High Level Synthesis of Pipelined Instruction Set Processors and Back-End Compilers

Ing-Jer Huang and Alvin M. Despain
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2561
ijhuang@usc.edu, despain@usc.edu

Abstract
Designing instruction set processors and constructing their compilers are mutually dependent tasks. Piper is a high level synthesis tool of ADAS which controls the hardware-software interactions at the micro-architecture level. The key function of Piper is to perform pipeline scheduling with any fixed instruction latency, and generate a reorder table for a back-end compiler to resolve potential pipeline hazards existing in the design. Models for performance and cost of both hardware and software are developed in order to characterize the design space. A synthesis example of a simple instruction set processor illustrates Piper’s synthesis capabilities and how the performance and cost of hardware and software are estimated.

1. Introduction

Pipelining is a technique to improve the throughput of digital systems. Throughput is improved by overlapping the execution of successive iterations. The time delay between the issues of consecutive iterations is called latency. However, two major problems limit the power of pipelining. First, loop carried dependencies (LCD’s) [7], the data dependencies between different iterations, prevent the next iteration from being fired right after the issue of the current iteration (latency of 1). In instruction set processor specifications, LCD appears as the inter-instruction dependency. A pipeline hazard is introduced when the LCD prevents a successive instruction from being executed at its designated cycle. A design automation (DA) system targeted at pipelined architectures has to construct a proper data path structure and a control scheme so as to ensure the satisfaction of LCD’s. Secondly, conditional branching, the conditional execution of operations within the loop body, further complicates the detection of existences and lengths of LCD’s. General approaches to these problems are either to find out the minimal achievable latency (MAL) [7] and issue iterations with latency no less than MAL, or to issue an iteration at a latency of one, but stall the hardware as soon as the pipeline hazards are detected. Some DA systems choose to restrict their applications to algorithms without conditional branches, whereas some other DA systems perform the worst case analysis by expanding traces with all possible branches to find the LCD’s with longest temporal distance which possibly happen. Such solutions can be found in DA systems for DSP applications [2][5][6][7]. The basic limitations of these approaches are that the idle cycles (bubbles) are wasted, and that the synthesized hardware is tuned to worst cases which may rarely happen. We will discuss these systems in Section 2.

One area which has not received much attention from researchers in design automation for instruction set processors is to construct the back-end compilers for the machines generated. The decisions made in the micro-architecture implementations may affect the implementations of some phases in back-end compilers. For example, the typical ‘delayed load’ and ‘delayed branch’ are caused by the pipeline hazards in the pipeline structure, and require a phase reorderer in the back-end compiler to reorder the instructions or insert nap’s to ensure the correctness of the instruction behavior and improve the performance.

In this paper we propose a synthesis methodology for high performance pipelined instruction set (IS) processors which out performs the current DA systems in that it is capable of making the most use of the idle cycles with the presence of LCD’s and conditional branching in the instruction set architecture (ISA) specification. Piper also distinguishes itself from the other DA systems in that it bridges the semantic gaps between hardware and software at the micro-architecture level.

The performance (execution time) and cost (memory usage) of the reorderer depend on the characteristics of the reorder table which is subject to the pipeline synthesis decision. Therefore, we are able to add an additional dimension to the design space such that designers can explore instruction-level parallelism between hardware (synthesized IS processors) and software (reorderer). We have developed a model for determining the performance and cost of both hardware and software to encapsulate the qualitative interaction between them. Instruction benchmarks are used to evaluate the run time performance of pipeline.

This synthesis methodology and performance/cost model have been implemented in a high level synthesis system Piper. It is the behavioral domain component of the Advanced Design Automation System (ADAS) which transforms an abstract description of instruction set architecture into a mask layout [1][13].

The rest of this paper is organized as follows. Section 2 reviews the related work. Our approach to pipeline synthesis with any fixed instruction-latency is described in Section 3. An algorithm to construct the reorder table for the back-end compiler (reorderer) is presented in Section 4. In Section 5 the model of the performance and cost for the pipelined instruction set processors and their reorderers is described. Our techniques are illustrated by a synthesis example of a simple instruction set processor sm1 and compared to other DA systems in Section 6. Conclusion and future work are presented in Section 7.

