Dynamically Scheduling VLIW Instructions with Dependency Information

Sunghyun Jee
Chonan College in Foreign Studies
Chonan, Chungnam, South Korea
jees@missouri.edu

Kannappan Palaniappan
University of Missouri
Missouri, Columbia, U.S.A.
palani@cecs.missouri.edu

Abstract

This paper proposes balancing scheduling effort more evenly between the compiler and the processor, by introducing dynamically scheduled Very Long Instruction Word (VLIW) instructions. Dynamically Instruction Scheduled VLIW (DISVLIW) processor is aimed specifically at dynamic scheduling VLIW instructions with dependency information. The DISVLIW processor dynamically schedules each instruction within long instructions using functional unit and dynamic scheduler pairs. Every dynamic scheduler dynamically checks for data dependencies and resource collisions while scheduling each instruction. This scheduling is especially effective in applications containing loops. We simulate the architecture and show that the DISVLIW processor performs significantly better than the VLIW processor for a wide range of cache sizes and across various numerical benchmark applications.

1. Introduction

Recent high performance processors have depended on Instruction Level Parallelism (ILP) to achieve high execution speed. ILP processors achieve their high performance by causing multiple operations to execute in parallel using a combination of compiler and hardware techniques. Very Long Instruction Word (VLIW) is one particular style of processor design that tries to achieve high levels of ILP by executing long instructions composed of multiple instructions. The VLIW processor has performance bottlenecks due to static instruction scheduling and the unoptimized large object code containing a number of NOPs (No OPerations) and LNOPs (Long NOPs), where the LNOP means a long instruction that has only NOPs [20~22]. Superscalar VLIW (SVLIW) is the improving style of VLIW processor design that tries to execute object code constructed by removing all LNOPs from VLIW code [14,15,21,22]. The SVLIW processor also has a performance limitation similar to the VLIW processor due to static scheduling. By making use of powerful features to generate high-performance code, the IA-64 architecture allows the compiler to exploit high ILP using Explicit Parallel Instruction Computing (EPIC) [23,24]. The IA-64 is a statically scheduled processor architecture where the compiler is responsible for efficiently exploiting the available ILP and keeps the executions busy [24]. Instead of the merits, the IA-64 processor has a performance limitation due to static instruction scheduling. In order to overcome current performance bottlenecks in modern architectures, a processor architecture that satisfies the following criteria is required: (1) balanced scheduling effort between compile-time and run-time, (2) dynamic instruction scheduling, and (3) reducing the size of object code.

This paper presents a new ILP processor architecture called Dynamically Instruction Scheduled VLIW (DISVLIW) that achieves these goals. The DISVLIW instruction format is augmented to allow dependent bit vectors to be placed in the same VLIW word. Dependent bit vectors are added to the instruction format to enable synchronization between prior and subsequent instructions. To schedule instructions dynamically, the DISVLIW processor uses functional unit and dynamic scheduler pairs. Every dynamic scheduler decides to issue the next instruction to the associated functional unit, or to stall the functional unit due to possible resource collisions or data dependencies among instructions per every cycle. Such features can reduce the total number of execution cycles of the DISVLIW processor better than those of the VLIW or the SVLIW processor that compulsorily schedules long instructions. The DISVLIW processor is reminiscent of the CDC-6600 Scoreboard, an early dynamically scheduled processor architecture [22]. A different with the CDC-6600 is that the compiler conveys more explicit information for managing the scoreboard, in the form of the dependence bit vectors. Besides, even though the superscalar processor is an effective way of exploiting ILP, this superscalar processor architecture requires complex devices and the impact of such complexity on the design cost and clock cycle time can be severe [20,21]. Consequently, the superscalar processor will not be evaluated in this paper.

The rest of the paper is organized as follows. Section 2 compares issue slots and instruction pipelines of various ILP processors, Section 3 introduces the DISVLIW processor architecture and instruction pipeline, in Section 4 we evaluate a performance of the DISVLIW processor, and conclusion follows in Section 5.

