JETPIPELINE: A HYBRID PIPELINE ARCHITECTURE FOR INSTRUCTION-LEVEL PARALLELISM

MASAYUKI KATAHIRA, HONG SHEN, HIROAKI KOBAYASHI AND TADAO NAKAMURA

Department of Computer and Mathematical Sciences, Graduate School of Information Sciences, Tohoku University, Aramaki Aza Aoba, Aoba-ku, Sendai 980-77, JAPAN

ABSTRACT

High performance processors based on pipeline processing play an important role in scientific and engineering computation. However, it is difficult to gain a satisfactory solution when taking both high degree of flexibility of parallel processing and low hardware complexity into account. This paper propose a hybrid pipeline architecture named Jetpipeline that possesses high degree of flexibility in parallel processing as well as suitable hardware complexity. Jetpipeline incorporates several instruction pipelines together. Multiple ALUs and floating point arithmetic pipelines are respectively used in the execution stages of these instruction pipelines. An instruction set for Jetpipeline is described in detail. Furthermore, a policy for scheduling instructions to instruction pipelines is discussed. The simulation results offer us the potential of Jetpipeline.

1. Introduction

The progress of modern science and technology has brought us with a rapid increase in fast and massive computation requirements for scientific and engineering applications such as weather forecast, nonlinear system calculation and real-time control. Although the important development of VLSI technology allows a stable increase in the speed of processors, it is quite difficult to keep the speed growth of processors through hardware technology alone.

On the other hand, a good processor architecture can provide us with high performance/price ratio by organizing existing hardware devices reasonably. In the area of high performance processors, the implementation of superscalar machines[1][2][3] and the VLIW machines[3][4][5][6][7] is one of the achievements drawing wide attention besides vector machines.

Superscalar machines were developed because of the introduction of parallel processing into microprocessors. A superscalar machine with degree n can issue n instructions per cycle. However, there must be n instructions executable in parallel at all pipeline cycles to fully utilize a superscalar machine. Moreover, the superscalar machine suffers quite complex hardware configuration because it has to support the selection of which operations to be issued in a given cycle at run time. On the other hand, VLIW machines were proposed to solve the hardware difficulty of superscalar machines. VLIW machines have instructions with hundreds of bits long, each of the instructions can specify multiple operations that are executed in parallel. Since VLIW instructions have a fixed format, it is easier to decode VLIW instructions than superscalar instructions. Thus the hardware of VLIW machines is more simple in comparison with superscalar machines. Unfortunately, not always can all fields of a VLIW instruction specified by a fixed format be utilized in usual programs. Therefore, the lack of flexibility leads to a waste of hardware resources of VLIW machines.

This paper proposes a hybrid pipeline architecture to absorb the advantages of superscalar machines and VLIW machines. We borrow the speedy image of jet airplanes, and name the proposing architecture Jetpipeline. Jetpipeline is composed of several instruction pipelines, and the execution stages of these instruction pipelines include integer ALUs and floating point arithmetic pipelines. Thus, Jetpipeline can issue multiple instructions in a cycle like a superscalar machine. However, the scheduling of Jetpipeline instructions is statically performed at compile time so that the complicated hardware operations examining a sequence of instructions and investigating data dependencies among operands can be prevented. Although the static scheduling is somewhat similar to that of a VLIW machine, Jetpipeline possesses more flexibility than a VLIW machine because instructions executed in parallel in Jetpipeline are just grouped rather than decoded in a fixed long word like a VLIW machine. Therefore, Jetpipeline can be treated as a hybrid architecture of superscalar architectures and VLIW architectures.

The rest of this paper is organized as follows. Section 2 describes the concept of Jetpipeline. Section 3 presents a prototype architecture for Jetpipeline. The design details such as an instruction set and instruction scheduling are also discussed in this section. Simulated performance studies are carried out in Section 4. Finally, the paper concludes with Section 5.

2. The Concept of Jetpipeline

As mentioned before, Jetpipeline exploits instruction-level parallelism. There are multiple instruction pipelines bundled into Jetpipeline. ALUs and floating point arithmetic pipelines correspond to the execution stages of these instruction pipelines. Figure 1 gives the concept of Jetpipeline architecture. As shown in Figure 1, when the inputted instructions, data and the outputted results are considered as air, fuel and burnt gas, the entire system can be considered as a jet engine. Therefore, we call the architecture Jetpipeline.

