An Extended Classification of Inter-instruction Dependency and Its Application in Automatic Synthesis of Pipelined Processors†

Ing-Jer Huang and Alvin M. Despain

Advanced Computer Architecture Laboratory
Department of Electrical Engineering — Systems
University of Southern California

Abstract

The conventional classification of inter-instruction dependencies (data, anti and output dependencies) provides a basic scheme for the analysis of pipeline hazards in pipelined instruction set processors. However, it does not consider the relative spacial positions of micro-operations in the pipeline, thus providing limited hints to hardware designers and compiler writers about the hazard resolution in generalized pipeline structures. In this paper we propose an extension to the conventional classification of dependencies, which is capable of encapsulating the spacial/temporal relationship and providing precise hardware/software resolution strategies. The extension is derived by differentiating each class into three sub-classes: forward, stationary, and backward, according to the register access patterns in the pipeline.

Hardware and software resolutions are associated to each class of dependency. These resolutions include forwarding and duplicate registers in the micro-architecture and instruction reordering (up/down) in the compiler back-end. With the extended classification and its associated hardware/software resolution strategies, we are able to systematically analyze the potential register-related pipeline hazards for a given pipeline structure, determine appropriate resolution strategies, and explore the tradeoff between hardware and software complexities. The methodology enables the systematical synthesis of high performance pipelined micro-architectures, and is useful to derive the back-end of the supporting compilers.

Keywords: *inter-instruction dependency, pipeline hazard resolution, high level synthesis, compiler back-end generation, hardware/software tradeoffs*

1. Introduction

We will present an extension to the conventional classification of inter-instruction dependencies for pipelined instruction set processors (ISPs). We will show how it can be used to help design automation systems to improve the throughput of synthesized pipelined ISPs as well as to generate an interface to the compiler back-end so as to explore the hardware/software design space.

Pipelining is a well-recognized technique to improve system performance by pursuing instruction-level parallelism (ILP). However, the efficiency of pipelining is often defeated by pipeline hazards, which prevent instructions from being fired at the maximal rate. There are two categories of pipeline hazards: *structural* and *data dependent*[1] [4][11]. The structural hazards can be resolved by allocating sufficient hardware. However, the data dependent hazards, caused by inter-instruction dependencies in the instruction stream, require more careful consideration in both hardware and compiler back-ends. In this paper we will focus on the detection and resolution of the data dependent hazards.

Conventionally, the data dependent hazards are associated with a class of three inter-instruction dependencies: *data*, *anti*, and *output* dependencies (or read-after-write, write-after-read, and write-after-write, respectively) [9]. Although this classification provides helpful insight into the understanding of detection and resolution of hazards in some simple pipeline structures such as typical five-stage F-D-E-M-W (fetch, decode, execute, memory access, register write back) machines, it is not powerful enough for general pipeline structures. In general pipeline structures such as pipelined CISC architectures, register accesses may happen in many pipeline stages in order to support multiple-cycle instructions. For example, consider a seven-stage ISP shown in Figure1. Suppose that the instruction

†. This work was supported by the DARPA under contract No. J-FBI-91-194.

1. There is a third category *control* hazard defined in [4]. A control hazard involves the change of the program flow, which is due to the change of program counter’s value. If we treat the program counter as one of the regular registers, then the control hazards can be encapsulated as a special case of data dependent hazards.

