HIGH-LEVEL OPTIMIZATION OF PIPELINE DESIGN

Jennifer P. L. Campbell, Dept. of Comp. Sci., University of Toronto, Toronto, ON, Canada
Nancy A. Day, School of Comp. Sci., University of Waterloo, Waterloo, ON, Canada

Abstract

We describe an automatic method for synthesizing pipelined processors that optimizes throughput and automatically resolves control and data hazards. We present rules that describe how to resolve hazards based on the data dependencies between functional units. We demonstrate our method by showing optimal pipeline configurations of the DLX.

Introduction

Pipelining is a ubiquitous optimization used in circuit design. By partitioning the computation of a circuit into multiple stages that execute concurrently, the clock period can be decreased and, often, the throughput increased. However, when the staging of the computation over time requires the addition of control circuitry to handle dependencies among the stages, pipelining can be complex to design and difficult to optimize. We describe an automatic method for synthesizing pipelined processors with optimal throughput and the necessary circuitry to resolve control and data hazards. Our method is applicable to a very abstract description of the circuit, thus allowing it to be applied early in the design process.

Overview

Figure 1 provides an overview of our method. The input includes the dataflow of the functional units in the circuit, and the execution times for each functional unit. We also require *data dependency (DD)* information about the functional units similar to a def-use relation.

We have hazard rules that describe how hazard resolution circuitry must be added to the circuit based on the data dependencies between functional units and the staging of the computation. For example, if a unit *needs* data that a unit immediately following it in the dataflow *computes*, then the results of the second must be forwarded to the first when these two units are in separate stages of the pipeline.

We use Boolean variables to represent the presence or absence of pipeline registers and hazard resolution circuitry. The application of the hazard rules creates a Boolean satisfaction problem that can be

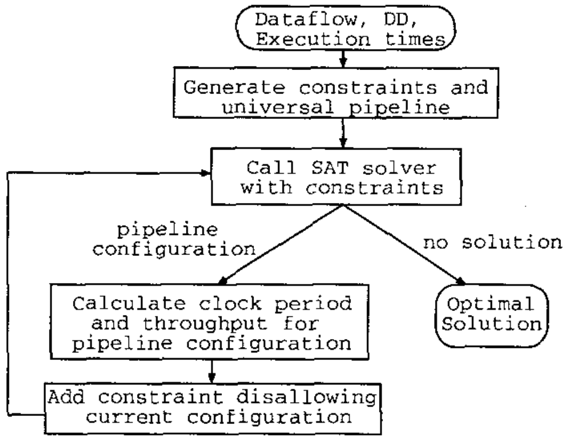
Dataflow, DD, Execution times
Generate constraints and universal pipeline
Call SAT solver with constraints
pipeline configuration
no solution
Calculate clock period and throughput for pipeline configuration
Optimal Solution
Add constraint disallowing current configuration

Figure 1: Optimal Pipeline Synthesis Method

solved using a SAT solver to produce a pipeline configuration. We use the term ***pipeline configuration*** to mean a pipeline layout with its control and data hazards resolved.

In the process of generating the constraints, we also create a pipeline configuration that is the union of all possible pipeline configurations, which we call the ***universal pipeline***. The presence or absence of the pipeline registers and hazard resolution circuitry in the universal pipeline is dependent on the Boolean variables used in the constraints.

Using the universal pipeline and the values for the Boolean variables that result from the SAT solver, we calculate the clock period including the time taken by the hazard resolution circuitry. Resolving the hazards may introduce stalls or kills into the pipeline, which decrease the throughput and, therefore, change the cost function to be optimized. Using information about the frequency of the data dependencies (e.g., how many times a Load instruction is immediately followed by a dependent Alu instruction), we calculate the throughput for the pipeline configuration.

We iterate this process of determining a pipeline configuration and calculating the throughput until all pipeline configurations have been considered and we know the optimal solution.