2. Related work

In the pursuit of high performance in digital circuit designs,
many high level synthesis systems focus on automating the design of pipeline structures. Several pipeline scheduling algorithms have been developed. These algorithms differ in their application scope and achieved performance. SEHWA uses two algorithms: feasible scheduling and maximal scheduling [2]. HAL applies a modified force-directed scheduling algorithm to achieve better quality of pipeline scheduling and allocation at the cost of higher complexity [5]. These two systems do not work with applications which exhibit loop carried dependences. PLS and CATHEDRAL II are capable of working on applications with LCD’s by means of an iterative folding algorithm [7] and loop folding algorithm [4], respectively. Both systems adopt an iterative approach to find the MAL. This MAL then serves as the upper bound of the performance of the synthesized designs. The major application domain of the above synthesis systems is DSP. ASPD further improves the pipeline scheduling performance by applying an enhanced percolation scheduling algorithm and using a wide-instruction-word (WIW) architecture as a template [6]. ASPD is capable of synthesizing WIW-based processors with instruction latency less than MAL at the cost of control for flushing pipeline when the pipeline hazards are detected. However, the practicality of ASPD is limited by the facts of microcode explosion and the complexity of pattern merging.

The System Architect’s Workbench consists of two synthesis methodologies: one is tuned specifically to microprocessor designs, and the other supports general design style [8]. In the latter approach, pipelined designs are derived by interactively performing transformations at the system level. However, it does not attempt to solve the pipeline hazard automatically.

The difference between Piper and these systems is that Piper focuses on instruction set processor designs, and further improves the performance of the pipeline structures by considering simultaneously the synthesis of hardwares and the construction of back-end compilers.

In compiler designs, there has been some work in machine-independent code generation and optimization to automate the generation of code-generators and -optimizers by separating the machine-dependent information from the machine-independent algorithms. The machine-dependent information is derived from a machine description. In most systems the machine-dependent information is tabularized. Cattell’s and Graham’s works focus on the generation of code-generators [10][11], while Giegerich’s work is on the derivation of machine-specific optimizers [12]. They derive their machine-dependent tables from instruction set semantics which describes the instruction formats, addressing modes, and operations. However, the micro-architectural features such as pipeline configuration are not considered in these systems. Piper is different from these works in that it focuses on the construction of back-end compiler phases which are related to the micro-architecture. Currently, the focus is restricted to generating the reorderers for instruction set processors with various pipeline configurations.

3. Pipeline synthesis with any fixed instruction latency

It is commonly found in the designs of modern pipelined microprocessors that a microprocessor operates with the instruction latency shorter than the MAL. For example, in the MIPS R2000/R3000 microprocessor which operates with instruction latency of one, the minimal achievable latency found in the data/control flow is two (read after load from memory, for example) [9]. The processor does not flush its pipeline, nor does it insert nop’s itself in order to resolve pipeline hazard. The optimizing compiler is thus responsible for inserting nop’s or reordering the instructions when such patterns of instructions are found. Compared with previous approaches in Section 2, this approach has higher performance and lower cost because of the shorter instruction latency and the absence of the flush/stall circuitry. However, a phase in the compiler has to be constructed to properly reorder instructions or insert nop’s, and, therefore, the time and space complexities are increased in the compilation process.

We adopt this approach in Piper to achieve high performance comparable to the manually-designed pipelined processors. This approach is generalized in Piper such that scheduling with any fixed instruction latency is supported, regardless of the MAL. This allows designers to explore design spaces ranging from a non-pipelined design to a maximal throughput pipeline (instruction latency of one). The scheduling process of Piper begins with a global data-flow analysis for intra-instruction and inter-instruction dependencies. It tags the conditional branches and associates the tags to the dependencies. The scheduler of Piper then schedules the micro-operations according to the desired instruction latency given by the designers. After the micro-operations are scheduled, the data path and control path can be constructed.

Note that this approach is independent of the scheduling algorithm used by the synthesis system. It lifts the MAL constraint such that scheduling algorithms are free to schedule with any fixed instruction-latency. This enhances the capability of the scheduling algorithms, and provides more flexibility for the designers.

