International Journal of Enhanced Research in Science Technology & Engineering, ISSN: 2319-7463
Vol. 3 Issue 4, April-2014, pp: (561-566), Impact Factor: 1.252, Available online at: www.erpublications.com.

A Survey on techniques to increase Instruction Level Parallelism

Jimcy Babu[1], Kavitha V[2]
[1] M. Tech., CMRIT Bangalore, India
[2] Ph.D Scholar, Jain University, India

Abstract: Current microprocessors incorporate techniques to aggressively exploit Instruction Level Parallelism (ILP). ILP can be exploited by different methods like Very-Long Instruction Word (VLIW) processing, Vector processing and array processing. This paper reviews various techniques to increase Instruction Level Parallelism. Techniques reviewed in this paper are Balanced Scheduling with Compiler Optimization, Instruction Pre-computation and Micro-threading. The survey reveals that, balanced scheduling with compiler optimization had a 10% advantage over traditional schedulers, Instruction Pre-computation had speedups of 11.0%, while Micro-threading produce a speedup making it useful for Chip Multiprocessors.

Keywords: Instruction Level Parallelism, Instruction Pre-computation (IP), Value Reuse Table (VRT), Pre-computation Table (PT), Micro-threading.

1. Introduction

Pipelining can overlap the execution of instructions, when they are independent of one another. This potential overlap among instruction is called instruction level parallelism (ILP), since instruction can be evaluated in parallel. It is a measure of how many number of the operations in a computer program can be performed simultaneously. There are two approaches to ILP: Hardware and Software. Hardware level works upon dynamic parallelism whereas; the software level works on static parallelism. The Pentium processor works on the dynamic sequence of parallel execution but the Itatnium processor works on the static level parallelism. A goal of compiler and processor designers is to identify and take advantage of as much ILP as possible.ILP allows the compiler and processor to overlap the execution of multiple instruction or even to change the order in which instruction are executed.

There are different parallel processing techniques to exploit ILP: VLIW Processing, Vector Processing and Array Processing. In VLIW, ILP relies on the compiler to determine which instruction may be executed in parallel and providing that information to the processor hardware. Programs are required to be recompiled for new architecture, but achieve very good performance on programs written in sequential language such as C or FORTRAN when these programs are recompiled for a VLIW processor. In Vector Processing, the operations such as multiply are first divided into several steps and a stream of operands (vectors) operated in for each step parallel processing units. A Vector processor functions for ILP by using IP Architecture along with vector element operands in the parallel processing pipelines. It is most common special case of pipelining.

2. Techniques to increase ILP

A. The Balanced Scheduling with Compiler Optimization

Traditional instruction schedulers order load instructions based on an optimistic assumption that all loads will be cache hits. On most machines, this optimistic estimate is accurate because the processors block on cache misses. But processors with non-blocking Architecture, instruction latency is exposed to the compiler and becomes uncertain. Not only will the processor see both cache hits and misses, but also each level in the memory hierarchy will also introduce a new set of latencies.

(i) Balanced Scheduling (BS)

Balanced Scheduling is an algorithm that can generate schedules that adapt more readily to the uncertainties in memory latency. Rather than being determined by a predefined, Architecture based value load latency estimates are based on the number of independent instructions that are available to hide the latency of a particular load. Previous work has demonstrated that balanced schedules show speedups averaging 8% for several Perfect Club Benchmarks for two different cache hit/miss ratios, assuming a workstation like memory model in which cache misses are normally

distributed. This study combines Balanced Scheduling with 3 compiler optimizations that increase instruction level parallelism.