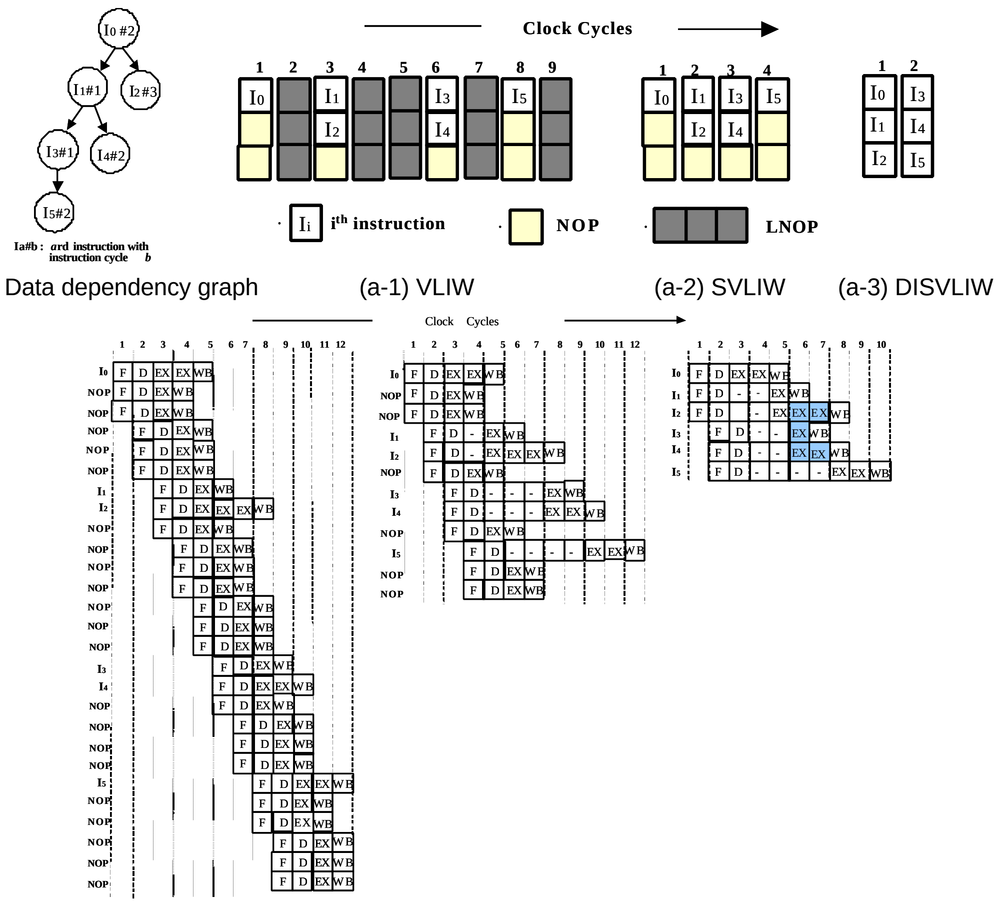
2. Instruction level parallelism

Figure 1 shows issue slots and execution images of the VLIW, the SVLIW, and the DISVLIW processors. The processors execute their own object code generated from given data dependency graph. In the data dependency graph, a node represents an instruction and a drected edge is annotated with data dependencies and resource collisions between instructions. We assume that every processor has three untyped functional units that can execute any instruction and a long instruction has three instructions. Figure 1 illustrates issue slots of each object code using rectangles repeated in the horizontal direction to represent consecutive clock cycles. Squares placed vertically in each rectangle represent the per cycle utilization instruction issue slots, where a rectangle and a square mean a long instruction and an instruction. An instruction is executed in the following four stages: F (Fetch), D (Decode), EX (EXecute), and WB (Write Back).

Figure 1(b-1) shows an execution image for the VLIW processor to execute VLIW code. The VLIW code contains a number of LNOPs and NOPs in order to solve data dependencies and resource collisions between long instructions as shown in Figure 1(a-1). During execution, the VLIW processor does not allow the next long instruction to enter into the execution stage until functional units have finished executing all instructions within the scheduled long instruction.

Figure 1(b-2) shows an execution image for the SVLIW processor to execute SVLIW code. The SVLIW code is constructed by removing all LNOPs from the VLIW code as shown in Figure 1(b-2).

In order to execute the SVLIW code, the SVLIW processor schedules the next long instruction after checking for data dependencies and resource collisions with the scheduled long instructions in advance. When a collision occurs, the processor is stalled as indicated by dash (–) in Figure 1(b-2) until all collisions are resolved. The SVLIW processor uses the same scheduling strategies used for the VLIW processor.



(b-1) VLIW (b-2) SVLIW (b-3) DISVLIW

Figure 1. Comparison of issue slots and execution images

Figure 1(b-3) shows an execution image for the DISVLIW processor proposed in this research. Since instructions within a long instruction may depend on each other as shown in Figure 1(a-3), we assume that each instruction contains dependency information in order to achieve synchronization. The DISVLIW processor issues one long instruction per cycle and dynamically executes each instruction using dependency information. As shown in the shaded pipelines in Figure 1(b-3), instructions I_2, I_3, and I_4 are simultaneously executed during the 6th clock cycle although the instructions are fetched on different clock cycle. Instructions I_2 and I_4 are also executed during the 7th clock cycle at the same time.

This example demonstrates the process by which the DISVLIW processor can achieve better performance in comparison to the VLIW or the SVLIW processor. The main insight is that in the DISVLIW processor each instruction within a given long instruction is dynamically processed. Therefore, the DISVLIW processor decreases the waiting time to process a given set of long instructions in comparison to other processors.

