Software Pipelining for Jetpipeline Architecture

Masayuki Katahira, Takehito Sasaki, Hong Shen,
Hiroaki Kobayashi and Tadao Nakamura
Graduate School of Information Sciences
Tohoku University, Sendai 980-77, JAPAN

Abstract

High performance processors based on pipeline processing play an important role in scientific computation. We have proposed a hybrid pipeline architecture named Jetpipeline in our former work. The concept of Jetpipeline comes from the integration of superscalar, VLIW and vector architectures. Jetpipeline has multiple instruction pipelines, which execute multiple instructions like superscalar architectures. Instructions to be executed simultaneously are statically scheduled by a compiler like VLIW architectures. Therefore, parallelism derivation and instruction scheduling are very important for Jetpipeline. Software pipelining is one of the well-known techniques to achieve high throughput when processing loop programs. In this paper, we propose software pipelining for Jetpipeline. Firstly, the overview of the Jetpipeline architecture is described. Then the banked register configuration of Jetpipeline for reducing hardware complexity and supporting software pipelining is presented. Finally, the effectiveness of software pipelining for Jetpipeline is discussed by simulation.

1 Introduction

The progress in modern science and technology has brought us with a rapid increase in fast and massive computation requirements. 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 depending on hardware technology alone.

On the other hand, a novel processor architecture can provide us with high performance/price ratio by reasonably organizing existing hardware devices. The implementations of superscalar machines[1][2][3] and VLIW machines[3][4][5][6][7] are two good examples drawing wide attention besides vector machines.

We have proposed Jetpipeline in [8], which is a hybrid pipeline architecture absorbing advantages of superscalar machines, VLIW machines and vector machines. Jetpipeline can issue multiple instructions in one clock cycle like a superscalar machine. However, the scheduling of Jetpipeline instructions is statically performed at compile time to prevent complicating hardware configuration. Jetpipeline possesses higher flexibility than a VLIW machine because instructions executed simultaneously in Jetpipeline are just grouped together rather than encoded in a fixed long word like VLIW machines.

In Jetpipeline, extracting parallelism from programs and code scheduling are statically performed by a compiler. Therefore, it is important to pay sufficient attention to the compiler so as to make Jetpipeline achieve its expected high performance. In this paper, we introduce the software pipelining technique[9] into the compiler of the Jetpipeline architecture. Software pipelining is one of the well-known techniques deriving parallelism from loop programs so that high performance can be achieved. We also propose an instruction code scheduling strategy. The effectiveness of the scheduling strategy is evaluated and discussed by examining the results of software simulation.

The rest of this paper is organized as follows. Section 2 gives the overview of Jetpipeline. The design details such as hardware configuration and the instruction set of Jetpipeline are presented. In addition, a basic scalar instruction scheduling strategy is also discussed in this section. Section 3 presents software pipelining for Jetpipeline. In this section, a code scheduling scheme and register renaming is described. Simulated performance studies are presented in Section 4. Finally, the paper concludes with Section 5.

2 The overview of Jetpipeline

2.1 Basic concept of Jetpipeline

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

In Jetpipeline, multiple instructions can be issued in one pipeline cycle like a superscalar machine. The compiler for Jetpipeline schedules instructions and packs several instructions for the same pipeline cycle as a group. At run time, each of the packed instructions is issued to a corresponding instruction pipeline, and is executed in parallel with other packed instructions. Hence, instruction scheduling is performed statically by a compiler like a VLIW machine. Therefore, Jetpipeline can be treated as a hybrid architecture of superscalar machines and VLIW machines.

However, Jetpipeline differs much from both superscalar machines and VLIW machines. Although Jetpipeline issues multiple instructions per cycle, it does not schedule instructions relying on complicated hardware at run time such as superscalar machines. On the other hand, instructions executed simultaneously in Jetpipeline are just packed together rather than fixed in a long word as VLIW machines. Thus, Jetpipeline is not only more flexible than VLIW machines, but also more concise than superscalar machines. A waste of instruction bits in VLIW machines can also be avoided.

