Performance Evaluation For a Compressed-VLIW Processor

Sunghyun Jee
Chonan College in Foreign Studies
Department of Computer Information
393 Anseo Dong, Cheonan, Chungcheong Namdo
South Korea, 330-705
jees@missouri.edu

Kannappan Palaniappan
University of Missouri
Multimedia Communications and Visualization Lab
Department of Computer Science & Computer Engineering
University of Missouri, Columbia, MO 65211-2060
palani@cecs.missouri.edu

ABSTRACT
This paper presents a new ILP processor architecture called *Compressed VLIW* (CVLIW). The CVLIW processor constructs a sequence of long instructions by removing nearly all NOPs (No OPerations) and LNOPs (Long NOPs) from VLIW code. The CVLIW processor individually schedules each instruction within long instructions using functional unit and dynamic scheduler pairs. Every dynamic scheduler in the CVLIW processor individually checks for data dependencies and resource collisions while scheduling each instruction. In this paper, we simulate the architecture and show that the CVLIW processor performs better than the VLIW processor for a wide range of cache sizes and across various numerical benchmark applications. These performance gains of the CVLIW processor result from individual instruction scheduling and size reduction of object code. Even though we assume a cache with a zero miss rate, the CVLIW's performance is still 9%~15% higher than that of the VLIW processor regardless of benchmark applications.

Keywords
ILP, VLIW, CVLIW processor, Individual Instruction Scheduling

1. INTRODUCTION
An important focus of activity regarding computer architecture in recent years has been the exploitation of Instruction Level Parallelism (ILP), that is, the ability to execute several operations simultaneously. Processors which are capable of exploiting ILP contain multiple functional units, fetch several instructions per cycle from the instruction cache, and in a given cycle may dispatch multiple operations for execution [3]. Such processors are referred to as a *superscalar* processor and a *Very Long Instruction Word* (VLIW) processor.

The superscalar processor executes all parallel processing steps directly in hardware at run-time [5]. 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 [1,8]. 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 [2,6,7].

To balance a load between compile-time and run-time on the above processors, *Superscalar VLIW* (SVLIW) processor architecture has been studied. The SVLIW processor executes object code acquired by removing LNOPs (Long NOPs) from the VLIW code, where the LNOP equals a long instruction that is composed of only NOPs (No OPerations) [6]. The SVLIW processor schedules the next long instruction after checking out data dependencies and resource collisions with the scheduled long instruction in advance. When any collision occurs, the SVLIW processor generates LNOPs automatically until all collisions are removed. However, despite the merits, the SVLIW processor has a performance limit the same as the VLIW processor because the SVLIW processor can't execute the next long instruction until all instructions within the scheduled long instruction are executed. To solve the performance limit problem, a processor architecture that satisfies the following criteria is required: (1) load balance between compile-time and run-time parallelization, (2) individual instruction scheduling, and (3) reducing the size of object code [2].

This paper presents a new ILP processor architecture called a *Compressed VLIW* (CVLIW) processor architecture that achieves these goals. The CVLIW processor individually schedules each instruction in long instructions constructed by removing LNOPs and NOPs from the VLIW code. To schedule each instruction independently, the CVLIW processor has a number of functional unit and individual scheduler pairs. Every scheduler individually decides to issue the next instruction to the associated functional unit, or to stall the functional unit for the next pipeline cycle due to possible resource collisions or data dependencies. Such features can reduce execution cycles of the CVLIW processor better than those of the VLIW or the SVLIW processor that has to schedule long instructions compulsorily. Even though the superscalar processor is an effective way of exploiting ILP, this architecture requires complex devices and the impact of such complexity on the design cost and clock cycle time can be severe [7]. Consequently, the superscalar processor will not be evaluated in this paper. We believe that the CVLIW processor is simpler and more cost effective than the superscalar processor. The reason is that the CVLIW processor requires simple hardware units to schedule instructions due to nearly parallelized object code constructed from the VLIW code.

