Hardware/Software Resolution of Pipeline Hazards in Pipeline Synthesis of Instruction Set Processors

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

Abstract—One major problem in pipeline synthesis is the detection and resolution of pipeline hazards. In this paper we present a new solution to the problem in the domain of pipelined application-specific instruction set processors, based on hardware/software concurrent engineering approach. An extended taxonomy of inter-instruction dependencies is proposed for the analysis of pipeline hazards. Hardware/software resolution candidates are then associated with these dependencies. Algorithms using the taxonomy and the resolutions are developed to detect and resolve pipeline hazards, and to explore the hardware and software design space. Application benchmarks are used to evaluate the designs and guide the design decision. The power of these tools are demonstrated through the pipeline synthesis of two processors including industrial one. Compared with other approaches, our method achieves higher throughput, and provides a way to explore the hardware/software tradeoff. Our method can be combined with current approaches to achieve even higher performance since they are orthogonal.

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).

Pipeline hazards are the major hurdle in pipeline synthesis. They degrade the performance by preventing the subsequent task from being executed at its designated cycle. Therefore, resolving pipeline hazards becomes a major challenge to the power of synthesis techniques. To resolve pipeline hazards for pipelined ISPs is a more difficult problem since ISPs involve both hardware and software components. All the current approaches focus on hardware and do not consider the implication and design alternatives in software.

Most of the existing techniques, such as CATHEDRA II [4], HAL [12], PLS [9] and the work of Lee, et al. [11], are developed for DSP applications. They avoid pipeline hazards by limiting the degree of pipelining: a task should be started in the pipelined design with an initiation latency no less than the minimal achievable latency (MAL) [9]. On the other hand, a method called “Snapshot” is developed by Cloutier for pipelined ISP synthesis [2]. His approach is subject to the same limitation of MAL as in the DSP-oriented techniques. We will refer to the synthesis techniques reviewed in this paragraph as MAL-based techniques.

The ASPD system breaks the MAL barrier with an enhanced percolation scheduling algorithm [1]. The pipelined ISPs generated by ASPD execute at one instruction per cycle. The pipeline hazards are resolved 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. This approach leaves no room for compilers to improve the performance by reordering the instruction streams.

We presented a synthesis system Piper for pipelined ISPs in [6], which further improves the performance by simultaneously generating pipelined micro-architectures and their compiler backends. Higher throughput can be obtained by synthesizing pipelined designs with initiation latencies less than MALs. The induced pipeline hazards are resolved by generating reorder tables to guide the compiler backends (reorderers) to properly reorganize the instruction streams. This approach has the advantage over ASPD’s in that the compiler backends can make use of the cycles which would otherwise be stalled in ASPD’s designs.

In this paper we present an extension to our previous work, which provides the capability of trading off hardware and software while resolving register-related pipeline hazards (RPHs). There are two key issues in this extension. First, we need a new taxonomy of inter-instruction dependencies and their hardware/software resolution candidates, which are summarized in Section2. (For detailed discussion please refer to [8]). Second, related algorithms are necessary to apply the taxonomy and the associated resolutions to detect and resolve pipeline hazards, and to explore the design space. These algorithms are described in Section3. We demonstrate our synthesis power in two ISPs including industrial one in Section4. Summaries of the contributions, limitations, and future directions are given in Section5.

†. This work was supported by the DARPA under contract No. J-FBI-91-194.
2. An extended taxonomy of inter-instruction dependencies and their resolutions

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/output>.

An inter-instruction dependency with respect to a register X in a linear pipeline structure can be described in terms of the triple \((P_x, R_p, R_s)\). \(R_p\) and \(R_s\) are the register access patterns (read/write) of the preceding instruction instP and succeeding instruction instS, respectively. The conventional taxonomy of dependencies is encapsulated by this pair of parameters: data (\(R_p=\text{write}, \ R_s=\text{read};\) or, read after write), anti (\(R_p=\text{read}, \ R_s=\text{write};\) or, write after read), and output (\(R_p=\text{write}, \ R_s=\text{write};\) or, write after write) dependencies. The third parameter \(P\) describes the relative position of register accesses by a dependent instruction pair in the pipeline. Assume that instP and instS access the register X at the \(C_p\)’th and \(C_s\)’th cycle after they enter the processor, respectively. The possible values of \(P\) are: forward \((C_p<C_s)\), backward \((C_p>C_s)\), and stationary \((C_p=C_s)\). Table 1 summarizes the resulting nine types of dependencies from all possible cases of the triple.