(ii) Compiler Optimization

Balanced Scheduling utilizes load level parallelism to hide the longer load latencies exposed by non-blocking processors. In this study, the effect of 3 techniques to increase ILP has been analyzed.

a) Loop Unrolling

Loop Unrolling increases the size of basic blocks by duplicating iterations a number of times equal to the unrolling factor. It contributes to increased performance in two ways. First, by creating multiple copies of the loop body; it decreases conditional branch and loop indexing overhead from all but the last copy. Second, the consequent increase in the size of the basic block can expose more ILP, thereby providing more opportunities for code scheduling.

b) Trace Scheduling

Trace Scheduling enables more aggressive scheduling by permitting code motion across basic block boundaries. It creates traces of paths through each procedure, guided by the profiled execution frequencies for each basic block. The trace scheduler then picks a trace, in order of decreasing execution frequencies, and schedules the basic blocks in the trace as if they were a single block. Code motion takes into account the effects of scheduling instructions across conditional jumps and merges, following specific rules. The final schedule effectively combines into a single block which has been multiple blocks if generated by traditional scheduler. Figure 1 illustrates an example of TS. The trace scheduler identifies basic blocks 1,2,4, and 5 as single block(trace A) and block 3 forms its own trace(trace B).

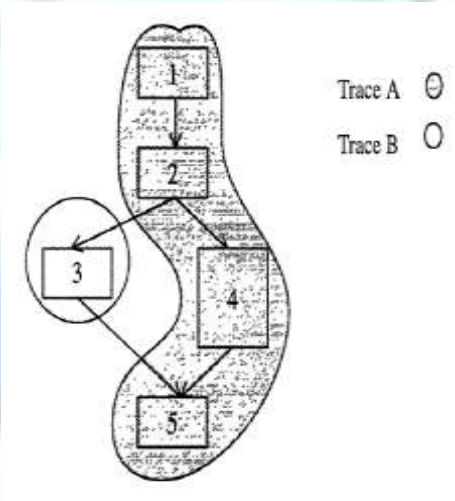

Fig 1: Example of Trace scheduling (Ref 1)

c) Locality Analysis

If the compiler can determine cache behavior, it can treat cache hits and misses differently. Cache hits can be scheduled using the traditional scheduling schemes.

(iii) Effect of Optimization

Balanced scheduled code consistently produced speedups over that generated by traditional scheduling. With only two exceptions, balanced scheduled code produced fewer load interlocks than that of traditional scheduler for all programs on all levels of optimization. The differences ranged from two or three times as many interlocks for the traditional scheduler.

Locality analysis contributed additional speedup when applied along with other optimization. When used with loop unrolling, speedups of 1.28 and 1.31 were obtained over balanced scheduling alone, for unrolling factors of 4 and 8. When Trace scheduling was alone applied, these speedups reached 1.29 and 1.40.Table 1 shows the comparison of locality analysis results

Table 2 shows the comparison between balanced scheduling and traditional scheduling. Optimization has increased the instruction level parallelism; therefore balanced scheduling was able to extend its advantage over traditional scheduling by exploiting the additional instruction level parallelism.

The analysis reveals that balanced scheduling had a 10% advantage over traditional scheduling with simple model. It validates that balanced scheduling is on average superior to traditional scheduling.

Optimization	Speedup relative to locality analysis alone	Speedup relative to balanced scheduling with no unrolling and no trace scheduling
Locality analysis	na	1.15
Locality analysis with loop unrolling by 4	1.11	1.28
Locality analysis with loop unrolling by 8	1.14	1.31
Locality analysis with trace scheduling and loop unrolling by 4	1.12	1.29
Locality analysis with trace scheduling and loop unrolling by 8	1.21	1.40

Table 1: Summary Comparison of locality analysis result (Ref 1)

Optimization in addition to balanced scheduling	Relative to traditional scheduling with the same optimization		Relative to balanced scheduling with no other optimization		Load interlock cycles remaining after applying optimization(% of total cycles)	
	Program speedup	Percentage decrease in load interlock cycles	Program speedup	Percentage decrease in load interlock cycles	Balanced Scheduling	Traditional scheduling
No optimization	1.05	51	na	na	7	15
Loop unrolling by 4	1.12	61	1.19	23	6	16
Loop unrolling by 8	1.18	62	1.28	26	6	16
Trace scheduling with loop unrolling by 4	1.14	65	1.19	42	5	15
Trace scheduling with loop unrolling by 8	1.16	56	1.26	34	5	15

Table 2: Summary comparison of balanced and traditional scheduling (Ref 1)

B. Instruction Precomputation(IP)

Value Reuse improves a processor's performance by dynamically caching the results of previous instructions into the Value Reuse Table (VRT) and reusing those results to bypass the execution of future instructions that have the same opcode and input operands. This reuse increases the amount of ILP. Replacing the least recently used entries with the results of the current instruction could eventually fill the VRT with instructions that are not frequently used. This decreases the effectiveness of this method.

