VLIW 명령어의 동적 스케줄링을 위한 컴파일러와 프로세서간 상호보완

(Compiler Processor Trade-offs for Dynamic Scheduling of VLIW Instructions)

지 승 현 †

(Sunghyun Jee)

요 약 본 논문에서 제안한 DISVLIW(Dynamically Instruction Scheduled VLIW) 프로세서 구조는 자료종속성 정보를 이용하여 VLIW(Very Long Instruction Word) 명령어들을 동적으로 스케줄링할 수 있다. 이러한 동작을 수행하기 위해서, DISVLIW 프로세서는 연산처리기와 동적스케줄러의 쌍들로 구성되었다. VLIW 명령어들의 동적 스케줄링, 컴파일시간과 실시간의 균등한 작업분배, 명령어내의 명백한 병렬성 표현 등의 특징은 성능향상에 중요한 영향을 미쳤다. DISVLIW 프로세서 구조의 시뮬레이션 결과, 다양한 벤치마크들과 캐쉬메모리 사이즈들을 이용할 경우에도 DISVLIW 프로세서 구조가 VLIW 프로세서 구조에 비하여 항상 높은 성능향상을 가짐을 확인하였다.

키워드 : ILP, VLIW, DISVLIW, 동적 스케줄링, 동적스케줄러

Abstract This paper describes a processor architecture, named Dynamically Instruction Scheduled VLIW (DISVLIW). The DISVLIW processor architecture is designed for dynamic scheduling VLIW instructions using dependency information. The DISVLIW instruction format is augmented to allow dependency bit vectors to be placed in the same VLIW word. The DISVLIW processor dynamically schedules each instruction in long instructions using functional unit and dynamic scheduler pairs. Features such as explicit parallelism, balanced scheduling effort, and dynamic scheduling of VLIW instructions can be used to provide a sound frustructure for supercomputing. We simulate the DISVLIW processor architecture and show that the DISVLIW processor performs significantly better than the VLIW processor for a wide range of cache sizes and across numerical benchmark applications.

Key words : ILP, VLIW, DISVLIW, Dynamic instruction scheduling, Dynamic scheduler

1. INTRODUCTION

Recent high performance processors have depended on Instruction Level Parallelism(ILP) to achieve high execution speed[1-12]. ILP processors achieve their high performance by causing multiple operations to execute in parallel using a combination of compiler and hardware techniques.

The superscalar processor executes all parallel processing steps directly in hardware at run-time [2-4]. Therefore, the superscalar processor uses complex hardware units and the object code is simply the same as sequential code. Due to the unbalanced optimization between compile-time and run-time parallelization, the superscalar processors typically have a performance bottleneck from excessive run-time overhead. On the other hand, the VLIW processor constructs a parallelized long instruction sequence at compile-time[2-6]. Therefore, the VLIW processor can be implemented using simple hardware units, but object code is more complex since it contains groups of long instructions each of which is composed of a number of instructions. The VLIW processor has performance bottlenecks due to the unoptimized large object code and compulsory instruction scheduling[4,6]. To balance a load between compile-time and run-time

† 정 회 원 : 백석대학 컴퓨터학부 교수
jsh@ccfs.ac.kr
논문접수 : 2003년 6월 17일
심사완료 : 2003년 12월 30일

on the above processors, 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[10,11]. The SVLIW processor also has a performance limitation similar to the VLIW processor due to static scheduling and biased parallelism exploitation at only compile time.

Another technique to overcome the VLIW performance limitation problem is to allow the compiler to exploit high ILP using Explicit Parallel Instruction Computing(EPIC)[7,8]. The basic EPIC principle is that the compiler should be able to indicate the inherent parallelism of programs explicitly in the instruction sequence, rather than obliging the processor to reconstruct it from a particular sequence of serial instructions. By making use of powerful features to generate high-performance code, the IA-64 architecture allows the compiler to exploit high ILP using EPIC techniques [7,8]. IA-64 processor architecture implementing this concept is the processor architecture where the compiler is responsible for efficiently exploiting the available ILP and keeps the executions busy [8]. Instead of the merits, the IA-64 processor has performance limitations due to static instruction scheduling and the difficulty of complicated compiler design. 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. In the DISVLIW processor, the instruction format is augmented to allow dependency bit vectors to be placed in the same VLIW word. Dependency 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 designing of the DISVLIW processor results in four positive characteristics. First, the DISVLIW compiler can maximize the use of current VLIW compiler techniques to generate object code for DISVLIW. Second, the DISVLIW processor can get higher cache effects, high cache hit ratio or reduced instruction fetch time, due to the reduced object code size. Third, the DISVLIW processor can dynamically schedule each instruction using dependency information in the instruction. Fourth, the task of finding parallelism is balanced between compile time and run time. Such features can reduce the total number of execution cycles of the DISVLIW processor better than those of other ILP processors that statically schedule long instructions.