In the current implementation of Piper, the pipeline scheduling is done in the stage partitioning phase by decomposing an abstract finite state graph generated by Piper, the preprocessor of Piper, into several smaller state graphs, according to the desired instruction latency specified by designers. Each smaller finite state graph represents a stage in the pipeline. Figure 1 illustrates the concept of stage partitioning. An ISA specification trans-
formed into an abstract finite state machine consisting of four states which is then decomposed into two cascading finite state machines. Each smaller finite state machine iterates between its two states, and is synchronized with each other by a system counter counting between $t1$ and $t2$. The result is a two-stage machine with instruction latency of two. Piper is able to deal with conditional branches inside the loop body. In the case where nondeterministic inner loops exist in the loop body, the current implementation of Piper does not attempt to pipeline the design.

Figure 4 shows the inter-instruction dependencies after the stage partitioning for an instruction set processor with three instructions foo, bar, and goo. In this example the processor is partitioned into three stages with the stage time of two cycles. This processor consists of fourteen states, $S1$ to $S14$ shown as bubbles in the figure. For each state there is a state label shown at the left side of the bubble, and a conditional tag shown at the right side. A conditional tag with non-empty string indicates that the state is conditional executed. For example, state $S1$ with tag of [ ] is non-conditionally executed, whereas $S3$ with tag of [foo] is conditionally executed when the opcode is foo. The tags are derived from the conditional branches such as opcode decoding and the values of status registers. The state transitions are represented by light arcs. The body of the state represents the active register transfers. For simplicity, we only show the register transfers of interest in this discussion. For example, in state $S1$, PC<- indicates a write to register PC, and in state $S7$, <-X indicates a read from register X. The heavy bi-directional arcs indicate the inter-instruction dependencies. The longest inter-instruction dependency spans over four cycles. However, the machine is implemented with latency of two. Therefore this is not a hazard-free pipeline implementation.

Note that better resource utilization may be obtained if rescheduling after the stage partitioning is performed. For example, assume that there is an addition operation in states $S1$ and $S13$ (in Figure 2), respectively. Then it would require that two adders be allocated since these two additions may be executed concurrently at cycle 2. When the mobility allows, the former addition can be rescheduled to state $S12$ or the latter to state $S14$, then only one adder will be sufficient since these two additions do not execute concurrently. However, while this rescheduling may improve the resource utilization, it may degrade the performance. For example, by delaying the addition in $S11$ to $S12$ (with tag goo) may introduce a pipeline hazard between an instruction

```
1: for each upward dependency pair $Si$-$Sj$ do
2:    for each $Displacement$ in Inst1 do
3:        if either $Si$ or $Sj$ is in an inner loop then
4:            terminate; % Not pipelinable
5:        if $Si$ and $Sj$ are not of the same stage then
6:            if cycle index of $Sj$ > cycle index of $Si$ then notInTime=1
7:                else notInTime = 0;
8:                Displacement = stage index of $Si$ - stage index of $Sj$ + notInTime;
9:                Nops = Displacement - 1;
10:       if tag of $Si$ contains opcode then Inst2<is this opcode
11:          else Inst2<allInstructions;
12:         if tag of $Sj$ contains opcode then Inst1= this opcode
13:             else Inst1= allInstructions;
14:         report table entry: reorder(Inst1 after Inst2 by Nops nop)
15:     } do
16:     } delete entries which are subsets of others
```

Figure 3 Reorder table generation which follows goo and uses the result of that addition. The rescheduling and its impact on the pipeline hazards will be integrated into the future version of Piper.

An algorithm described in Section 4 is then invoked to solve the pipeline hazards by extracting a reorder table for the reorderer.

4. Construction of the reorder table

To solve the pipeline hazards existing in the high performance implementations of pipelined micro-architectures, Piper assumes that there is a retargetable reorderer to reorder the instructions. The reorderer takes in the instruction format specification and a reorder table to properly reorder instructions in order to generate hazard-free and efficient target codes. An exemplary entry of the table is reorder(foo-after-bar-by-3-nop). This means that there should be at least three $nop's$ or other independent instructions (safe instruction distance=3) in between the instruction pattern of foo following bar.