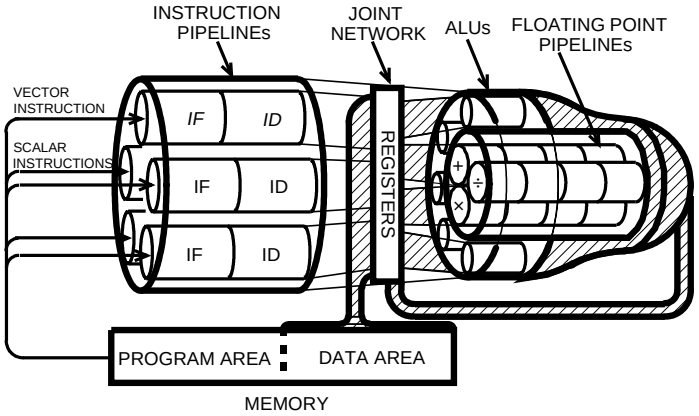


Figure 1 : Overview of Jetpipeline.

In Jetpipeline, multiple instructions can be issued in one pipeline cycle like a superscalar machine. The compiler for Jetpipeline schedules instructions from a source program, and packs instructions for the same pipeline cycle as a group. At run time, packed instructions are scheduled to respective instruction pipelines, and executed in parallel. Hence, scheduling instructions is performed statically by a compiler like a VLIW machine. Therefore, Jetpipeline may be treated as a hybrid architecture of superscalar machines and VLIW machines.

However, Jetpipeline differs much from both superscalar machines and VLIW machines. Table 1 compares the major features of Jetpipeline with superscalar machines and VLIW machines. Although Jetpipeline issues multiple instructions per cycle, it does not schedule instructions relying on hardware at run time. It is unnecessary to keep the compatibility of instructions with an expensive cost of hardware such as superscalar machines. Besides, it is quite difficult for superscalar machines to schedule vector instructions that will spend lots of cycles on hardware without any waste. On the other hand, instructions executed in parallel in Jetpipeline are just packed together different from VLIW machines. The format of instructions in VLIW machines is fixed, and each long instruction word specifies multiple operations. A VLIW machine issues only one instruction per pipeline cycle. Operations specified in a VLIW instruction are unfolded at the execution stage. In Jetpipeline, however, instruction pipelines are independent with each other. These instruction pipelines are controlled by a group of instructions packed together at compile time, not by a long instruction word. Thus, Jetpipeline is more flexible than VLIW machines and a waste of instruction bits can be avoided.

Table 1 : The comparison of superscalar machines, VLIW machines and Jetpipeline.

	Instruction fetch	Stage with par-allel processing	Controlling/ Scheduling policy	The support of compiler	Instruction compatibility
Superscalar	Multiple instructions are fetched per cycle.	Execution stage e.g. IU, FPU, Load/Store	The scheduling is performed by hardware at run time. (The kinds of concurrent operations are restricted.)	It is unnecessary, but specialized compiler can improve the system performace.	Yes
VLIW	One long instruction is fetched per cycle.	Execution Stage	The scheduling is performed by software at compile time. (The operation of each field in instruction is fixed.)	Necessary	No
Jetpipeline	One group with multiple instructions is fetched per cycle.	Execution Stage	The scheduling is performed by software at compile time. (Any combination of concurrently executed instructions is possible.)	Necessary	No

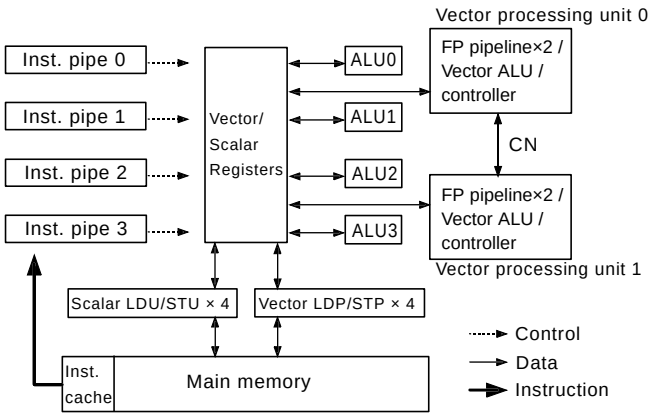
3. The Prototype Architecture

In this section, we present the prototype architecture to implement the Jetpipeline model. We examine the prototype architecture at the functional block level. The instruction set and scheduling policy are also discussed in this section.