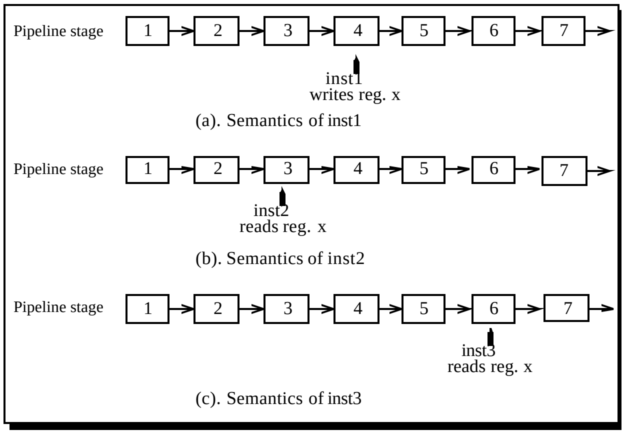

Figure 1. Instruction semantics at the micro-architecture level

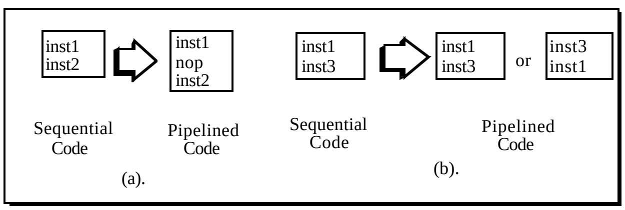

Figure 2. Inter-instruction dependencies vs. pipeline hazards

semantics and the pipeline implementation require that instruction inst1 write to register X in stage 4 (Figure 1 (a)), and instructions inst2 and inst3 read register X in stages 3 and 6, respectively (Figure1 (b) and (c)). Under this condition, both the instruction pairs inst1-inst2 (inst1 followed by inst2) and inst1-inst3 (inst1 followed by inst3) exhibit data dependency on register X. However, only the former pair causes a pipeline hazard, which forces the pair to be separated by one nop as shown in Figure2 (a). The latter pair does not cause any pipeline hazard; instead, this pair can be scheduled as in the non-pipelined case. This even allows inst3 to be moved ahead of inst1 and still retains the correct semantics, as shown in Figure2 (b). An example of this case is where a 'compare' instruction generates the status bit at the fourth stage, and the following 'branch' instruction checks the status bit at the eighth stage. This feature gives the possibility of scheduling the 'branch' ahead of the 'compare'. A technique called 'prophetic branch' was developed to take advantage of this feature [14].

This observation motivated us to develop an extension to the conventional classification of inter-instruction dependencies, based on the register access patterns in the pipeline. In the extension, each class of dependency defined in [9] is differentiated further into three classes: *forward*, *backward* and *stationary*. For each class of dependency we are able to associate with it applicable hardware and/or software resolution strategies, including forwarding and duplicate registers in hardware, and instruction reordering (up and down) in compiler back-ends. The extension and its hardware/software resolutions provide a better understanding of pipeline hazards in general pipeline structures and a systematic approach to the analysis of hardware/software tradeoff in pipeline hazard resolution.

To validate the potential contribution of our extension, we found an interesting application in high level synthesis (design automation) for pipelined ISPs. The basic task of high level synthesis for pipelined ISPs is to generate a pipelined micro-architecture from a behavioral (sequential) description of any given instruction set specification (either RISC or CISC styles). However, due to the limited power of pipeline synthesis techniques currently available, the automatically synthesized pipelined ISPs can not achieve the same throughputs as in manually designed ones in most cases, as pointed out in [2] and [5]. The key bottleneck is the lack of a thorough understanding of pipeline hazards with respect to the inter-instruction dependencies and the possible resolution strategies. Our proposed classification helps bridging this gap. And since our resolution strategies include both hardware and software, a high level synthesis system incorporating our method is able to synthesize both pipelined micro-architectures and interfaces to compiler back-ends. This approach makes it possible to explore the hardware and software design spaces simultaneously.

The rest of the paper is organized as follows. Section 2 presents the extended classification of inter-instruction dependencies. Section 3 associates each class of dependency with applicable hardware/software resolution strategies. Section 4 describes an application of our classification to high level synthesis. Section 5 shows some experiments with automatically synthesized ISPs based on the proposed classification. Section 6 concludes with a discussion of the contributions and limitations of the current work, and some future directions in how to extend our work for superscalar processor designs.

2. The extended classification of inter-instruction dependencies

Data dependent pipeline hazards 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 classification of inter-instruction dependencies that consists of nine classes, derived from the cross products of <forward/backward/stationary> and <data/anti/output>.

The general model of the pipeline structures that we consider in this paper is a single pipeline, with linearly cascaded stages and constant stage latency[2]. This model is

2. The *stage latency* is the number of clock cycles that an instruction spends in one pipeline stage before it advances to next stage.

similar to the pipelined SISD machine model defined by Kogge in [11], with the extension that the execution phase of instructions allows multiple register accesses and ALU operations. These operations may span multiple pipeline stages. For simplicity, we assume a pipelined machine with one-cycle stage latency 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 being stage latency of multiple cycles will be discussed in Section 3.

2.1. Definitions

To make the following discussion valid, we assume that a pair of dependent instructions travelling along the pipeline closely enough such that the pipeline structure will interfere with the dependencies of the pair.

An inter-instruction dependency in a pipeline structure can be described in terms of the triple (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 classification of dependencies is encapsulated by this pair of parameters: data (R_p =write, R_s =read; or, read after write), anti (R_p =read, R_s =write; or, write after read), and output (R_p =write, R_s =write; or, write after write) dependencies. On the other hand, P describes the relative position of register accesses by a dependent instruction pair in a pipeline structure. The possible values of P are: forward, backward, and stationary. Thus P provides a classification of dependencies with a pipeline structure's point of view: a new dimension is created to differentiate the relative timing behavior, in addition to the conventional classification for dependencies. Figure3 (a) 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 the pair (C_a, C_b) and the precedence relationship 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 stages (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.