Jetpipeline also has vector operation facilities to obtain high performance on vectorable

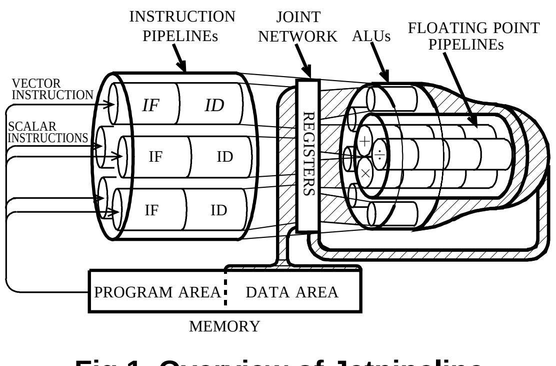

Fig.1 Overview of Jetpipeline.

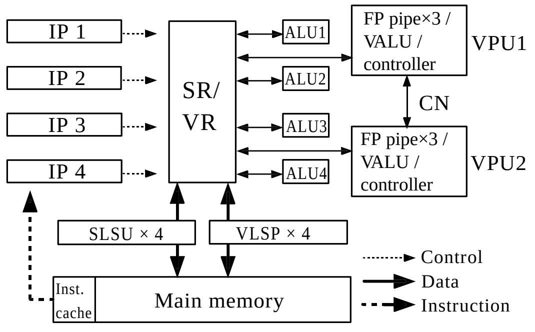

Fig.2 System architecture of the functional block level.

parts of programs. Since we concentrate on parallelization of non-vectorable parts of programs, the vector facilities in Jetpipeline will not be discussed in this paper.

2.2 Hardware configuration

Figure 2 shows the system configuration of Jetpipeline with four instruction pipelines. Each of the instruction pipelines(IP) consists of six stages. They are instruction fetch, instruction decode and register fetch, ALU operation, memory access, writeback and floating point data writeback stages. These stages are overlapped every one clock cycle. Because arithmetic logic units(ALUs) is used most frequently for executing scalar instructions, it is necessary to provide one ALU for each instruction pipeline. In addition, each instruction pipeline has an address ALU to calculate target addresses for jump instructions. In Figure 2, each of the two vector processing units(VPUs) consists of three floating point calculation pipelines(FP pipes) for floating point arithmetic operations, an ALU for vector operations(VALU), and a controller for vector operations. Since these two units are shared by arbitrary adjacent instruction pipelines, the same kind of vector instructions cannot be executed in parallel at two adjacent instruction pipelines. In addition, the three floating point operation pipelines(add/sub, mult, div) in each unit are shared by both vector instructions and scalar instructions. The interconnection network(CN) between these two units is designed for chaining multiple vector operations.

In Jetpipeline, the data transmission between registers and the main memory relies on the scalar load/store units(SLSUs) and the vector load/store pipelines(VLSPs). Moreover, the instruction cache(Inst.Cache) is employed to decrease the memory access latency at the instruction fetch stage. Registers used in Jetpipeline are divided into vector registers (VRs) and scalar registers(SRs). In order to support conditional instructions, the vector mask register(VMR) is also prepared for vector instructions.

2.3 The banked and overlapped register file

In the original concept of Jetpipeline, scalar registers are fully shared by all four instruction pipelines to realize flexible accesses from any of the four instruction pipelines. However, the connection network between registers and instruction pipelines will become quite complicated in this case. For example, assuming that a crossbar network were used here, the number of all datapath switches would reach 65536, and this seems to be an impractical number.

To moderate the hardware complexity while keeping the flexibility for register accesses, we develop a banked and overlapped register file as scalar registers. Figure 3 depicts the configuration of the register file, which is divided into one global block and eight local/overlap blocks. The global block can be accessed by all IPs and is used to broadcast commonly used data among IPs. The local blocks are accessed by their corresponding IPs only, and used to hold IP-local data. The forward/back blocks are shared by two adjacent IPs and used to exchange data each other. Since the forward block of IP3 is wrapped around to the back block of IP0, it takes two hops at most to send data between any two IPs.

We discuss an implementation example of the banked and overlapped register configuration, where each register block is constructed from a conventional register file with two read ports and two write ports and the switches are 1-to-4 selector/multiplexers. In this case, the number of all datapath switches is reduced to 2048, and this is approximately about 1/32 of that in crossbar implementation.

It is not difficult to find that the banked and overlapped register file satisfies both

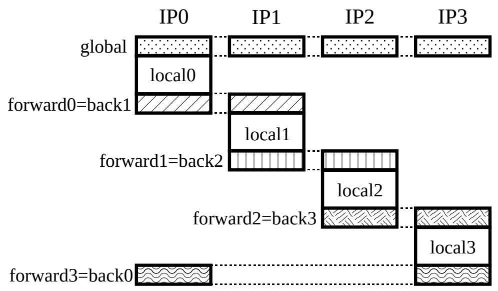

Fig.3 The configuration of the banked and overlapped register file.

moderate hardware complexity and register access flexibility. Furthermore, this register configuration provides hardware supports useful for software pipelining that will be discussed in Section 3.