Therefore, Instruction pre-computation is used to address the issue of frequency of execution. IP has two main steps: profiling and execution. In profiling step, the redundant computations with the highest frequencies or highest frequency/latency products (F/LPs), are found. The opcodes and input operands for these redundant computations are loaded into the pre-computation table (PT) before the program executes. During execution, PT functions like VRT but with two key differences: (a) The PT stores only highest frequency (F/LP) redundant computation, (b) The PT does not replace or update any entries. Therefore, it selectively targets those redundant computations that have an impact on the program. Table 3 shows the profiled benchmarks using two different input sets, A and B.

Benchmark	Suite	Type	Input Set A	Input Set B
099.go	SPEC 95	Integer	Train	Reduced Test
124.m88ksim	SPEC 95	Integer	Train	Test
126.gcc	SPEC 95	Integer	Test	Train
129.compress	SPEC 95	Integer	Train	Test
130.li	SPEC 95	Integer	Train	Test
132.ijpeg	SPEC 95	Integer	Test	Train
134.perl	SPEC 95	Integer	Test	Train
164.gzip	SPEC 2000	Integer	Reduced Small	Reduced Medium
175.vpr	SPEC 2000	Integer	Reduced Medium	Reduced Small
177.mesa	SPEC 2000	Floating-Point	Reduced Large	Test
181.mcf	SPEC 2000	Integer	Reduced Medium	Reduced Small
183.equake	SPEC 2000	Floating-Point	Reduced Large	Test
188.ammp	SPEC 2000	Floating-Point	Reduced Medium	Reduced Small
197.parser	SPEC 2000	Integer	Reduced Medium	Reduced Small
255.vortex	SPEC 2000	Integer	Reduced Medium	Reduced Large
300.twolf	SPEC 2000	Integer	Test	Reduced Large

Table 3: Selected Characteristics for the benchmarks previously tested (Ref 2)

After profiling each benchmark, the unique computations were sorted by their frequency of execution. Figure 2 shows what percentage of the total instructions are due to the top 2048(by frequency) Arithmetic Unique Computations.

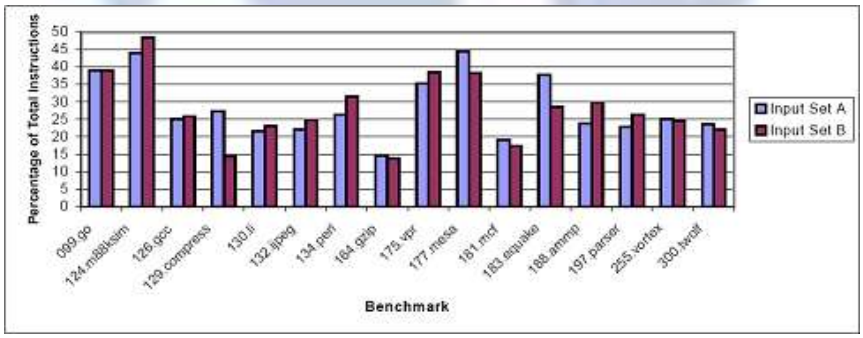

Fig 2: Percentage of instructions that is due to the top 2048 arithmetic Unique Computations (Ref 2).

As can be seen in figure2, the top 2048 Arithmetic Unique Computations account for 14.7% to 44.5% (Input set A) and 13.9% to 48.8% (B) of total instructions executed by the program. Furthermore, a small number of unique computations account for 3.1% to 19.6% (A) and 2.8% to 16.0% (B). Therefore, profiling a program to determine the highest frequency (F/LP) unique computations and putting them into a PT can significantly improve processor's performance by reducing the effective latency of each instruction that matches a unique computation in the PT, even for very small tables.

