Automatic Resolution of Pipeline Hazards in Pipleline Synthesis of Instruction Set Processors

Ing-Jer Huang and Alvin M. Despain

CENG Technical Report 93-21

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-6006

May 1993

Automatic Resolution of Pipeline Hazards in Pipeline Synthesis of Instruction Set Processors

Ing-Jer Huang, Alvin M. Despain
Advanced Computer Architecture Laboratory
Department of Electrical Engineering - Systems
University of Southern California
ijhuang@usc.edu, despain@usc.edu

Abstract

Pipeline hazards may occur in a heavily pipelined hardware when inter-task dependencies are not satisfied. To avoid pipeline hazards, a common approach adopted in pipeline synthesis is to limit the task initiation interval to be no less than the minimal achievable latency [13]. However, this approach could be insufficient in synthesizing high performance pipelined circuits. The aim of our research is to achieve higher performance in pipelined application-specific instruction set processor synthesis by dealing with a subset of pipeline hazards (register related) with a hardware/software concurrent engineering approach.

The register-related pipeline hazards (RPHs) are caused by various types of inter-instruction dependencies. However, the conventional taxonomy for dependencies (data, anti-data, output) is insufficient for differentiating dependencies in a pipeline structure. We propose an extended taxonomy consisting of nine classes <forward/backward/stationary, data/anti-data/output> for the analysis of RPHs. Hardware and software resolution candidates are then provided for each class of dependency, including forwarding/duplicate registers in hardware and up/down instruction reordering in the compiler back-end. The complete resolutions are selected from the resolution candidates weighted according to the characteristics of application benchmarks. A set of analysis and synthesis tools were developed and integrated into our high level synthesis system Piper. The power of these tools are illustrated through the pipeline syntheses and design space exploration for several processors including industrial one.

Automatic Resolution of Pipeline Hazards in Pipeline Synthesis of Instruction Set Processors

1. Introduction

This paper presents a hardware/software concurrent engineering approach to the resolution of pipeline hazards in pipeline synthesis for application-specific instruction set processors (ISPs). The pipeline hazard is the major hurdle in pipeline synthesis. It degrades the pipeline performance and complicates the interaction between micro-architectures and compiler back-ends. We first investigate the complication of hardware and software.

Pipelining is an effective way to improve the throughputs of ISPs by overlapping the computation of consecutive instructions. However, pipelining may affect or modify both the hardware and software aspects of the systems. In hardware, micro-operations with higher degree of concurrency result in faster instruction firing rate at the costs of some extra hardware and more complex control such as duplicate functional units and bypassing buses. The relative timing of micro-operations is modified such that their external behavior may be different from the expected behavior of the original sequential semantics. The 'delayed load' in today's pipelined RISC processors is a typical example: the consumer instruction immediately following the producer 'load' will not be able to use the loaded data since they won't be available until several cycles later; therefore, the consumer of the data has to be scheduled several cycles after the producer, leaving some NO-OP slots between the producer/consumer pair. Thus the sequential semantics of instruction execution is not preserved in pipelined RISC processors.

The incompatibility between pipelined and sequential implementations calls for the modification of the software to compensate for such difference. Moreover, the software may be modified to take advantage of the hardware's pipeline structure. For example, modern compilers for pipelined ISPs have the explicit knowledge of the pipeline structures. A proper amount of NO-OPs is inserted between some particular dependent instruction pairs such as instructions after 'delayed load' or 'delayed branch', in order to preserve the desired behavior. The compilers can optimize the performance of the compiled codes by reordering instructions to eliminate the NO-OP slots. On the other hand, the limitation of compilers and the characteristics of application benchmarks may in turn pose their impact on the hardware pipeline structure. For example, the small average size of basic blocks in application benchmarks and the complexity of instruction reordering in a compiler back-end may void a highly pipelined hardware design: most of the stages are executing NO-OPs since the compiler is unable to find useful instructions to be scheduled into the NO-OP slots.

The driving force behind such hardware and software interactions is the pipeline hazards caused by the desire to highly pipeline the ISPs. A *pipeline hazard* happens when the previous instructions do not release resources in time for the next instruction that depends on the same resources such that the next instruction is prevented from executing during its designated cycle. Pipeline hazards are characterized into three categories: *structural, data*, and *control* [6]. They are

caused by the conflicting use of functional units, registers, and program counter, respectively. In fact, the control hazard is a special case of data hazards with the data being the program counter.

In this paper we will address the problem of analyzing and resolving register-related pipeline hazards (RPHs)[1] in pipeline synthesis for application-specific ISPs. First, we need a model to classify the RPHs. RPHs are caused by various types of inter-instruction dependencies. However, while trying to relate RPHs to these dependencies, we found that the conventional taxonomy of dependencies, i.e., data, anti-data, and output dependency (or read-after-write, write-after-read, and write-after-write, respectively) [6][13], is insufficient to encapsulate the relative timing of register accesses in pipeline stages. Therefore, we developed an extended taxonomy by differentiating each class of dependency into forward, backward, and stationary dependencies.

Second, we will provide hardware and software resolutions for each type of dependency. These resolution techniques include inserting forwarding/duplicate registers in hardware, and generating compilation guides (instruction reordering) to compiler back-ends. Third, we will provide the design procedure and algorithms that analyze and resolve RPHs. These techniques have been integrated into Piper, our pipeline synthesis system for instruction set processors. Given an abstract behavioral description of an ISP, Piper automatically generates a pipelined RTL implementation and an interface to the compiler back-end [7]. In this paper, Piper will serve as the platform for discussion and demonstration of these techniques.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 overviews the structure of Piper. Section 4 lists the extended taxonomy of inter-instruction dependencies and their relation to pipeline hazards. The hardware and software resolution strategies are outlined in Section 5. Section 6 presents the synthesis procedure of RPH resolutions. Section 7 shows examples of applying above techniques to synthesize some pipelined ISPs including industrial one. Section 8 reports the conclusion and future direction.

2. Related work

Most of the existing pipeline synthesis techniques, such as SEHWA [15], HAL [14], CATHEDRA II [4] and PLS [9], are developed for DSP applications. These synthesis techniques avoid pipeline hazards by limiting the degree of pipelining: in order to prevent pipeline hazards, the pipelined machines generated should not have a *task initiation interval* (the number of clock cycles between the firing of consecutive computing tasks) less than the *minimal achievable latency* (MAL) (the minimal number of clock cycles before the execution of the next task) [13]. Pipelined ISPs synthesized with these DSP-oriented techniques could suffer from the lower degree of pipelining. For example, the MIPS R2000, if it were synthesized with these techniques, would have to fire an instruction every two cycles (MAL=2), resulting in three pipeline stages as opposed to five [7]. Piper differs from these techniques in that Piper was developed for ISPs such that it is able to consider the interaction with the software environment (compiler back-end) in order to further increase the degree of Pipelining for ISPs.

1. The structural hazards exist where the functional units have multi-cycle execution time but are not fully pipelined [6]. This problem can be bypassed in a synthesis environment where this type of functional units is not supported.

ASPD improves the achievable degree of pipelining, at one instruction per cycle, with an enhanced percolation scheduling algorithm [1]. ASPD deals with the pipeline hazards by flushing the pipeline. As soon as an instruction which may cause hazards is decoded, the pipeline is flushed, no matter whether it really causes hazards or not. For example, whenever a 'delayed load' is decoded, the pipeline is flushed, regardless of whether the succeeding instructions depend on the 'delayed load'. This approach leaves no room for compilers to utilize the delay slots.