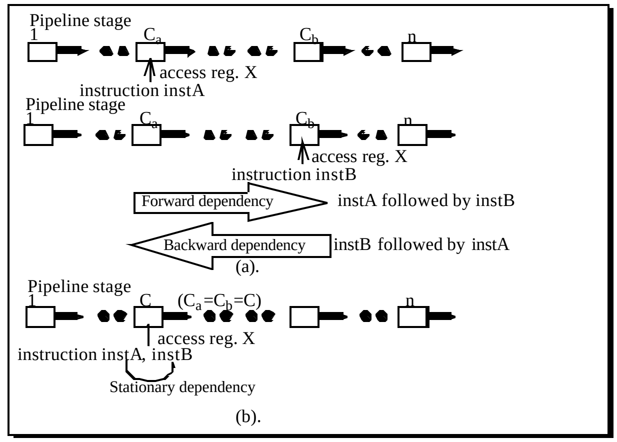

Figure 3. Forward, backward and stationary dependencies and pipeline stages

The third class of dependency is *stationary dependency* which happens when the preceding and succeeding instructions access the same register at the same stage ($C_a = C_b$) as shown in Figure3 (b).

Please note that our definition of inter-instruction dependency is different from the definition in [11] in that our dependency is defined based on the access of a single register, as opposed to a set of registers. In [11], for every instruction, a *domain* and a *range* are defined as the set of registers that the instruction reads and writes, respectively. A pair of instructions exhibits an inter-instruction dependency as long as their domains or ranges have some overlap. This definition is insufficient since a single instruction may access more than one register, resulting in different types of pipeline hazards, each of which requires a different resolution. Therefore, a pair of dependent instructions may be involved in more than one class of dependency. With our classification, these hazards can be differentiated such that each dependency can be separately resolved.

The forward, backward and stationary dependencies can be further refined into nine types. By applying the conventional classification to each class of dependency, we have forward data, forward anti, forward output, backward data, backward anti, backward output, stationary data, stationary anti, and stationary output dependency. Table1 summarizes these nine types of dependencies.

2.2. Pipeline hazards and inter-instruction dependencies

A pair of instructions exhibiting a backward dependency, when travelling closely in a pipeline, will cause a pipeline hazard because when the succeeding instruction reaches the stage where it accesses a register, the preceding

The preceding instruction accesses register X at cycle C_p The succeeding instruction accesses 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 dependency	forward output dependency
$C_p > C_s$ (*P* = backward)	backward data dependency	backward anti dependency	backward output dependency
$C_p = C_s$ (*P* = stationary)	stationary data dependency	stationary anti dependency	stationary output dependency

Table 1: Inter-instruction dependencies for pipelined instruction set processors

instruction hasn't arrived at the stage (a latter stage) where it accesses the same register (as the instB-instA pair in Figure3). For example, a 'delayed load' instruction immediately followed by another instruction that uses the loaded data in an earlier stage results in a pipeline hazard involving a backward dependency.

A forward dependency does not cause any pipeline hazard. The instA-instB pair in Figure3is 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 Figure3). We will further explain this issue in Section 3.

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 design. This translates to a design goal that requires accesses to the same register in different instructions be scheduled to the same pipeline stage. However, this goal might not be achievable due to other design constraints. For example, aligning register accesses to the same pipeline stage 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 hardware and software resolutions for pipeline hazards caused by forward and backward dependencies.

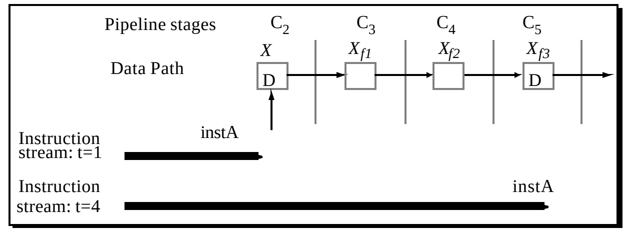

Figure 4. Hardware resolution: forwarding registers (X_{f1}, X_{f2}, X_{f3})

3. Hardware/software resolution strategies for pipeline hazards

The most powerful and practical way of resolving data dependent pipeline hazards is a data-flow like approach, such as Tomasulo's approach [15]. Data are assigned tags, and a data flow engine, such as the reservation station and its supporting circuitry in Tomasulo's approach, is used to track the relationships among data. In this paper, we consider an alternative approach employing relatively simple hardware and software techniques. An advantage of this approach is that it takes advantages of the classification of dependencies we presented in the previous section. We first summarize these resolution techniques in Section 3.1 and relate them to pipeline hazards in Section 3.2.

3.1. General hardware and software resolution strategies

Hardware resolution

The most straightforward way to resolve hazards 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. For example, in Figure4 a datum D is written to register X by instruction instA in stage C_2 at time t=1, and is forwarded along the pipeline via forwarding registers (X_{f1}, X_{f2}, X_{f3}) until D arrives at stage C_5 at time t=4. The datum moves along the pipeline synchronously with instA. The advantage of forwarding is that as soon as the current datum of register X is forwarded to next stage, X is free for more data. This is analogous to adding extra latches (delays) in a pipeline in order to improve the system throughput [12].