3. DISVLIW processor architecture

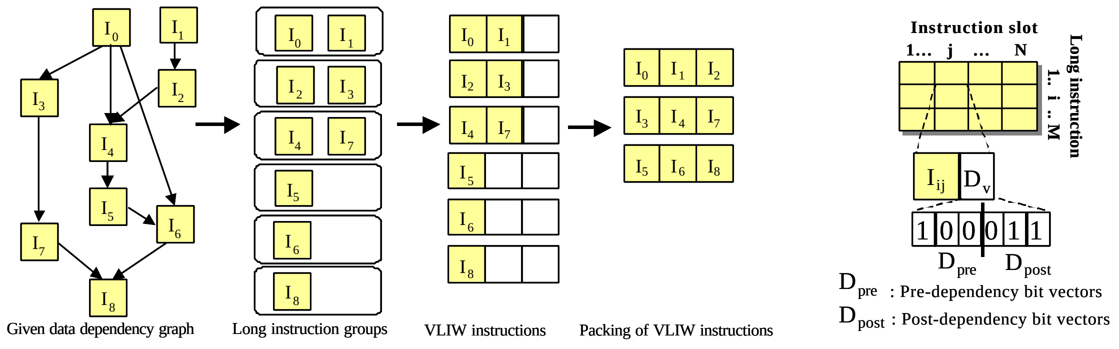
3.1 Long instruction format

To dynamically schedule VLIW instructions, the DISVLIW instruction format is augmented to allow dependent information to be placed in the same VLIW instruction. Dependent information is added to the instruction format to enable synchronization between prior and subsequent instructions.

The problem of optimal DISVLIW code generation can be subdivided into two phases as shown in Figure 2. In the remainder of this paper we will refer to the first phase as *VLIW instruction generation* and to the second phase as *packing*; the result of both phases represents the final DISVLIW code composed of long instructions. Each long instruction has multiple instructions that may depend on each other due to data dependencies or resource collisions.

In the VLIW instruction generation phase, the compiler first generates VLIW code from given data dependency graph where each instruction is assigned to a long instruction as shown in Figure 2. The result is a sequence of long instructions so that one long instruction can be executed per clock cycle without violating data dependences or resource constraints. Empty instruction slots within a long instruction have to be filled with *NOPs* (NOPs are depicted with a white background). In the packing phase, the compiler constructs DISVLIW code by removing nearly all LNOPs and NOPs from the generated VLIW code and by inserting dependency information to each instruction.

To store the dependency relations between instructions, each instruction format consists of an instruction I_{ij} and dependency vector D_V, which has pre-dependency D_{pre} and post-dependency D_{post}. I_{ij} refers to the j^{th} $(j=1,..,N)$ instruction within the i^{th} $(i=1,..,M)$ long instruction. D_{pre} provides information about functional units executing prior instructions that have dependencies with I_{ij}. D_{post} provides information about functional units that will execute subsequent instructions that depend on I_{ij}. D_{pre} and D_{post} are individually composed of a bit vector that has (N-1) bits, where N equals the number of functional units. To store the information to a bit vector, the compiler allocates one bit for every other functional unit. If I_{ij} depends on a prior instruction I_{lk} ($k<j$ *if* $l=i;k=1,..,n$ *if* $l<i$) being executed by functional unit F_k, the bit designating F_k in the D_{pre} is set to 1. Otherwise, it is set to zero. Although DISVLIW code contains dependency information composed of many bits, the processor can still achieve a reduction in object code size in comparison to the VLIW processor [21]. Figure 2(b) shows the example of DISVLIW code (N=4).



(a) code generation (b) code structure

Figure 2. DISVLIW code generation

3.2 DISVLIW processor implementation

Figure 3 shows the DISVLIW processor architecture. The DISVLIW processor has FU (Functional Unit) and DS (Dynamic Scheduler) pairs, a number of IQs (Instruction Queue) and DCs (Dependency Counter), a register file, an instruction cache, a data cache, and a BTB (Branch Target Buffer). IQs are placed in front of each FU. It seems like instructions within a IQ issue in order, but instructions among IQs slip with respect to each other, dynamic scheduling allows instructions in different IQs(i.e. different FUs) are synchronized by having counters (DC) at each FU. If there are *N* FUs, then each FU has a DC composed of *N-1* counters, 1 counter for every other FU. Each DC saves D_{post} of executed instructions on the associated FU. Using the DC, each DS dynamically decides whether to assign the next instruction to the associated FU, or to stall the FU due to resource collisions or data dependencies. The processor also utilizes the BTB structure for branch prediction [6,9].

Every DS checks for data dependencies and resource collisions among instructions per each cycle using both D_{pre} of the next instruction and counter values in the associated DC. In Figure 4, we assume that the DISVLIW processor has five pairs of FU and DS. In order to schedule instruction, Each DS compares D_{pre} of the next instruction to counter values in the associated DC per each cycle. If any bit in D_{pre} is set to 1, DS checks the counter in the corresponding location in the DC. If the counter is 0, it means that the execution of prior dependent instruction hasn't finished. That is, *d(i)* returns zero. Otherwise, the execution of prior dependent instruction has finished. That is, *d(i)* returns 1. After the DS confirms that the execution of all prior dependent instructions is finished (all of *d(i)* return 1), the DS decrements the counter values in corresponding location in its DC using the set bits in given D_{pre}. It is necessary to clear the D_{post} of the prior instructions from the DC before execution. Simultaneously, each DS individually assigns the next instruction to the associated FU.