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

SAC '02, Madrid, Spain
Copyright 2002 ACM 1-58113-445-2/02/03...$5.00

2. INSTRUCTION LEVEL PARALLELISM

Figure 1 shows issue slots and execution images of the VLIW, the SVLIW, and the CVLIW processors that execute their own object code generated from given instruction graph. In the instruction graph, a node represents an instruction and a directed edge is annotated with data dependencies and resource collisions among instructions. We assume that every processor has three untyped functional units that can execute any instruction and a long instruction is composed of three instructions. Figure 1(a-1), (a-2), and (a-3) illustrate 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 individually mean a long instruction and an instruction. Every 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 that executes VLIW code. The VLIW code contains a number of LNOPs and NOPs to solve data dependencies and resource collisions between long instructions as shown in Figure 1(a-1). 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 executing SVLIW code. The SVLIW code is constructed by removing all LNOPs from the VLIW code as shown in Figure 1(a-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. In Figure 1(b-1) and 1(b-3), The VLIW and SVLIW processors equally require1 12 cycles for execution in this experiment.

Figure 1(b-3) shows an execution image for the CVLIW 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 among instructions in order to admit synchronization. The CVLIW processor issues one long instruction per cycle and individually executes each instruction using dependency information contained within the instruction. As shown in the shaded pipelines in Figure 1(c-3), instructions I_2, I_3, and I_4 are simultaneously executed during the 6^{th} cycle although the instructions are individually fetched on different clock cycle. The CVLIW processor requires 10 cycles for execution.

This simple example demonstrates the process by which the CVLIW processor can achieve better performance in comparison to the VLIW or the SVLIW processor. The main insight is that in the CVLIW