(i) Instruction Pre-computation performance

Figure 3 shows the speedup due to instruction pre-computation for various numbers of entries in the PT when input Set A is used for profiling and for execution. As shown in this figure, instruction pre-computation improves the performance of all benchmarks by an average of 4.6% (16 entries) to 12.2% (2048 entries). The average is the mean weighted by execution time.

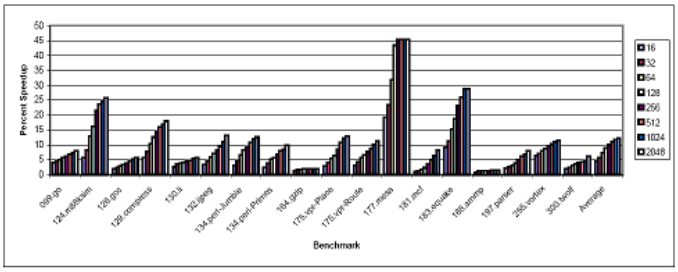

Fig 3: Percent speedup due to Instruction Precomputation for various Table sizes; Profile Input set A, Run input set A (Ref 2)

(ii) Comparison with value reuse

Since instruction pre-computation is related to value reuse, it is necessary to compare the speedups of the two techniques. The following figure 4 compares the speedups of value reuse and instruction pre-computation for various table sizes.

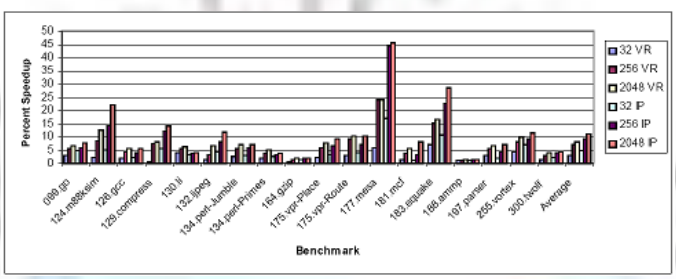

Fig 4: Speedup comparison between Value Reuse (VR) and Instruction Pre-computation (IP) for various table sizes; Profile input set A, Run input set B (Ref 2)

In the above figure 4, three table sizes are shown-32, 256, 2048 entries. VR corresponds to Value Reuse and IP corresponds to Instruction Pre-computation. Figure shows two main results: First, Instruction Pre-computation outperforms value reuse for almost all benchmarks and table sizes. Second, for smaller table sizes, which are less expensive in terms of area and cycle time, instruction pre-computation has non-trivial speedups (4.1% for 16-bit entry) while value reuse has a much smaller speedup(1.7% for 16-bit entry).

Finally, instruction pre-computation produces speedups that are almost always higher than the speedups produced by value reuse for same table size. Instruction pre-computation has lower area cost and lower access time. Instruction pre-computation can easily use the instruction's execution latency to determine the unique computations that could yield the most performance difference. This is beneficial for multimedia applications.

Instruction pre-computation is effective for small table sizes due to its profiling step. For small pre-computation table, it produces average speedups of 4.1% in terms frequency and 4.4% in terms of F/LP for the same program.

C. Micro-threading

Most microprocessors use out-of-order execution techniques. This allows superscalar processors to extract high levels of ILP. But the most significant problem with this approach is a large instruction window and logic to support instruction issue from it. The Micro-threaded model avoids the complexity in instruction issue and eliminates speculative execution. The model is based on decomposing a sequential program into small fragments of code called micro-threads. These micro-threads are scheduled dynamically and can communicate and synchronise with each other efficiently. This process allows sequential code to be compiled for execution on a scalable chip multiprocessor. As the code is schedule invariant, the same code will execute on any number of processors limited only by problem size.

The block diagram of a micro-threaded chip multiprocessor is shown in the figure 5. N micro-threaded pipelines are connected to these two shared communications systems. The first is the broadcast bus, used for creating threads and distributing invariants. The second is the shared-register ring network used to perform communication between the register files in the producer and consumer threads.