An algorithm which generates the reorder table is described in Figure 3. The algorithm takes as inputs the pipeline schedule and the dependencies from the global data-flow analysis, and generates as output a reorder table. The details of the algorithm is described as follows.

The algorithm first collects all the upward dependencies (from deeper pipeline stage to shallower one). The downward dependencies do not cause pipeline hazards. For example, in Figure 2, the dependency due to a read of register X in state $S7$ and a write of register X in state $S14$ causes a hazard for bar-after-goo but not for goo-after-bar since the potential hazard for the latter one is automatically solved by the structure of the pipeline: bar reads X at stage 2, and at that time goo which follows bar has not arrived at the stage 3 where it writes to X.

The next step is to generate the reorder entry for each upward dependency. This task consists of three sub-tasks. First, we check that whether any one of the involving states $Si$ and $Sj$ is in an internal loop. If true, pipelining is not a suitable implementation for this processor, and thus we terminate this algorithm. Second, the safe instruction distance for the dependent instruction pair has to be determined. Third, the names of the instructions which involve in this dependency have to be identified. Line 7 to 10 calculate the safe instruction distance. Line 11 to 14 determine the names of the instructions from the tags of the involving states. In the case where the opcode can not be inferred from the tags, we unify the name of the instruction to allInstructions which means
this reorder case apply to all instructions. In the example of Figure 2, the S14-S7, and S3-S1 dependencies result in the reorder entries reorder(bar-after-goo-by-1-nop) and reorder(allInstructions-after-foo-by-1-nop), respectively. The last step deletes the reorder entries which are subsets of others.

In Figure 4 we show the reorder tables for instruction latencies of 1, 2, 3, and 4 for the example in Figure 2. As we expect, the larger the instruction latency, the less number of instruction pairs need reordering. The instruction latency of 1 results in three reorder pairs while the instruction latency of 3 resulting in only one reorder pair. Also the safe instruction distances decrease as the instruction latency getting larger. The longest safe instruction distance is four for instruction latency 1, while being one for instruction latency 3. Note that the reorder table for instruction latency 4 has zeros for all entries which means that this processor can be pipelined with instruction latency of 4 (2 stages) with no pipeline hazards for all instruction patterns. Therefore, this implementations makes the pipelined micro-architecture transparent to the compiler.

The generated reorder table affects the execution time and memory space of the reorderer. The more entries in the reorder table and the larger the safe instruction distance, the longer time the reorderer takes to execute and the more memory space it consumes. We will discuss these effect in the following section.

5. Performance/cost evaluation

We characterize the synthesized design by the performance and cost of the hardware and its supporting software (the reorderer). The important metrics are peak/run-time performance and cost of hardware, and time/space complexities of compilers. With these metrics we are able to provide the designers with a broader application scope of the design automation systems into embedded systems design.

5.1. The performance estimation of hardware

Two performance metrics are used to evaluate designs: maximal and run-time performances. The maximal performance can be obtained when the instruction latency and the clock rate are known. It does not vary with the applications. However, the run-time performance depends on the characteristics of the applications. The instruction trace expansion due to inter-instruction dependencies and pipeline hazards degrades the run-time performance of a deeply pipelined processor from its maximal performance. Consider a fragment of instructions compiled for non-pipelined machine in case (a) of Figure 5. Suppose that there is a synthesized pipelined machine which requires two delay slots between branch brn and its successive instruction of any type. In case (b) two nop’ s are inserted between brn and the successive sub r2. The instruction trace is expanded by two nop slots which degrade the pipeline performance by two pipeline cycles. In case (c) the size of the nop slots is compressed from two to one by reordering an independent instruction add r1 into the nop slots.

The instruction set simulator of ADAS provides a set of instruction benchmark characteristics. One of the characteristics used by Piper is the pattern frequency of instruction pairs which is the frequency of the consecutive instruction pairs in the instruction trace. Shown in Figure 3 is the pattern frequency of a small instruction benchmark for the instruction set in Figure 7. For example, the first entry represents that load immediately followed by add (the load-add pair) happens twelve times in the trace which has a frequency of 14.5%.