While these systems deal with hardware only, the PEAS project addresses the hardware/software concurrent engineering for ISPs [10]. It automatically generates a micro-architecture and software components such as compilers and simulators from a set of application benchmarks. A proper set of instruction is first selected from a pre-defined super set according to the benchmarks. A parameterized micro-architecture is then specialized. A compiler and a simulator are also customized from their super sets (the GNU's C compiler and simulators). The difference between Piper and PEAS is that PEAS currently does not handle pipelined processors, and deals with a super set of pre-defined instructions, while Piper does not select instructions and does not generate the complete compiler and simulator, but accepts any given instruction set written in a subset of Prolog [7], produces an interface to the compiler back-end, and perform pipeline synthesis.

Instruction reordering (or instruction scheduling) is a software technique used at compile time to avoid pipeline hazards in pipelined ISPs [6]. Instructions are reorganized such that a safe distance (proper number of other instructions including `NO-OP`s) is maintained between any pair of instructions that causes pipeline hazards. We will call the minimal safe distance as the *reorder distance* through the rest of the paper. Instruction reordering is often integrated into the back-end of the compiler. Several approaches have been adopted to reorder instructions. A simple approach is to view this problem as a peephole optimization problem [2]. The peephole optimizer keeps a window while scanning through the machine codes. Once a pair of instructions that causes pipeline hazards is identified, a proper number of `NO-OP`s is inserted between the pair, and other independent instructions from this window are reordered into the `NO-OP` slots to improve performance. More complicated approaches include rescheduling instructions within a basic block [5], a trace [3], or a region (a set of basic blocks) [17]. An instruction reorderer usually consist of two modules: a reorder engine (machine independent) and a reorder table (machine dependent). The reorder engine is the reorder algorithm itself, and the reorder table provides the engine with a set of reordering constraints specifying instruction pairs to be separated and the reorder distances to be kept. The time/space complexities of the reorder algorithm are usually related to the characteristics of the reorder table such as the size of the table. Therefore, Piper produces a reorder table for every pipelined ISP it synthesizes, and the characteristics of the reorder table to measure the performance/cost impact on the reorderer.

3. Overview of Piper: a high level synthesis system for pipelined instruction set processors and compiler back-ends [7]

Piper and its pre-processor, Fiper, serve as the behavioral domain of ADAS, a full-range design automation system for micro-processors [16]. They translate the abstract specification of an instruction set architecture (ISA), into pipelined register-transfer level designs consisting of data paths and control paths. Piper also generates an interface to the compiler back-end (reor-

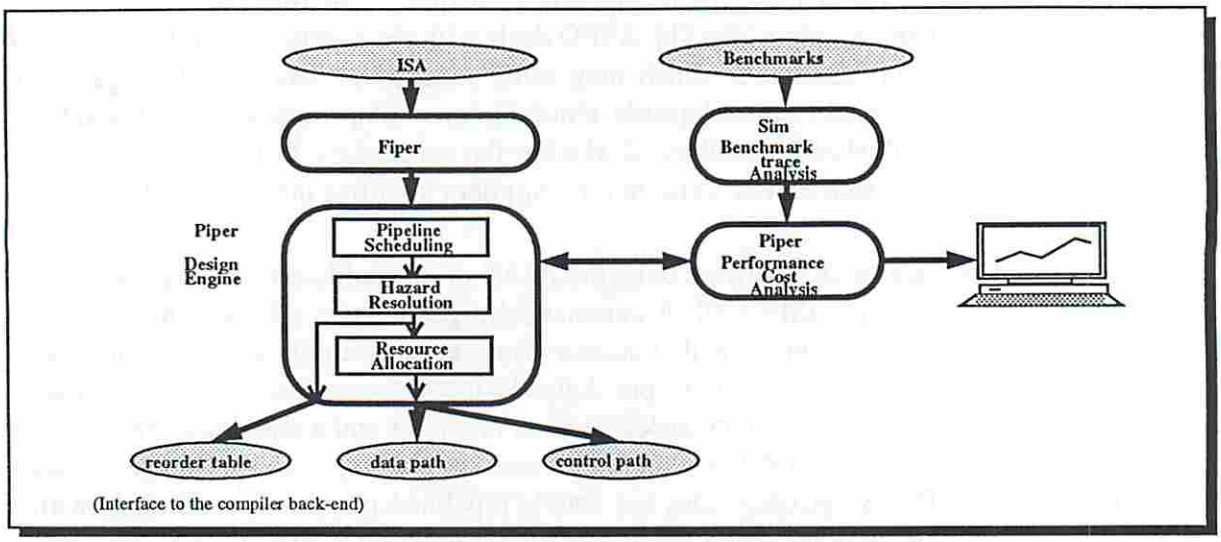


Figure 1 The structure of Piper system

derer), and measures its time and space complexities. The benchmark characteristics is applied to evaluate the quality of designs.

Figure 1 illustrates the conceptual structure of the Piper system. Fiper translates the ISA specification in Prolog into an abstract finite state machine. Piper takes the output of Fiper and performs the following tasks: (1). pipeline scheduling; (2) pipeline hazard resolution; (3). resource allocation.

The first phase, pipeline scheduling, assigns micro-operations into pipeline stages. Pipeline hazards may be introduced by the pipeline scheduler in highly pipelining cases. These hazards are resolved in the second phase by a combination of hardware and software resolution strategies. This is accomplished in two conceptual steps: analysis of inter-instruction dependencies, and application of resolution strategies. In this phase Piper generates a reorder table consisting of reordering constraints which instruct the compiler back-end (reorderer) to properly organize the codes for the pipelined machine synthesized. At the last phase, the hardware resources are allocated, producing a pipelined RTL level design.

Design decisions made by the pipeline scheduling and hazard resolution affect the performance and cost of hardware and the time/space complexities of the compiler back-end. Therefore, in addition to the design engine, there are a set of estimators and application benchmarks for the estimation of those effects.

4. The analysis of pipeline hazards

RPHs are caused by inter-instruction dependencies. To resolve a hazard, we have to determine the type of dependency it involves and choose an appropriate resolution strategy. We first provide a taxonomy of inter-instruction dependencies which consists of nine types, derived from the cross products of <forward/backward/stationary> and <data/anti-data/output>.

For simplicity, we assume a pipelined machine with stage time of one cycle throughout this section. With this assumption, a micro-operation of an instruction executed at its C'th cycle belongs to the C'th stage of the pipeline. The generalized case of stage time being multiple cycles will be discussed in Section 5.

4.1 An extended taxonomy of inter-instruction dependencies in pipelined ISPs

An inter-instruction dependency in a pipeline structure can be described in terms of the ternary tuple (P, R_p, R_s). R_p and R_s are the register access patterns (read/write) of the preceding and succeeding instruction, respectively. The conventional taxonomy of dependencies is encapsulated by this pair of parameters: data (R_p =write, R_s =read; or, read after write), anti-data (R_p =read, R_s =write; or, write after read), and output (R_p =write, R_s =write; or, write after write) dependency. On the other hand, P describes the relative position of register accesses in a pipeline structure of a dependent instruction pair. The possible values of P are: forward, backward, and stationary. Thus P provides a classification of dependencies from a pipeline structure's aspect. Figure 2 shows the relative positions for the register accesses of preceding and succeeding instructions. Suppose the instruction `instA` accesses register x at C_a'th cycle (at C_a'th stage), and the instruction `instB` accesses register x at C_b'th cycle (at C_b'th stage), with $C_a < C_b$.

Now we define the first two classes of dependencies with respect to C_a and C_b and the precedence of instruction pairs: forward and backward dependency. A *forward dependency* happens when a preceding instruction accesses a register at an earlier stage and a succeeding instruction accesses the same register at a latter stage such as the `instA-instB` pair (`instA` followed by `instB`); a *backward dependency* happens when a preceding instruction accesses a register at a latter stage and a succeeding instruction accesses the same register at an earlier stage such as the `instB-instA` pair (`instB` followed by `instA`). Note that both forward and backward dependencies potentially co-exist in the hardware for any pair of instructions that access the same register at different cycles (except the read-read case). The actual direction of the dependency (forward or backward) in an application program is determined by the relative precedence relationship of the instruction pair; for example, the `instA-instB` pair has a forward dependency and the `instB-instA` pair has a backward dependency.