Duplicate registers release the burden of temporary registers. In Figure5 (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 T 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. Adding a duplicate register T', to create one additional data path, will ease the traffic and improve the

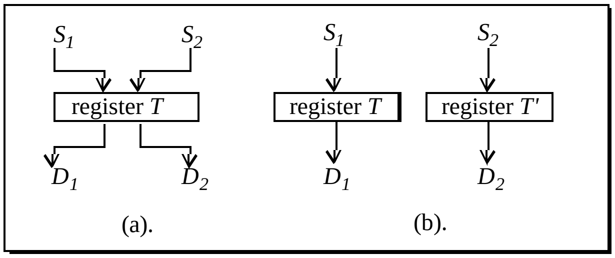

Figure 5. Hardware resolution: duplicate registers

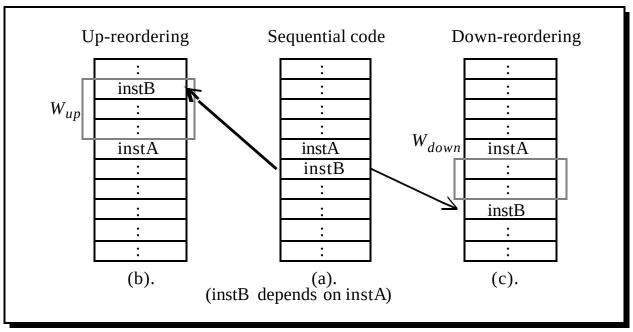

Figure 6. Software resolution: instruction reordering

performance as shown in Figure5 (b).

Software resolution

Instruction reordering is the major technique used in compiler back-ends to resolve pipeline hazards. The desired behavior of an instruction stream may be distorted by pipelining. 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 6 (a) is a sequential code where instruction instB follows and depends on instA. The cases (b) and (c) of Figure6 are the reordered codes for two different 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) to 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 Figure6, 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.

3.2. Inter-instruction dependencies and their applicable resolution

In this subsection we provide applicable hardware and software resolution strategies to pipeline hazards caused by forward and backward dependencies. As pointed out in Section 2.2, the stationary dependencies do not cause any hazard. Therefore, they do not require any resolution. For ease of discussion, we will use the same pipeline architecture and instruction pairs in Figure3 as an example, assuming stage latency of one cycle. The generalized case where a pipeline stage takes multiple cycles will be provided in Section 3.2.3.

3.2.1 Forward dependencies

A forward dependency is the case of the instA-instB pair (instA followed by instB) (Figure3). As mentioned in Section 2.2, a forward dependency does not cause pipeline hazards; however, some resolutions can be optionally applied to improve the system performance.

Forward data dependency

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-1} (totally C_b-C_a-1 forward registers). A side effect of this approach is that the associated backward anti dependency (the 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 has a ‘delayed’ operation to other register *Y*), at the cost of longer compilation time in the compiler back-end.

Forward anti dependency

The applicable resolution is the up-reordering in the compiler back-end as described previously with W_{up}=C_b-C_a (assuming the register is master-slaved).

Forward output dependency

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 (the instB-instA pair) is 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.

3.2.2 Backward dependencies

A backward dependency is the case of the 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 nops 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. The side effect of the hardware resolution is that the associated forward output dependency is automatically resolved. Circular dependency check has to be performed before a duplicate register can be inserted [6]. Unfortunately, there is no hardware resolution for backward data and anti dependencies.

3.2.3 Summary and extension

Here we summarize the resolution strategies for inter-instruction dependencies in Table2. In this table we extend the resolution strategies for pipeline machines with multi-cycle stage latency: 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 the same register 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-slave registers.

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

Table 2: Hardware/Software resolution strategies for inter-instruction dependencies that are not hazard-free

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

The first column contains the six classes of inter-instruction dependencies that require resolution. (Remember that the stationary dependencies do not cause pipeline hazards.) For each class of dependency, a dependent instruction pair taken from Figure 3 is listed as an example. Applicable hardware and software resolution strategies are listed in the second and third columns for each class of dependency. In the second column we also list the side effects of hardware resolutions. Note that forward data and forward/backward output dependencies have both hardware and software resolution candidates available, which allow tradeoff between hardware and software. The tradeoff can be based on the characteristics of the application domain (the frequency of the dependency in the applications) and the time/space complexities of the compiler back-end. We will present a way to perform the tradeoff analysis in Section 4.3, and show examples in Section 5.

Having discussed the classification and their resolutions, we now present their application in high level synthesis to automatically analyze pipeline hazards and apply the resolutions.

4. The application in high level synthesis for pipelined ISPs

In this section we first point out the inadequacy of typical high level synthesis systems for pipelined ISPs and then present how our classification can be incorporated into the synthesis process to improve the design quality. Finally, we show how a high level synthesis system is able to explore the hardware/software design space with our classification and resolution strategies.

4.1. The inadequacy of current synthesis techniques

The basic task of high level synthesis for pipelined ISPs is to construct pipelined micro-architectures at the RTL level, given any behavioral (abstract and sequential) specification of the instruction set architecture to be synthesized. It is assumed that the machine to be synthesized repeatedly executes some computation task. For ISPs, the computation task is the operations of fetch, decode and execution of instructions. The computation task is represented as a loop body in the specification, and the repetition is represented as the iteration of the loop body. Figure7 (a) shows a state transition diagram of a three-instruction ISP: ovals are states with each state taking one clock cycle, thin arcs are state transitions, and the outer feedback arc represents the iteration of the body. For simplicity we only show register accesses of PC and X in the states. The two grey bi-directional arcs are inter-iteration dependencies[3] (inter-instruction dependencies) for registers PC and X, respectively.