2. THE DISVLIW ARCHITECTURE

2.1 Compiler Design for DISVLIW

The DISVLIW compiler consists of multiple phases occurring in the order as shown in Figure 1. The problem of optimal DISVLIW code generation can mainly be subdivided into two phases. We refer to the first phase as *VLIW code generation and to the second phase as dependency information insertion* The first phase, *VLIW code generation*, contains two schedulers: the software pipeliner for targeted cyclic regions and the global code scheduler for all remaining regions. After the first phase, the result is VLIW code composed by a sequence of long instructions so that each long instruction can be executed per clock cycle without violating data dependences or resource constraints. Empty instruction slots within long instruction have to be filled with *NOPs*. We can maximize the use of the VLIW compiler techniques for the first phase. The second phase, *dependency information insertion*, consists of VLIW code compactor and dependency inserter. The VLIW code compactor compacts the VLIW code by removing nearly all LNOPs and NOPs from the generated VLIW code, and the dependency inserter then inserts dependency information into each instruction in the compacted VLIW code. It is necessary to express

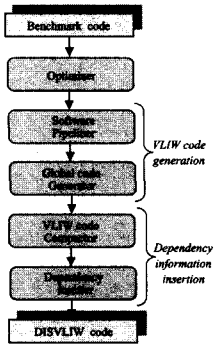

Figure 1 DISVLIW compiler phases

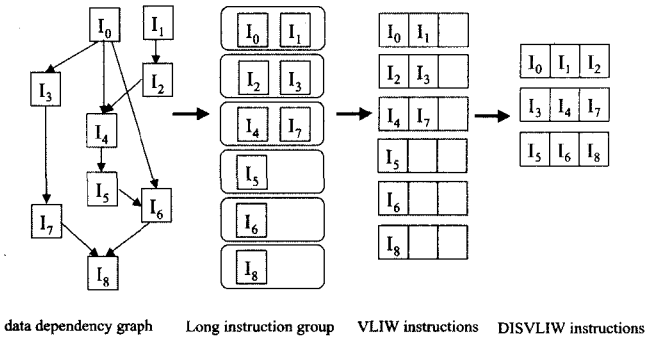

Figure 2 Example of DISVLIW code generation

explicit parallelism within the DISVLIW object code for synchronization. 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.

Figure 2 shows how to generate DISVLIW code from the given data dependency graph. In the data dependency graph, a node represents an instruction Ii that equals i^{th} instruction and a directed 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 that a long instruction has three instructions. At the first step, the compiler explicits long instruction groups from the data dependency graph. Each instruction group is composed of instructions that are executed at the same time. The compiler then generates VLIW code from the long instruction groups. In the VLIW code, unused instruction slots within a long instruction have to be filled with NOPs(each NOP is depicted with a white background). The compiler then fills available instructions into the unused instruction slots and finally inserts dependency information into each instruction.

DISVLIW instruction format consists of an instruction I_{ij}, 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. To store the information as 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[10,11].

2.2 Processor Design for DISVLIW

In order to dynamically schedule each instruction using dependency information, the DISVLIW processor require additional hardware units which can analyze data dependencies and resource collisions among instructions using the dependency information at run time, and which manage dependency information of the executed instructions for synchronization.

The symbolic diagram of the DISVLIW processor architecture is shown in Figure 3. 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). Each IQ stores an instruction(separated into a long instruction) in its own tail, and individually provides an instruction and dependency information stored in its own head to decode unit and the associated DS. IQs are placed in front of each FU. It seems like instruc-

