Beniamino Murgante · Sanjay Misra
Ana Maria A.C. Rocha · Carmelo Torre
Jorge Gustavo Rocha · Maria Irene Falcão
David Taniar · Bernady O. Apduhan
Osvaldo Gervasi (Eds.)

Computational Science and Its Applications – ICCSA 2014

LNCS 8580

14th International Conference
Guimarães, Portugal, June 30 – July 3, 2014
Proceedings, Part IV

Springer

A Two-Way Loop Algorithm for Exploiting Instruction-Level Parallelism in Memory System

Sanjay Misra[1], Abraham Ayegba Alfa[2], Sunday Olamide Adewale[3], Michael Abogunde Akogbe[2], and Mikail Olayemi Olaniyi[2]

[1] Covenant University, OTA, Nigeria
[2] Federal University of Technology, Minna, Nigeria
[3] Federal University of Technology, Akure, Nigeria
Sanjay.misra@covenantuniversity.edu.ng,
abrahamsalfa@gmail.com, adewale@futa.edu.ng,
{michael.akogbe,mikail.olaniyi}@futminna.edu.ng

Abstract. There is ever increasing need for the use of computer memory and processing elements in computations. Multiple and complex instructions processing require to be carried out almost concurrently and in parallel that exhibit interleaves and inherent dependencies. Loop architectures such as unrolling loop architecture do not allow for branch/conditional instructions processing (or execution). Two-Way Loop (TWL) technique exploits instruction-level parallelism (ILP) using TWL algorithm to transform basic block loops to parallel ILP architecture to allow parallel instructions processes and executions. This paper presents TWL for concurrent executions of straight forward and branch/conditional instructions. Further evaluation of TWL algorithm is carried out in this paper.

Keywords: Branch/conditional, loops, ILP, multiple issues, parallelism.

1 Introduction

Pipelining enables infinite number of instructions to be executed concurrently (or at the same time), though in distinct pipeline stages at a particular moment [1]. Pipelining is used to overlap the execution of instructions and to attain better performance. This prospective overlap for several sets of instructions is known as instruction-level parallelism (ILP) [1]; reason being that these instructions can be executed in parallel. There are two very distinct approaches for exploiting ILP: hardware and software technology [1], [2].

The factors limiting exploiting of ILP in compilers/processing elements continue to widen because of complexity in size and nature of instructions constructs. Some of the short coming observed with existing techniques include: majority support multiple issues of straight instructions, prediction techniques are required to transform conditional/branch instructions to straight instructions, which often cause stall of memory/compiler system [3].

In BB architecture, there is only one entry and exit points in loop body (that is inflexible to accept interrupts for new instructions scheduling until the entry or exit is

B. Murgante et al. (Eds.): ICCSA 2014, Part V, LNCS 8583, pp. 255–264, 2014.

encountered) [3]. There is basically little or no overlap among instructions in basic block architecture. The opportunities to speed up program execution through parallelism exploitation became feasible; that is executing parts of the program at the same time as against increasing sequential execution speeds. Unrolling loop technique is an attempt to achieve that by replicating original loop body into several other sub-loops for multiple issues of instructions for concurrent processing and execution.

The remainder of the paper is organized as follows. The next section introduces unique features and problems in existing techniques for exploiting ILP in memory system. In section 3, challenges are identified. Following that, a new technique is formed and results related to ILP exploitations are discussed in section 4 and, finally, conclusions are drawn in section 5.

2 Related Works

2.1 Pipeline Parallelism

Flynn; Smith and Weiss [4], [2] introduced pipelining into the architecture of CPUs to permit instruction execution in minimum time possible. Pipeline parallelism technique involves partitioning of each instruction set into several other segments that required different hardware resources for purpose of completing execution within a cycle. A CPI (cycle per instruction) equals to one can be attained given any scenario to be true [4]. Multiple issues architecture takes advantage of parallelism inherent in programs of application such as sequential stream of codes, compare and control data dependencies found in instruction, identifying sets of independent instructions to be issued concurrently without altering the correctness of program [5].