3.1 The Hardware Configuration

In order to realize Jetpipeline, we make PEs be tightly coupled with registers to form a high performance processor capsule. Figure 2 shows the Jetpipeline system configuration with four instruction pipelines at the functional block level. Figure 3 shows the stages of instruction pipelines and the functions of each stage.

As shown in Figure 3, each of the instruction pipelines(Inst.pipe, IP) consists of six stages that are instruction fetch stage(IF), instruction decode and register fetch stage(ID) and four execution stages. The stages are overlapped



Inst. pipe : Instruction pipeline
Inst. cache : Instruction cache
FP pipeline : Floating point calculation pipeline
CN : Interconnection network
LDU/STU : Load unit/Store unit
LDP/STP : Load pipeline/Store pipeline

Figure 2 : System architecture of the functional block level.

every two clock cycles.

Because arithmetic logic unit(ALU) is utilized most frequently for executing scalar instructions, it is necessary to set up an ALU for each instruction pipeline. Therefore, there are four ALUs to satisfy the requirements from the four instruction pipelines. In Figure 2, each of the two vector processing units consists of two floating point calculation pipelines(FP pipeline) for the four fundamental arithmetic operations, an ALU for vector operation, and a controller for vector operation. Since these two units are shared by any two neighboring instruction pipelines, the same kind of vector instructions cannot be executed in parallel at two neighboring instruction pipelines. In addition, the two floating point operation pipelines in each of the block are shared by both vector instructions and scalar instructions. Consequently, the same kind of scalar floating point instructions cannot be concurrently executed during the execution of vector floating point instructions. However, scalar instructions except floating point instruction can be executed in parallel with vector instructions because vector instructions are executed under the control of vector operation controller in the block. Figure 4 shows these restrictions of

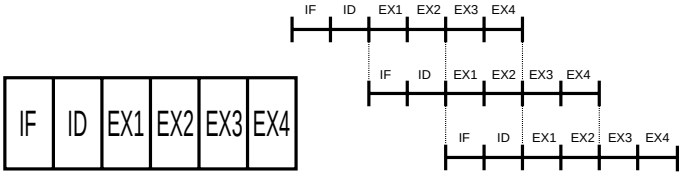


Figure 3 : The structure of instruction pipelines.

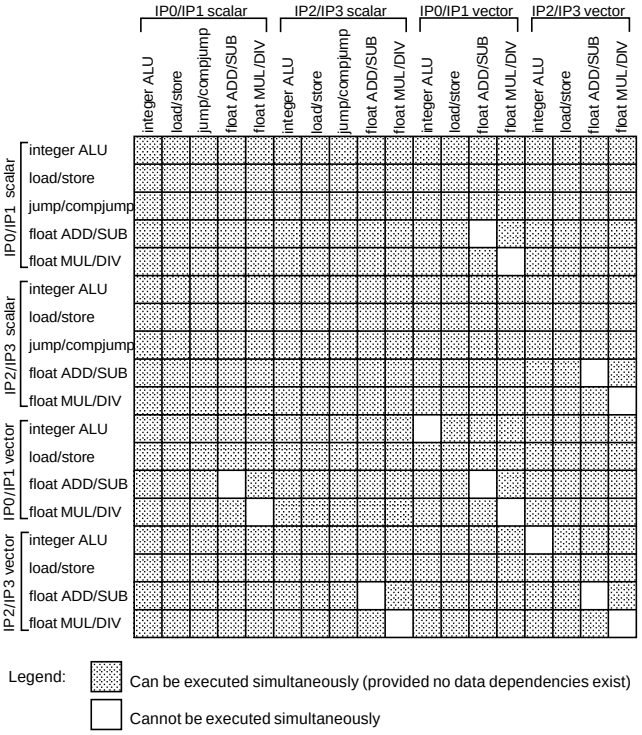


Figure 4 : Simultaneous instruction execution restrictions.

simultaneous instruction execution. The interconnection network(CN) between the two blocks is designed for vector chaining operations.