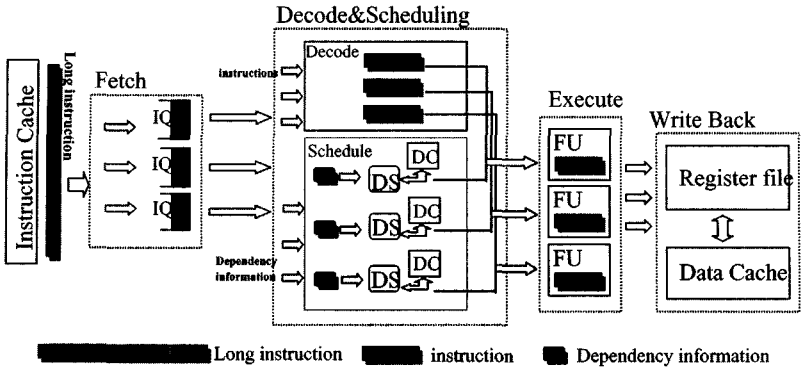

Figure 3 DISVLIW processor architecture

tions within IQ issue in order, but instructions among IQs slip with respect to each other. Each DC saves D_{post} of executed instructions on the associated FU. Using the DC values, 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. Dynamic scheduling allows instructions in different IQs(i.e. different FUs) are synchronized by having 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. The processor also utilizes the BTB structure for branch prediction[2].

2.2.1 The dynamic scheduler units.

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 dynamically schedule instruction, each DS compares D_{pre} of the next instruction to counter values in the associated DC per each cycle. If i^{th} bit in D_{pre}, D^i_{pre}, is set to 1, the DS checks C_i in the corresponding location in the DC. If C_i 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. In order to assign the next instruction to the associated FU, the DS should confirm that the execution of all prior dependent instructions is finished (all of d_i return 1).

check signal of D^i

$$= d_0.d_1...d_{i-1}.d_{i+1} \ ... \ d_{N-1}$$

$$= (D^0_{pre} \otimes C_0).(D^1_{pre} \otimes C_1)...(D^{i-1}_{pre} \otimes C_{i-1}).(D^{i+1}_{pre} \otimes C_{i+1})...(D^{N-1}_{pre} \otimes C_{N-1})$$

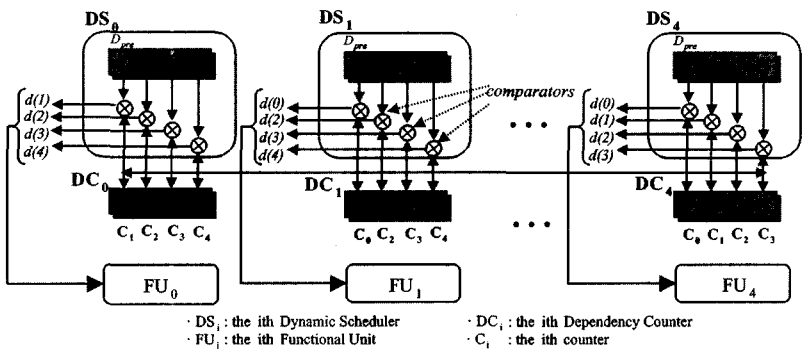

Figure 4 Dynamic scheduler units

The equation below represents the logic for DS_i necessary to check the dependency between instructions for dynamic scheduling: If check signal is 1, the DS_i assigns the next instruction to FU_i. Otherwise, DSi have waited until the the data dependencies or resource collision between instructions are solved. The operator "⊗" means binary vector comparison. $D^{i-1}_{pre} \otimes C_{i-1}$ evaluates to true (That is, 1) if both of D^{i-1}_{pre} and C_{i-1} are 0 or C_{i-1} is greater than D^{i-1}_{pre}(when D^{i-1}_{pre} is set to 1). the operator "·" means logic *and*. After assigns the next instruction, the DS_i simultaneously 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 next execution.

2.2.2 Instruction pipeline stages.

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, separates it into instructions, and then stores each instruction to the associated IQ. 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, each DS assigns the next instruction to the associated FU and simultaneously decrements counter values in its DC in order to clear the D_{post} of the prior instructions from its DC. 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}. That is, every FU can achieve 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.

The DISVLIW processor manipulates the BTB for branch instruction. The BTB provides the answer before the current instruction is decoded and therefore enables fetching to begin after IF-stage. The BTB provides the branch target if the prediction is a taken direct branch (for not taken branches the target simply is *PC* (Program Counter) +1). The DISVLIW processor duplicates the values of all DCs and the register files in temporary storag as soon as a prediction is taken. Then the DISVLIW processor has updated the values of DCs and the register files in the temporary storage according to result values of the executed instructions. When the predicate is true, The DISVLIW processor duplicates the values of DCs and the register files in the temporary storage into original DCs and register files. Otherwise, the DISVLIW processor clears the temporary storage and also removes the instructions of all IQs since the instructions were fetched after mispredicted instructions. Finally, the processor updates the branch information in BTB according to result of the prediction.