2.2 Loop Architectures

Data and control dependencies form the basis for execution of a cyclic code containing conditions (or branch). Loop dependencies analysis continues to widen in complexity especially when every statement can be executed severally giving rise to two (or more) separate iteration executions (i.e. loop-carried dependencies; as a result of dependencies existing between statements) in the same loop body [6]. There exists greater responsibility to identify which paths are most repeated, that every path of the program for real time application exhibit constraints in time, overall time of execution must be minimized [5].

Unrolling of Loop: This technique involves many enlarged basic blocks for the purpose of exploiting parallelism by avoiding branch instructions. Enlarging basic block means repetitive operations in form of loops using an algorithm to achieve an efficient and small scale code. Unrolling loop algorithm repeats a loop body for a number of times, if the bound variables are unchanged and defined at compile time [8], [9]. The loop unrolling architecture and code are illustrated in Figure 1. Unrolling loop resolves no control issue for instruction at the start of every iteration (runs check on the body of the loop for entry point) [9], [8] and [5].

2.3 Execution Cycle of Instructions

The execution of a single basic block of instructions on compiler is partitioned into a series of independent operations referred to as execution cycle of instruction. Parathasarathy [7] gives description of five basic steps for executing a compiler instruction using memory operand is as follows:

Fetch: The control unit fetches the instruction from the instruction queue and the instruction pointer (IP) is incremented.

Decode: The function of instruction is decoded by the control unit to ascertain what the instruction does. Input operands of instruction are pushed to the arithmetic logic unit (ALU) and signals are transmitted to the ALU specifying the operation to be carried out.

Fetch operands: If the instruction requires an input operand stored on memory, the control unit takes advantage of a *Read operation* to recall the operand and copy it into internal registers. Internal registers are not visible to program of the user.

Execute: The instruction is executed by the ALU using the named registers and internal registers as operands and sends the result to named registers and/or memory. The status flags rendering information about the state of processor is updated by the ALU.

Store output operand: If the output operand is located in memory, the control unit takes advantage of a write operation to store the data.

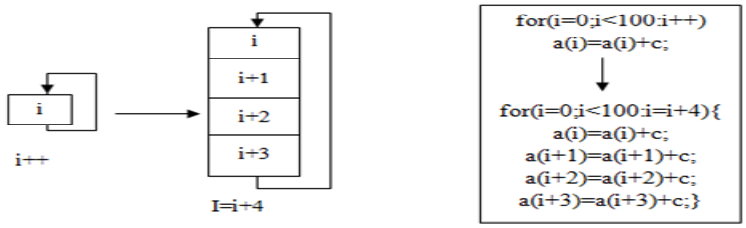

Fig. 1. Loop unrolling architecture and code. *Source- [5]*

3 Two-Way Loop Technique

This section discusses the TWL algorithm, its implementation and its mathematical modeling.

3.1 Two-Way Loop Algorithm

Two-Way Loop algorithm supports multiple issues/concurrent instructions executions of straight and branch paths of loops. It modifies unrolling of loop technique by severally enlarging basic block for parallelism exploitation by allowing multiple branch instructions executions.

I. Identify conditional branch instructions //across several loop unrolling
II. Transform instructions in Step I into predicate defining instructions // instructions that set a specific value known as a predicate
III. Instructions belonging to straight and branch constructs are then modified into predicate instructions // both of them execute according to the value of the predicate
IV. Fetch and execute predicated instructions irrespective of the value of their predicate// across several loops unrolling
V. Instructions retirement phase
VI. If predicate value = TRUE // continue to the next and last pipeline stage
VII. If predicate value = FALSE // nullified: results produced do not need to be written back and hence lost

3.2 Evaluation

Time of Execution: The impact of ILP can be measured by the speedup in execution time (that is speedup of ILP) is defined by Equation 1

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

where,

T0 = execution time of pipelining technique
T1 = execution time of TWL technique

Performance: According to Flynn Benchmark, Execution time = total time required to run program (that is wall-clock time for product development and testing) [10].