<table>
<thead>
<tr>
<th>The preceding instruction instP accesses register (s) at cycle (C_p). The succeeding instS instruction accesses register (s) at cycle (C_s).</th>
<th>Types of register accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read after Write ((R_p=\text{write}, \ R_s=\text{read}))</td>
</tr>
<tr>
<td>(C_p &lt; C_s) (P=forward)</td>
<td>forward data dependency</td>
</tr>
<tr>
<td>(C_p &gt; C_s) (P=backward)</td>
<td>backward data dependency</td>
</tr>
<tr>
<td>(C_p = C_s) (P=stationary)</td>
<td>stationary data dependency</td>
</tr>
</tbody>
</table>

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

Note that only the backward dependencies cause RPHs. The forward dependencies do not cause any RPH. However, since forward and backward dependencies always happen in pairs in the data path, some resolution strategies can be optionally applied to the forward dependencies to eliminate their interference with the backward ones. The stationary dependencies do not cause any RPH, nor do they interfere with other types of dependencies. Therefore, they are the most desired ones in scheduling micro-operations for pipelined machines.

The hardware resolution currently adopted in Piper is the use of additional registers. Two types of additional registers can be employed: forwarding and duplicate registers. Forwarding registers carry the data along the pipeline synchronously with the instruction stream. This is analogous to adding extra latches (delays) in pipeline synthesis for DSP applications. Duplicate registers release the burden of temporary registers by creating additional data paths to eliminate false dependencies.

The major technique used in software (compiler back-end) to resolve pipeline hazards is instruction reordering, which 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. Assume that in a sequential instruction stream, instruction instB logically follows and depends on instA. The up reordering refers to moving the instruction instB up and ahead of instA, whereas the down reordering refers to moving instB 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. We define \(W_{up}\) and \(W_{down}\) as the reorder distances for up and down reordering, respectively.

Table 2 associates the applicable resolution strategies and the appropriate settings with inter-instruction dependencies. In this table it is assumed that instA and instB access register X at the \(C_{a}\)'th and \(C_{b}\)'th cycle of their sequential execution paths, respectively. The initiation latency is \(S\). In the first column is the nice types of dependencies and their exemplar pairs are given in parenthesis. For example, the instA-instB pair means instA followed by instB. In the second and third columns are the hardware and software resolutions, respectively.

Note that: 1) hardware resolutions has side effects. For example, the hardware resolution (forward registers) for a forward data dependency of a register X by an instruction pair instA-instB automatically resolves the backward anti dependency of the same register by the instruction pair instB-instA; 2) Some dependencies have both hardware and software resolutions available. These observations show that there are several combinations of hardware and software resolutions such that they provide chances to explore hardware/software tradeoff for different performance/cost criteria. In the following section, we present algorithms that apply Table 2 to resolve pipeline hazards and explore the hardware/software design space.

<table>
<thead>
<tr>
<th>Inter-instruction dependency</th>
<th>Hardware resolution</th>
<th>Software resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward data (instA-instB)</td>
<td>• Forward registers total number of forward registers (\lceil(C_p-C_s+1-M/A)\rceil\times S+1)</td>
<td>• Optional up-reordering (W_{up}=\lfloor(C_p-C_s+1-M/A)\rceil\times S+1)</td>
</tr>
<tr>
<td>Forward anti (instA-instB)</td>
<td>• (nA)</td>
<td>• Optional up-reordering (W_{up}=\lceil(C_p-C_s+1-M/A)\rceil\times S+1)</td>
</tr>
</tbody>
</table>

Table 2. Hardware/Software resolution strategies for inter-instruction dependencies
resolutions. This is accomplished in three steps. These
pipeline schedule, and outputs a set of hardware/software
producing a pipelined RTL level design.

In the second phase, application benchmarks are used to evaluate the design choices. At the last phase, these hazards are resolved in highly pipelined cases. These hazards are resolved in the second phase. In this phase, application benchmarks are used to evaluate the design choices. At the last phase, pipeline hazards may be introduced by the pipeline scheduler in highly pipelined cases. These hazards are resolved in the second phase.