The third class of dependency is *stationary dependency* which happens when the preceding and succeeding instructions access the same register at the same cycle ($C_a = C_b$).

After defining the forward, backward and stationary dependencies, let's further refine dependencies into nine types. By applying the conventional taxonomy to each class of dependency, we have forward data, forward anti-data, forward output, backward data, backward anti-data, backward output, stationary data, stationary anti-data, and stationary output dependency. Table 1 summarizes these nine types of dependencies.

4.2 Pipeline hazards and inter-instruction dependencies

A backward dependency causes a pipeline hazard because when the succeeding instruction reaches the stage where it accesses a register, the preceding instruction hasn't arrived at the stage (a latter stage) where it accesses the same register (as the `instB-instA` pair in Figure 2). For

The preceding instruction access register x at cycle C_p The succeeding instruction access register x at cycle C_s	Types of register accesses		
	Read after Write (R_p=write, R_s=read)	Write after Read (R_p=read, R_s=write)	Write after Write (R_p=write, R_s=write)
$C_p < C_s$ (P=forward)	forward data dependency	forward anti-data dependency	forward output dependency
$C_p > C_s$ (P=backward)	backward data dependency	backward anti-data dependency	backward output dependency
$C_p = C_s$ (P=stationary)	stationary data dependency	stationary anti-data dependency	stationary output dependency

Table 1 Inter-instruction dependencies for pipelined instruction set processors

example, a 'delayed load' instruction immediately followed by another instruction which uses the loaded data results in a pipeline hazard involving a backward dependency.

A forward dependency does not cause any pipeline hazard. The `instA-instB` pair in Figure 2 is an example of a forward dependency. Instruction `instA` accesses register x at stage C_a, leaving enough time (C_b - C_a +1 cycles) for the succeeding instruction `instB` to reach stage C_b to access X. However, even though a forward dependency does not cause any pipeline hazard, properly handling it may eliminate the associated backward dependency (for example, the `instB-instA` pair in Figure 2). We will further explain this issue in Section 5.

The stationary dependency does not cause any pipeline hazard, nor does it interfere with other classes of dependencies. The succeeding instruction can access the same register in the next cycle right after the preceding instruction's access. Instructions exhibiting stationary dependencies never access the same register simultaneously. There is no delay slot required for instruction pairs with stationary dependencies. Therefore, the stationary dependency is the most desired way of handling inter-instruction dependency in the pipeline synthesis. This translates to a design goal in

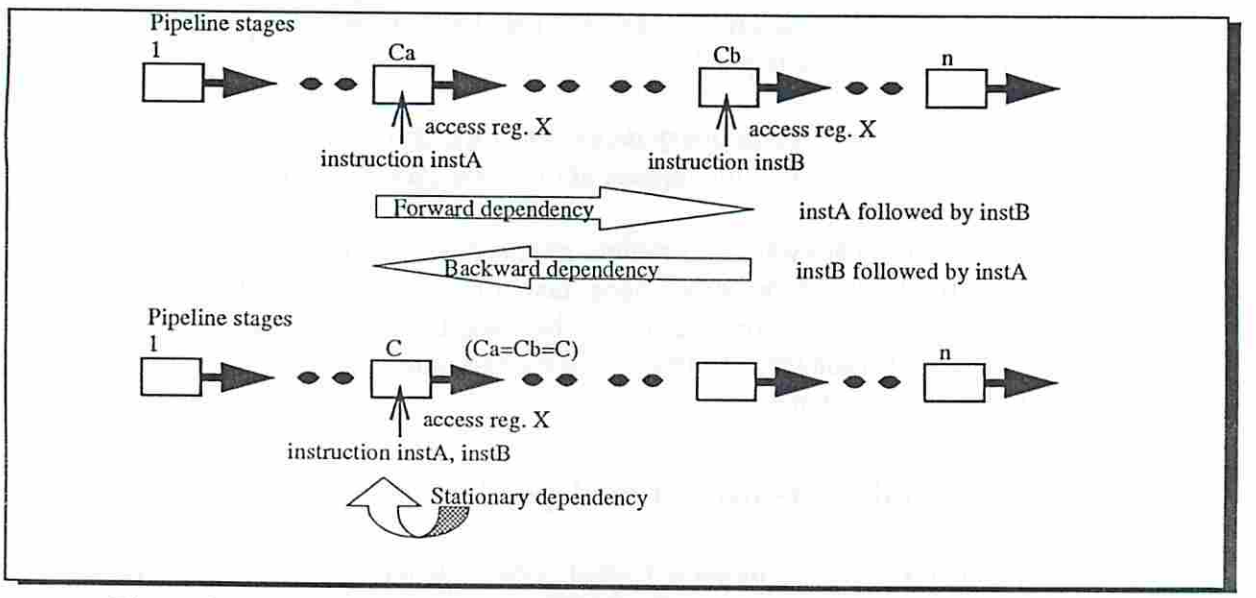


Figure 2 Forward, backward and stationary dependencies and pipeline stages

the scheduling phase of pipeline synthesis that requires accesses to the same register in different instructions be scheduled to the same cycle. However, this goal might not achievable with respect to other design constraints. For example, aligning register accesses to the same cycle may effectively lengthen the critical path. When a stationary dependency can not be preserved, forward and backward dependencies occur, which necessitate some forms of resolution to ensure the proper behavior.

In the following section, we will present the hardware and software resolutions for pipeline hazards caused by forward and backward dependencies.

5. Hardware and software resolution strategies

We first overview the general hardware/software resolution strategies, and then associate them with various types of pipeline hazards.

5.1 General hardware and software resolution strategies

Hardware resolution

For some RPHs, the most straightforward way to resolve them in hardware is to use additional registers. Two types of additional registers can be employed: forwarding and duplicate registers.

Forwarding registers carry the data along with the instruction stream in the pipeline. In Figure 3 a data D is written to register X by instruction `instA` in stage C_2, and is forwarded along the pipeline via forwarding registers (X_{f1},$X_{f2}X_{f3}$). The data moves along the pipeline synchronously with `instA`. The advantage of forwarding is that as soon as the current data of register X is forwarded to next stage, X is free for next data. This is analogous to adding extra latches (delays) in pipeline synthesis for DSP applications.

Duplicate registers release the burden of temporary registers. In Figure 4 (a) a temporary register T connects two sources S_1 and S_2 and two destinations D_1 and D_2. There are four possible connection patterns. All connections are mutually exclusive, with Y being the bottleneck of the data traffic. Suppose that the actual connections to be established by T are S_1->D_1 and S_2->D_2, and better performance will be achieved when these two connections can be made concurrently.

