Compiler Processor Tradeoffs for DISVLIW Architecture

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

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

Abstract

The Dynamically Instruction Scheduled VLIW (DISVLIW) processor architecture is designed for balancing scheduling effort more evenly between the compiler and the processor. The DISVLIW instruction format is augmented to allow dependency bit vectors to be placed in the same VLIW word. Dependency bit vectors are added to each instruction format within long instructions to enable synchronization between prior and subsequent instructions. The DISVLIW processor dynamically schedules each instruction in long instructions using functional unit and dynamic scheduler pairs. Each dynamic scheduler dynamically checks for data dependencies and resource collisions while scheduling each instruction. Features such as explicit parallelism, balanced scheduling effort, and dynamic scheduling can be used to provide a sound frastructure for supercomputing. We simulate the DISVLIW 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.

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 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 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 difference 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 describes the compiler support for the DISVLIW, Section 3 describes the processor hardware support of the DISVLIW. In Section 4, we evaluate a performance of the DISVLIW processor and conclusion follows in Section 5.

2. Compiler support for DISVLIW

To dynamically schedule instructions, the DISVLIW instruction format is augmented to allow dependency information to be placed in the same VLIW instruction. Dependency information is required for synchronization between prior and subsequent instructions.

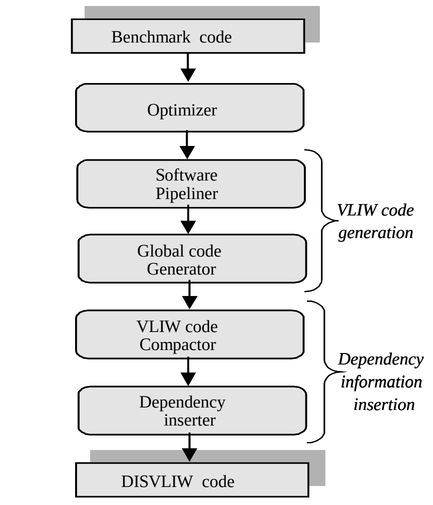
Figure 1(a) shows the phases for the DISVLIW code generation. The problem of optimal DISVLIW code generation can mainly be subdivided into two phases. In the remainder of this paper we will refer to the first phase as *VLIW code generation* and to the second phase as *dependency information insertion*; 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 code generation phase, the compiler first generates VLIW code from the benchmark program. 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*. In the dependency information insertion phase, the compiler constructs the DISVLIW code by removing nearly all LNOPs and NOPs from the generated VLIW code and by inserting dependency information into each instruction.

To store the dependent relations between instructions, each instruction format consists of an instruction I_{ij} and dependency information 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} $(k<j \text{ if } l=i; k=1,..,n \text{ if } l<i)$ 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 the 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).

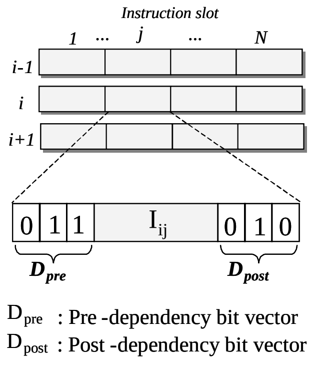
3. Processor support for DISVLIW

3.1 Processor implementation

Figure 2(a) 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



(a) code generation phase order


Instruction slot
1 ... j ... N
i-1
i
i+1
0 1 1 I_ij 0 1 0
D_pre
D_post
D_pre : Pre -dependency bit vector
D_post : Post -dependency bit vector

(b) code structure

Figure 1. DISVLIW code generation

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 other FUs. 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].

