Scheduling Pipelined Circuits

Gordan Konevski
gordan.konveski@stud.hshl.de

***Abstract*—This paper deals with the usages and relevancy of instruction scheduling, and specifically with pipelined scheduling, elaborating on their hazards and common issues in the meantime.**

I. PIPELINING

Pipelining is a commonly used concept that aims to improve on parallelism, in cases where the underlying software has to follow through with separate tasks, either completely unique in their nature, or similar, but simply used in a different environment. Variations in the time needed to complete the tasks can be accommodated by ”buffering” and/or by ”stalling”, until the necessary requirements are met.

Suppose that assembling one car requires three tasks that take 20, 10, and 15 minutes, respectively. Then, if all three tasks were performed by a single station, the factory would output one car every 45 minutes. By using a pipeline of three stations, the factory would output the first car in 45 minutes, and then a new one every 20 minutes.

As this example shows, pipelining does not decrease the latency, that is, the total time for one item to go through the whole system. It does however increase the system's throughput, that is, the rate at which new items are processed after the first one.

Pipelines are usually divided into two classes:

- instruction pipelines
- arithmetic pipelines

A pipeline in each of these classes can be designed in two ways: static or dynamic. A static pipeline can perform only a single operation (for e.g. addition or multiplication) at one time. The operation of a static pipeline can only be changed after the pipeline has been drained. (A pipeline is said to be drained when the last input data leave the pipeline.) For example, consider a static pipeline that is able to perform addition and multiplication. Each time that the pipeline switches from a multiplication operation to an addition operation, it must be drained and set for the new operation. The performance of static pipelines is severely degraded when the operations change often, since this requires the pipeline to be drained and refilled each time. A dynamic pipeline can perform more than one operation at a time. To perform a particular operation on an input data, the data must go through a certain sequence of stages. In dynamic pipelines the mechanism that controls when data should be fed to the pipeline is much more complex than in static pipelines.

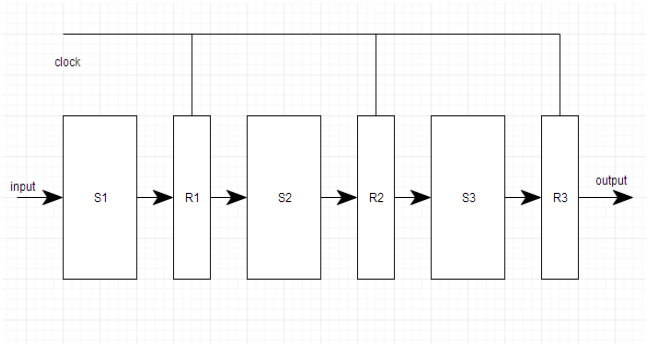

Fig. 1. Pipelining Diagram

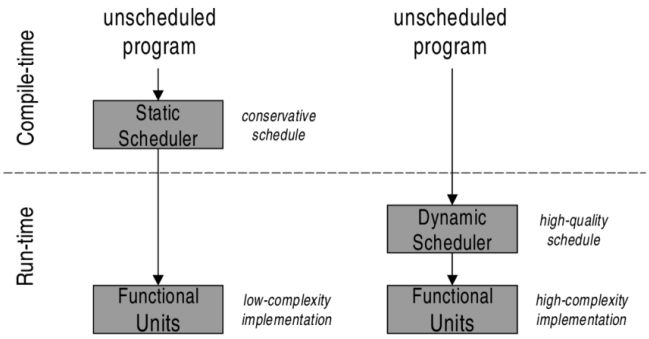

Fig. 2. Tasks Being Scheduled

II. INSTRUCTION SCHEDULING

In computer science specifically, instruction scheduling is used to improve instruction-based code, commonly on a single CPU, which improves performance on machines with instruction pipelines, which will be elaborated further below. It ultimately tries to do the following without changing the meaning of the code:

- Avoiding pipeline stalls by rearranging the order of instructions
- Avoiding illegal or semantically ambiguous operations (typically involving subtle instruction pipeline timing issues or non-interlocked resources). The pipeline stalls can be caused by hazards, which are described in further detail below.

Instruction scheduling is typically done on a single basic block. In order to determine whether rearranging the block's instructions in a certain way preserves the behavior of that block, we need the concept of a data dependency. There are three types of dependencies, which also happen to be the three data hazards and to make sure we pay heed to the three types of dependencies, we construct a dependency graph, which is a directed graph where each vertex is an instruction and there is an edge from I1 to I2 if I1 must come before I2

due to a dependency. If loop-carried dependencies are left out, the dependency graph is a directed acyclic graph. Then, any topological sort of this graph is a valid instruction schedule. The edges of the graph are usually labelled with the latency of the dependence. This is the number of clock cycles that needs to elapse before the pipeline can proceed with the target instruction without stalling.