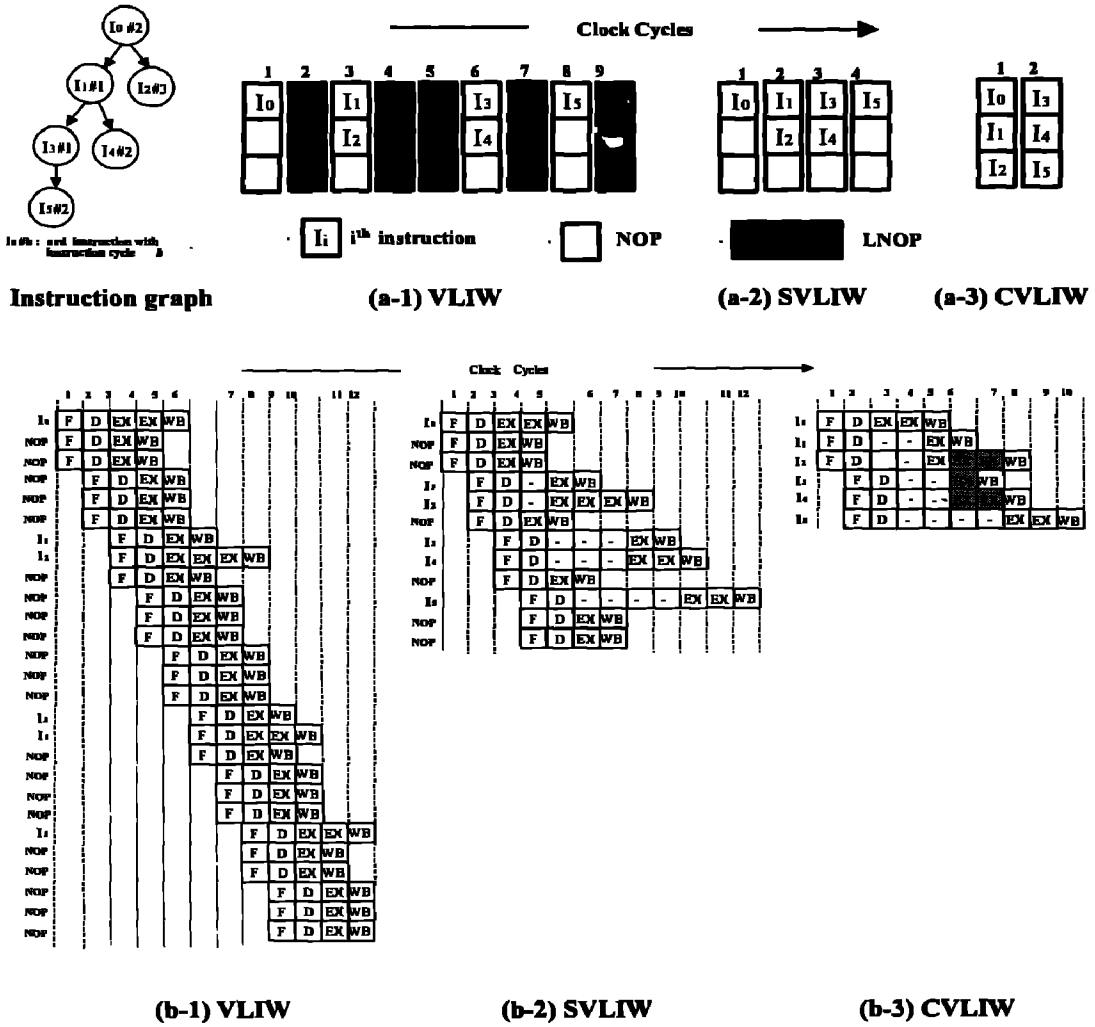


Figure 1. Comparison of Issue slots and execution images

processor each instruction within a given long instruction is independently processed. Therefore, it decreases the waiting time to process a given set of long instructions, however, the VLIW and the SVLIW processors schedule each long instruction according to fetched order of long instructions.

3. CVLIW PROCESSOR MODEL

3.1 long Instruction Format

To work every functional unit individually, the CVLIW processor needs object code including dependent information that express dependent relations among instructions.

Figure 2 shows CVLIW code structure composed of m long instructions. A long instruction has n instructions that may depend on each other due to data dependencies or resource collisions. Each instruction format consists of pre-dependency Pre_{dep}, an instruction I_{ij}, and post-dependency $Post_{dep}$. The Pre_{dep} provides information about functional units executing previous instructions that have dependencies with the instruction I_{ij}. I_{ij} refers to the j^{th} $(j=1,..,n)$ instruction contained within the i^{th} $(i=1,..,m)$ long instruction. The $Post_{dep}$ provides information about functional units that will execute following instructions that depend on the instruction I_{ij}.

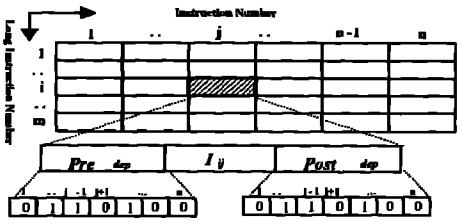


Figure 2. CVLIW long instruction format

To store the dependency relations between the I_{ij} and other instructions, the Pre_{dep} and the $Post_{dep}$ of I_{ij} are individually composed of a bit vector that has $(n-1)$ bits, where n equals the number of functional units. To store the information in the bit vector, the compiler allocates one bit per each functional unit within the bit vector. The bit corresponding to the functional unit F_j that will execute I_{ij} is omitted in the bit vector, since there is no need for the F_j to store information about itself (as Figure 2). If I_{ij} depends on a previous instruction I_{ik} being executed by functional unit F_k, the bit designating F_k in the Pre_{dep} is set to 1. Otherwise, it is set to zero. Although CVLIW code contains dependency information composed of many bits, the CVLIW processor can still achieve a reduction in object code size in comparison to the VLIW processor [7].

$$00I_011 \quad 10I_110 \quad 10I_200$$
$$10I_301 \quad 00I_400 \quad 10I_500$$

The CVLIW code shown in the above is generated from the instruction graph of Figure 1. A long instruction consists of three instructions. From this CVLIW code, we know that instruction I_1 within the 1st long instruction depends on previous instruction I_0 executed by the 1st functional unit since the first bit in the Pre_{dep} is set to 1. We also know that I_1 also has dependent relations with following instruction I_3 executed by the 1st functional unit because first bit in the $Post_{dep}$ is set to 1.

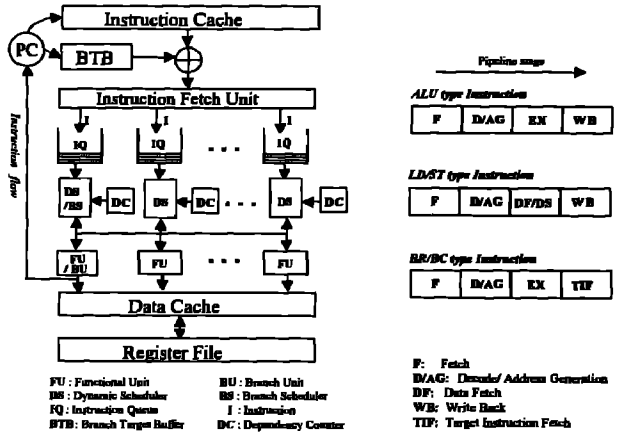
3.2 Processor Architecture

To schedule each instruction in CVLIW code independently, the CVLIW processor must be able to synchronize instructions and to check data dependencies and resource collisions among instructions using dependency information contained within each instruction.