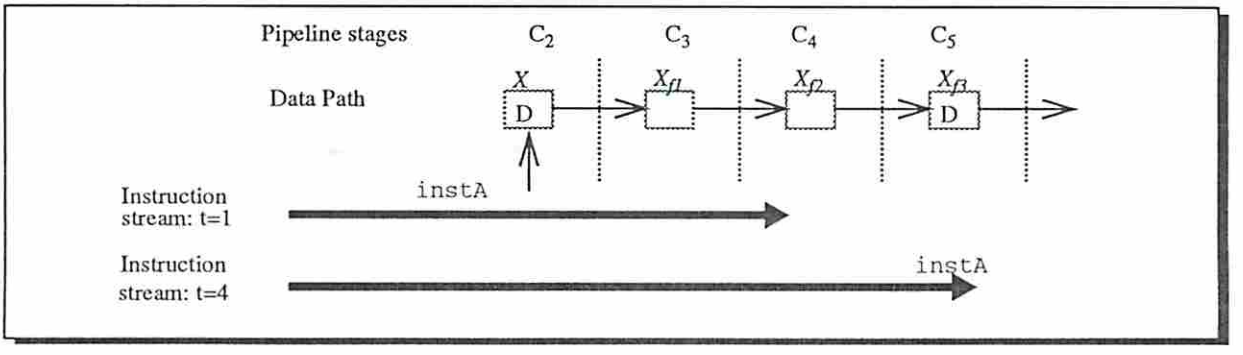


Figure 3 Hardware resolution: forwarding registers (X_{f1}, X_{f2}, X_{f3})

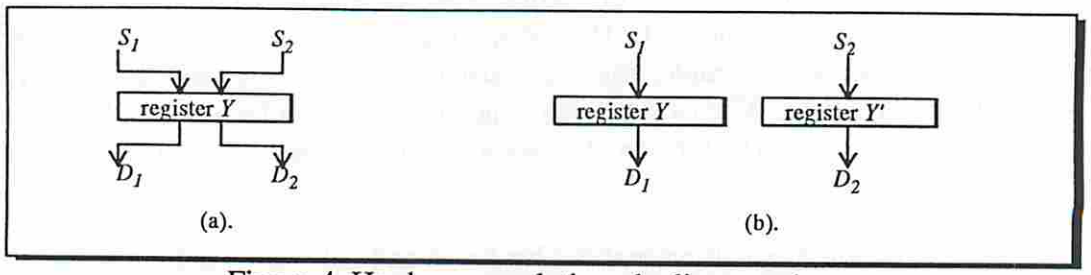


Figure 4 Hardware resolution: duplicate registers

Then adding a duplicate register Y' to create one additional data path will ease the traffic and improve the performance as shown in Figure 4 (b).

Software resolution

The major technique used in software (compiler back-end) to resolve pipeline hazards is instruction reordering. As described in Section 1, the desired behavior of an instruction stream may be distorted due to the change in the relative timing of micro-operations in pipelined ISPs. Instruction reordering restores the desired sequential semantics of an instruction stream by reordering the sequence of instructions.

There are two directions in reordering: up and down reordering. In Figure 5 (a) is a piece of sequential codes where instruction `instB` follows and depends on `instA`. The cases (b) and (c) of Figure 5 are the reordered codes for some pipeline structures. In case (b) the instruction `instB` is moved up and ahead of `instA`, whereas in case (c) `instB` is moved down and apart from `instA`. There are usually constraints (windows) about these movements: W_{up} being the maximal numbers of slots `instB` can be moved ahead of `instA` and W_{down} being the minimal number of slots `instB` has to be moved down from `instA`. For the example in Figure 5, W_{up} and W_{down} are three and two, respectively. We define W_{up} and W_{down} as the reorder distance for up and down reordering, respectively

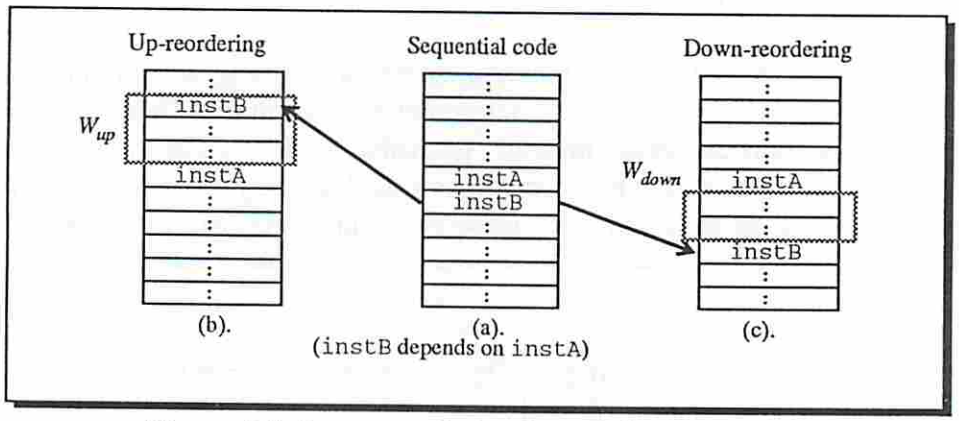


Figure 5 Software resolution: instruction reordering

5.2 Inter-instruction dependencies and their applicable resolutions

In this subsection we provide applicable hardware and/or software resolutions to pipeline hazards caused by forward or backward dependencies. For ease of discussion, we will use the same pipeline architecture and instruction pairs in Figure 2 as an example, assuming stage time of one

cycle. The generalized case where a pipeline stage takes multiple cycles will be provided in Section 5.2.3.

5.2.1 Forward dependencies

We first discuss forward dependencies. A forward dependency is the case of `instA-instB` pair (`instA` followed by `instB`) (Figure 2). As mentioned in Section 4.2, a forward dependency does not cause pipeline hazards; however, some resolutions can be optionally applied to improve the system performance.

Forward data dependency

A forward data dependency is the case where `instA` writes to register X and `instB` reads X. There are two ways to resolve this type of dependency: forwarding registers or up-reordering.

First, one forward register per stage can be allocated to stages C_{a+1} to C_b (total of C_b-C_a-1 forward registers). A side effect of this approach is that the associated backward anti-data dependency (`instB-instA` pair) is automatically resolved. This approach is preferable in the case where both `instA-instB` and `instB-instA` pairs happen very frequently in the application programs.

Secondly, instead of hardware resolution, one can choose to optionally move instruction `instB` ahead of `instA` (up-reordering) at most W_{up} ($W_{up}=C_b-C_a-1$) slots (cycles). This has the advantage of hiding the possible delay slots associated with the instruction `instB` (for example, `instB` being a 'delayed load' instruction), at the cost of longer compilation time in the compiler back-end.

Forward anti-data dependency

The forward anti-data dependency is the case where `instA` reads register X and `instB` writes to X. The applicable resolution is the up-reordering in the compiler back-end as described above with $W_{up}=C_b-C_a$ (assuming the register is master-slaved).

Forward output dependency

The forward output dependency is the case where both `instA` and `instB` write to register X. There are two ways of resolving this type of dependency: duplicate registers and up-reordering.

Duplicate registers eliminate the forward output dependency such that `instA` and `instB` become independent instructions. The side effect is that the backward output dependency (`instB-instA` pair) is also eliminated as well. Forward output dependencies can also be resolved by the optional up-reordering in the compiler back-end as described previously with $W_{up}=C_b-C_a-1$.

5.2.2 Backward dependencies

A backward dependency is the case of `instB-instA` pair (`instB` followed by `instA`). There is a software resolution (down-reordering) available for all backward dependencies. The

dependent instruction `instA` has to be moved away, downward from its predecessor `instB` for at least C_b-C_a+1 slots ($W_{down}=C_b-C_a+1$) to ensure that `instA` access the target register after `instB`'s access. The compiler back-end can fill in these slots with `NO-OPs` or reorder other independent instructions into these slots.

The backward output dependency can also be resolved in hardware with the same technique, duplicate registers, as in the case of forward output dependency. Circular dependency check has to be performed before a duplicate register can be inserted [8]. Unfortunately, there is no hardware resolution for backward data/anti-data dependencies.

5.2.3 Summary and extension

Here we summarize the resolution strategies for inter-instruction dependencies in Table 2. In this table we extend the resolution strategies for pipeline machines with multi-cycle stage time: every instruction spends S cycles in a stage before it advances to next stage. Therefore, a micro-operation executed at the $C'th$ cycle of its execution path will be executed at the `mod` $(C/S)+1$'th cycle of the $\lceil C/S \rceil$'th stage. In this table it is assumed that `instA` and `instB` access register X at the C_a'th and C_b'th cycles of their execution paths, respectively. Please note that the constant M is used to adjust for the case of master-slaved registers. Due to space limitation, we are not able to discuss the table in details. Interested readers please refer to [8].