The procedure of pipeline hazard resolution

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

<table>
<thead>
<tr>
<th>Inter-instruction dependency</th>
<th>Hardware resolution</th>
<th>Software resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward output (instA-instB)</td>
<td>Duplicate register</td>
<td>Optional up-reordering: $W_{up} = (C_0 - C_1)/S \times 1$</td>
</tr>
<tr>
<td>Backward data (instB-instA)</td>
<td>n/a</td>
<td>Down-reordering $W_{down} = l(C_1 - C_3)/S \times 1$</td>
</tr>
<tr>
<td>Backward anti (instB-instA)</td>
<td>n/a</td>
<td>Down-reordering $W_{down} = l(C_1 - C_3)/S \times 1$</td>
</tr>
<tr>
<td>Backward output (instB-instA)</td>
<td>Duplicate register</td>
<td>Down-reordering $W_{down} = l(C_1 - C_3)/S \times 1$</td>
</tr>
<tr>
<td>Stationary data (instB-instA)</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Stationary anti (instB-instA)</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Stationary output (instB-instA)</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

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

3. The procedure of pipeline hazard resolution

The pipeline synthesis process of Piper consists of three phases: 1) pipeline scheduling, 2) pipeline hazard resolution, and 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. In this phase, application benchmarks are used to evaluate the design choices. At the last phase, the hardware resources for the data path are allocated, producing a pipelined RTL level design.

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.

3.1. 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: 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 1 (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 1 (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 1 (a)). In the second step, for each register and each set of exclusive blocks, cross-trace dependencies are generated. They are formed by pairing the earliest/latest register read/write of one block with the earliest/latest register read/write of the other block, for every pair of exclusive blocks within the set.

1. There may be more than one set of exclusive blocks. A set of exclusive blocks consists of blocks that are exclusive to each other.

Figure 1. (a). in-trace and cross-trace dependencies; (b). algorithm for in/cross-trace dependency analysis

International Conference on Computer Aided Design, 1993 3
3.2. 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 Table2. 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 $i$; $InstPair_i$ 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 were 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.

3.3. Generation of final resolutions

After the designer has selected the desired hardware resolutions, the software resolutions can be obtained. Figure2 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 Table2, 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 $ro(\text{add,load},3)$ (three delay slots between add followed by load) and $ro(\text{add,load},4)$ exist, then the former can be deleted from the resolutions since it can be covered by the later constraint. The second step in Figure2 is shown for the down-reordering case. Up-reordering constraints can be obtained by interchanging ‘>’ and ‘<’ operation in the second step.

1: GIVEN: selected hardware resolutions, software resolution candidates (reordering constraints), and inter-instruction dependencies
2: OUTPUT: software resolutions
3: 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
4: For the remaining dependencies, perform a merge process on their software resolutions (reordering constraints). A reordering constraint is a record with the format:
   $ro(\text{instruction1, instruction2, displacement})$
   repeat the following until no further change can be made:
   (1). if $ro(\text{instruction1, instruction2, d1})$ and $ro(\text{instruction2, instruction3, d2})$ exist and $d_1 > d_2$, delete
       $ro(\text{instruction1, instruction2, d2})$;
   (2). if $ro(\text{allInsts, instruction2, d})$ exists, delete all $ro(\text{allInsts, instruction3, dx})$ with $dx < d$;
   (3). if $ro(\text{allInsts, instruction2, d})$ exists, delete all $ro(\text{allInsts, instruction3, dx})$ with $dx < d$

Figure 2. algorithm for the generation of software resolutions

4. Examples

In this section, we applied our resolution techniques to explore the design space of two ISPs: SM2a and TDY-43. We were interested in the sub-space of highly pipelined designs where pipeline hazards exist, since the instruction sets were not designed specifically for pipelined implementation. Several features of them provide a challenge to the pipeline synthesis power of design automation systems. 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 [6] was used to evaluate the effective speed-ups (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 grows linearly with the size of the reorder table (the number of reordering constraints).