A. Instruction pipelining

In a pipelined computer, instructions flow through the CPU in stages. For example, it might have one stage for each step of the von Neumann cycle: Fetch the instruction, fetch the operands, do the instruction, write the results. A pipelined computer usually has ”pipeline registers” after each stage. These store information from the instruction and calculations so that the logic gates of the next stage can do the next step.

This arrangement lets the CPU complete an instruction on each clock cycle. It is common for even numbered stages to operate on one edge of the square-wave clock, while odd-numbered stages operate on the other edge. This allows more CPU throughput than a multicycle computer at a given clock rate, but may increase latency due to the added overhead of the pipelining process itself. Also, even though the electronic logic has a fixed maximum speed, a pipelined computer can be made faster or slower by varying the number of stages in the pipeline. With more stages, each stage does less work, and so the stage has fewer delays from the logic gates and could run at a higher clock rate.

A pipelined model of computer is often the most economical, when cost is measured as logic gates per instruction per second. At each instant, an instruction is in only one pipeline stage, and on average, a pipeline stage is less costly than a multicycle computer. Also, when made well, most of the pipelined computer's logic is in use most of the time. In contrast, out of order computers usually have large amounts of idle logic at any given instant. Similar calculations usually show that a pipelined computer uses less energy per instruction.

However, a pipelined computer is usually more complex and more costly than a comparable multicycle computer. It typically has more logic gates, registers and a more complex control unit. In a like way, it might use more total energy, while using less energy per instruction. Out of order CPUs can usually do more instructions per second because they can do several instructions at once.

In a pipelined computer, the control unit arranges for the flow to start, continue, and stop as a program commands. The instruction data is usually passed in pipeline registers from one stage to the next, with a somewhat separated piece of control logic for each stage. The control unit also assures that the instruction in each stage does not harm the operation of instructions in other stages. For example, if two stages must use the same piece of data, the control logic assures that the uses are done in the correct sequence.

Time Slot ->	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	IF	ID	OF	EX	SR						
Instruction 2		IF	ID	OF	EX	SR					
Instruction 3			IF	ID	OF	EX	SR				
Instruction 4				IF	ID	OF	EX	SR			
Instruction 5					IF	ID	OF	EX	SR		
Instruction 6						IF	ID	OF	EX	SR	
Instruction 7							IF	ID	OF	EX	SR

Instruction Pipeline

Fig. 3. Instruction Pipeline Showcase

III. Hazards

Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycles. Any condition that causes a stall in the pipeline operations can be called a hazard. These are the three primary hazards mentioned in the sections above:

- Data Hazards
- Control Hazards or instruction Hazards
- Structural Hazards.

A. Data Hazards

A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result of which some operation has to be delayed and the pipeline stalls. Whenever there are two instructions one of which depends on the data obtained from the other.

A=3+A

B=A*4

For the above sequence, the second instruction needs the value of ‘A’ computed in the first instruction. Thus the second instruction is said to depend on the first. If the execution is done in a pipelined processor, it is highly likely that the interleaving of these two instructions can lead to incorrect results due to data dependency between the instructions. Thus the pipeline needs to be stalled as and when necessary to avoid errors.

B. Structural Hazards

This situation arises mainly when two instructions require a given hardware resource at the same time and hence for one of the instructions the pipeline needs to be stalled. The most common case is when memory is accessed at the same time by two instructions. One instruction may need to access the memory as part of the Execute or Write back phase while other instruction is being fetched. In this case if both the instructions and data reside in the same memory. Both the instructions can't proceed together and one of them needs to be stalled till the other is done with the memory access part. Thus in

general sufficient hardware resources are needed for avoiding structural hazards.

C. Control Hazards

The instruction fetch unit of the CPU is responsible for providing a stream of instructions to the execution unit. The instructions fetched by the fetch unit are in consecutive memory locations and they are executed. However the problem arises when one of the instructions is a branching instruction to some other memory location. Thus all the instruction fetched in the pipeline from consecutive memory locations are invalid now and need to removed(also called flushing of the pipeline).This induces a stall till new instructions are again fetched from the memory address specified in the branch instruction. Thus the time lost as a result of this is called a branch penalty. Often dedicated hardware is incorporated in the fetch unit to identify branch instructions and compute branch addresses as soon as possible and reducing the resulting delay as a result.