Figure 3(a) shows a block diagram of the CVLIW processor architecture. The CVLIW processor has a number of FU (Functional Unit) and DS (Dynamic Scheduler) pairs, a number of IQs (Instruction Queues) and DCs (Dependency Counters), register file, instruction cache, data cache, and BTB (Branch Target Buffer). Each IQ stores an instruction (separated into a long instruction) in its own tail, and provides an instruction stored in its own head to the associated DS. Each DC is necessary to check Pre_{dep} of the next instruction, which will be executed by the associated FU, using $Post_{dep}$ of executed instructions in order to achieve synchronization. To accomplish this, each DC, a 32 bit register, is composed of $n-1$ counters. Each counter is associated with one FU and counts the number of previously executed instructions the FU depends on. Since there is no need for an FU to count its own instructions, there is no counter for the associated FU in the DC. Using the DC, each DS individually decides whether to assign the next instruction to the associated functional unit, or to stall the functional unit. The processor also utilizes the BTB structure for branch prediction.

3.3 Instruction Pipeline Algorithm

In the CVLIW processor, every instruction is classified as ALU (Arithmetic Logic Unit), LD/ST (LoaD/STore), or BR/BC (BRanch unconditional/Branch Conditional) type instruction. Every instruction is executed in four stages as shown in Figure 3(b). Each stage requires only one cycle except the execution stage that requires various execution cycles according to an instruction.



(a) Processor architecture **(b) Pipeline stage**

Figure 3. The CVLIW processor architecture

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 in the tails of the 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/Address Generation* (D/AG) stage, the decode unit analyzes the next instructions located at the head of each IQ. Every DS simultaneously checks for data dependencies and resource collisions using both Pre_{dep} in the next instruction and counter values in the associated DC. If any bit in Pre_{dep} is set to 1, DS checks the counter in the corresponding location in the associated DC. If the counter is 0, it means that the execution of previous dependent instruction hasn't finished. Otherwise, the execution of previous dependent instruction has finished. After the DS confirms that the execution of all previous dependent instructions is finished, the DS decrements the counter values in corresponding location in its DC

using the set bits in Pre_{dep} of the next instruction. It is necessary in order to clear the $Post_{dep}$ of the previous instructions. Simultaneously, the DS assigns the next instruction to the associated FU.

In the *Execute* (EX) stage, every FU executes instructions and simultaneously announces that its execution is finished using $Post_{dep}$ of its currently executing instruction. That is, in the case of floating-point instructions requiring multi-cycles, the FU announces its execution is 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 $Post_{dep}$. To facilitate this, we designed the EX stage with the ability to control the D/AG stage. Finally, in the *Write Back* (WB) stage, the results of the executed instructions are stored in the register file. In the case of a branch instruction, the processor decides whether or not to branch using information included in the BTB. If a branch occurs, the results of the consecutively executed instructions are kept in temporary storage until the branch is proven to be correct. If the branch prediction is correct, then the values in temporary storage are restored to the register file. Otherwise, the values are discarded.