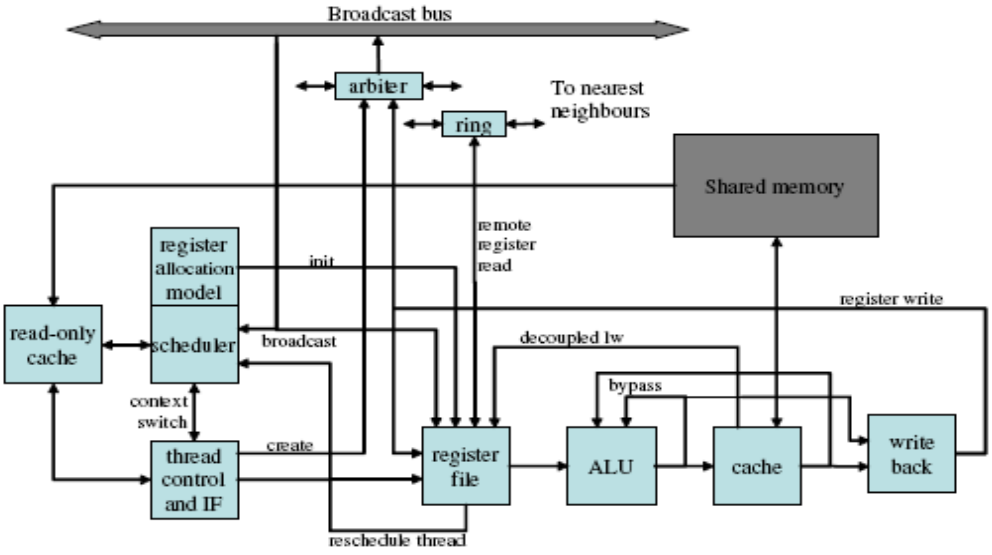

Fig 5: Micro-threaded CMP Architecture (Ref 3)

This model exploits Instruction Level Parallelism within basic blocks and across loop bodies. Micro-threading approach also supports a pre-fetching mechanism that avoids many instruction cache misses in the pipeline.

Conclusions

Each technique used to increase Instruction Level Parallelism has its own advantages as well as disadvantages. Balanced scheduling with compiler optimization has 10% advantageous than traditional schedulers. Balanced scheduling with compiler optimization produces average speedups that range from 1.15 to 1.40. It is more advantageous if more instruction level parallelism is available. Instruction Pre-computation produces speedups of 1.6% to 45.5%, with an average speed up of 11.0%.Inaddition to its superior performance; instruction pre-computation also comes up with less area and has lower access time. Instruction Pre-computation is beneficial for multimedia application. But Instruction Pre-computation is only effective for small table sizes due to its profiling step. In micro-threading, the micro-threads (fragments) capture Instruction Level Parallelism and loop concurrency. These fragments can be interleaved to single processor or distributed to multiple processors to achieve speedup. Hence it find useful in Chip Multiprocessors. Therefore micro-threaded CMP based on a fully distributed and scalable register file organization and asynchronous global communication buses is a good candidate for future Chip Multiprocessor.

References

[1]. Jack L.Lo and SusanJ.Eggers; “Improving Balanced Scheduling with Compiler Optimization that Increase Instruction Level Parallelism”, Department Of Computer Science and Engineering, University Of Washington, 1995.
[2]. NJoshua J. Yi, Resit Sendag, and David J. Lilja; “ Increasing Instruction Level Parallelism with Instruction Precomputation” Department Of Electrical And Computer Engineering, Minnesota Supercomputing Institute, University Of Minneeivabsota
[3]. Kostas Bousias, Nabil Hasasneh, Chris Jesshope, “Instruction-level parallelism through Micro-threading- A Scalable Approach to Chip Multiprocessor”, Computer Journal, March 2006,49(21): 211-233.
[4]. F. Gabbay and A. Mendelson; “Improving Achievable ILP through Value Prediction and Program Profiling”, Microprocessors and Microsystems, Vol.22.No.6, November 30, 1998.
[5]. http://www.cs.iastate.edu
[6]. http://www.da.univ.ac.in