Registers used in the Jetpipeline architecture are divided into vector registers and scalar registers. In order to support conditional operations, the vector mask register(VMR) is prepared. The registers are banked to meet the requirements of high speed accesses from ALUs and floating point calculation pipelines.

In Jetpipeline, the data transmission between registers and the main memory relies on the scalar load/store units(LDU/STU) and the vector load/store pipelines(LDP/STP). The number of the LDU/STUs and the LDP/STPs corresponds to the number of the instruction pipelines. Therefore, it is possible to perform high speed data transmission for both scalar and vector load/store instructions. Moreover, the instruction cache(Inst.Cache) is provided to decrease the latency at the instruction fetch stage.

3.2 The Instruction Set

The instruction set of an architecture reflects the function of the architecture. The instruction set for Jetpipeline is not an exception. In order to keep the conciseness of the Jetpipeline

architecture, we build up the instruction set for Jetpipeline based on RISC approaches[8][9][10][11][12]. The length of Jetpipeline instructions is fixed to 32 bits. Table 2 lists out the major kinds of Jetpipeline instructions, and their execution processes on the instruction pipelines. Figure 5 gives the format of the Jetpipeline instructions.

The scalar instructions of Jetpipeline are designed based on the Stanford MIPS architecture[8][13]. Concretely, the scalar instructions consist of load/store instructions, register operation instructions and branch instructions. The branch instructions include the compare-and-jump instructions in addition to the branch instructions based on conditional codes.

The vector instructions are designed referring the NEC SX supercomputer[14]. The vector instruction set includes vector load/store instructions, vector register instructions of integer and floating point operations, other vector instructions such as mask register operation, and so on. Besides this, chaining operation is also controlled at the instruction level.

Table 2 : The Jetpipeline instructions.

Instruction Type		Pipe-stage	Description
All instructions		IF	fetch instruction from memory PC + 1 -> PC
		ID	decode instruction
Scalar instruc-tion	integer operation	EX1	src1,src2 -> ALU -> dest
	floating operation	EX1	src1,src2 -> floating(A/S) pipeline (add/sub) src1,src2 -> floating(M/D) pipeline (mult/div)
		EX2	no operation
		EX3	floating(A/S) pipeline -> dest (add/sub)
		EX4	floating(M/D) pipeline -> dest (mult/div)
	load/store	EX1	src1,0 -> ALU -> dest (load immediate) src1,src2 -> ALU -> LU addr. latch (load) src1,src2 -> ALU -> SU addr. latch (store)
		EX2	LU data latch -> dest (load) dest -> SU data latch (store)
	jump/ compare&jump	EX1	src1,src2 -> ALU -> tempreg
		EX2	(if cond then) tempreg -> PC (jump) src1,src2->ALU->cond,if cond then tempreg->PC (compare and jump)
Vector instruc-tion	vector load/store	EX1	parameters(addr etc.)->Vector Load/Store Unit initiate VLU/VSU
	vector int/logic operation	EX1	VRsrc1[0],VRsrc2[0] -> vector int/logic pipe VRsrc1[1],VRsrc2[1] -> vector int/logic pipe
		EX2	initiate vector int/logic pipe controller
	vector floating operation (add/sub)	EX1	VRsrc1[0],VRsrc2[0] -> floating add/sub pipe VRsrc1[1],VRsrc2[1] -> floating add/sub pipe
		EX2	VRsrc1[2],VRsrc2[2] -> floating add/sub pipe VRsrc1[3],VRsrc2[3] -> floating add/sub pipe
		EX3	initiate vector float add/sub pipe controller
	vector floating operation (mult/div)	EX1	VRsrc1[0],VRsrc2[0] -> floating mult/div pipe VRsrc1[1],VRsrc2[1] -> floating mult/div pipe
		EX2	VRsrc1[2],VRsrc2[2] -> floating mult/div pipe VRsrc1[3],VRsrc2[3] -> floating mult/div pipe
		EX3	VRsrc1[4],VRsrc2[4] -> floating mult/div pipe VRsrc1[5],VRsrc2[5] -> floating mult/div pipe
		EX4	initiate vector float mult/div pipe controller