$$Performance = \frac{1}{(execution\ time)} \leq 1\ , \tag{2}$$

Utilization: Is number of instructions issued/number completed per second. The mean time that a request speeds in the system exposes more ILP. Cantrell [11] develops a benchmark and formula to compute number of instructions executed (μ) if the mean time of execution is T seconds, is given by: $\mu = \frac{1}{T}$

$$Utilization: \rho = \lambda T = \frac{\lambda}{\mu} = 1\ . \tag{3}$$

The mean waiting time (i.e. no parallelism is present in program), Tw = ∞
$\lambda = rate\ of\ issue\ of\ instructions.$

3.3 Mathematical Model for Two-Way Loop Algorithm

These mathematical notations show relationships between loops and instructions in a two-way loop.

I. *Loops /iterations constructs:* Unrolling is replicating the body of a loop multiple times to reduce loop iteration overhead.

```
for h = [ 0…Q]
 X(h) = Y(h) + Z(h)
```

Unrolled 3 times, gives;

```
 for h =  [0…Q] by 3
 X(h) = Y(h) + Z(h)
 X(h+1) = Y(h+1) + Z(h+1)
 X(h+2) = Y(h+2) + Z(h+2)
 X(h+3) = Y(h+3) + Z(h+3)
```

II. *Unrolling construct:* Unrolling loop can be transformed from original basic-block architecture into several deep inner loop bound for any given loop of the form:

```
 for h = [ 0 ... Q]
  X(p) = f(p)
```

The target machine/memory system usually has its vector length and several inner deep loop nests, where the inner loop has a constant loop bound without loop dependencies

```
 for h =  [0 …Q] by 32
 for k =[ 0 …32]
 X(h) = f(h)
```

III. *Interleave/overlap constructs:* To overlap (or interchange) the order of nested loops in the instructions basic-block transformation. Estimated by:

```
for h =  [0…Q]
for p = [ 0…Q]
A(h,p) = f(h,p)
```

Then, an interchange of the h and p loops gives:

```
 for p = [0...Q]
 for h = [0...Q]
 X(h,p) = f(h,p)
```

This interchange code proceeds for column-major order computation (preferably if X is stored in column-major order). To improve memory system and increase parallelism exploited for processing elements by transforming loops of basic-block architecture to several nested loops and performs interchange of loops order inwards to bring about overlapping of processes. Consider the multiple-nested loops of a matrix multiplication:

```
for h = [ 0…R]
for p = [ 0…Q]
for k =  [0…k]
Z(h,p) += X(h,k) * Y(k,p)
```

Applying tiling technique, all three loops gives:

```
for hh = [0…R] by Y
for pp = [0…Q] by Y
for kk =  [0…k] by Y
for h = hh…hh + Y
for p = pp…pp + Y
for k = kk…kk + Y
for Z(h,p) + =  X(h,k) * Y(k,p)
```

Suppose that: Y(h) is not compiled after this code executes, it need not be stored explicitly and memory space is saved.

IV. *Branch and conditional constructs*: When instructions execute in loop with conditions or branches, it can be expressed as:

```
for h = [ 0...Q]
for p = [ 0…R]
for k =  [0…K]
If  k = h
Y(h,k) = f(h,k)
Else k ≠ h; i.e. (k = p)
Z(p,k) = g(p,k)
```

4 Results

Experimental execution time for TWL technique against pipelining technique is presented in Table 1.