2.3 Dynamic scheduling strategies

Figure 5 shows dynamic execution examples of the DISVLIW code. We assume that every processor has three untyped functional units that can execute any instruction and a long instruction has three instructions. The DISVLIW compiler generates the DISVLIW code from the given simple assembly code as shown in Figure 5a). From the DISVLIW code of Figure 5(b), we know that instruction *sub.d* within the 2nd long instruction depends on previous instruction *lwc1* executed by FU_1 since the first bit in D_{pre} is set to 1. We also know that *sub.d* also has dependent relations with following instruction *add.d* executed by FU_1 because the first bit in D_{post} is set to 1. Figure 5(c) shows the changes of DC values according to the execution of DISVLIW code in Figure 5(b). FU_0 first executes instruction *lwc1* since D_{pre} of *lwc1* is

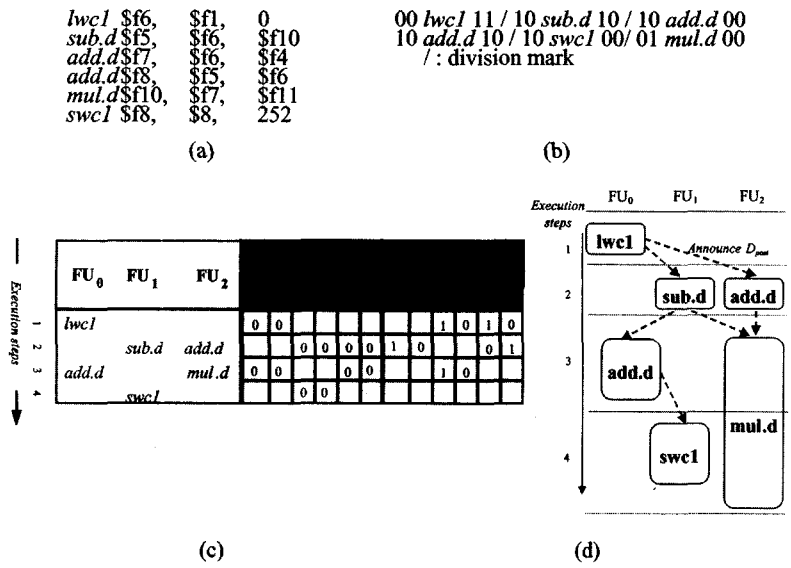

Figure 5 Example of dynamic scheduling DISVLIW instructions

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

The main insight of this example 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. But the VLIW or the SVLIW 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[3,6, 9-11].

3. EXPERIMENT METHODOLOGY

The performance of the DISVLIW processor was accurately analyzed using a simulator testbed. Using a 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 assembly code by compiling a C-language benchmark applications. Next, the macro expander inputs the MIPS assembly code while simultaneously expanding macros into static instructions. Macro means a dynamic instruction that consists of a lot of static instructions. The Macro expander then passes the expanded 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[2,3,6,9].

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.

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. Figure 6 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. The proportion of I/F is an important factor to compare what processor architecture is effective according to changes in the proportion of I/F. By the proportion of I/F, we can also know the proportion of static/dynamic instructions in this experiment since static instruction means an integer instruction that requires a static instruction cycle and most floating-point instructions implies dynamic instructions that require variable instruction cycles.

Figure 7 provides 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

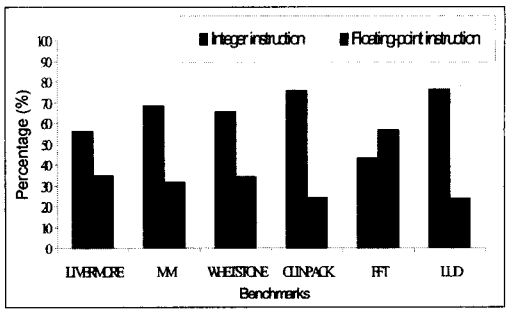

Figure 6 Ratios of integer/floating-point instructions

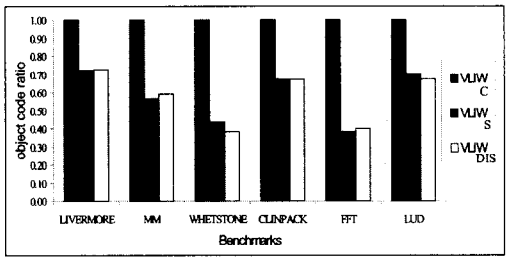

Figure 7 Relative ratios of object code size

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, Figure 7 indicates that $VLIW_{DIS}$ averages 45% smaller than $VLIW_C$ and is almost the same size as $VLIW_S$.