IV. ARCHITECTURAL PROBLEMS

Three sources of architectural problems may affect the throughput of an instruction pipeline. They are:

- Fetching problems
- Bottleneck problems
- Issuing problems

A. The fetching problem

In general, supplying instructions rapidly through a pipeline is costly in terms of chip area. Buffering the data to be sent to the pipeline is one simple way of improving the overall utilization of a pipeline. The utilization of a pipeline is defined as the percentage of time that the stages of the pipeline are used over a sufficiently long period of time. A pipeline is utilized 100% of the time when every stage is used (utilized) during each clock cycle. Occasionally, the pipeline has to be drained and refilled, for example, whenever an interrupt or a branch occurs. The time spent refilling the pipeline can be minimized by having instructions and data loaded ahead of time into various geographically close buffers (like on-chip caches) for immediate transfer into the pipeline. If instructions and data for normal execution can be fetched before they are needed and stored in buffers, the pipeline will have a continuous source of information with which to work. Prefetch algorithms are used to make sure potentially needed instructions are available most of the time. Delays from memoryaccess conflicts can thereby be reduced if these algorithms are used, since the time required to transfer data from main memory is far greater than the time required to transfer data from a buffer.

B. The bottleneck problem

The bottleneck problem relates to the amount of load (work) assigned to a stage in the pipeline. If too much work is applied to one stage, the time taken to complete an operation at that stage can become unacceptably long. This relatively long time spent by the instruction at one stage will inevitably create a bottleneck in the pipeline system. In such a system, it is better to remove the bottleneck that is the source of congestion. One solution to this problem is to further subdivide the stage. Another solution is to build multiple copies of this stage into the pipeline.

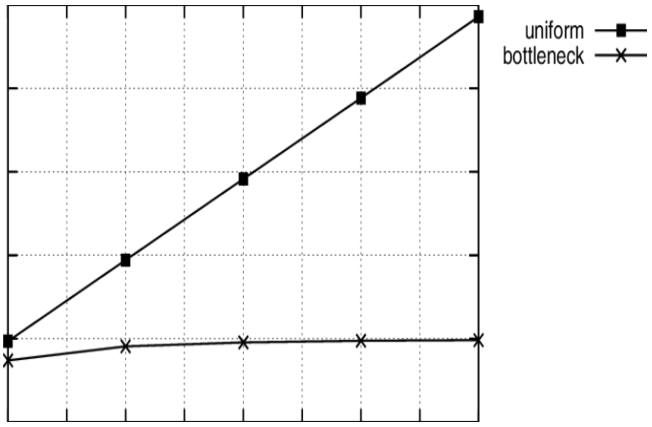

Fig. 4. Bottleneck in Pipelining

C. The issuing problem

If an instruction is available, but cannot be executed for some reason, a hazard exists for that instruction, which is where e have the three hazards mentioned above.

V. LIST SCHEDULING

Even a simple formulation of optimal instruction scheduling is an NP-complete search problem. A search for the optimal solution can take exponential time. Therefore most instruction schedulers try to find good, but possibly suboptimal schedules using heuristic algorithms.

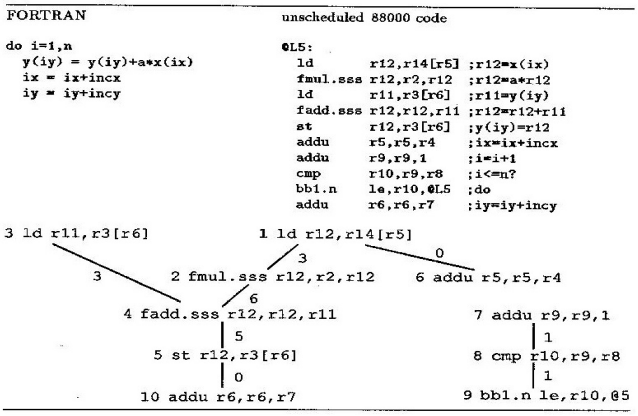

Fig. 5. List Scheduling Example

The most common algorithm is list scheduling. It builds a data dependence graph for each basic block. Figure 4 shows an example graph. An edge from instruction a to instruction b indicates that a must be executed before b to preserve the correctness of the overall program. Dependence edges exist between reads and writes, writes and reads and between writes

to the same register or memory location. After building the dependence graph the algorithm selects one of the leaders (instructions without predecessors) and removes it from the graph. This step is repeated until the graph is empty. The order in which the instructions are removed is the new instruction order of the basic block. The selection function determines the quality of the schedule. A typical selection function uses:

A. smallest earliest execution time (EET)

The EET of an instruction is the cycle when the instruction can start executing, because it is no longer delayed by any hazards. Nothing can be gained by choosing an instruction with a higher EET, because (in the absence of structural hazards) a leading instruction Cannot be delayed by instructions that are executed before its EET has arrived. Ties are broken by the maximum path length.