As an example of Figure 4, DS_0 checks D_{pre} (1010) of the next instruction and counter values (1030) in its DC_0. Since counters in corresponding position in DC_0 are greater than 0, DS_0 decrements counters in DC_0 using set bits in D_{pre}. As soon as DC_0 turns from (1030) to (0020), then DS_0 assigns the next instruction to FU_0.

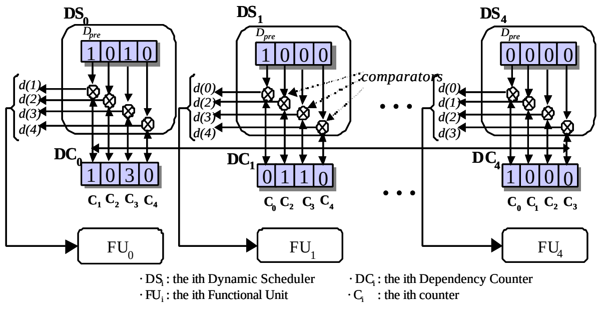


Figure 4. Dynamic scheduler units

3.3 Instruction pipeline algorithm

Each instruction on the DISVLIW processor is executed in four stages as shown in Figure 3. Each stage requires one cycle except the execution stage that requires various execution cycles according to an instruction type. In the Fetch (F) stage, the fetch unit gets one long instruction from the instruction cache each clock cycle and separates it into instructions to store IQs. If IQ is in the full state, the fetch unit cannot fetch the following long instruction, which prevents the IQ from overflowing. In the Decode/Scheduling (D/S) stage, the decode unit analyzes the next instruction at the head of each IQ. Every DS simultaneously checks for data dependencies and resource collisions using both D_{pre} of the next instruction and counter values in the its DC. If there are no data dependencies and resource collisions,

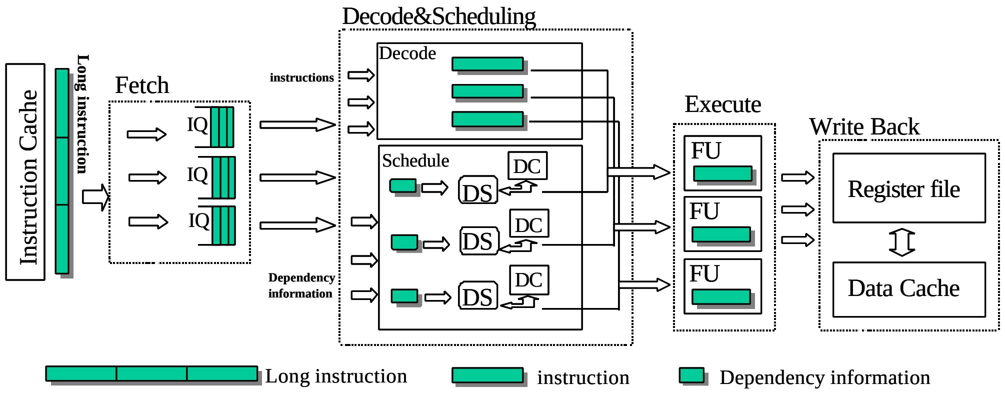
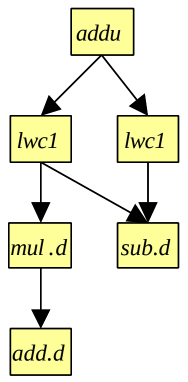


Figure 3. DISVLIW processor architecture



(a) data dependency graph

00 *addu* 11	10 *lwc1* 11	10 *lwc1* 00
10 *mul.d* 10	10 *ladd.d* 00	01 *sub.d* 00

(b) DISVLIW code

Clock Cycles	FU_0	FU_1	FU_2	D stage						EX stage					
				DC_0		DC_1		DC_2		DC_0		DC_1		DC_2	
				C_1	C_2	C_0	C_2	C_0	C_1	C_1	C_2	C_0	C_2	C_0	C_1
1	*addu*			0	0							1	0	1	0
2		*lwc1*	*lwc1*			0	0	0	0	1	0			0	1
3	*mul .d*		*sub.d*	0	0			0	0			1	0		
4		*add.d*				0	0								

(c) Execution steps according to clock cycle

· DC_i : the ith dependency counter · C_i : the ith counter · FU_i : the ith functional unit

Figure 5. Changes of DC according to instruction execution

each DS decrements counter values in its DC in order to clear the D_{post} of the prior instructions from its DC and assigns the next instruction to the associated FU. In the Execute (EX) stage, every FU executes instruction and announces to other FUs that its execution will be finished during the execution of the final cycle. To accomplish this, the FU increments counters (indicating the FU) in DCs in corresponding location using set bits in the D_{post}. Thus, every FU achieves synchronization since it decrements counter values in its DC at D/S stage and increments it at EX stage. To facilitate this, we designed the EX stage with the ability to control the D/S stage. Finally, in the Write Back (WB) stage, the results of the executed instructions are stored in the register file.