We currently have several restrictions on the types of circuits that our method can optimize. The dataflow must be for in-order execution of instructions. We do not have rules that check for write-after-write or write-after-read hazards. We assume that the instruction path is linear, meaning that there no forks or joins in the dataflow. We assume writebacks occur at the beginning of the computation of the next step (i.e., we have a write-before-read register file). Also, our method does not handle structural hazards.

c

Method

In this section, we describe our method in more detail, using the DLX microprocessor [7] as given in Figure 2 as an example. The functional units have the names used by Hennessy and Patterson, e.g., "ID" is the decode unit. These units are partitioned to illustrate the instruction set architecture (ISA) state (program counter (*pc*), register file (*rf*), memory (*mem*)). The arrows denote wires carrying data from one hardware component to another.

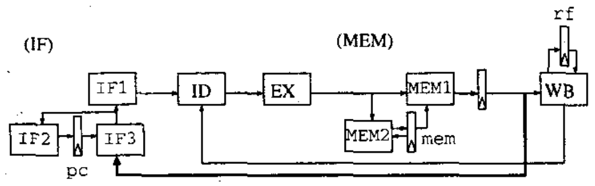

Figure 2: Unpipelined (dataflow) DLX

Data Dependency (DD) Information

The data dependency (DD) information that we require is similar to a def-use relation among the functional units. It describes what data is *computed*, *needed*, or *written* to ISA state by each functional unit for data that is affected by multiple functional units. For the DLX, the DDs appear in Figure 3.

```
Needs(IF3, pc)
Needs(ID, regVal), Computes(ID, pc)
Computes(EX, regVal), Computes (EX,pc)
Computes(MEM1, regVal)
Writes(WB, regVal)
```

Figure 3: Unpipelined (dataflow) DLX

The ID unit *needs* the values of the source operands from the register file and we express this as *Needs*(*ID*, *regVal*). The EX unit performs Alu operations, so it is *computing* register values and has the DD *Computes*(*EX*, *regVal*). The destination register value is *written* to ISA state in the WB functional unit. For unconditional branch instructions using direct addressing the *pc* is *computed* in the ID stage, but for conditional branches or branches using relative addressing the *pc* is not *computed* until after an Alu operation is performed in the EX stage. Therefore, we say that both ID and EX *compute* the *pc*.

Hazard Resolution Rules

Figure 4 lists our rules that determine where to place hazard resolution hardware depending on the DD information and on how functional units are partitioned into stages (i.e., the location of the pipeline registers). The result of the application of these rules is a functionally correct pipeline, but not necessarily an optimal pipeline.

Bypass Rules (Rules 1, 2 and 3)

When a *need* for a value and its *computation* (without a *write* of the value to ISA state) are separated by 1 stage then the hazard can be resolved by forwarding the data back to the previous stage. An instance of Rule 1 that satisfies the DLX data dependencies involves the ID in stage 1 immediately followed by the EX in stage 2 and no other functional units in stage 2:

$$\begin{array}{ll} & \text{Stage}(ID,1) \wedge \text{Stage}(EX,2) \\ \wedge & \text{Needs}(ID, regVal) \wedge \text{Computes}(EX, regVal) \\ \Leftrightarrow & \text{forward}(EX, ID, regVal) \end{array}$$

Figure 5 illustrates how this rule introduces a bypass (BYP1) if there are pipeline registers between ID and EX (IDEX), and between EX and MEM1 (EXMEM1). The presence of IDEX ensures that the *needs-computes* dependency between ID and EX is 1 stage apart. The EXMEM1 register guarantees that no functional units later in the stage with EX *write* to the register file. If the EXMEM1 register is absent and the MEM1WB register is present, then the data would be forwarded back to ID from MEM1 rather than EX. In this instance of the rule, no mention is made of the IFID register. The presence or absence of IFID does not have any affect on the rule conditions being satisfied. If it were present, then the stage numbers would be increased by one.