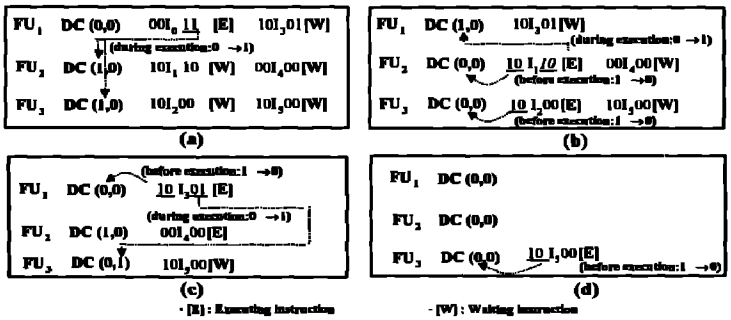


Figure 4. Example of instruction execution steps

Figure 4 shows execution steps of the CVLIW code fragment shown in Section 3.1, where FU_i corresponds to the i^{th} functional unit. As an example, FU_1 first executes instruction I_0 since the Pre_{dep} of I_0 is 00. Simultaneously, FU_1 increments the first counters(indicating FU_1) in the DCs of FU_2 and FU_3, because the $Post_{dep}$ of I_0 is 11. In Figure 4(b), FU_2 and FU_3 individually check their Pre_{dep} bits of the next instruction and the counter values in the associated DC. If both of them are greater than 0, FU_2 and FU_3 decrement the first counter in its DC because Pre_{dep} of I_1 and I_2 are 10. This is required in order to clear the $Post_{dep}$ of I_0. Then, FU_2 and FU_3 simultaneously begin the execution of I_1 and I_2.

4. PERFORMANCE EVALUATION

4.1 Simulation System

The performance of the CVLIW processor was accurately analyzed using a simulator testbed. We measured the total number of execution cycles for various numerical benchmark applications on the VLIW, the SVLIW, the CVLIW 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 [4]. 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. In the diagram, $VLIW_T$, $VLIW_S$, and $VLIW_C$ correspond to VLIW, SVLIW, and CVLIW 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.

The simulators receive the object code and then calculate the total number of execution cycles required for execution. We can then compute performance by comparing the three processors' total number of execution cycles. 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 CVLIW or the SVLIW processor. We assume input parameters which appreciate their influence on the simulation performance as the follows: each processor simulator has four functional units composed of two integer units and two floating-point units, the cache replacement is LRU (Least Recently Used), and memory reference latency is four cycles when cache miss occurs.

Table 1. Benchmark applications

Benchmarks	I/F(%)	$VLIW_T$	$VLIW_S$	$VLIW_C$
LIVERMORE	65.3/34.7	1	0.723	0.725
MM	68.4/31.6	1	0.568	0.591
WHETSTONE	65.6/34.4	1	0.438	0.385
FFT	43.3/56.7	1	0.385	0.400

Table 1 provides proportions of I/F (Integer instructions and Floating-point instructions) in benchmark programs used in this research. We choose benchmark programs that have a high proportion of floating-point instructions. This choice was appropriate because the CVLIW processor is more effective given individual instruction scheduling and reduced object code size. These applications all use double precision. Table 1 also tabulates the ratios of object code size of the VLIW to both the SVLIW and CVLIW processors. Even though $VLIW_C$ contains many bits of dependency information, $VLIW_C$ averages 45% smaller than $VLIW_T$ and is almost the same size as $VLIW_S$.

4.2 Experimental Results

In this section, we present and discuss the experiments carried out to evaluate the performance of the CVLIW processor. We start by examining the effects of scheduling strategies and cache size on the CVLIW, the SVLIW, and the VLIW processors.

4.2.1. Effect of scheduling strategies

Figure 5 shows the speedup of the CVLIW 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 and instruction fetch cycles. We assume that an instruction cache size is perfect (no miss penalty). Therefore, there is no instruction that occurs cache miss penalty. In this experiment, we reduced the number of loop iterations in each benchmark application in order to reduce simulation duration.

Figure 5 illustrates that even though we assume a cache with a zero miss rate, the CVLIW's performance is still 9%-15% higher than that of the VLIW processor regardless of benchmark application. We have the CVLIW's scheduling to thank for this speedup. This individual 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 previous long instruction finishes. Besides, in simulation environment of Figure 5, the SVLIW processor shows same performance in comparison to the VLIW processor.

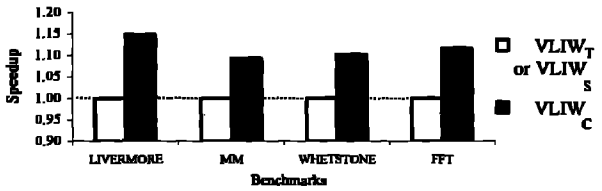


Figure 5. Speedup according to scheduling strategies

4.2.1. Effect of Cache Size

Figure 6 illustrates the impact of cache size on speedup of the CVLIW 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 CVLIW and the SVLIW processors were measured relative to the VLIW processor regardless of cache size. In this experiment, we reduced the number of loop iterations in each benchmark application in order to reduce simulation duration.