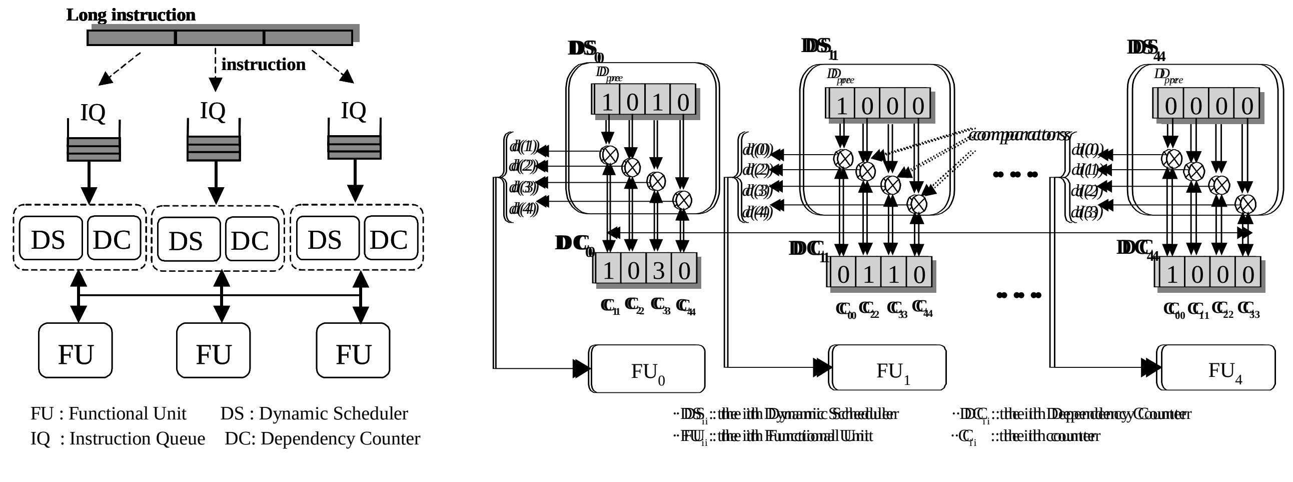
Figure 2(b) shows DS architecture. We assume that the DISVLIW processor has five pairs of FU and DS. Every DS checks for data dependencies and resource collisions among instructions using both D_{pre} of the next instruction and the associated DC. If any bit in D_{pre} is set to 1, DS checks the counter 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 2(b), 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 $FU_{0.}$

3.2 Instruction pipeline stages

An instruction is executed in four stages as shown in Figure 2(a). 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 each IQ. Every DS simultaneously checks for data dependencies and resource collisions using both D_{pre} of the next instruction and counter in the its DC. If there are no dependencies and resource collisions, each DS decrements counter values in its DC 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. In the Write Back (WB) stage the results of the execution are stored in the register file.

The DISVLIW processor manipulates the BTB for predication of 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



(a) DISVLIW processor architecture (b) dynamic scheduler units

Figure 2. DISVLIW processor architecture

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 predication 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 predication.

4. Dynamic scheduling strategies

Figure 3 shows the execution examples of the DISVLIW code. The DISVLIW compiler generates an object code from the given data dependency graph at compile time. In the data dependency graph, a node represents an 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 a long instruction has three instructions. From the DISVLIW code of Figure 3(b), we know that instruction *sub.d* within the 2nd long instruction depends on previous instruction *lwc1* executed by the 2nd functional unit 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 the 2nd functional unit because the first bit in D_{post} is set to 1.

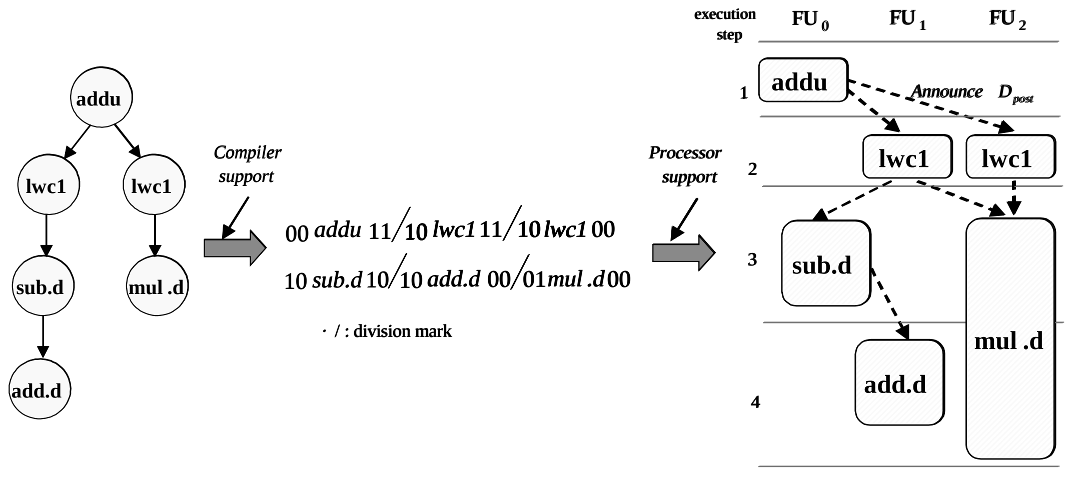
Figure 3(c) shows the execution images of the given DISVLIW code. FU_0 first executes instruction *addu* since D_{pre} of *addu* is 00, and simultaneously increments the first counters (indicating FU_0) in the DC_1 and DC_2 because D_{post} of *addu* is 11. Then, FU_1 and FU_2 individually check D_{pre} of instruction *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 D_{pre}. It is necessary to clear D_{post} of *addu* from each *DC* before the execution of FU_1 and FU_2. Then, FU_1 and FU_2 simultaneously begin the execution of *lwc1*.

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 [20,21,25,26].