The second bypass rule is similar to the first, but it involves *needs-computes* dependencies between functional units that are 2 or more stages apart. A bypass

$$\textbf{Rule 1 (byp):}\quad \begin{array}{ll} & \mathrm{Stage}(A,i) \wedge \mathrm{Stage}(B,i+1) \\ \wedge & \mathrm{Needs}(A,x) \wedge \mathrm{Computes}(B,x) \wedge (x \neq pc) \\ \wedge & (\forall C.\, \mathrm{Stage}(C,i+1) \wedge \mathrm{Earlier}(B,C) \Rightarrow \neg \mathrm{Writes}(C,x)) \\ \Leftrightarrow & \mathrm{forward}(B,A,x) \end{array}$$

$$\textbf{Rule 2 (byp):}\quad \begin{array}{ll} & \mathrm{Stage}(A,i) \wedge \mathrm{Stage}(B,i+y) \wedge (y \geq 2) \\ \wedge & \mathrm{Needs}(A,x) \wedge \mathrm{Computes}(B,x) \wedge (x \neq pc) \\ \wedge & (\forall C. \mathrm{Stage}(C,i+y) \wedge \mathrm{Earlier}(B,C) \Rightarrow \neg \mathrm{Writes}(C,x)) \\ \Leftrightarrow & \mathrm{forward}(B,A,x) \end{array}$$

$$\textbf{Rule 3 (byp):}\quad \begin{array}{ll} & \mathrm{Stage}(A,i) \wedge \mathrm{Stage}(B,i) \wedge \mathrm{Earlier}(A,B) \\ \wedge & \mathrm{Needs}(A,x) \wedge \mathrm{Computes}(B,x) \wedge (x \neq pc) \\ \wedge & (\forall C.\, \mathrm{Stage}(C,i+1) \Rightarrow \neg \mathrm{Writes}(C,x)) \\ \Leftrightarrow & \mathrm{delayedForward}(B,A,x) \end{array}$$

$$\textbf{Rule 4 (stall):}\quad \begin{array}{ll} & \mathrm{Stage}(A,i) \wedge \mathrm{Stage}(B,j) \wedge \mathrm{Stage}(C,j+y) \wedge (i \leq j) \wedge (y \geq 2) \\ \wedge & \mathrm{Needs}(A,pc) \wedge \mathrm{Needs}(B,x) \wedge \mathrm{Computes}(C,x) \\ \Leftrightarrow & \mathrm{stall}(j-1) \wedge \mathrm{stall}(j-2) \wedge \ldots \wedge \mathrm{stall}(i) \wedge \mathrm{stallpc} \end{array}$$

$$\textbf{Rule 5 (kill):}\quad \begin{array}{ll} & \mathrm{Stage}(A,i) \wedge \mathrm{Stage}(B,i+x) \wedge x > 0 \\ \wedge & \mathrm{Needs}(A,pc) \wedge \mathrm{Computes}(B,pc) \\ \Leftrightarrow & \mathrm{kill}(i+x-1) \wedge \ldots \wedge \mathrm{kill}(i) \wedge \mathrm{fixpc} \end{array}$$

A, B and C are functional units, i is a stage number, and x is the data being manipulated.

Figure 4: Hazard Resolution Rules

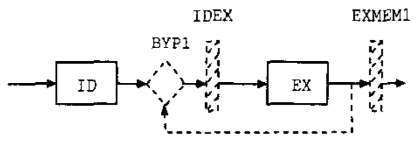

Figure 5: Example Instance of Rule 1

is used to pass data from the functional unit that *computes* it back to the functional unit that *needs* it. At least one stall is also required in this case, as described in Rule 4.

The third bypass rule is necessary for functional units within the same stage but where writing the data to ISA state does not occur until later in the pipeline. In this case, we delay the forwarding of data until the next stage to avoid a combinational loop.

The *pc* is excluded from the bypass rules, because data dependencies involving the program counter are resolved using stalls and kills (Rules 4 and 5). The same bypass may satisfy multiple rules.

Stalling Rules (Rule 4)

Because stalls and kills affect the program counter, the DD information must include a single functional unit that *needs* the *pc*, which outputs the *pc* of the next instruction to fetch, and the circuit must have a distinguished signal that is the feedback program counter based on the computation of a instruction. In Figure 2, these are IF3 and the bolded line respectively.

A pipeline stall is required when the *need* for a value and its *computation* are separated by two or more stages. This hazard is resolved by detecting the need to stall in the stage that needs the data, and stalling all previous stages the appropriate number of times. The *pc* is also stalled (not incremented) by sending a signal to the functional unit that *computes* the *pc*.