Inter-instruction dependency	Hardware resolution	Software resolution
Forward data (`instA-instB`)	• Forward registers total number of forward registers: $\lceil (C_b-C_a+1-M^*)/S-1$	• Side effect of h/w resolution: backward anti-data dependency (`instB-instA`) is resolved
	• N/A	• Optional up-reordering: $W_{up}=\lceil (C_b-C_a+1-M^*)/S \rceil-1$
Forward anti-data (`instA-instB`)	• N/A	• Optional up-reordering $W_{up}=\lceil (C_b-C_a+M^*)/S \rceil-1$
Forward output (`instA-instB`)	• Duplicate register	• Side effect of h/w resolution: backward output dependency (`instB-instA`) is resolved
	• N/A	• Optional up-reordering: $W_{up}=\lceil (C_b-C_a)/S \rceil-1$
Backward data (`instB-instA`)	• N/A	• Down-reordering $W_{down}=\lfloor (C_b-C_a)/S \rfloor$
Backward anti-data (`instB-instA`)	• N/A	• Down-reordering $W_{down}=\lfloor (C_b-C_a-M^*)/S \rfloor$

Table 2 Hardware/Software resolution strategies for inter-instruction dependencies

Inter-instruction dependency	Hardware resolution	Software resolution
Backward output (`instB-instA`)	• Duplicate register	• Side effect of h/w resolution: forward output dependency (`instA-instB`) is resolved
	• N/A	• Down-reordering $W_{down}=\lfloor(C_b-C_a)/S\rfloor$

Table 2 Hardware/Software resolution strategies for inter-instruction dependencies

*. M= 1 for master-slaved registers; M=0 otherwise.

Having discussed the RPHs and their resolutions, we now present the design procedure adopted in Piper to automatically analyze the RPHs and apply the resolutions.

6. The procedure of pipeline hazard resolution

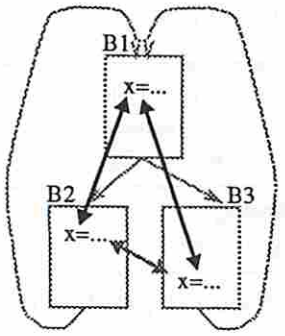
The pipeline hazard resolution phase takes as input a pipelined schedule, and outputs a set of hardware/software resolutions. This is accomplished in three steps. These steps are described in the following subsections.

6.1 Analysis of inter-instruction dependencies

In the first step, inter-instruction dependencies in the given pipelined schedule are identified. The inter-instruction dependencies appear as the *inter-iteration* dependencies in the pipelined schedule (pipelined loop body). There are two types of inter-iteration dependencies we are interested: *in-trace* and *cross-trace*. The in-trace dependencies are the dependencies that lie in the same execution trace within a single iteration, i.e., dependencies that can be detected without unrolling the loop. On the other hand, the cross-trace dependencies are those that lie across different execution traces within a single iteration, i.e., those that can be detected only when the loops are unrolled. For example, Figure 6 (a) shows a schedule consisting of three basic blocks (`B1`, `B2`, `B3`). The root block `B1` conditionally branches to block `B2` or `B3`. Blocks `B2` and `B3` are exclusive blocks and loop back to `B1`. There is a write to register `X` in every block. The two dark bi-directional arcs connecting register `X` accesses in block `B1` and `B2`, `B1` and `B3`, respectively, are in-trace dependencies, while the grey bi-directional arc connecting register `X` accesses in block `B2` and `B3` is a cross-trace dependency.

In Figure 6 (b) we present an algorithm to identify both in-trace and cross-trace dependencies without actually unrolling the loop. In the first step, global data flow analysis is performed on the loop body to identify the in-trace dependencies. While traversing through basic blocks, the analyzer records the earliest and latest read/write for each register, each exclusive block (such as `B2` and `B3` in Figure 6). In the second step, for each register and each set of exclusive blocks[2], cross-trace dependencies are generated. They are formed by pairing the earliest/latest register read/write

2. There may be more than one set of exclusive blocks. A set of exclusive blocks is that every block is exclusive to any other block in the set.



(a).

GIVEN: a pipelined schedule
OUTPUT: in-trace and cross-trace dependencies
1: Perform a global data flow analysis for the loop body in the schedule:
(1). identify any in-trace dependency;
(2). for each basic block belonging to a branch (exclusive blocks):
record the earliest/latest read and write for each register
2: Perform pairing between earliest/latest register accesses in basic blocks for each set of exclusive blocks:
(1). N= number of exclusive blocks;
(2). label exclusive blocks: 1 to N;
(3). for i= 1 to N do
for each register in exclusive block i
pair the earliest read with earliest and latest write in blocks i to N
pair the earliest write with earliest and latest read/write in blocks i to N
pair the latest read with earliest and latest write in blocks i to N
pair the latest write with earliest and latest read/write in blocks i to N
report these pairs as cross-trace dependencies

(b).

Figure 6 (a). in-trace and cross-trace dependencies;
(b). algorithm for in/cross-trace dependency analysis

of one block with the earliest/latest register read/write of the other block, for every pair of exclusive blocks within the set.

6.2 Generation and weight assignment of resolution candidates

For each inter-instruction dependency identified in the first step, all possible hardware and software resolutions are generated in the second step, according to Table 2. Some inter-instruction dependencies may have both hardware and software resolutions available. For such cases, we assign weights to hardware resolution candidates to help the designer to select the appropriate resolutions. In our current implementation, the weight is derived from the frequency in the application benchmarks and the reorder distance of the dependency that the hardware is to resolve. The following equation defines the weights. W_i is the weight assigned to the hardware resolution i;

$$W_i = \sum_{InstPair_i} Freq(InstPair_i) \times Dist(InstPair_i)$$

$InstPair_i$ is the instruction pair that contains the dependency the hardware is to resolve. $Freq(InstPair_i)$ is the frequency of the instruction pair $InstPair_i$ in the benchmark. $Dist(InstPair_i)$ is the reorder distance of the instruction pair $InstPair_i$ due to the dependency if it is not resolved by the hardware. Since a hardware resolution may resolve multiple dependencies (instruction pairs), the weight is calculated as a summation of the product $Freq(InstPair_i)Dist(InstPair_i)$ over all related instruction pairs. If application benchmarks are not available, an equal frequency is assumed. This equation intends to measure the hardware utilization and effectiveness of eliminating instruction reordering. However, the limitation of this simple model is that it does not consider the interaction between resolutions of different dependencies. We will examine this limitation in the example section.

GIVEN: selected hardware resolutions, software resolution candidates (reordering constraints), and inter-instruction dependencies
OUTPUT: software resolutions

1: For each hardware resolution
(1). delete the dependencies that is related to the hardware resolution;
(2). delete the software resolutions which belong to these dependencies;
(3). if the hardware resolution has a side effect on the related dependency, delete that dependency and its software resolutions

2: For the remaining dependencies, perform a merge process on their software resolutions (reordering constraints). A reordering constraint is a record with the format:
`ro(instruction1, instruction2, displacement)`
repeat the following until no further change can be made:
(1). if `ro(i1,i2,d1)` and `ro(i1,i2,d2)` exist and d1 > d2, delete `ro(i1,i2,d2)`;
(2). if `ro(allInsts,i2,d)` exists, delete all `ro(ix,i2,dx)` with dx < d
(3). if `ro(i1,allInsts,d)` exists, delete all `ro(i1,ix,dx)` with dx < d