For each pipelined processor p, the run time performance metrics with respect to a benchmark can be approximated by the following equations:

\[ \text{Exp} = \sum_j \text{Dist}_{pj} \cdot \left(1 - C\right) \cdot R_j \cdot A + A \]

\[ S_p = \frac{A \cdot M}{L_p \cdot \text{Exp}} \]

\[ T_p = \frac{A}{L_p \cdot \text{Exp}} \]

where \( E_{\text{Exp}} \) is the trace expansion due to the insertion of nop’s, \( S_p \) is the speedup of pipelined processor \( p \) with respect to non-pipelined processor, and \( T_p \) is the throughput of pipelined processor \( p \).

\( \text{Dist}_{pj} \) is the number of nop’s (in the worst case) to be inserted between a consecutive instruction pattern \( j \) for pipelined processor \( p \), \( R_j \) is frequency of the consecutive instruction pattern \( j \), \( L_p \) is the instruction latency of pipelined processor \( p \), \( M \) is the instruction cycle time of non-pipelined processor, \( A \) is the total number of instructions

- original code
- reordered code

Figure 5 Instruction reordering

Figure 6 Pattern frequency of instruction pairs
of instructions in the simulation trace, and \( C_j \) is the compression factor for consecutive instruction pattern \( j \) which means that, empirically, the percentage of the number of nop slots which can be filled by reordering instructions into the slots. \( C_j \) can be set to zero for the worst case analysis.

The product \( AM \) is the total number of cycles when the benchmark is executed on the non-pipelined processor while the product \( L_p E_{up} \) is the total number of cycles executed on the pipelined processor \( p \).

5.2. The cost estimation of hardware

We represent the cost of the hardware by the sizes of the data path and control path. The size of the data path is obtained by estimating the number of transistors in the data path. The wiring cost is not considered in our current version of Piper. The control model adopted by Piper is data stationary model. Each stage has its own controller. We implement these controllers as PLAs. Therefore, we derive the control path cost by summing up the estimated PLA size of each stage.

5.3. The time/space complexities of instruction set compilers

When comparing the time and space complexities of instruction set compilers for different pipeline implementations of the same instruction set architecture, we focus on the reorder phase of the compiler since the other phases of the compiler remain constant. We apply the characteristics of the reorder table to the equations of time and space complexities to derived the relative performances. The possible characteristics of reorder table include the number of reorder entries (consecutive instruction patterns) \( E \), the maximal safe instruction distance \( D_{max} \), the minimal safe instruction distance \( D_{min} \), and the average safe instruction distance \( D_{ave} \).

The time/space complexities of the reorderer can be considered as the constraint imposed on the simulation system by the software (the reorderer). For example, a reorderer with time complexity of \( E^2 D_{max} \) implies that it is not feasible for a synthesis system to synthesize a design which produces a larger reorder table; however, a design with large safe instruction distance but a smaller reorder table may still be an acceptable design.

To demonstrate the idea, let’s consider a simple reorderer which reads in the reorder table of size \( E \), and then sweeps a window of size \( 2^*D_{max} \) through the instructions to look for the target patterns, and then searches within the window for independent instructions to be inserted into the nop slots. Assume that the table is kept in an array. Then for each iteration, the reorderer takes \( O(E^2D_{max} + D_{max}) \) time. At least, memory of size \( O(E^2D_{max}) \) is necessary. Its time and space complexities may be expressed as:

\[
\text{time complexity} = L \cdot D_{max} \cdot (E + D_{max})
\]

\[
\text{space complexity} = E + D_{max}
\]

where \( L \) is the benchmark length. By substituting the variables for values, we can obtain the relative complexities. Similarly, we can derive the time/space complexities in terms of reorder table parameters for any reordering algorithms. This set of equations is kept in a system configuration file in Piper and should be modified if the reordering technique used in the compiler is changed.