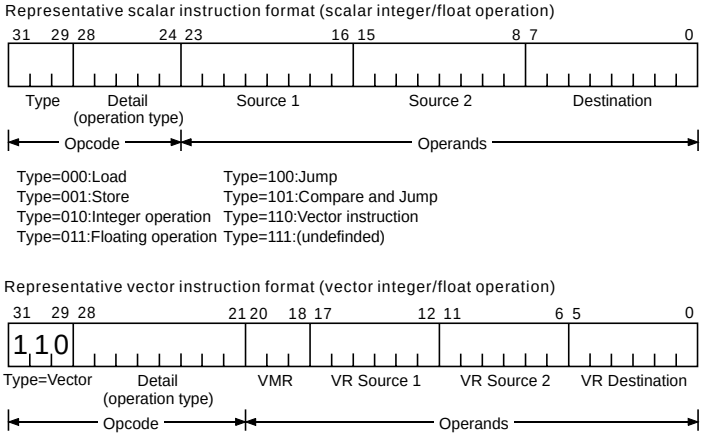


Figure 5 : Instruction format.

3.3 Instruction Scheduling

Instruction scheduling is one of the crucial issues affecting processor performance in a pipeline processor. In Jetpipeline, instruction scheduling controls the distribution of instructions from the instruction cache to the four instruction pipelines. This section discusses the principle of Jetpipeline instruction scheduling.

The instruction scheduling of Jetpipeline is performed statically by software at compile time. As illustrated in Section 3.1, Jetpipeline hardware configuration affects the possible execution order of instructions. In order to achieve satisfactory performance of every operation units of Jetpipeline, basic scheduling rules are necessary. We describe the scheduling rules as follows.

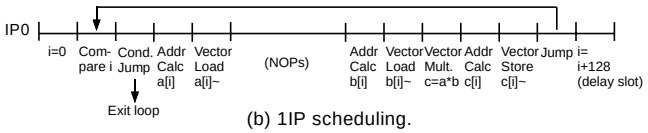
- The execution of an instruction that uses the results of its predecessor scalar floating point instruction must be delayed until the results are obtained. During this period, NOP or other instructions that do not rely on the execution order should be inserted into the pipeline.

- It takes multiple clock cycles to execute a vector instruction. NOP or concurrently executable instructions should be inserted between continuously executed vector instructions with data-dependent relation to adjust the timing of execution.

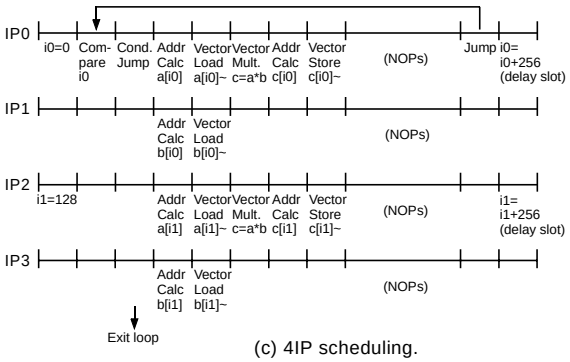
- Delayed branches are also adopted as general RISC architectures in Jetpipeline. Some instructions or NOP are also inserted in delay slots after a branch instruction.

```
for i=0 to n do c[i] = a[i] * b[i]
```

(a) Source Program.



(b) 1IP scheduling.



(c) 4IP scheduling.

Figure 6 : Examples of scheduling.

Figure 6 shows a practical scheduling example, where the compilation of the loop program in Figure 6(a) is examined.

Figure 6(b) depicts the scheduling results when the number of instruction pipelines is 1. Because the two vector instructions of LOAD a[i] and LOAD b[i] cannot be continuously executed, other executable instructions should be inserted between the two vector instructions.

Figure 6(c) gives the scheduling results when the number of instruction pipelines is 4. It can be found that other concurrently executable instructions are also inserted to prevent the vector load instructions from overlapped execution.

4. The Simulated Performance Studies

In order to evaluate the performance of the proposed Jetpipeline architecture, simulation experiments have been carried out. This section reports the experimental results.

4.1 The Simulation Model

The simulation experiments are held based on the hardware configuration shown in Figure 7, where four instruction pipelines are included. In order to evaluate the Jetpipeline operation units and their controlling mechanism which reflect the peculiarity of Jetpipeline, the following assumptions are made.

- The restriction of the interconnection network and the bank of registers are not taken into account in the simulator. The parallel accesses to the registers are performed with a satisfied speed.

- The main memory accesses are finished within one clock, and the memory latency are ignored.