Branching Rules (Rule 5)

Instructions in the shadow of a mispredicted branch must be killed in the pipeline. Similar to stalls, the resolution for this hazard involves sending a signal to kill instructions in all previous stages, and sending the correct *pc* to the program counter.

The resolution strategy for stalls and branches must work together as they both affect the *pc*. The signals to revise the *pc* (either by stalling or setting it to the destination of the branch) are coordinated so that the later instructions in the pipeline have precedence. The signals sent back to the *pc* are combined in a *pc* selector unit, illustrated in Figure 7.

Constraints

Instantiations of the rules create Boolean constraints. We associate a Boolean variable with each pipeline register and each hazard resolution component (e.g., bypass). For example, the instance of Rule 1 given above, creates the Boolean constraint:

$$\mathrm{BYP1} \Leftrightarrow \mathrm{IDEX} \wedge \mathrm{EXMEM1}$$

These constraints are given to the SAT solver SATO [14] to determine a pipeline configuration. For n functional units in a linear order, there are at least 2^{n-1} possible pipeline configurations based on the possible placements of the pipeline registers. A pipeline configuration is a pipeline layout with its hazards resolved; if it has m pipeline registers there are $m + 1$ stages.

Universal Pipeline

From the hazard resolution rules, we also generate the universal pipeline, which is the union of all possible pipeline configurations. The presence or absence of the hazard resolution circuitry and pipeline registers in the pipeline configuration depends on the Boolean variables used in the constraints. In Figure 5, if the Boolean variable BYP1 is true, then its bypass is part of the pipeline.

The application of the rules in creating the universal pipeline is ordered to ensure that bypasses are placed before kill units, which are placed before stall units. This order ensures that kills take precedence over stalls and that forwarding of data happens before determining whether an instruction should be stalled or killed.

The SAT solver determines a satisfying assignment to the Boolean variables. The universal pipeline is instantiated with these variable values to create a pipeline configuration, which includes the necessary hazard resolution circuitry.

Clock Period

To calculate the clock period for a pipeline configuration, we use reinterpretation of the circuit description [12]. Reinterpretation allows us to use the same description of the circuit for simulation, timing analysis and other interpretations. In reinterpretation, we redefine the functional units to accomplish different tasks. For simulation, we use a stream-based interpretation similar to that used in the Hawk HDL [11]. For the timing interpretation, each unit takes as input the time taken by the functional units and hazard circuitry between it and the previous pipeline register, and outputs this time plus its execution time. Cycles created by feedback loops are broken by naming the pipeline registers and requiring that these names are distinct. This method calculates the maximum time between pipeline registers in the pipeline configuration, thus determining the clock period.

Throughput

The throughput of a pipeline configuration depends on the clock period and on how many times instructions will stall or be killed, which makes its cost function specific to a particular pipeline configuration and program. From the application of the rules, we know which *computes* and *needs* pairs of dependencies in the DD information result in particular hazard resolution circuitry. For a pipeline configuration that requires stalls or kills, the user must provide the frequency of the occurrence of this dependency. For example, the MEM1 unit computes the value of operands when we have a Load instruction. This value is needed by the ID unit when that operand is used in a following instruction. Thus, in asking the user how many times the ID unit depends on the MEM1 unit, we are asking for the frequency of Loads followed immediately by dependent Alu operations. The instruction frequencies can be determined from benchmark programs. A different benchmark could result in different optimal pipeline configurations. Using the frequencies of the dependencies, we calculate the throughput as defined in Figure 6.

$$\mathit{throughput} = \frac{1}{\sum_{j=0}^{n}(j+1) * p_j * cp}$$

cp	clock period
p_j	percentage that result in an j cycle delay
n	maximum number of cycle delays required

Figure 6: Throughput Calculation

Once the throughput has been calculated, an additional constraint disallowing the current solution is added to the set of constraints. We iterate the process of finding a satisfying assignment, calculating the clock period and throughput, and recording the maximal throughput and its pipeline configuration. This cycle continues until there are no more solutions, meaning that every possible pipeline configuration has been considered.