4. EXPERIMENTS

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 8k bytes to 32k bytes to compare performance according to

Table 1 Input Parameters

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

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. These results indicate 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 high cache hit ratio due to the DISVLIW's reduced object code size, which decreases average fetch cycles and also reduces cache misses, as shown in Figure 7. Figure 8 also indicates that larger cache sizes result in smaller speedup differences among the VLIW, the SVLIW, 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 compare with VLIW's object code. But as cache size increases, performance difference decreases and the VLIW's performance approaches that of the DISVLIW. For example, the DISVLIW's performance is 20% higher than that of the VLIW processor on the experiment of LIVERMORE benchmark in 8kbyte cache size. The DISVLIW's performance is 16% higher than that of the VLIW processor on the same benchmark in 32kbyte cache size. As cache size increases, performance difference slowly decreases. Yet, even assuming perfect cache, the DISVLIW is still faster than the VLIW's because of dynamic scheduling strategies.

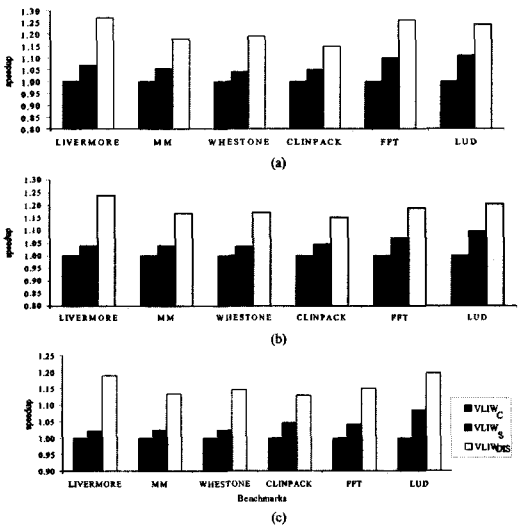

Figure 8 Comparison of speedup according to changes in Cache Sizes; (a) cache size=8k bytes, (b) cache size=16kbytes, (c) cache size=32kbytes

Figure 9 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 9 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 9, the SVLIW processor shows same performance when compared to the VLIW processor.

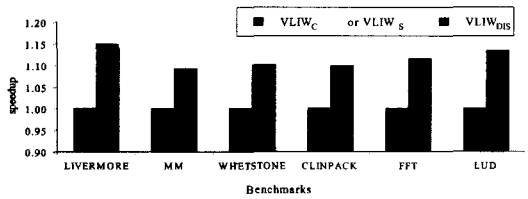

Figure 9 Comparison of speedup according to different scheduling strategies

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.

References

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

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

[3] 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.

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

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

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

[7] Intel, IA-64 Architecture Software Developer's Manual, Volume 1:IA-64 Application Architecture, Revision 1.1, July 2000.

[8] Intel, Itanium Processor Microarchitecture Reference for Software Optimization, Aug. 2000.

[9] P. Faraboschi, J.A. Fisher, and C. Young, "Instruction Scheduling for Instruction Level Parallel Processors," Proceedings of the IEEE Microprocessor Architecture&Compiler Technology, Vol. 89, No. 11, pp. 1638~1659, Nov 2001.

[10] Sunghyun Jee and Kannappan Palaniappan, "Performance Evaluation For a Compressed-VLIW Processor," the 17th ACM Symposium on Applied Computing, March 2002.

[11] Sunghyun Jee and Kannappan Palaniappan, "Compiler Processor Tradeoffs for DISVLIW Architectures," the 6th Workshop on Interaction between Compilers and Computer Architectures, IEEE CS Press, May 2002.

[12] Michael J. Bass and Clayton M. Christensen, "The Future of the Microprocessor Business," IEEE SPECTRUM, pp. 34~39, Apr 2002.

지 승 현

1999년 3월~현재 백석대학 컴퓨터학부 교수로 재직중. 2002년 3월~2003년 2월 미주리 주립대학교 연구교수(University of Missouri(U.S.A)). 2001년 1월~2002년 3월 미주리 주립대학교(University of Missouri(U.S.A)) 박사후 연수과정 수료. 1996년 3월~2000년 2월 충북대학교 전자계산학과 이학박사학위 취득. 1993년 3월~1995년 2월 충북대학교 전자계산학과 이학석사학위 취득. 1988년 2월~1993년 2월 충북대학교 전자계산학과 이학학사학위 취득