4.2 The Assembler

The simulator is executed on the UNIX operating system. Considering the software environment of Jetpipeline, the assembler is already put into use while the compiler is currently under construction.

The assembler translates the Jetpipeline instructions into the Jetpipeline machine code, and offers the programs in the Jetpipeline assembly language to the Jetpipeline simulator. With the help of the assembler, the facility for debugging programs can be utilized. The simulator internal states, e.g. contents of registers during executing programs, can be traced, and break point of program execution can be set.

4.3 The Simulation Results and Analysis

We run some benchmark programs on the simulator to investigate the system performance of the proposed Jetpipeline architecture. In our

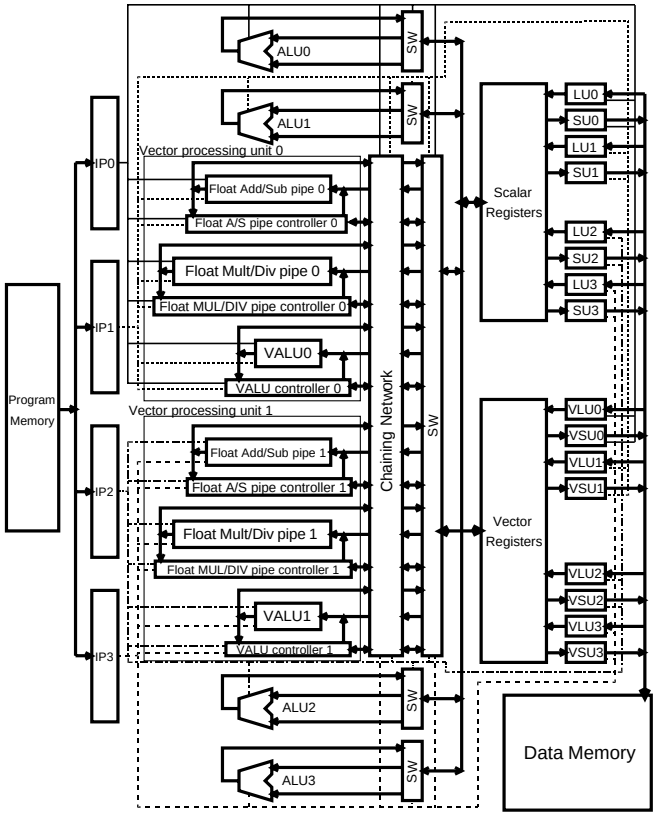


Figure 7 : Block diagram of the Jetpipeline simulator.

research, we chose Livermore loops[15] that are well used to evaluate performance of supercomputers as benchmarks. Livermore loops are constructed for a spectrum of computation-intensive benchmarks by Lawrence Livermore National Laboratory. A number of loops are composed in the Livermore loops including those both easy to vectorize and difficult to vectorize.

In this section, we present the simulation results of three representative Livermore loops (KERNEL 1, KERNEL 3, and KERNEL 5). These loops written in FORTRAN are translated into the form of the Jetpipeline assembly language manually, and then are assembled into the Jetpipeline machine code by the assembler. The simulator executes these programs and outputs the number of clock cycles as execution time for each loop.

Figure 8 shows the execution results in terms of speedup. The speedup is measured against the execution cycles when the number of instruction pipeline is 1 and only scalar instructions are used. The notation of nIP-Scalar in Figure 8 means the case that the number of instruction pipeline is n and just scalar instructions are used. Similarly, the notation of nIP-Vector means the case that the number of instruction pipelines is n and both scalar and vector instructions are used.

KERNEL 1 calculates hydro fragment, and it is easy to be vectorized. Therefore, high speedups were achieved as the result of its high degree of parallelism. Especially, in case of 4IP-Vector, the speedup rises up till 28.9. Because the jump instructions and instructions calculating array address for loop counter are also parallelized, one instruction cycle was removed when executing instructions of the loop. Consequently, the speedup went up to 4.2 over the optimal one in the case of 4IP-scalar.