Examples

Figure 7 shows two optimal DLX pipeline configurations that result from our method. The functional units and hazard resolution circuitry are labeled with their execution times. The cost of a pipeline register is 3 time units. The upper pipeline has four stages and has a clock period of 66 time units and a throughput of 0.01358 instructions/time unit. The lower one has three stages and has a clock period of 70 time units and a throughput of 0.01350 instructions/time unit. The input used to generate these pipelines differs only in the cost of the EX stage. The upper pipeline requires a bypass from the third stage (EX and MEM) to the second (ID). It requires a 2-cycle kill. The lower pipeline requires a bypass and a 1-cycle kill. The frequency of mispredicted branches used in this example is 5.8%. Running our implementation to generate an optimal pipeline configuration takes less than 15 seconds on a 1.8 GHz Pentium III Xeon server with 4G of RAM.

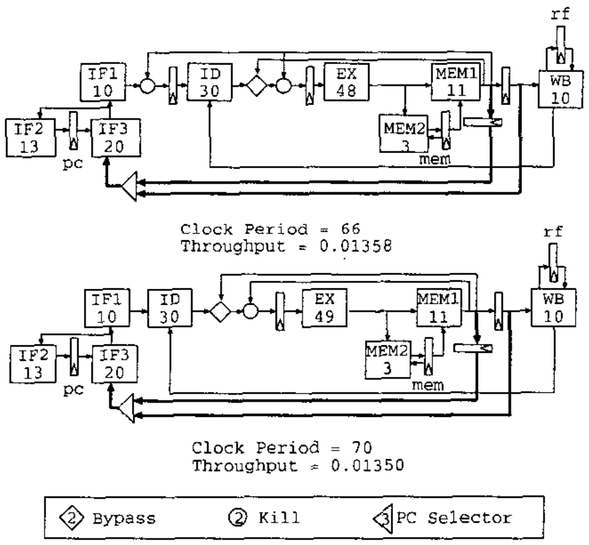

Figure 7: Optimal DLX Pipeline Configurations

Figure 8 contains two more examples of optimal pipeline configurations that result from our method. The lower pipeline contains a 1-cycle stall situation, which is caused by a Load-Alu instruction pair. In this example, the frequency of 1-cycle stalls and mispredicted branches are 6.8% and 5.8%. The results in Figure 8 demonstrate that a smaller clock period does not necessarily correlate to higher throughput. The lower pipeline has a smaller clock period than the upper pipeline, but the presence of stall and kill circuitry in the lower pipeline decreases instruction throughput.

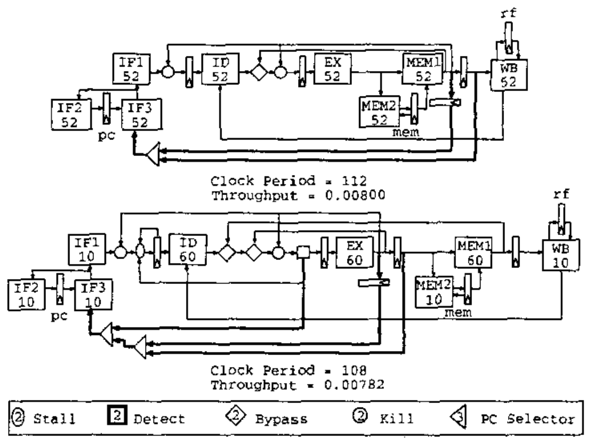

Figure 8: Optimal DLX Pipeline Configurations

The user can control the possible pipeline configurations considered by grouping multiple functional blocks into one in the input. The user can also place pipeline registers into the dataflow input, making them required components, which must appear in all pipeline configurations.

Validation

In order to validate that our rules are correct and complete, we tested the DLX pipeline configurations using simulation. Simulation allows us to "execute" instructions and examine the results to ensure that they are as expected. We took a systematic approach and tested an instance of each of the known hazard situations on all of the pipeline configurations generated by our method. We confirmed that the output of the pipeline configurations is the same as that of the unpipelined processor.