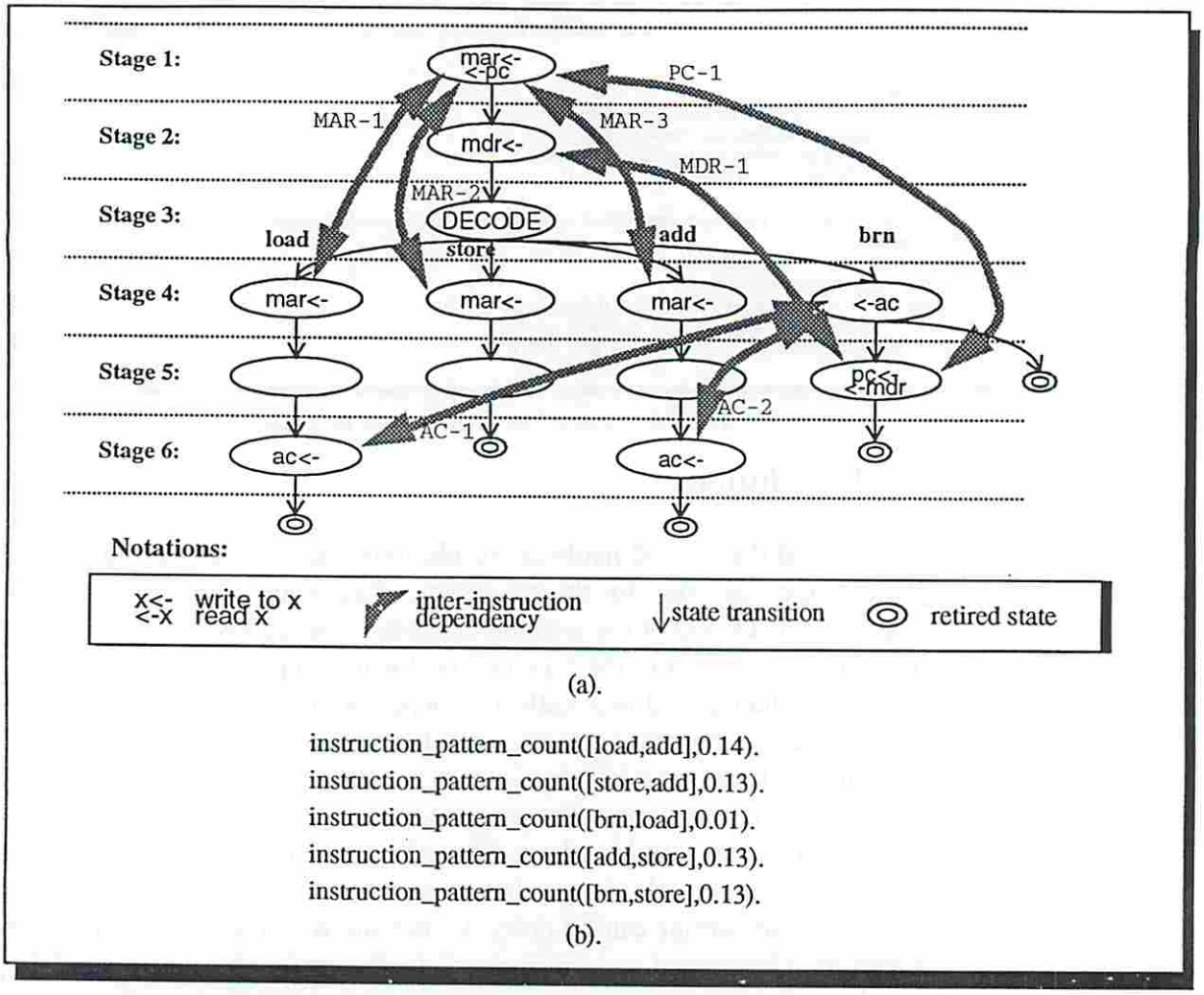
Figure 7 algorithm for the generation of software resolutions

6.3 Generation of final resolutions

After the designer has selected the desired hardware resolutions, the software resolutions can be obtained. Figure 7 presents the algorithm for the generation of software resolutions from the candidates. In the first step, the side effects of the selected hardware resolutions are examined. As described in Table 2, in addition to resolving the forward (backward) dependency it involves, a hardware resolution has a side effect of automatically resolving the associated backward (forward) dependency as well. Therefore, reordering constraints that are introduced by these dependencies are deleted from the software resolutions. In the second step, a merge process is performed on the remaining software resolutions. The purpose of this merge process is to remove the software resolutions that can be covered by others. For example, for down-reordering, suppose both `ro(add,load,3)` (three delay slots between `add` followed by `load`) and `ro(add,load,4)` exist, then the former can be deleted from the resolutions since it can be covered by the later constraint. The second step in Figure 7 is shown for the down-reordering case. Up-reordering constraints can be obtained with the second of the algorithm by interchanging '>' and '<'.

7. Examples

In this section, we applied our resolution techniques to exploring the design space of several ISPs. Specifically, we were interested in the sub-space of highly pipelined designs where pipeline hazards exist. We began with a four-instruction processor to demonstrate the design process, followed by the results for two real processors. Note that in order to simplify the presentation, we only list the down-reordering constraints for the software resolutions. In the following examples, the analytical model developed in [7] was used to evaluate the effective speedups (w.r.t. the non-pipelined design) and relative time complexities of the reorderer (compiler back-end) for pipelined designs synthesized with various resolution strategies.The reorder algorithm described in [5] was used as our reorderer. The time complexity of this algorithm glows linearly with the size of the reorder table (the number of reordering constraints).



(a).

```
instruction_pattern_count([load,add],0.14).
instruction_pattern_count([store,add],0.13).
instruction_pattern_count([brn,load],0.01).
instruction_pattern_count([add,store],0.13).
instruction_pattern_count([brn,store],0.13).
```

(b).

Figure 8 (a). Pipeline stages and state transitions for instructions: load, store, add, brn
(b). Instruction pair frequency

7.1 A small processor with four instructions

This small processor has a minimal achievable latency of five; i.e., any effort to pipeline it with an instruction initiation interval less than five will experience pipeline hazards. We now show how we can pipeline this processor efficiently at the instruction initiation interval of *one*, for which other pipeline synthesis techniques provide very limited synthesis power.

Figure 8 (a) shows the pipeline schedule of this small processor: a simplified finite state representation of the loop body and the pipeline stages. Bubbles in the figure represent states.The state in stage 1 sets the instruction address to MAR. The state in stage 2 fetches the instruction. The state in stage 3 is the decode state. The opcode is forwarded to latter stages. States in stage 1, 2, and 3 are active in every clock cycle. For every clock cycle, there is an active state in stage 4, 5, and 6, respectively. States in stage 4, 5 and 6 are conditionally executed, according to the opcodes in the instruction stream. The contents of bubbles are register accesses we are interested in this discussion. Empty bubbles contain RTLs which are not of interest here. The thick bi-directional arcs are forward/backward inter-instruction dependencies associated with these register accesses. These

arcs are labelled as PC-1, MAR-1, MAR-2, etc. There are fourteen dependencies in the figure, such as the forward anti-data dependency of PC, backward data dependency of PC (shown as the bi-directional arc PC-1), etc.

Figure 8 (b) shows the instruction pair analysis for a synthetic benchmark which we used to evaluate the various resolution strategies. The first field is the instruction pair: [preceding instruction, succeeding instruction]. The second field is the frequency with which the associated instruction pair exists in the dynamic execution trace. The average instruction level parallelism for this benchmark is about one since most of the instructions access the register AC. This observation implies that hardware resolutions may be preferable over software resolutions since there will be very few independent instructions which can be reordered into the NO-OP slots.

The hardware/software resolution candidates were generated for each dependency. Some of these candidates are listed in Table 3. Note that some reordering constraints can be covered by others. For example, reordering constraints derived for the instruction pair (allInstructions, brn) from dependency MDR-1 can be covered by the ones derived from PC-1. Two hardware resolution candidates are available: a duplicate register for mar (resolving MAR-1:f and MDR-1:b) and forward registers for mdr (resolving MDR-1:f). The weights assigned to these hardware resolution candidates by the benchmark analysis is 2.46 and 0.41, respectively, implying that the former hardware resolution candidate is much more effective than the latter.