The task of pipeline synthesis is then to find an appropriate overlapping of multiple iterations of the loop body subject to the inter-iteration dependencies. As shown in Figure7 (a), the longest inter-iteration dependency spans

3. The inter-instruction dependencies of ISPs appear as inter-iteration dependencies in the behavioral specification.

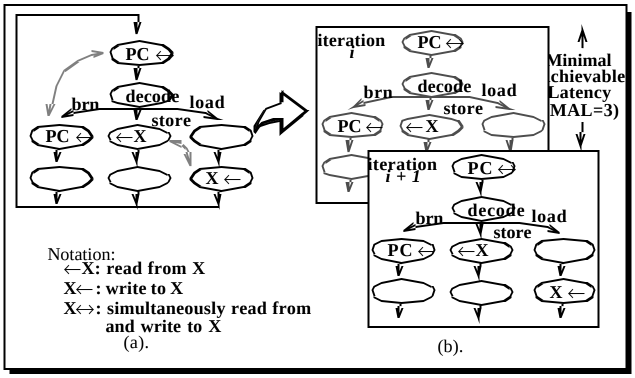

Figure 7. Pipeline synthesis and inter-iteration dependencies

three cycles (the PC dependency); therefore, the optimal degree of overlapping without causing pipeline hazards is to issue an iteration every three cycles as shown in Figure7 (b). The length of the longest inter-iteration dependency is called *minimal achievable latency* (MAL) [7], which has been used by several pipeline synthesis systems as the lower bound of the instruction initiation latency. There are several techniques available that try to minimize MAL by rearranging the micro-operations, in order to increase the degree of overlapping [2][7]. The limitation of these techniques is that they are still subject to the constraint of MAL. Pipelined ISPs with instruction initiation latency less than MAL are impossible. For example, the MIPS R2000 [8] has MAL of two. If it were synthesized with these techniques, it would have instruction initiation latency of two, instead of one as in the manually designed one.

There is one exceptional synthesis system ASPD [1] which is capable of synthesizing pipelined ISPs with instruction initiation latency of one, regardless of their MALs. To resolve the induced pipeline hazards, the synthesized machine flushes the pipeline as long as an instruction which may cause hazards is decoded. Take the MIPS R2000 as an example, if it were synthesized by ASPD, the pipeline would flush as soon as ‘delayed load’ and ‘brn’ are decoded, regardless whether the succeeding instructions are dependent instructions or not. This approach leaves no room for compilers to utilize the flushed (idle) cycles.

In summary, the pipeline synthesis techniques currently available are not adequate enough to synthesize high performance pipelined ISPs with quality of manually designed ones. Some better ways of dealing with pipeline hazards are necessary to overcome the difficulty.

4.2. Integrating hazard resolution into high level synthesis

Our extended classification of inter-instruction dependencies and their associated hardware/software resolution strategies provide a feasible way of fixing up the inadequacy mentioned in the previous subsection. Our method enables the pipeline synthesis system to generate high performance pipelined ISPs that might not be free of pipeline hazards, and then provide a hardware/software approach to resolve the induced pipeline hazards.

Piper is a high level synthesis system for pipelined ISPs [5] that incorporates our hazard resolution method. Figure8 illustrates the structure of Piper and how the pipeline hazards are resolved in Piper. Piper and its pre-processor, Fiper, translate the abstract specification of an instruction set architecture into pipelined RTL level designs consisting of data paths and control paths. Piper also generates an interface to the compiler back-end (reorderer), and measures its time and space complexities. A set of application benchmarks is given to Piper to evaluate the quality of designs.

The main design phases of Piper consist of the following: (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 pipelined 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 based on our extended classification, and application of resolution strategies based on Table2. In this phase Piper generates a reorder table consisting of reordering constraints that instruct the compiler back-end (reorderer) to properly organize the codes for the pipelined machine being synthesized. Readers interested in the algorithms used for pipeline hazard resolution may refer to [6]. 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.