2.4 The instruction set and the basic code scheduling scheme

The instruction set of an architecture reflects the function of the architecture. The instruction set of 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[10][11][12][13]. The length of Jetpipeline instructions is fixed to 32 bits.

The scalar instructions of Jetpipeline are designed based on the Stanford MIPS microprocessor architecture[10][14]. The scalar instructions include load/store, arithmetic operation and branch instructions. The branch instructions include the compare-and-jump instructions in addition to the branch instructions based on condition codes.

The vector instructions are designed referring the NEC SX supercomputer[15]. The vector instructions include vector load/store instructions, vector register instructions of integer and floating point operations, mask register operation instructions, and so on.

Instruction scheduling is one of the important issues affecting processor performance in a pipeline processor. The instruction scheduling of Jetpipeline is performed statically by software at compile time and there is no run time scheduling hardware like superscalar machines. To achieve the satisfactory performance of every operation units of Jetpipeline, the following basic scheduling strategies are necessary.

- The execution of an instruction that needs the results of its predecessor scalar load instructions or scalar floating point instructions must be delayed until the results are obtained. During this period, NOP or other instructions that do not depend on the execution order should be inserted.
- The technique of delayed branches is also adopted in Jetpipeline as the RISC architecture. Their delay slots are utilized to insert the instructions which do not affect to jump instructions.

3 Software pipelining for Jetpipeline

Software pipelining[9] is a technique effectively dealing with loop programs, which yields high instruction-level parallelism. It is applicable to RISC, CISC, superscalar, superpipelined, and VLIW processors to extract parallel processing capability of these processors. Therefore, we make use of software pipelining to perform loop scheduling, which is the crucial problem in instruction scheduling. In this section, we describe the software pipelining for Jetpipeline, and shows an scheduling example. We use the following source program example to explain our strategy.

```
for k := 1 to n do
  begin
      x[k] := q + y[k] * (r * zx[k+10] + t * zx[k+11]);
  end
```

The loop body of this example program is translated into assembly codes as follows, which only use local registers(%L):

```
 1: L5: ld    %L9,%L1,%L2   8:  add   %L6,%L2,%L2
 2:     mul   %L5,%L2,%L2   9:  st    %L7,%L1,%L2
 3:     ld    %L0,%L1,%L3  10:  addi  %L2,1,%L2
 4:     mul   %L4,%L3,%L3  11:  cmp   %L2,%L5
 5:     add   %L2,%L3,%L2  12:  jc    LE,L5
 6:     ld    %L4,%L1,%L3  13:  addi  %L1,4,%L1    ;delay slot of jc
 7:     mul   %L2,%L3,%L2
```

The software pipelining for Jetpipeline is performed based on four steps. They are: (1) extract innermost loop bodies; (2)calculate initiation interval; (3)unroll steady state; (4) rename registers. The following of this section gives a detailed discussion of these steps.

3.1 Calculating initiation interval

When software pipelining is applied to an innermost loop, both of the resource and the precedence constraints must be taken into account. The resource constraint means the number of execution units simultaneously available to execute software pipelined codes. The resource constraint in Jetpipeline can be considered as four, because Jetpipeline executes instructions issued from four IPs per cycle. On the other hand, the precedence constraint will appear when an instruction requires the result of its previous iteration. The iterations with the precedence constraint can not be initiated until the previous results are obtained. Thus, the initiation interval(II) can be determined by considering both resource constraints and precedence constraints. Because the example program has no precedence constraints in assembly code, II is given by the following equation:

II = [number of instructions / max (resource constraints, precedence constraints)]
 = [13 / 4]
 = [3.25]
 = 4

Here, [] is a ceiling function.

3.2 Unrolling steady state

When a software pipelined loop is in a steady state, several iterations of the original loop are processed in parallel. The variables of each iteration executed in parallel have the same name in the loop. If a certain register is assigned to such variables, the register is overwritten in every iteration. Therefore, correct results can not be obtained in such a situation, because register conflicts occur.