Figure 5 shows execution examples of DISVLIW code generated from Figure 5(a). FU_0 first executes instruction *addu* since the D_{pre} of *addu* is 00, and simultaneously increments the first counters (indicating FU_0) in the DC_1 and DC_2 because the D_{post} of *addu* is 11. Then, FU_1 and FU_2 individually check D_{pre} bits of the next instruction *lwc1* and the counter values in its DC_1 and DC_2. If both of them are greater than 0, FU_1 and FU_2 decrements the first counter value in its DC_1 and DC_2 using set bits in the D_{pre}. It is necessary to clear the D_{post} of the instruction *addu* from each DC before the execution of FU_1 and FU_2. Then, FU_1 and FU_2 simultaneously begin the execution of instructions *lwc1*.

3.4 Loop performance

The DISVLIW processor can significantly reduce the execution cycles of applications containing loops since the processor can simultaneously schedule the instructions fetched from different iterations in the loop.

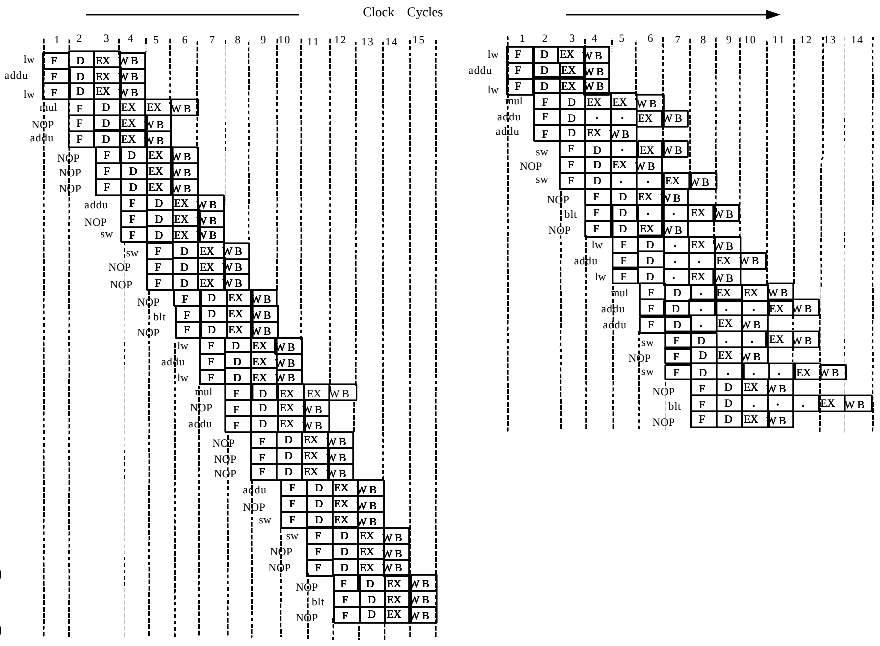
Figure 6 shows execution images of the VLIW and the DISVLIW processors that execute the ith iteration and the (i+1)th iteration of the loop. We generate VLIW code of Figure 6(a-2) and DISVLIW code of Figure 6(a-3) from the MIPS code of Figure 6(a-1). The MIPS code is to initialize integer array. A long instruction has three instructions each execution cycle of which is one cycle except that those of *mul* are two cycles. Every instruction is executed in the following four pipeline stages: F, D, EX, and WB. Figure 6(b) shows an execution image of the VLIW processor that executes Figure 6(a-2). The VLIW processor requires 15 cycles to execute two iterations of the loop. Figure 6(c) shows an execution image of the DISVLIW processor that executes Figure 6(a-3). Intructions *addu* and *sw* are simultaneously executed for the 6th cycle although the instructions are individually fetched at the 2nd and the 3rd clock cycle. Besides, instructions *blt* and *lw* are simultaneously executed at the 8th cycle although the instructions are fetched from different iterations. The DISVLIW processor requires 14 cycles to execute two iterations of the loop.

From the above observation, we know that the DISVLIW processor is more effective than the VLIW processor in applications containing loops. This is because the DISVLIW processor can simultaneously schedule instructions fetched from different iterations of the loop as long as the instructions don't depend on each other. Due to this feature, the larger the number of loop iterations the DISVLIW processor gets reduced execution cycles in proportion to those, when compared with the VLIW processor. Although is not shown in the Figure 6, the DISVLIW processor can reduce fetch cycles because the DISVLIW processor can simultaneously fetch a number of instructions that may depend on each other in a long instruction. The DISVLIW processor also has relative low cache miss rates due to its reduced object code [21].

```
$32:      lw     $14, 20($sp)
          mul    $15, $14, 4
          addu   $24, $sp, 0
          addu   $25, $15, $24
          sw     $14, 0($25)
          lw     $8, 20($sp)
          addu   $9, $8, 1
          sw     $9, 20($sp)
          blt    $9, 5, $32
```
(a-1)

```
$32:  lw      addu     lw
      mul     NOP      addu
      NOP     NOP      NOP
      add     NOP      sw
      sw      NOP      NOP
      NOP     blt      NOP
```
(a-2)

```
$32: 00lw00     00addu00   00lw00
     00mul10    10addu01   00addu00
     00sw00     00NOP00    01sw01
     00NOP00    01blt00    00NOP00
```
(a-3)