Table 1. The Mean Time of Executions for the Simulation Test

Execution Time	Loops fields			
	l1	l2	l3	l4
T0 (sec)	851	205	337	514
T1 (sec)	502	306	282	291

Table 1 gives the mean time of executions carried out in TWL technique test for 50 students (several instructions set), 4 sub-loops forms performed in parallel and concurrently with the pipelining technique (statistical sample error of ± 0.05) for students' record.

$$\text{Total execution time } T_{et} = \sum T_{e0} + \sum T_{e1}$$

where,
T_{e0} = execution time for pipelining technique
T_{e1} = execution time for TWL (new) technique

$\sum T_{e0} = 851 + 205 + 337 + 514 = [1] = 1907$
$\sum T_{e0} = 502 + 306 + 282 + 291 = [2] = 1381$

$$\therefore T_{et} = \sum T_{e0} + \sum T_{e1} = [1] + [2]$$
$$= 1907 + 1381$$
$$= 3288$$

Percentages of time of execution of pipelining and TWL techniques are given by:

$$Percentage\ of\ T_{e0} = \frac{\sum T_{e0}}{\sum T_{et}} \times \frac{100}{1}$$
$$= \frac{1907}{3288} \times \frac{100}{1} = 0.5799878 \times 100 = 579987 = 58\%\ (Approx.)$$

and,

$$Percentage\ of\ T_{e1} = \frac{\sum T_{e1}}{\sum T_{et}} \times \frac{100}{1}$$
$$= \frac{1381}{3288} \times \frac{100}{1} = 0.4200122 \times 100 = 4200122 = 42\%$$

The performances (P_0) of pipelining technique and (P_1) of new technique, are given by:

$$P_0 = \frac{1}{Te0} = \frac{1}{1907} = 5.2438 \times 10^{-4}\ (Approx.)$$
$$P_1 = \frac{1}{Te1} = \frac{1}{1381} = 7.2411 \times 10^{-4}\ (Approx.)$$

Units are (CPS) Cycle per Second. The performance (P_1) is better than (P_0) because the mean execution time is low for the TWL technique.

From equation 1, computing the speedup (n):

$$n = \frac{T_{e0}}{T_{e1}} = \frac{1907}{1381} = 1.381\ (Approx.)$$

This implies the TWL technique is 1.381 times faster than the pipelining technique.

Scale of Graphs: 100 (or 0.01) units are used to represent the quantities measured on Y- axes. While, 1 unit is used to capture the values of magnitude of loops (l) on X-axes as shown in Figures 2 and 3.

The difference in executions time between the pipelining technique (T_{e0}) and TWL technique (T_{e1}) is illustrated in Figure 2.

Remarks: The time of execution of 42% for TWL makes it more preferable to 58% for pipelining technique. The time taken to complete parallel executions in TWL is relatively same as time taken to complete one loop process/execution in the pipelining technique. Time spent for the pipelining technique is much due to sequential scheduling of the loops, while that of TWL technique is low, several loops are executed concurrently due to support for multiple issues and instruction level parallelism.

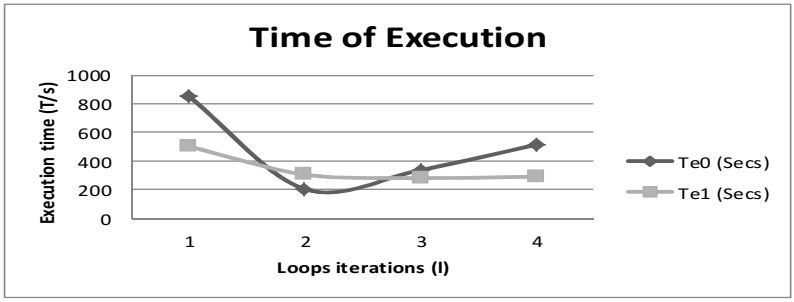

Fig. 2. A graph showing the difference between execution time of TWL technique and pipelining technique

The difference in the rate of loops execution per cycle (frequency) can be determine for the pipelining technique (F_0) and TWL technique (F_1) is shown in Figure 3.

Remarks: The frequency of pipelining technique (F_0) is very low because of lack of support for the ILP (i. e. more loops for less frequencies). While, the frequency of TWL (F_1) is relatively stable because of presence of multiple issues, and multiple processes/executions are carried out at the same time.