To solve this problem, we use multiple registers for one variable when variables of several iterations are allocated to the same register. It is necessary to unroll a steady state several times for allocating multiple registers to a variable. We determine the degree of unrolling based on lifetimes of the registers allocated to one variable. The degree of unrolling is obtained by dividing the maximum lifetime in the loop body with the initiation interval.

In the example program, the longest lifetime of registers is that of local register `%L2`, since `%L2` is alive from the first instruction to the ninth instruction. Therefore, we can obtain the degree of unrolling as shown in the following equation:

$$\begin{aligned} \text{N\_unrolling} &= [\max\text{ (lifetimes of each registers) / II}] \\ &= [9 / 4] \\ &= [2.25] \\ &= 3 \end{aligned}$$

Hence, the assembly sequence after software pipelining with the consideration of the initiation interval and the degree of unrolling can be shown as follows (only operation codes are shown here):

```
    IP1  IP2  IP3  IP4       IP1  IP2  IP3  IP4       IP1  IP2  IP3  IP4

 1  addi st   add  ld     5  addi st   add  ld     9  addi st   add  ld
 2  nop  addi ld   mul    6  nop  addi ld   mul   10  nop  addi ld   mul
 3  nop  cmp  mul  ld     7  nop  cmp  mul  ld    11  nop  cmp  mul  ld
 4  nop  jc   add  mul    8  nop  jc   add  mul   12  nop  jc   add  mul
```

Each of four-line blocks (1~4, 5~8, 9~12) shows the steady state and is unrolled three times, and one column shows instructions in an IP. After unrolling the steady state, each register is renamed to utilize the register file of Jetpipeline correctly.

3.3 Register renaming

The register reallocation of software pipelined codes is performed in order to assign independent registers to each variable of every iteration. Therefore, it is important to independently deal with each iteration in the software pipelined loop. The register file of Jetpipeline is divided into several banks, each of which includes local registers and overlapped registers. Iterations assigned to the local registers are kept independently.

When software pipelined codes are executed on Jetpipeline, each iteration occupies one instruction pipeline independently. However, when a new iteration starts, iterations currently being executed have to be shifted to their adjacent IPs, respectively, because of the resource constraint of Jetpipeline. It means that only four iterations can be executed simultaneously, since there are just four IPs. The register renaming strategy for software pipelined codes in Jetpipeline is simple. We just need to consider how to utilize the overlapped registers and the global registers. The overlapped registers are reallocated when an iteration is shifted to its adjacent IP. The global registers are assigned to a variable referred by all iterations. The other variables are allocated to the local registers. The following assembly sequence shows the results after renaming registers in the steady state of the example program.

```
IP1              IP2                  IP3                  IP4

addi %G1,4,%G1   st   %G7,%G1,%F2     add  %F2,%F3,%L2     ld  %G3,%G1,%L2
nop              addi %G2,1,%G2       ld   %G4,%G1,%L3     mul %G5,%L2,%B2
nop              cmp  %G2,%G5         mul  %L2,%L3,%L2     ld  %G0,%G1,%L3
nop              jc   LE,L5           add  %G6,%L2,%B2     mul %G4,%L3,%B3
```

The variables referred by every iteration are assigned to global registers(`%G`), which are the base addresses of the array in this example. The destination registers are assigned to the back block overlap registers(`%B`), when the destination registers are used as source registers in the next unrolled steady state. Of course, the source registers to read the results obtained in the previous steady state are also assigned to forward block overlap registers(%F).

4 The simulated performance studies

4.1 The simulation model

In order to evaluate the performance of Jetpipeline using software pipelining, simulation experiments have been carried out. We constructed a software simulator using the C language, in which all functional units depicted in Figure 2 are simulated except vector facility. In order to evaluate the effectiveness of software pipelining, the following assumptions are made.

- The scalar registers are banked and overlapped as shown in Figure 3. Thus, the maximum number of read/write accesses to each register bank in the same clock is limited to two.
- The accesses to the main memory are finished within one clock, and the memory latency are ignored because the benchmark programs are small enough to be contained in the cache.

4.2 The simulation results and analysis

We run some benchmark programs on the simulator to investigate the system performance of Jetpipeline. In our study, we chose Livermore loops[16] that are well used to evaluate performance of supercomputers as benchmarks. Livermore loops are composed of many kinds of loops in which some are easy to be parallelized and some are difficult.