4.3. Exploration of the hardware/software tradeoff

As shown in Table 2, pipeline hazards caused by some inter-instruction dependencies may have both hardware and software resolution strategies available. Deciding which strategy to pick requires a tradeoff analysis. The hardware resolution has the advantage of resolving both forward and backward dependencies at the same time, at the cost of some additional registers and connections in the data path. On the other hand, the software resolution has the advantage of not complicating the hardware. However,

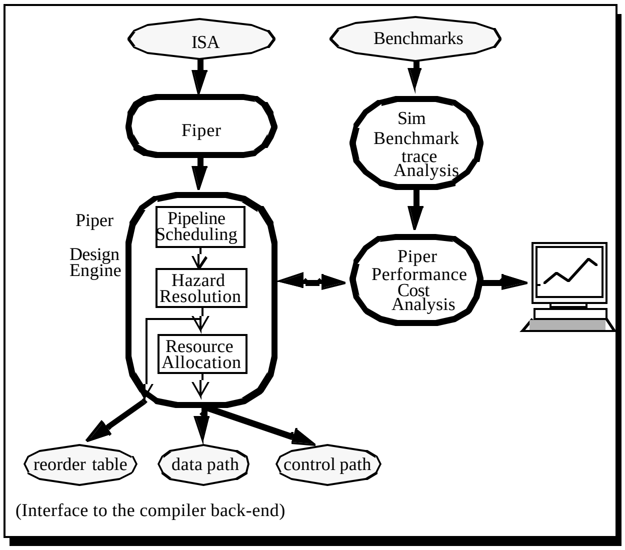

Figure 8. The structure of Piper system

it places a burden on the compiler back-end; the more reorder constraints, the longer the time to compile code.

The tradeoff analysis performed by Piper is as follows. On the hardware side, the cost of hardware is represented by the estimated size of the processor including data path, forwarding and duplicate registers (if adopted), and control path. A set of application benchmarks is used to measure the performance of the hardware. The performance is estimated with an analytical approach described in [5].

On the software side, the time and space complexities of the compiler back-end are expressed as functions with parameters related to the characteristics of the reorder constraints such as the size of the reorder table, the maximal/minimal/average reorder distances in the table, etc. For example, a straightforward implementation of the reorderer (compiler back-end) described in [3] has time and space complexities proportional to the size of the reorder table (the number of reorder constraints). Therefore, when comparing the software cost of several possible designs, we evaluate, for each design, the time and space complexity functions by substituting the characteristics of its reorder constraints into the functions. In practice, the space complexity is of less concern since the space (static code size of the reorderer) varies just slightly for hardware designs with different sizes of reorder tables. Only the space taken up by the table varies, which is usually small when compared to the space of the algorithm.

The performance and cost of hardware and software are then taken together to evaluate the global "goodness" of the designs. The objective function used by Piper for "goodness" is,

$$Goodness = P / C_{total} \qquad \text{(EQ 1)}$$

$$C_{total} = \alpha \cdot H + (1-\alpha) \cdot k \cdot S. \qquad \text{(EQ 2)}$$

H is the size of the hardware, S is the time complexity of the reorderer (representing the compilation difficulty), and k is a constant that adjusts the scales of H and S. The parameter α ($0 \le \alpha \le 1$) is used by the designer to assign the relative importance of hardware with respect to software (reorderer). C_{total} is a conceptual cost of the entire system including hardware and software. P is the estimated performance in term of speedup with respect to the non-pipelined design. The *Goodness* is then defined as the "performance/cost" ratio of the entire system.

With the concurrent hardware/software approach and the tradeoff model, Piper providers designers the capability of customizing designs for different application domains by selecting appropriate benchmarks and the weight constant α.

5. Experiments

In this section we demonstrate the usefulness of our dependency classification by showing how it improves the quality of automatically synthesized pipelined ISPs and helps the design automation system exploring the hardware/software design space. Two ISPs were synthesized by Piper: SM2a and TDY-43. It is interesting to note that these two instruction set architectures were not originally designed for pipelined implementation. They contain some architectural features that hinder good pipeline implementations, and thus provide a challenge to pipeline design automation systems.

We used the tradeoff analysis described in Section 4.3 to compare designs. Also as discussed in Section 4.3, the space complexity was ignored in the comparison since it is not significant. Note that we assumed the same clock cycle time when comparing designs. In order to simplify the presentation, we only listed the down-reordering constraints when comparing the software complexity.

5.1. The SM2a processor

The SM2a[4] processor is a 39-instruction micro-processor with both general and special purpose registers. In this experiment, the instruction pair frequency from a synthetic application benchmark was given to Piper to evaluate the average (run time) performance. The minimal achievable latency (MAL) of this processor is five; i.e., any effort to synthesize this processor with an instruction initiation latency less than five will introduce pipeline hazards. The speedup obtained by MAL-based pipeline synthesis techniques is 1.2 (with latency of five). ASPD system achieved better speedup of 1.5 (with latency of one), which is the

4. The SM2a processor was used by the Advanced Computer Architecture Laboratory at University of Southern California for the studies of Prolog compilation and design automation.

best performance of currently available synthesis techniques. Our technique produced a design with far better performance: a speedup of 2.37 (with latency of one). In the following paragraphs we present the pipeline hazard resolution in detail.