6. Synthesis example and comparison

In this section we present a synthesis example for an instruction set processor sm1. Its ISA specification in Prolog [14] is shown in Figure 7. It is an accumulator-based processor with eight instructions. Piper returns six pipelined designs with instruction latency ranging from one to six. The instruction latency of six is the non-pipelined design. In Figure 8 to Figure 10 we show the performance/cost evaluations for these designs. In these figures, we characterize the designs by their instruction latencies. From the dependency analysis, the MAL is five for sm1. Therefore, the designs with latency from one to four are not hazard-free, and require the reorderer to resolve the pipeline hazards.

In Figure 8 are the costs of hardware including data path and control path. Figure 9 shows the maximal and average (run-time) performance curves and the performance/cost curves. In this example, we use the speed-up to represent the performance. The average performance is obtained by applying the benchmark characteristics from a small straight-line instruction benchmark with conditional branches to the speed-up equation listed in Section 5. Note that the smaller the instruction latency, the greater the average performance degrades from the maximal performance due to the insertions of nop’s. The performance/cost curves indicate that pipelining, with higher utilization of resources, dramatically improves the performance at slightly extra cost. This is also observed in [3] and [4].

The relative time/space complexities of reordering phase of the compilers are illustrated in Figure 10. As we expect, the complexities increase dramatically as the instruction latency decreases. These curves show that the maximal throughput design (latency of one) has a relatively large compiler complexity, even though it offers the largest performance/cost ratio. Therefore it might not be a feasible design from a compiler’s point of view. Note that the relative complexities of time and space are zeros for designs of instruction latency 5 and 6. This means that no reordering is required for these two designs. This result is interesting in that if the reordering is not supported in a computing environment, then a speed-up of 1.2 (instruction latency 5) is still achievable at an additional 5.7% hardware cost of the non-pipelined design.

In Figure 10 we show a comparison for the synthesis capability of several DA systems with our sm1 example. PLS, HAL, and CATHEDRAL II perform pipeline synthesis under the constraint of MAL and thus are not able to produce the designs with latency less than five. SEHWA performs pipeline synthesis for data path only. Therefore, SEHWA is unavailable for designs with pipeline hazards since they require special cares in the control path design. ASPD is able to synthesize designs with pipeline hazards. The control path generated by ASPD stalls the pipeline as soon as a leading instruction which may cause hazards is decoded. ASPD
assumes the worst cases. Suppose that the only instruction-pair with pipeline hazard is loadR1-addR1. ASPD will always stall the pipeline as soon as loadR1 is decoded, no matter which instruction follows. This strategy leaves compilers no room to make use of the stalled cycles. Therefore, the performance of the synthesized designs by ASPD is inferior to Piper’s designs.

7. Conclusions and future work

We have presented a synthesis model which encapsulates the interactions between the hardware and software at the micro-architecture level. We have shown that even higher performance of pipelined instruction set processor designs can be achieved by our design automation system Piper with a scheduling technique which allows pipeline scheduling with any fixed instruction-latency, and with the help of the reordrer in the back-end compiler. We also have presented the performance and cost models for both hardware and software. This capability of modeling the aspects of the synthesis, performance, and cost for the software and hardware components of complete systems makes Piper a promising design automation system for embedded systems design.

In addition to the improvement of the automatically generated pipeline structures, Piper contributes to the knowledge of automating the generation of back-end compilers, especially the micro-architecture dependent phases, which has been lack of profound success during the past decade. We have presented an algorithm to extract the reorder table for the reorderer from a description of the pipelined micro-architecture.

The rescheduling guided by the characteristics of instruction benchmarks after stage partitioning in order to improve the resource utilization will be implemented into the future version of Piper. Piper will be able to deal with the non-deterministic inner loops by means of inserting internal opcodes. We will develop and implement further synthesis techniques to fine tune the micro-architecture to take advantages of some advanced compiling techniques such as delay slots with annulling or branch prediction.

Acknowledgments

The authors would like to thank Gino Cheng, Barton Sano, Apoorv Srivastava, Chi-Ying Tsui and anonymous reviewers for their suggestions and comments. We also appreciate the research supports from Yong-Seon Koh, Shih-Ming Liu, Ricky Pan, Iksoo Pyo, Ching-Long Su, and Shiqun Wu.

Reference