In this section, we present the simulation results of five representative Livermore loops (KERNELs 1, 3, 5, 7, and 9). First, source programs of Livermore loops are compiled to instructions from the SPARC instruction set by the GNU C compiler with optimization, and then are translated into Jetpipeline assembly codes. Unrolling loops and scheduling based on software pipelining are semi-automatically performed. Register renaming and scheduling to solve constraints from the configuration of instruction pipeline are manually carried out. Assembly codes are assembled into the Jetpipeline machine codes by the assembler. The simulator executes these programs and outputs the number of clock cycles as execution time.

Table 1 shows the execution results in terms of execution cycles and speedup ratio. The speedups are measured against the execution cycles when the number of instruction pipelines is 1. The case using the dispatch stack[17] for parallelization is also shown in Table 1 for comparison. A compiler with the dispatch stack scheme parallelizes the sequential codes by means of only checking dependencies of operands.

KERNELs 1, 7 and 9 are easy to be parallelized and vectorized because they do not have dependency between iterations of their loops. Therefore, high speedup can be achieved as a result of its high degree of parallelism. Especially, in the case of software pipelining the speedup of 3.70 ~ 3.79 has been achieved.

KERNELs 3 and 5 have the dependency between iterations of their loop. Hence it is not easy

Table 1 Experimental results of simulations.

	Sequential	Dispatch stack		Software pipelining	
	cycles	cycles	(speedup)	cycles	(speedup)
KERNEL 1	19032	8015	(2.37)	5024	(3.79)
KERNEL 3	9019	5012	(1.80)	3344	(2.70)
KERNEL 5	3083	1543	(2.00)	1212	(2.54)
KERNEL 7	11503	5383	(2.14)	3111	(3.70)
KERNEL 9	6110	2107	(2.90)	1650	(3.70)

to parallelize and vectorize these programs. However, Jetpipeline using software pipelining achieves the speedup of 2.70 for KERNEL 3 and 2.54 for KERNEL 5.

In all cases, software pipelining has obtained sufficient speedups for loop programs. The performance of dispatch stack scheme is not so good as that of the software pipelining in our results. If appropriate preprocesses such as the loop unrolling were introduced into the dispatch stack scheme, the performance may be improved. However, this causes a large amount of overhead for data independency check among operands. The dispatch stack scheme also has an advantage that it can be applied to the non-loop part of a program. Therefore, the combination of these two schemes can be used for practical applications.

The simulation results are measured without the consideration of memory latency. However, the data transfer delay between registers and memory may affect overall performance. The complete simulator including these factors is currently under construction, and the consideration about the memory system such as data cache or statically code scheduling scheme which takes memory latency into account must be studied for practical implementation.

5 Conclusions

This paper proposed the software pipelining scheme for Jetpipeline. Jetpipeline absorbs the flexibility of superscalar machines and the conciseness of VLIW machines. Simulation results shows a good performance by using software pipelining in Jetpipeline. Not only can programs easy to be parallelized be effectively accelerated, but also programs difficult to be vectorized or parallelized can be speeded up. Therefore, Jetpipeline with software pipelining is a promising candidate for architectures exploiting instruction-level parallelism.

References

[1] T. Agewala et al., "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] M. Katahira et al., "Jetpipeline: A Hybrid Pipeline Architecture for Instruction-Level Paralellism," *Proc. High Performance Computing Conference '94*, September 1994(to be published).

[9] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," *SIGPLAN Notice*, Vol. 23, No. 7, pp.318-328, 1988.

[10] J. L. Hennesy et al., "MIPS:A VLSI processor architecture," *Technical Report No.223*, Computer Systems Laboratory, Stanford University, November 1981.

[11] D. A. Patterson and C. H. Séquin, "A VLSI RISC," *Computer*, pp.8-22, September 1982.

[12] J. L. Hennessy, "VLSI Processor Architecture," *IEEE Trans. Comput.*, pp.1221-1246, December 1984.

[13] D. A. Patterson, "Reduced Instruction set computers," *Comm. ACM*, pp.8-21, January 1985.

[14] T. Gross and J. Gill, "A short guide to MIPS assembly instructions," *Technical Note No.83-236*, Computer Systems Laboratory, Stanford University, November 1983.

[15] A. Jippo et al., "The supercomputer SX system: hardware," *Proc. of the second international conference on supercomputing*, Vol.1, pp.57-64, 1987.

[16] 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, pp.143-186, 1988.

[17] R. D. Acosta et al., "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Trans. Comput*., Vol.C-35, No.9, pp.815-828, 1986.