The results of the pipeline hazard resolutions for heavily pipelined SM2a (latency= 1, 2, 3, 4) are presented in Table3. The header row is the instruction initiation latencies chosen by the designer. The second row lists the number of dependencies to be dealt with for each instruction initiation latency considered. This number implies the ‘difficulty’ of the problem. The third row lists the number of hardware resolution candidates for duplicate registers and forwarding registers, respectively. The fourth row lists the hardware resolutions selected by the designer. Note that several combinations of hardware resolutions are available for the designs with instruction initiation latency of 1. Same as latencies of 2 and 3. For example, in the third column under latency=1, the designer selected one duplicate register and one forwarding register (1D,1F); whereas in the fourth column under latency=1 the designer chose one duplicate and three forwarding registers (1D,3F). The entry with ‘no’ means that no hardware is selected. The fifth through eighth rows summarize the software resolutions with respect to the design decision made in the fourth row. In the last row a design identifier (#1~#9) is assigned for each design synthesized by Piper. As a comparison, the design #10 is the design synthesized by ASPD, the current best technique.

Instruction initiation latency	4	3		2		1				1
# of inter-instruction dependencies	54	90		342		484				0
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	0
max. reorder distance	1	1	1	2	2	4	4	4	4	0
min. reorder distance	1	1	1	1	1	1	1	1	1	0
average reorder distance	1.00	1.00	1.00	1.10	1.10	3.20	2.20	2.22	2.20	0
Design ID#	1	2	3	4	5	6	7	8	9	10*

Table 3: Results of pipeline hazard resolution for the sm2a processor

*. Synthesized by ASPD [1]

Figure9 shows the performance/cost metrics for candidate designs #1 to #10. Figure9 (a) is the average speedup of the given benchmark (w. r. t. non-pipelined design). Designs #7, #8 and #9 have the largest speedup. Figure9 (b) shows the hardware sizes of candidate designs. The hardware size, including both data and control paths, was estimated at the RTL level. We were not able to obtain the actual size of design #10 directly. However, it should have roughly about the same data path size as design #6’s since

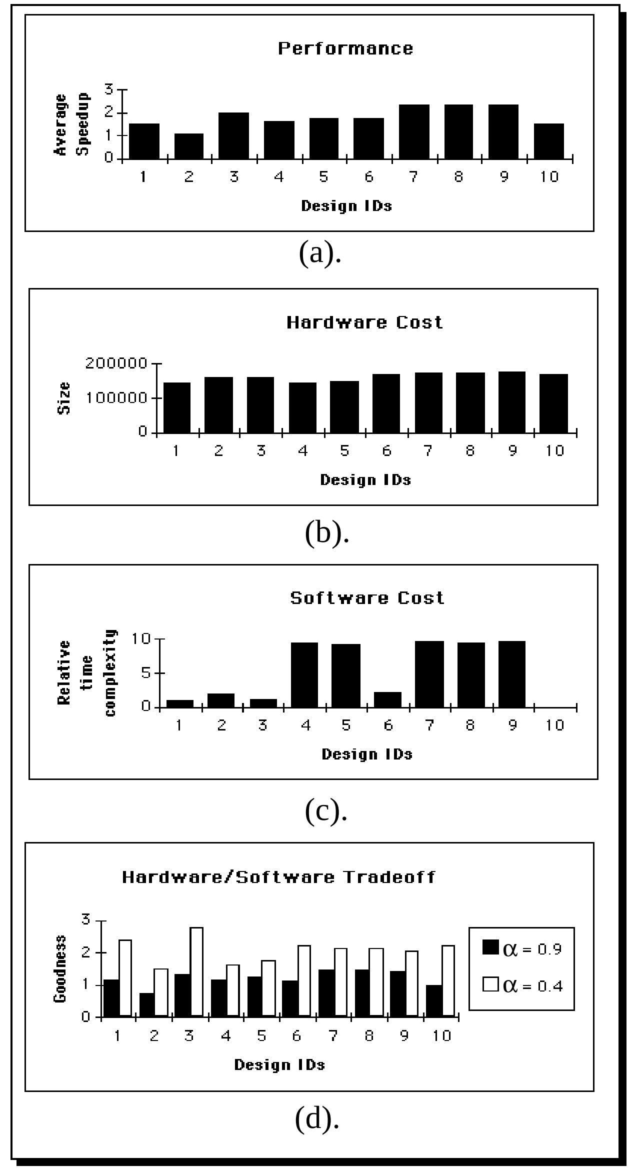

Figure 9. Performance/cost tradeoff analysis of SM2a

they both have no hardware resolution. The control path of design #10 may be more complicated than that of design #6, because the former has to flush the pipeline. As for the cost of software, the time complexities of design #1, #2, #3 and #6 are much smaller than others’, due to smaller number of reorder constraints, as shown in Figure 9 (c). Note that design #10 does not require reordering, thus it has zero time complexity in the figure.

Based on the performance and cost of hardware and software in Figure9 (a) through (c), the global hardware/software tradeoff analysis was conducted for two cases: α=0.9 and α=0.4 in EQ2, as shown in Figure9 (d).The case α=0.9 assigns more emphasis on the hardware cost and tolerates more software cost. The design with the best "performance/cost" ratio in this case is design #7 (with instruction initiation latency=1, one duplicate register, and 312 reorder constraints). On the other hand, the case α=0.4 assigns less importance to the hardware cost, and is more sensitive to the software cost. In this case, the best design is design #3 (with instruction initiation latency=3, one duplicate register, and 15 reorder constraints).