Dependent Pair: (f=forward, b=backward)	Software Resolution (reorder constraint) <direction, proceeding inst., succeeding inst, reorder distance>	Hardware Resolution <type,target register, number,weight> (d=duplicate register, f=forward register)
PC-1: f	<up, all instructions, brn, 3>	
pc-1: b	<down, brn, all instructions, 4>	
AC-1: f	<up, brn, load, 1>	
AC-1: b	<down, load, brn, 2>	
MAR-1: f	<up, all instructions, load, 2>	<d, mar, 1, 2.46>
MAR-1: b	<down, load, all instructions, 3>	<d, mar, 1, 2.46>
MDR-1: f	<up, all instructions, brn, 2>	<f, mdr, 2, 0.41>
MDR-1: b	<down, brn, all instructions, 2>	

Table 3 Hardware/Software resolution candidates

The column 'Design 1' of Table 4 lists the design with software-only resolutions. Piper found four down-reordering constraints. The maximal, minimal, and average reorder distances are 4, 3, and 3.25, respectively. This design has an estimated speedup of 2.13 (w.r.t. non-pipelined design).

We now add a duplicate register for mar to resolve its output dependencies between stage one and three which involves instructions 'add', 'load' and 'store'. These patterns exist in our benchmark with a total frequency of 40%. This hardware resolution removes one down-reordering constraint, improves the performance by 78% (2.13 to 3.80 in speedup) and reduces the reordering complexity by 24.8% (1.33 to 1 in relative time complexity). This is shown in column 'Design 2' of Table 4. The speedup improvement is very significant since for every 'add', 'load', and 'store'

in the execution trace, one 'nop' has to be inserted to avoid the output conflict of `mar`, if it were not duplicated. (`mar` is a special register. A duplicate of it requires a two-port memory support).

The column 'Design 3' shows the design with the hardware resolution of forward registers. This hardware resolution resolves the forward data dependency `MDR-1`. The reordering constraints by dependency `MDR-1` require that two 'nop' be inserted after 'brn'. A forwarding register chain consisting of two registers can be allocated in stage three and four such that 'brn' can carry its own copy of `mdr` along the pipeline until it reaches stage five where it accesses `mdr`, leaving stage two immediately available for next instruction. However, this strategy does not introduce the anticipated performance improvement. None of the reordering constraint is eliminated. The reason is that the reorder distance for the `brn-allInstructions` pairs is dominated by the dependency `PC-1`, instead of `MDR-1` that the forward registers are to resolve.

The column 'Design 4' is the design with both hardware resolutions. The result shows that this design has the same performance and reordering constraints as 'Design 2'. Therefore, the forward registers are wasted resources. The situations of 'Design 3' and 'Design 4' illustrate the interaction between the resolutions for different dependencies associated with the same instruction pair, and expose the limitation of the simple weighting model described in Section 6.2: it is effective in sorting the importance of hardware candidates; however, it is insufficient in estimating the exact performance impacts of the hardware candidates, especially for designs with a combination of many hardware resolutions.

One observation about the reordering constraints for various designs of this small processor is that the maximal reorder distance remains as four while the number of constraints decreases slightly as the hardware resource is invested. This is because that the maximal reorder distance in this example is derived from the backward data dependency `PC-1` between stage one and five which does not have applicable hardware resolution.

Most of the pipeline synthesis techniques described in Section 2. are not suitable in synthesizing circuits with pipeline hazards, except ASPD. For this small processor, with the synthesis goal of keeping instruction initiation interval as one, ASPD would produce a pipelined design that flushes the pipe for 3, 3, 3, 5 cycles as soon as the instruction 'load', 'store', 'add', and 'brn' are decoded, respectively. This design decision actually slows down the pipeline significantly to speedup (w.r.t. to non-pipelined case) of less than 1.5, much less than our case (speedup=3.80 for the 'Design 2' in Table 4). However, ASPD does not generate any compilation information for the compiler back-end, which simplifies the hardware/software complication.

	Design 1	Design 2	Design 3	Design 4	ASPD's
Registers (h/w resolution) <type, target register, number> `dg` = duplicate register `fg` = forward register		<dg,mar,1>	<fg,mdr,2>	<dg,mar,1> <fg,mdr,2>	n/a
Down-reordering constraint (s/w resolution) <# of constraint, maximal reorder distance,minimal reorder distance, avg. reorder dist.>	<4, 4, 3, 3.25>	<3, 4, 2, 2.67>	<4, 4, 3, 3.25>	<3, 4, 2, 2.67>	n/a

Table 4 Designs with various combinations of hardware/software resolutions

	Design 1	Design 2	Design 3	Design 4	ASPD's
Estimated speedup of the given benchmark (w.r.t. non-pipelined)	2.13	3.80	2.13	3.80	<1.5
Relative time complexity of the reorderer (compiler back-end)	1.33	1	1.33	1	0

Table 4 Designs with various combinations of hardware/software resolutions

7.2 SM2a processor

The SM2a processor is a 39-instruction micro-processor with both general and special purpose registers. It was used by the Advanced Computer Architecture Laboratory at University of Southern California for the studies of Prolog compilation and design automation.The minimal achievable interval of this processor is five; i.e., any effort to synthesize this processor with an instruction initiation interval less than five will introduce pipeline hazards.

The results of the pipeline hazard resolutions for heavily pipelined SM2a (interval= 1, 2, 3, 4) are presented in Table 5. The second row lists the number of dependencies to be dealt with for each instruction initiation interval considered. This number implies the 'difficulty' of the problem. The third row lists the number of hardware resolution candidates for duplicate registers, and forward registers, respectively. The fourth row lists the hardware resolutions selected by the designer. For example, in the third column under interval=1, the designer selected one duplicate register and one forward register resolution. The entry with 'no' means that no hardware is selected. The fifth through eighth rows summarized the software resolutions with respect to the design decision made in the fourth row. The ninth row is the estimated speedup with respect to the non-pipelined case. The last row is the relative time complexity of the reorderer. The experiment took 25 seconds on a HP750 workstation.

Instruction initiation interval	4	3		2		1			
# of inter-instruction dependencies	54	90		342		484			
possible hardware resolutions (D=duplicate register, F=forward register)	-	1D		1D,2F		1D,4F			
selected hardware resolutions	no	no	1D	no	1D	no	1D	1D,1F	1D,3F
# of down-reordering	12	23	15	113	111	27	115	113	115
max. reorder distance	1	1	1	2	2	4	4	4	4
min. reorder distance	1	1	1	1	1	1	1	1	1
average reorder distance	1.00	1.00	1.00	1.10	1.10	3.20	2.20	2.22	2.20
estimated performance for the benchmark (speedup w.r.t. non-pipelined case)	1.48	1.10	1.97	1.64	1.76	1.73	2.37	2.37	2.37
relative time complexity of the reorder	1	1.92	1.25	9.42	9.25	2.25	9.58	9.42	9.58

Table 5 Results of pipeline hazard resolution for the sm2a processor

Several comments can be drawn from the results. First, the maximal speedup of pipelined SM2a synthesized with DSP-oriented pipeline techniques is 1.2. We were able to increase the speedup to 2.37 for the given benchmark, and 6 for benchmarks that have no inter-instruction dependency (the 1D1F design with instruction initiation interval of one). Second, The number of dependencies to be resolved grows fast when the instruction initiation interval is decreased. This is because that the higher the degree of pipelining, the larger the number of pipeline stages; the larger the number of pipeline stages, the more interactions between stages. Third, adopting more hardware resolutions does not necessarily reduce the number of reordering constraints. One of the reasons is that the adopted hardware may deleted a constraint of general cases such as `ro(load,allInstructions,5)` which originally covers `ro(load,add,3)` and `ro(load,sub,4)`. By deleting the general case, special cases such as the latter two will get exposed to the final software resolutions. Fourth, for each set of hardware resolution candidates, there exists a minimal subset which provides the maximal performance gain such as the subset (1D,1F) of the candidate set (1D,4F) of interval=1. Currently this subset is empirically identified through estimation and experiment. It appears to us that a systematic search for such subset is an interesting future research direction.