(a) object code (b) VLIW (c) DISVLIW

Figure 6. Comparison of loop execution

4. Experiment and analysis

4.1 Simulation system

The performance of the DISVLIW processor was accurately analyzed using a simulator testbed. Using the simulator testbed, we measured the total number of execution cycles for various numerical benchmark applications on the VLIW, the SVLIW, the DISVLIW processor architectures.

The simulator starts with the MIPS assembler, a Mipspro C++ compiler using optimization flag –O and assembly code generation flag –S, generating MIPS R4000 assembly code by compiling a C-language benchmark applications [17]. Next, the macro expander inputs the MIPS R4000 assembly code while simultaneously expanding macros. The Macro expander then passes the assembly code to each parallelizer. Three parallelizers, each of which is associated with a unique processor, are designed with the ability to exploit ILP across basic blocks using compile techniques such as register renaming, branch prediction, invariant code motion from loops, common subexpression elimination, function inlining, and loop unrolling [6,9,10,11]. Generally, the VLIW's effectiveness depends on how good the compiler is: the VLIW processor using a compiler with higher ILP will produce better performance, and will get higher cache hit rates because of the reduced object code size. However, the DISVLIW processor accomplishes this same goal since it constructs object code using the VLIW code. From now on, VLIW$_c$, VLIW$_s$, and VLIW$_{DIS}$ individually mean VLIW, SVLIW, and DISVLIW code, respectively. The parallelizers then use the MIPS code to generate parallelized code for its processor simulator and then translate this parallelized code into object code.

For these experiments, processor speedups are calculated by dividing the total number of execution cycles of the VLIW processor by the total number of cycles of the DISVLIW or the SVLIW processor. In the Table 1, the fixed parameters and the variable parameters are also shown. Except when stated otherwise, the default values were used in the simulations.

Table 1. Input parameters

Fixed Parameters	
Processor pipeline	Four-stage(F,D,EX, WB)
Decoded instruction size	4 bytes
integer instruction latency	1 cycle
Floating point instruction latency	1~32 cycle(depend on instruction)
Data cache size	Perfect(no miss penalty)
cache mapping method	direct mapped
cache replacement policy	LRU(Least Recently Used)
Variable Parameters	
Parameter	Default Value
A number of integer unit	2
A number of floating-point unit	2
next long instruction miss penalty	4
Instruction cache size	16k bytes

Table 2 provides the benchmark applications and the proportion of I/F (Integer instructions and Floating-point instructions) of each benchmark application. These applications all use double precision.

Table 2. Benchmark applications

Benchmarks	Description	I/F(%)
LIVERMORE	Do loop for various kernel operations	65.3/34.7
M M	Matrix Multiply using floating point instructions	68.4/31.6
CLINPACK	Set of linear algebra subroutine	75.7/24.3
WHETSTONE	Loop instructions for arithmetic computation	65.6/34.4
FFT	Matrix Fourier Transformation	43.3/56.7

Table 3 tabulates the ratios of object code size of the VLIW to both the SVLIW and DISVLIW processors for each benchmark. In this experiment, we chose numerical benchmarks that have a high proportion of floating-point instructions. This choice was appropriate because the DISVLIW processor is more effective given dynamic instruction scheduling and reduced object code size. Even though $VLIW_{DIS}$ contains many bits of dependency information, Table 3 indicates that $VLIW_{DIS}$ averages 45% smaller than $VLIW_c$ and is almost the same size as $VLIW_s$.

Table 3. Comparison of object code size

Benchmarks	$VLIW_C$	$VLIW_S$	$VLIW_{DIS}$
LIVERMORE	1	0.723	0.725
M M	1	0.568	0.591
CLINPACK	1	0.673	0.673
WHETSTONE	1	0.438	0.385
FFT	1	0.385	0.400
AVERAGE	1	0.557	0.554

4.2 Experimental results

Figure 7 shows the speedup of the DISVLIW processor over the VLIW (or the SVLIW) processor using different scheduling strategies. In order to evaluate scheduling performance only, we ignore cache effects such as cache miss rates. We assume that an instruction cache size is perfect (no miss penalty). In this experiment, we reduced the number of loop iterations in each benchmark application to reduce simulation duration.

Figure 7 illustrates that even though we assume a cache with a zero miss rate, the DISVLIW's performance is still 9%-15% higher than that of the VLIW processor regardless of benchmark application. We have the DISVLIW's scheduling strategies to thank for this speedup. This scheduling decreases the waiting time to process a set of long instructions when compared to the VLIW and SVLIW processors. By contrast, the VLIW and the SVLIW processor can't execute pending long instructions until the execution of all instructions in the previous long instruction finishes. In Figure 7, the SVLIW processor shows same performance when compared to the VLIW processor.