These results indicate that the CVLIW processor is faster than the SVLIW processor regardless of both benchmark applications and cache size. This is due to the CVLIW's scheduling strategies. Another factor is the CVLIW's reduced object code size, which decreases average fetch cycles and also reduces cache misses. But cache size does play a role in performance *difference*. Figure 6 indicates that larger cache sizes result in smaller speedup differences between the VLIW and CVLIW processors. At smaller cache sizes, the VLIW's performance is slower due to higher cache miss rates. Unlike the VLIW, the CVLIW'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 CVLIW. Yet, even assuming perfect cache, the CVLIW is still faster than the VLIW's because of its individual scheduling strategy.

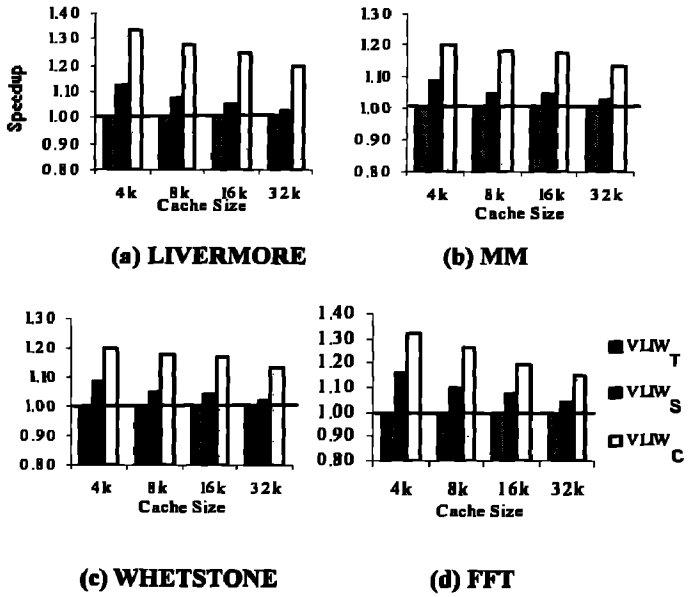


Figure 6. Speedup according to changes in cache size

Overall, we attribute these CVLIW's performance gains to the balanced benefits of compile-time and run-time parallelization, individual instruction scheduling, and size reduction of object code as previously described.

5. CONCLUSION

This paper describes a new ILP processor architecture referred to as Compressed *VLIW* (CVLIW). The proposed CVLIW processor is a hybrid architecture that has inherited features as ILP exploitation at compile-time of the VLIW processor and individual instruction scheduling at run-time of the superscalar processor. The CVLIW processor can individually schedule each instruction using dependency information contained within the CVLIW code. To schedule each instruction independently, the CVLIW processor has a number of functional unit and individual scheduler pairs. In this paper, the experimental evaluations have shown that the CVLIW processor achieves a high speedup over the VLIW and the SVLIW processors for a wide range of cache sizes and across various numerical benchmarks. The performance gains result from individual instruction scheduling and size reduction of object code. Even though we assume a cache with a zero miss rate, the CVLIW's performance is still 9%-15% higher than that of the VLIW processor regardless of benchmark application.

The CVLIW processor architecture opens several new avenues of research. Optimization of dependency information within object code, CVLIW 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 to achieve synchronization.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] Arthur Abnous and Nader Bagherzadeh, "Pipelining and bypassing in a VLIW processor,"*Trans. Para. Dist. Sys.*, Vol. 5, No. 6, pp. 658-664, June 1994.

[2] Erik R. Altman, R. Govindarajan , and Guang R. Gao, "A Unified Framework for Instruction Scheduling and Mapping for Functional Units with Structural Hazards, "Journal of Parallel and Distributed Computing, PP 259-293, 1998.

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

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

[5] Pohua P. Chang, Daniel M. Lavery, Scott A. Mahlke, and William Chen, "The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors," *IEEE Transactions on Computers*, Vol. 44, No. 3, March 1995.

[6] 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.

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

[8] Thomas M. Conte and Sumedh W. Sathaye, "Dynamic Rescheduling: A technique for object code compatibility in VLIW architecture," *Proc. 28th Inter. Symp. Micro.*, 1995.