In summary, we have shown that our method is able to improve the throughput of the pipelined SM2a. The performance is much better than that of pipeline synthesis techniques currently available. Besides, our method provides better flexibility in producing designs for different application domains. The designers can customize the designs either for embedded systems where most of the applications are compiled beforehand, thus making software cost (compilation time) more affordable as shown in our α=0.9 case, or for application development systems where lots of applications are constantly compiled during the life time of the machines, thus making software cost less affordable as shown in our α=0.4 case.

5.2. The TDY-43 processor

The TDY-43 processor was designed about twenty years ago, and was used for aviation control in helicopters [16]. 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 re-implementation is desirable. The ADAS design automation system [13] was used to generate a gate-array implementation from the instruction set specification. Piper is the high level synthesis tool of ADAS, and was used to explore possible pipeline implementations of TDY-43.

Instruction initiation latency	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 4: Results of pipeline hazard resolution for the TDY-43 processor

Table4 shows some results of TDY-43. The design with latency of one was not completed since Piper 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. According to the experiment, TDY-43 can be pipelined at the latency of two, with 2814 reorder constraints. The large number of reorder constraints is primarily due to complex instructions such as normalization of fractional numbers, multiplication/division, and shifting by an arbitrary amount (0~64 bits). Even at the latency of eight, there are still 2180 reordering constraints required. At this latency, 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. Judging from this experiment as well as the fact that the company that designed TDY-43 would have difficulty in re-compiling their applications, we concluded that pipelining with software support (reordering) was not a feasible design style for TDY-43, instead, a sequential implementation was recommended.

6. Concluding remarks

We have shown the inadequacy of the conventional classification of inter-instruction dependencies, i.e., data, anti and output dependencies, and presented an extension to fix the problem. The extension was derived by further differentiating each class of the conventional inter-instruction dependency into three classes: forward, backward and stationary dependencies, according to register access patterns in the pipeline. The extension provides a better understanding of pipeline hazards in general pipeline structures, and allows hardware and software resolution strategies being associated precisely to pipeline hazards according to the classes of inter-instruction dependencies the hazards involve. The resolution strategies considered in this paper are, in hardware, the use of forwarding registers and duplicate registers, and in software, the use of instruction up/down reordering in the compiler back-end.

An interesting application of this extension and its resolution strategies was found in high level synthesis for pipelined ISPs. The state of the art high level synthesis techniques for pipelined ISPs haven't been powerful enough to synthesize high throughput pipelined ISPs, primarily due to insufficient understanding of pipeline hazards and their resolutions. This need has been filled by our extension and its resolution strategies. We have shown that a high level synthesis system Piper, which incorporates our methods, is able to produce designs much better than the results of other techniques. Another contribution is that our method adopts a concurrent hardware/software approach that helps the design automation systems explore both the hardware and software design spaces, as opposed to leaving the software design space intact.

Our current work was developed based on a machine model with a single pipeline, linearly cascaded stages and constant stage latency. While this assumption limits the applicability of our work, it does not prevent application to fairly complex machines such as superscalar processors. On the other hand, these techniques may be insufficient for machines where ‘autonomous’ hardware components are employed to resolve pipeline hazards dynamically.

This methodology needs the following extensions: First, more elaborated dependency analysis considering the dynamic behaviors of instructions in the data path needs to be developed; Second, the behavior of those ‘autonomous’ hardware components and their supporting circuitry, such as the reservation station, has to be formalized and integrated into the resolution strategies; Third, methods have to be developed to efficiently explore a much richer design space.

Acknowledgment

We appreciate the comments by Dr. Paul Suhler and the anonymous reviewers who pointed out some error in our draft and suggested several improvements to the final version of this paper.

7. Reference

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

[2] R. J. Cloutier and D. E. Thomas, “Synthesis of Pipelined Instruction Set Processors,” *Proc. of 30th DAC*, June 1993

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

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

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

[6] Ing-Jer Huang and Alvin Despain, “Hardware/Software Resolution of Pipeline Hazards in Pipeline Synthesis of Instruction Set Processors,” *Proc. ICCAD*, 1993

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

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

[9] Robert M. Keller, “Look Ahead Processors,” *Computing Surveys*, vol. 7, no. 4, December, 1975

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

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

[12] J. H. Patel and E. S. Davidson, “Improving the Throughput of a Pipeline by Insertion of Delays,” *IEEE/ACM 3rd Ann. Symp. Computer Arch.*, IEEE No. 76CH0143-5C, 1976

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

[14] Apoorv Srivastava and Alvin Despain, “Prophetic Branches: A Branch Architecture for Code Compactness and Efficient Execution,” *Proc. MICRO-26*, December, 1993

[15] R. M. Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” *IBM J. Res. Dev.*, January 1967, pp. 25-33.

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