Figure 8 illustrates the impact of cache size on speedups of the DISVLIW processor with respect to both the SVLIW and VLIW processors. We varied the instruction cache size from 4k bytes to 32k bytes to compare performance according to changes in cache size. The speedups of the DISVLIW and the SVLIW processors were measured relative to the VLIW processor regardless of cache size. In this experiment, we also reduced the number of loop iterations in each benchmark to reduce simulation duration.

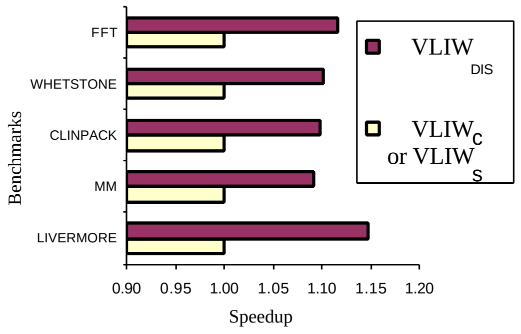


Figure 7. Comparison of speedups for different scheduling strategies

These results indicates that the DISVLIW processor is faster than the SVLIW processor regardless of both benchmark applications and cache size. This is due to the DISVLIW's unique instruction scheduling strategies. Another factor is the DISVLIW's reduced object code size, which decreases average fetch cycles and also reduces cache misses, as shown in Table 3. Figure 8 indicates that larger cache sizes result in smaller speedup differences between the VLIW and DISVLIW processors. At smaller cache sizes, the VLIW's performance is slower due to higher cache miss rates. Unlike the VLIW, the DISVLIW's performance is not as sensitive to cache size due to its smaller object code. But as cache size increases, performance difference decreases and the VLIW's performance approaches that of the DISVLIW. Yet, even assuming perfect cache, the DISVLIW is still faster than the VLIW's.

Overall, the performance of DISVLIW processor is faster than the VLIW and the SVLIW processors over a wide range of cache size and across various numerical benchmark applications. We attribute these performance gains to the balanced benefits of compile-time and run-time parallelization, dynamic instruction scheduling, and size reduction of object code as previously described.

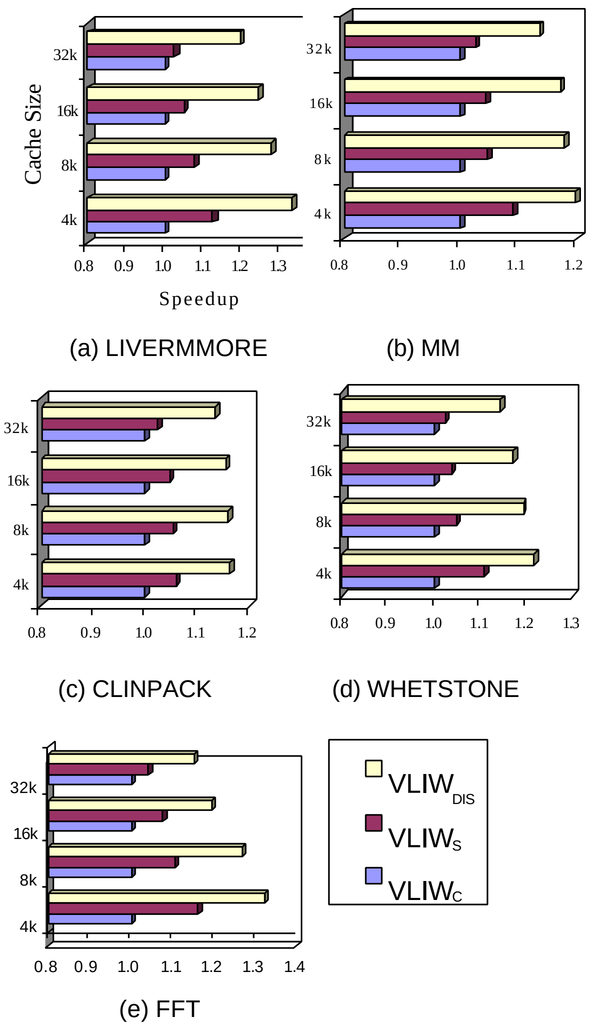


Figure 8 Comparison of the speedups according to changes in cache sizes

5. Conclusion

This paper describes a new ILP processor architecture referred to as Dynamically Instruction Scheduled VLIW (DISVLIW). The DISVLIW processor is a hybrid architecture that has inherited features as ILP exploitation at compile-time of the VLIW processor and dynamic scheduling at run-time of the superscalar processor. The experimental evaluations presented in this paper have shown that the DISVLIW processor achieves a high speedup over the VLIW and the SVLIW processors for a wide range of cache sizes and across various numerical benchmark applications. These performance gains of the DISVLIW processor result from dynamic instruction scheduling and size reduction of object code.