B. maximum path length

The path length is the sum of the latencies along the longest path to the end of the basic block 2. This heuristic exposes delay slots early, while there are other instructions to fill them.

VI. Switching Between Heuristics

In mainly sequential code data hazards are likely to occur. Structural hazards are easy to avoid, because the pipeline is often idle. On the other hand, in code with much instruction-level parallelism the execution speed is limited by pipeline contention. Structural hazards are common. Data hazards are easy to avoid, because there are many independent instructions that can be scheduled into delay slots.

```
seq_par(leaders)
  if parallelism<threshold
    return seq(leaders)
  else
    return par(leaders)
```

Fig. 6. seq_par

Therefore our selection function seq_par switches between a selection function for sequential code and one for parallel code. Both selection functions are described below. If there are no structural hazards, they produce the same results as "ep". The choice is based on the parallelism of the basic block. We define the parallelism as:

$$parallelism = \frac{cycles\ needed\ by\ the\ most\text{-}used\ stage}{critical\ path\ length}$$

Fig. 7. Parallelism

A related idea is used in Integrated Prepass Scheduling, which tries to reconcile instruction scheduling with register allocation. It switches between scheduling for pipelining and scheduling for register allocation based on the number of used and available registers.

A. The Sequential Heuristic Seq

In sequential code the instruction with the longest path length must be executed as soon as possible. On a machine with structural hazards scheduling an early instruction can delay a later instruction. Since we want to execute the instruction with the longest path length as soon as possible, so we select only this instruction or an instruction that does not delay its execution. Among those we choose with a secondary selection function. See figure 8. We tested several reasonable secondary selection functions and found that they did not make any difference. In the measurements we seq itself as secondary function.

```
seq(leaders)
  critical_leaders←{leaders with maximal path length}
  ready_critical← one of critical_leaders with minimal EET
  earlier←{leaders that do not increase the EET of ready_critical when one of
   them is scheduled before ready_critical}
  if earlier= ∅
    return ready_critical
  else
    return secondary(earlier)
```

Fig. 8. Sequential Heuristic Seq

Let's see how seq handles the SAXPY loop. After scheduling the first ld the fmul becomes the ready_critical instruction, because it has maximal path length. The second ld still fits in front of it, but the addu does not - it would delay the fmul as we have already seen. So the fmul is scheduled immediately. The addu easily fits in one of the later delay slots. The resulting schedule is optimal (see figure 9).

```
@L5:
  ld       r12,r14[r5] ;r12=x(ix)
  ld       r11,r3[r6]  ;r11=y(iy)
  fmul.sss r12,r2,r12  ;r12=a*r12
  addu     r9,r9,1     ;i=i+1
  cmp      r10,r9,r8   ;i<=n?
  addu     r5,r5,r4    ;ix=ix+incx
  fadd.sss r12,r12,r11 ;r12=r12+r11
  st       r12,r3[r6]  ;y(iy)=r12
  bb1.n    le,r10,@L5  ;do
  addu     r6,r6,r7    ;iy=iy+incy
```

Fig. 9. Sequential Heuristic Results

B. The Parallel Heuristic Par

In parallel code the scheduler has two goals: to avoid structural hazards and to keep the bottleneck stages busy. Therefore it should select instructions that use the bottleneck stages and do not cause structural hazards. While this is a good solution for problems like nonpipelined functional units and nonreplicated functional units in superscalar processors, it is not the whole story. Depending on the pipeline structure, such a heuristic can lead to the suppression of some instruction

classes, causing unbalanced and bad scheduling. It could suppress non-integer instructions, if they do not use the writeback stage as early as integer instructions. Therefore our selection function delays non-integer instructions only if it knows that non-integer instructions will be preferred soon. I.e., if an integer instruction in the next cycle after the current one would cause a writeback collision. The parallel selection function uses the EET as the primary criterion, the instruction class heuristic described above as the second, and path length as the least significant criterion. The EET is first, because in parallel code we cannot afford to miss a cycle. Path length is last, because it is not so important in parallel code. The switching scheme protects from the problems that this selection function may exhibit in sequential code.

```
par(leaders)
  ready_leaders←{leaders with minimal EET}
  non_conflicting←{ready_leaders that do not conflict with the scheduled
   instructions}
  if non_conflicting= ∅ non_conflicting←ready_leaders
  if an instruction that uses the bottleneck stage early causes a collision when
   executed in the next cycle
    early_users←{non_conflicting that use the bottleneck stage early}
    if early_users= ∅ early_users←non_conflicting
  else
    early_users←non_conflicting
  return one of the early_users with maximal path length
```