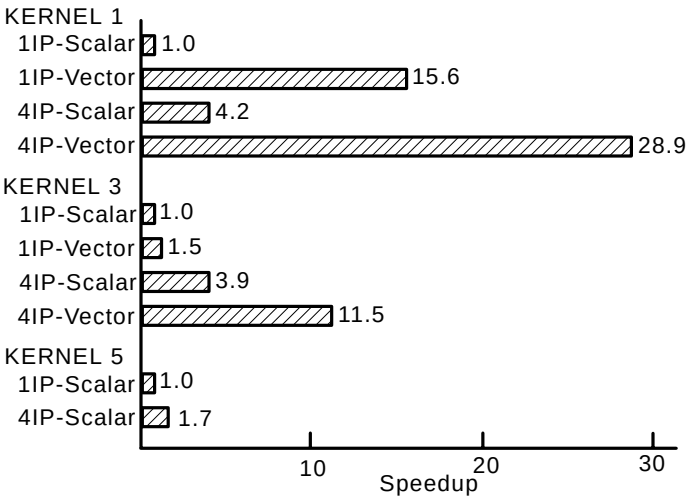


Figure 8 : Experimental results of simulations.

KERNEL 3 calculates vector inner product. Although it possesses quite amount of parallelism, just a part of its operations can be vectorized. The part of multiplication is performed by the vector instructions while the part of sum is performed by the scalar instruction. Lower speedups were obtained when compared with KERNEL 1. However, in the case of 4IP-Vector, the speedup of 12 times was obtained because the calculation of sum can be unfolded and parallelized.

KERNEL 5 calculates tri-diagonal elimination of below diagonal which is a sequential program. Neither can it be parallelized, nor vectorized. No speedup can be obtained with traditional methods. However, Jetpipeline aims the lower level parallelism so that the speedup of about 1.7 was achieved.

The simulation results are measured without the consideration of the interconnection network and memory latency. However, the executed benchmark programs are not so large that whole of each of the programs can be contained in cache. Therefore, the simulation results will not vary greatly even if taking the above factors into account. The complete simulator is now being constructed, and practical implementation should be done.

5. Conclusions

This paper proposed a hybrid processor architecture named Jetpipeline for instruction-level parallelism. Jetpipeline absorbs the flexibility of superscalar machines and the conciseness of VLIW machines. Performance simulation results shows a good performance. Not only can the programs easy to vectorize be effectively accelerated, but also the programs difficult to vectorize can be speeded up. Therefore, Jetpipeline is a promising candidate for architectures exploiting instruction-level parallelism.

References

[1] T. Agewala and J. Cocke, "High performance reduced instruction set processors," IBM Tech. Rep., March 1987.

[2] "MC88110 Second generation RISC microprocessor user's manual", Motorola Inc., 1991
[3] N. P. Jouppi, "The nonuniform distribution of instruction-level and machine parallelism and its effect on performance," IEEE Trans. Comput., vol.38, No.12, pp.1645-1658, December 1989.
[4] A. E. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," Computer, vol.14, No.9, pp.18-27, September 1981.
[5] J. A. Fisher, "Very long instruction word architectures and the ELI-512," Proc. 10th Annual International Symposium on Computer Architecture, pp. 140-150, June 1983.
[6] J. R. Ellis, "Bulldog: A compiler for VLIW architectures," The MIT Press, 1986.
[7] R. P. Colwell et al., "A VLIW architecture for a trace scheduling compiler," IEEE Trans. Comput., vol.C-37, No.8, pp.967-979, August 1988.
[8] J. L. Hennesy et al., "MIPS:A VLSI processor architecture," Technical Report No.223, Computer Systems Laboratory, Stanford University, November 1981.
[9] D. A. Patterson and C. H. Séquin, "A VLSI RISC," Computer, pp.8-22, September 1982.
[10] J. L. Hennessy, "VLSI Processor Architecture," IEEE Trans. Comput., pp.1221-1246, December 1984.
[11] D. A. Patterson, "Reduced Instruction set computers," Comm. ACM, pp.8-21, January 1985.
[12] D. Tabak, "RISC System," Research Studies Press Ltd., 1990.
[13] T. Gross and J. Gill, "A short guide to MIPS assembly instructions," Technical Note No.83-236, Computer Systems Laboratory, Stanford University, November 1983.
[14] A. Jippo et al., "The supercomputer SX system: hardware," Proc. of the second international conference on supercomputing, vol.1, pp.57-64, 1987.
[15] F. H. McMahon, "The Livermore Fortran Kernels test of the numerical performance range," in Performance evaluation of supercomputers, J. L. Martin ed., North Holland, Amsterdam, 1988, pp.143-186.