utdallas.edu/~cantrell/ee4304/perf.pdf
12. Flynn, M.J.: Computer Architecture: Pipelined and Parallel Processor Design, 1st edn, pp. 34–55. Jones and Bartlett Publishers, New York (1995)
13. Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., Chang, K.: The case for a single-chip multiprocessor. Conf. ACM SIGPLAN Not. Stanf. **31**(9), 2–11 (1996)