5. Experiment and analysis

5.1 Simulation system

The performance of the DISVLIW processor was accurately analyzed using a simulator testbed. Using the simulator testbed of Figure 4, we measured the total number of execution cycles for various numerical benchmark applications on the VLIW, the SVLIW, the DISVLIW processor architectures. The simulator has the compiler part and simulator part. In the compiler part, each of the parallelizer is designed with the ability to exploit ILP across basic blocks using compile techniques such as branch prediction and loop unrolling [6,9~11]. Generally, the VLIW's effectiveness depends on how good the compiler is: But, the DISVLIW processor accomplishes this same goal since it



(a) data dependency graph (b) an example of DISVLIW code (c) execution steps

Figure 3. Execution of the DISVLIW code

constructs object code using the VLIW code. From now on, $VLIW_C$, $VLIW_S$, and $VLIW_{DIS}$ individually mean VLIW, SVLIW, and DISVLIW code.

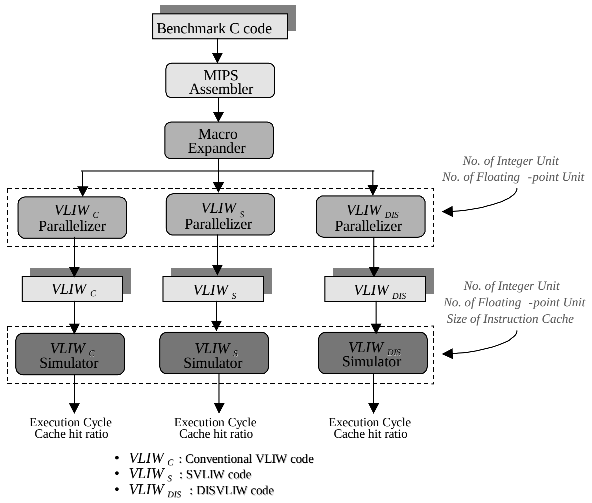


Figure 4. Simulator testbed

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, $VLIW_{DIS}$ is not larger than $VLIW_S$, and $VLIW_C$ [26].

5.2 Experimental results

Figure 5 shows the speedup of the DISVLIW processor over the VLIW (or SVLIW) processor using different scheduling strategies. 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) and each processor has 2 integer units and the 2 folating-point units. We also reduced the number of loop iterations in each benchmark to reduce simulation duration.

These results indicates that the DISVLIW processor is faster than other processors regardless of both benchmarks and cache size. This is due to the DISVLIW's dynamic scheduling strategies. the VLIW and the SVLIW processor can't execute pending long instructions until the execution of all instructions in the previous long instruction finishes. Another factor is the DISVLIW's reduced object code size, which decreases average fetch cycles and also reduces cache misses. Besides, in the Figure 5, the SVLIW processor shows same performance when compared to the VLIW processor.

We also analyze the impact of cache size on speedups of the DISVLIW processor with respect to both the SVLIW and the VLIW processors. We assumed that memory reference latency is four cycles when cache miss occurs and that instruction cache size is varied from 4k bytes to 64k bytes to compare performance according to changes in cache size. 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. 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 as shown in Figure 5.

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.

6. Conclusion

The DISVLIW processor is a hybrid architecture designed for balancing scheduling effort more evenly between the compiler and the processor. The DISVLIW processor has inherited features as ILP exploitation of the VLIW processor and dynamic scheduling 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 benchmarks. These performance gains result from dynamic scheduling and size reduction of object code. The DISVLIW processor architecture opens several new avenues of research. Optimization of dependency information in 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 and dynamic scheduling.

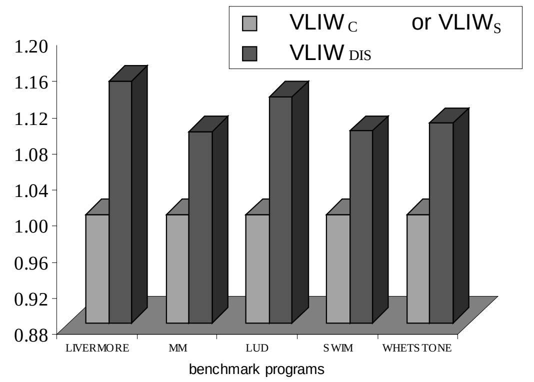


Figure 5. Comparison of speedup according to different scheduling strategies

7. Acknowledgments

The first author was supported by a postdoctoral fellowship program from Korea Science & Engineering Foundation (KOSEF). The research was completed using the facilities of the MCVL at University of Missouri-Columbia.

8. 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, 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," *Proc. 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," *Proc. The 28th International Symposium on Microarchitecture*, 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," *Proc. 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*, 2000.

[25] Sunghyun Jee and Kannappan Palaniappan, "Dynamically Scheduling VLIW Instructions with Dependency Information," *the 6th Workshop on Interaction between Compilers and Computer Architectures*, IEEE CS Press, Feb 2002.

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