The DISVLIW processor architecture opens several new avenues of research. Optimization of dependency information within object code, DISVLIW compilers, and scalability of functional units in the system are just a few examples that will be investigated in future work. Particularly, our research will focus on optimization and management of the dependency information required in order to achieve synchronization.

6. Acknowledgments

This work was supported by postdoctoral fellowship program from Korea Science & Engineering Foundation (KOSEF)

7. References

[1] Ken Sakamura, '21st-century microprocessors,' IEEE Micro, pp.10~11, July/Aug 2000.

[2] Michael J. Flynn, Computer Architecture, Jones and Bartlett Publishers, 1995

[3] P. P. Chang, D. M. Lavery, S. A. Mahlhe, W. Y. Chen, and Wen-Mei. W. Hwu, "The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors," IEEE Transactions on Computers, Vol. 44, No. 3, pp. 353~370, March 1995.

[4] Shyh-Kwei Chen, W. Kent Fuchs, and Wen-Mei W. Hwu, "An analytical approach to scheduling code for superscalar and VLIW architectures," Proc. International Conference on Parallel Processing., pp. I258-I292, 1994.

[5] J. A. Fisher, The VLIW machine: A multiprocessor for compiling scientific code, IEEE Transactions on Computers, pp. 45~53, July 1984.

[6] Barry Fagin, "Partial Resolution in Branch Target Buffers'" IEEE Computers, Vol. 46, No. 10, October 1997

[7] Joseph A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computers., Vol. C-30, No. 7, pp. 478~490, July 1981.

[8] Roger Espasa and Mateo Valero, "Exploiting instruction- and data-level parallelism," IEEE Micro, Vol. 17, No. 5, Sept 1997.

[9] S. A. Mahlke, R. E. Hank, J. E. M. McCormick, D. I. August, and W. W. Hwu, "A Comparison of Full and Partial Predicated Execution Support for ILP Processors," Proceedings of the 22th international Symposium on Computer Architectures, pp. 138~150, 1995.

[10] Thomas M. Conte and Sumedh W. Sathaye, "Dynamic Rescheduling: A technique for object code compatibility in VLIW architecture,"

Proceedings of 28th International Symposium on Microarchitecture, March 1995.

[11] Arthur Abnous and Nader Bagherzadeh, "Pipelining and bypassing in a VLIW processor," Transactions on Parallel and Distributed Systems, Vol. 5, No. 6, pp. 658~664, June 1994.

[12] T. M. Conte and S. W. Sathaye, "Dynamic rescheduling; a technique for object code compatibility in VLIW architecture," proceedings of the 28th Annual International Symposium on Micro architecture, pp. 208~218, March 1995.

[13] Kevin W. Rudd and Michael J. Flynn, "Instruction-level parallel processors-dynamic and static scheduling tradeoffs," Proc. The Second AIZU International Symposium on Parallel Algorithms/ Architecture Synthesis., pp. 74~80, March 1997.

[14] Shusuke Okamoto and Masahiro Sowa, "Hybrid processor based on VLIW and PN-Superscalar," Proc. DPTA'96 International Conference., pp. 623~632, 1996.

[15] Sunghyun Jee and Sukil Kim, "Performance analysis of caching instructions on SVLIW processor and VLIW processor," Journal IEEE Korea Council, Vol. 1, No. 1, December 1997.

[16] Susan J. Eggers, Joel S. Emer, Henry M. Levy, and Jack L. Lo, "Simultaneous multithreading," IEEE Micro, Vol. 17, No. 5, Sep 1997.

[17] MIPS R4000 Microprocessor User's Manual, MIPS Computer Systems, Inc., 1991.

[18] B. R. Rau, "Dynamically scheduled VLIW processors," Proceedings the 26th Annual International Symposium on Microarchitecture, pp. 138~148, Mach 1997.

[19] T. Hara and H. Ando, "Performance comparison of ILP machines with cycle time evaluation," Proceedings of the 23rd Annual International Symposium on Computer Architecture, pp. 213~224, Mach 1996.

[20] A. F. de Souza and P. Rounce, "Dynamically Scheduling VLIW instructions," Journal of Parallel and Distributed Computing, pp. 1480~1511, 2000.

[21] Sunghyun Jee and Sukil Kim," A Design of A Processor Architecture for Codes With Explicit data Dependencies," Proc. tenth SIAM Conference on Parallel Processing for Scientific Computing 2001, March 2001.

[22] Patterson D. A., and J. L Hennessy, *Computer Architecture A Quantitive Approach 2nd edition*, Morgan Kaufmann, pp. 240~261, 1996.

[23] Intel, http://www.intel.com/ia64, *IA-64 Architecture Software Developer's Manual, Volume 1:IA-64 Application Architecture, Revision 1.1,* July 2000.

[24] Intel, http://www.intel.com/ia64, *Itanium Processor Michroarchitecture Reference for Software Optimization*, Aug. 2000.