Related Work

Compared to commercial tools, such as those from Synopsys and Get2Chip [6], we work at a very high-level of description of the pipeline, thus allowing our techniques to be applied early in the design process. Compared to the work of Kroening *et al* [8], we optimize for throughput and do not include unneeded hazard resolution circuitry.

Chang and Hu [3] presented an automated tool to derive simulators from specifications. They assume that the microprocessor has already been pipelined and they are not trying to optimize the pipeline. Marinescu and Rinard [9, 10] use a context-based approach, similar to Chang and Hu, as they further partition functional units to create an efficient synchronous (pipelined) circuit, from a set of user-specified modules.

We considered using linear programming (LP), where there is a set of variables with constraints on or between them, and an objective to attain [4]. However, because we are maximizing throughput and this calculation varies with pipeline configurations, we cannot use LP. Weinhardt uses LP to optimize throughput for FPGAs, but does not consider stalls [13].

Conclusion

We have presented an automatic method for synthesizing pipeline configurations with optimal throughput and the necessary hazard resolution circuitry. The time taken by the hazard resolution circuitry is considered in the throughput calculation. Our method uses hazard resolution rules based on data dependencies among functional units to create a Boolean satisfaction problem. We are still implementing the rules that require a kill in more than one previous stage, which are not required by the DLX. Our method is applicable to a very abstract description of the circuit, thus allowing it to be applied early in the design process. Further details on our work can be found in Campbell [2].

We plan to verify our generated pipelines using the method described in Day *et al* [5] based on the Burch-Dill commuting diagrams [1]. Our long-term plans include verifying the hazard resolution rules themselves, which would allow us to conclude that the generated pipelines are correct by construction.

We also plan to investigate the scalability of our approach by applying it to more complex and realistic circuits, which may require us to overcome the limitation of only being able to apply our stall and kill rules to linear datapaths.

Acknowledgements

We thank Mark Aagaard and Jason Higgins of the University of Waterloo for providing their comments on this work, and assistance in determining realistic timing estimates for the functional units. The authors are funded by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Jerry Burch and David Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification*, volume 818 of *Lecture Notes In Computer Science*, pages 68–80. Springer Verlag; New York, 1994.

[2] Jennifer P. L. Campbell. High-level optimization of pipeline design. Master's thesis, School of Computer Science, University of Waterloo, 2003.

[3] Felix Sheng-Ho Chang and Alan Hu. Fast specification of cycle-accurate processor models. In *IEEE International Conference on Computer Design*, pages 488–492. IEEE Computer Society Press, 2001.

[4] George B. Dantzig. *Linear Programming and Extensions.* RAND Corporation, 1963.

[5] N. A. Day, M. D. Aagaard, and B. Cook. Combining stream-based and state-based verification techniques for microarchitectures. In *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes In Computer Science*, pages 126–142. Springer Verlag, 2000.

[6] R. Goering. Get2chip describes pipeline synthesis advances. *EE Times*, December 2001.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture (3rd ed.): A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2003.

[8] D. Kroening and W. J. Paul. Automated pipeline design. In *Design Automation Conference*, pages 810–815. ACM Press, 2001.

[9] Maria-Cristina Marinescu and Martin Rinard. High-level specifciation and efficient implementation of pipelined circuits. In *ASP-DAC Asia and South Pacific Design Automation Conference*, pages 655–661, 2001.

[10] Maria-Cristina V. Marinescu and Martin Rinard. High-level automatic pipelining for sequential circuits. In *14th International Symposium on System Synthesis (ISSS)*, pages 215–220, 2001.

[11] J. Matthews, John. Launchbury, and Byron Cook. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, 1998.

[12] J. O'Donnell. Generating netlists from executable functional circuit specifications in a pure functional language. In *Functional Programming Glasgow*, Workshops in Computing, pages 178–194. Springer Verlag, 1992.

[13] M. Weinhardt. Pipeline synthesis and optimization of reconfigurable custom computing machines. Technical Report iratr-1997-1, Universitat Karlsruhe, 1997.

[14] H. Zhang. SATO: an efficient propositional prover. In *Conference on Automated Deduction*, volume 1249 of *LNAI*, pages 272–275, 1997.