4.1. 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 latency 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 results of the pipeline hazard resolutions for heavily pipelined SM2a (latency= 1, 2, 3, 4) are presented in Table3. Designs made by MAL-based techniques and ASPD are listed in the table for comparison. Since we were not able to get access to these tools, the data produced for
these tools were manually derived based on our best knowledge of their algorithms. The third row is the latency. The fourth 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 fifth row lists the number of hardware resolution candidates for duplicate registers, and forward registers, respectively. The sixth row lists the hardware resolutions selected by the designer. For example, in design #8, the designer selected one duplicate register and one forward register resolution (1D,1F) out of the possible hardware candidates (1D,4F). The entry with ‘no’ means that no hardware is selected. The seventh through tenth rows summarize the software resolutions with respect to the design decision made in the sixth row. The eleventh 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.

Several comments can be drawn from the results. First, our techniques outperform other techniques. The maximal speedup (for the given application benchmark) of pipelined SM2a synthesized with MAL-oriented pipeline techniques is 1.2 (design #11, with instruction initiation latency of five). The design #10 synthesized by ASPD can execute at the latency of one; however, for a large portion of the execution time the design flushes the pipeline for both necessary and unnecessary cases. Therefore, it achieved an overall speedup of 1.5. On the other hand, we were able to increase the speedup to 2.37 for the given benchmark, and 6 for benchmarks that have no inter-instruction dependency in the design #8 (instruction initiation latency=1, with one duplicate and one forward register as the hardware resolution). The improvement is achieved at the cost of instruction reordering. The compiler backend has to examine 113 pairs of down-reorder constraints.

Second, we were able to generate several design alternatives consisting of different hardware/software combinations. The designs #1–#9 provide choices of instruction initiation latencies from 1 to 4, speedups from 1.1 to 2.37, and relative compiler backend (reorderer) time complexities from 1 to 9.58. The designer can select an appropriate design based on the application environment. For example, for embedded applications such as the cruise control of vehicles, the software is compiled only once and then loaded into the system. Therefore, the designs with higher reorderer time complexity are justifiable. On the other hand, if the designs are to be used as programming tools where compilation happen very often, then the designs with lower reorderer time complexity are more feasible.

Third, The number of dependencies to be resolved grows fast when the instruction initiation latency 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.

Fourth, 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.

Fifth, 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 latency=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.

4.2. TDY-43 processor

The TDY-43 processor was designed about twenty years ago, and was used for aviation control in helicopters [13]. It has a fairly complex instruction set consisting of 256 instructions supporting fix-point, fractional, and two’ comlement 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 experiment showed that we were able to implement this complex instruction set architecture with a highly pipelined style, at the cost of large reorderer time complexity. Due to the space limitation, interested readers may refer to [7] for the experiment data.

<table>
<thead>
<tr>
<th>Synthesis System</th>
<th>Piper</th>
<th>ASPD</th>
<th>MAL-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design ID#</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Instruction initiation latency</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td># of inter-instruction dependencies</td>
<td>54</td>
<td>90</td>
<td>342</td>
</tr>
<tr>
<td>possible hardware resolutions (D=dupicate register, F=forward register)</td>
<td>-</td>
<td>1D</td>
<td>1D,2F</td>
</tr>
<tr>
<td>selected hardware resolutions</td>
<td>no</td>
<td>no</td>
<td>1D,4F</td>
</tr>
<tr>
<td># of down-reordering</td>
<td>12</td>
<td>23</td>
<td>15</td>
</tr>
<tr>
<td>max. reorder distance</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>min. reorder distance</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>average reorder distance</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>estimated performance for the benchmark (speedup w.r.t. non-pipelined case)</td>
<td>1.48</td>
<td>1.10</td>
<td>1.97</td>
</tr>
<tr>
<td>relative time complexity of the reorderer</td>
<td>1</td>
<td>1.92</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Table 3. Results of pipeline hazard resolution for the SM2a processor
5. 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 (smaller initiation latencies) 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. Two 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, 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 design complexity. With its pipeline synthesis technique and hardware/software concurrent engineering approach, Piper is an appropriate design tool for these two design domains.

Note that our approach is orthogonal to MAL-based approaches. The MAL-based techniques can be integrated into Piper to reduce the MALs in our pipeline scheduling algorithm, which may decrease the number of pipeline hazards. High throughput designs with less requirements on the hardware and software resolutions can be produced.

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. (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. We will eliminate the limitations in our future work.

6. Reference