7.3 TDY-43 processor

The TDY-43 processor was designed about twenty years ago, and was used for aviation control in helicopters [18]. It has 256 instructions supporting fix-point, fractional, and two's complement operations on nine registers, a wide variety of addressing modes, and some external I/O controls. It was built on six boards. While it is still widely in service, its parts become obsolete and raise a difficult maintenance problem. Therefore, a customized single-chip reimplementation is desirable. The ADAS design automation system [16] was used to generate a gate-array implementation from the instruction set specification. One of the design decision to be made was whether to pipeline TDY-43 or not. Judging from its fairly complex instruction set, it was believed to be difficult. Therefore, one of Piper's synthesis task was to investigate such guess.

Instruction initiation interval	8	6	4		3		2		1
# of inter-instruction dependencies	3968	5167	7924		16307		47091		105654
possible hardware resolutions (D=duplicate register, F=forward register)	2D,1F	3D,2F	4D,7F		3D,8F		4D,19F		n/a
selected hardware resolutions	no	no	no	3D,2F	no	3D,7F	no	4D,19F	n/a
# of down-reordering	2180	2199	2406	2319	5245	5191	2814	11991	n/a
max. reorder distance	3	4	6	6	8	8	13	13	n/a
min. reorder distance	1	1	1	1	1	1	1	1	n/a
average reorder distance	2.14	2.93	4.36	4.48	3.24	3.27	7.95	2.62	n/a

Table 6 Results of pipeline hazard resolution for the TDY-43 processor

Table 6 shows some synthesis results for TDY-43. The design with interval=one was not completed since the computation could not finish within a reasonable time (more than two weeks).

The application benchmark was not available to us at the time of experiment. Therefore, we do not show the estimated performance with respect to the benchmark. The experiment confirmed that TDY-43 is difficult to pipeline, primarily due to complex instructions such as normalization of fractional numbers, multiplication/division, and shifting of arbitrary amount (0~64 bits). Even at the interval of eight, there are still 2180 reordering constraints, and two duplicate registers and one forward register are required. At this interval, most of the simple instructions can be completed within a single stage. The hardware/software resolutions are mainly for resolving the hazards for the instruction pairs involving those complex instructions. From this experiment we concluded that pipelining was not a feasible design style for TDY-43, instead, a sequential implementation was recommended. The experiment took about two CPU days on a HP750 workstation

8. Concluding remarks

We have presented a pipeline synthesis system for instruction set processors, Piper, which offers a greater synthesis power in providing higher pipeline rates that other pipeline synthesis techniques have difficulty or are not able to offer. This is accomplished by dealing with the pipeline hazards of a highly pipelined processors with a hardware/software concurrent engineering approach: we generate simultaneously both pipelined micro-architectures and their associated interface (reorder table) to the compiler back-end (reorderer). The key to the solution of the problem is our capability of systematically analyzing the potential pipeline hazards and adopting the appropriate resolutions.

We have proposed an extended taxonomy of inter-instruction dependencies for the analysis of register-related pipeline hazards in instruction set processors, and then presented hardware and software resolutions for the hazards. These resolutions include forwarding/duplicate registers in hardware, and up/down instruction reordering in software (compiler back-end). The register-related pipeline hazards are resolved according to the types of the inter-instruction dependencies they involve. In an application specific environment, the combination of hardware and software resolutions can be tuned towards the characteristics of the application benchmarks. We also have described the design procedure for pipeline hazard resolutions and the related algorithms. Both illustrative and real synthesis examples have been provided, and experiments of exploring the design space for those examples have been presented.

The promising applications of Piper are in two major design domains: reimplementation of existing ISPs and design aid for superscalar/superpipeline architecture designs. On one hand, as shown in our TDY-43 example, there are lots of ISPs built with obsolete architectures and technologies but they are still in active service. Maintenance is a serious problem for these ISPs. Low-cost reimplementation with current technologies and opportunities to patch the software environment are keys to improve their performance and extend their life spans. On the other hand, the current trend in computer architectures is shifting towards superscalar and superpipeline architectures [12], which usually exhibit fairly complex pipeline stages. Controlling the interlocking between these pipeline stages as well as investigating the hardware/software interaction become very difficult problems. Systematic approaches, other than manual ones, to these problems are necessary to handle such high design complexity. With its pipeline synthesis technique and hardware/software concurrent engineering approach, Piper is an appropriate design tool for these two design domains.

Current limitations include: (1). we are not able to resolve structural pipeline hazards since Piper does not handle multi-cycle non-fully-pipelined functional units (only single-cycle and fully-pipelined functional units are supported). (2). All various pipelined designs for an instruction set specification assume the same clock cycle length which is not realistic in a real design and may distort the performance estimation. (3). Better dependency analysis and resolution strategies for instructions involving register files are yet to be developed since the actual register access patterns can not be determined from the instruction set specification. (4). Better designs can be obtained if the analysis of pipeline hazards is fed back to the pipeline scheduling phase. (5). Finer control between hardware and software such as the architectural support for delayed branch with annulling slots is to be investigated. (6). More sophisticated approach to the systematic selection of the hardware resolution candidates is to be developed. These limitations will be our focus in future work.

Reference

[1] Mauricio Breternitz Jr. and John Paul Shen, "Architecture Synthesis of High-Performance Application-Specific Processors", *Proc. 27th DAC*, 1990

[2] Mike Carlton, Source codes of the Aquarius Prolog Compiler (Back-end), University of California, Berkeley, 1991

[3] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. on Computers*, Vol. 30, No. 7, 1981

[4] Gert Goossens, Jan Rabaey, Joos Vandewalle and Hugo De Man, "An Efficient Microcode Compiler for Application Specific DSP Processors," *IEEE Trans. on Computer-Aided Design, Vol. 9, No. 9*, September 1990

[5] John Hennessy and Thomas Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Tran. on Programming Languages and Systems*, July 1983, pp. 422-448

[6] John L. Hennessy and David A. Patterson, "Computer Architecture A Quantitative Approach," *Morgan Kaufmann Publishers*, pp. 257-278, 1990

[7] Ing-Jer Huang and Alvin Despain, "High Level Synthesis of Pipelined Instruction Set Processors and Back-End Compilers," *Proc. of 29th DAC*, June, 1992

[8] Ing-Jer Huang "Hardware/Software Resolutions for Pipeline Hazards in Instruction Set Processor," *CENG Technical Report 92-18*, University of Southern California, 1992

[9] Cheng-Tsung Hwang et al., "Scheduling for Functional Pipelining and Loop Winding", *Proc. 28th DAC*, 1991

[10] Masaharu Imai, Alauddin Alomary et al., "An Integer Programming Approach to Instruction Implementation Method Selection Problem," *Proc. of Euro-DAC*, 1992

[11] Gerry Kane, *MIPS RISC architecture*, Prentice Hall, 1989

[12] Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall, 1991

[13] Peter M. Kogge, *The Architecture of Pipelined Computers*, MacGraw-Hill, 1981

[14] Pierre B. Paulin and John P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Trans. on Computer-Aided Design, Vol. 8, No. 6*, June 1989

[15] Nohbyung Park and Alice Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," *Trans. on CAD, Vol. 7, No. 3*, March 1988

[16] Iksoo Pyo, et al., "Application-Driven Design Automation for Microprocessor Design," *Proc. 29th DAC*, June, 1992

[17] Ching-Long Su and Alvin Despain, "*An Instruction Scheduler and Register Allocator for Prolog Parallel Microprocessors,*" International Computer Symposium, 1992

[18] *Programming Manual for the Teledyne TDY-43 Computer*, Teledyne Systems Company, 1988