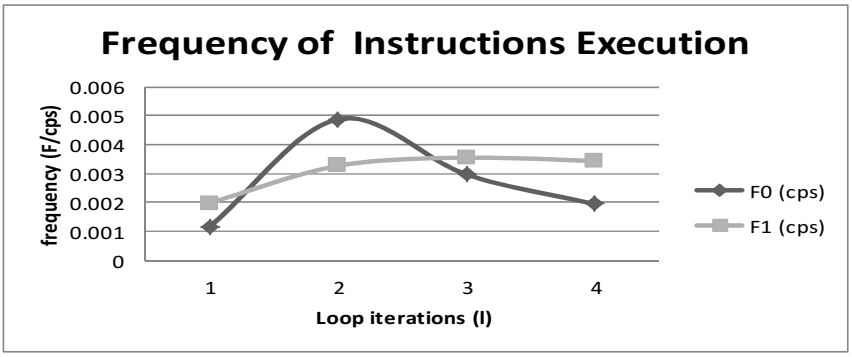

Fig. 3. A graph of frequency of TWL technique compared to pipelining technique

Capacity: Performance can be perceived as throughput and utilization. Throughput is total work done per unit time (measured as number of requests completed per second). While, utilization is number of instructions issued and completed successfully per second.

Number of instructions executed (μ), if the mean time of execution is T_e seconds (per day), is given by:

$$\mu = \frac{1}{T_{e0}} = 0.00052438 \times 3600 \times 24 = 0.3020451$$

$$\mu = \frac{1}{T_{e1}} = 0.00072411 \times 3600 \times 24 = 62.563104$$

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

From Equation 4, utilization for:

pipelining technique, $\rho = \lambda T_{e0}$

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

and, TWL, $\rho = \lambda T_{e1}$

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

∴ This means that ρ for TWL is less than 1 according to Cantrell's benchmark, which suggests more instructions and parallel processes can be carried-out in a day.

5 Conclusion

The paper presented a machine design using TWL algorithm for exploiting ILP, decreases time of execution over the existing technique by two, the utilization of 0.068 gives rise to increased capabilities to processing elements/compiler to issue multiple instructions at the same time, perform parallel executions, transforms basic block of instructions to several dependent and independent instructions as against pipelining technique.

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. IEEE Journal of Computer 27(6), 46–58 (1994)
3. Jack, W.D., Sanjay, J.: Improving Instruction-Level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation. An M.Sc. Thesis of Department of Computer Science, Thornton Hall, University of Virginia, Charlottesville, USA, pp. 1–8 (1995)
4. Flynn, M.J.: Computer Architecture: Pipelined and Parallel Processor Design, 1st edn., pp. 34–55. Jones and Bartlett Publishers, Inc., USA (1995) ISBN: 0867202041
5. Pozzi, L.: Compilation Techniques for Exploiting Instruction Level Parallelism, A Survey. Department of Electrical and Information, University of Milan, Milan. Italy Technical Report 20133, pp. 1–3 (2010)
6. Bacon, D.F., Graham, S.L., Sharp, O.J.: Complier Transformations for High Performance Computing. Journal of ACM Computing Surveys, 345–420 (1994)
7. Rau, B.R., Fisher, J.A.: Instruction-Level Parallel Processing: History Overview and Perspective. The Journal of Supercomputing 7(7), 9–50 (1993)
8. Pepijn, W.: Simdization Transformation Strategies - Polyhedral Transformations and Cost Estimation. An M.Sc Thesis, Department of Computer/Electrical Engineering, Delft University of Technology, Delft, Netherlands, pp. 1–77 (2012)

9. Vijay, S.P., Sarita, A.: Code Transformations to Improve Memory Parallelism. In: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, pp. 147–155. IEEE Computer Society, Haifa (1999)
10. Cantrell, C.D.: Computer System Performance Measurement. In: Unpublished Note Prepared for Lecture CE/EE 4304, Erik Jonsson School of Engineering and Computer Science, pp. 1–71. The University of Texas, Dallas (2012), http://www.utdallas.edu/~cantrell/ee4304/perf.pdf
11. Marcos, R.D.A., David, R.K.: Runtime Predictability of Loops. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization, I.C., Ed., Austin, Texas, USA, pp. 91–98 (2001)