Fig. 10. Parallel Heuristic

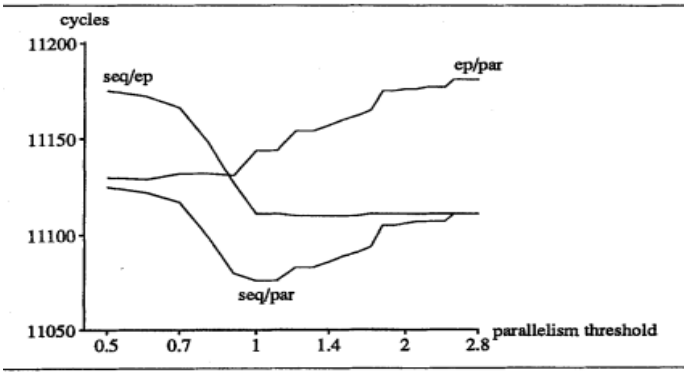

Fig. 11. Seqs Compared

Par is shown in figure 10. As we present it, this heuristic is quite specific for the writeback problem, but it can be adapted to other situations easily. Just change the instruction class heuristic appropriately. The first instruction chosen is again the first ld. The instruction class heuristic then selects the addu and saves the second ld for the third cycle to avoid ep's writeback collision. Then the scheduler is faced with integer instructions and the fraul. Since the instruction class heuristic has higher precedence than path length, an integer instruction is chosen (to save the fmul for the next cycle, when an integer instruction would cause a writeback collision with the second ld). Pat's schedule is as bad as ep's, which demonstrates the need to switch to seq when scheduling sequential code.

VII. Conclusion

Although the examples above are highly specific, it goes to show just how flexible piplining is as a concept. But its high degree of flexibility and, almost paradoxically, its high degree of specificity (at least in terms of where this concept is applied) would realistically render pipelining complex and costly to implement. Nevertheless, keeping in mind the increase in latency, it is a reliable and comprehensive solution for systems that require a higher throughput and less processing cycle times used in any given CPU.

References

[1] G. De Micheli *Synthesis and Optimization of Digital Circuits*.
[2] C. ButtazzoC. Buttazzo *Hard real-time computing systems: predictable scheduling algorithms and applications*. 3rd ed. New York: Springer, 2011
[3] R. Mall *Real-Time Systems: Theory and Practice*.
[4] Andrew Matthew Lines *Pipelined Asynchronous Circuits*.
[5] Maria-Cristina Marinescu, Martin C. Rinard *High-Level Synthesis of Pipelined Circuits from Modular Queue Based Specifications*.
[6] Mehdi Zargham, Prentice Hall *Computer Architecture*.
[7] Azeddien M. Sllame *A Pipeline Scheduling Algorithm for High-Level Synthesis*.
[8] Achim Rettberg, Mauro Zanella, Christophe Bobda, Thomas Lehmann *A Fully Self-Timed Bit-Serial Pipeline Architecture for Embedded Systems*.
[9] Achim Rettberg, Raphael Weber *Implementation of the AES Algorithm for a Reconfigurable, Bit Serial, Fully Pipelined Architecture*.
Other material used:

- https://www.wikiwand.com/en/Pipeline_(computing)
- https://www.wikiwand.com/en/Instruction_pipelining
- http://www.cs.cmu.edu/afs/cs/academic/class/15745-s06/web/handouts/11.pdf
- http://users.ece.northwestern.edu/ seda/pipeline.pdf
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-004-computation-structures-spring-2017/c7/c7s2/c7s2v2/pipelined-circuits-6-11-/
- fig 1: http://cssimplified.com/wp-content/uploads/2014/11/Instruction_Pipeline.jpg
- fig 2: https://www.researchgate.net/figure/Dependency-graph-b-of-an-instruction-block-a_fig7_299520102
- fig 3: http://cssimplified.com/computer-organisation-and-assembly-language-programming/explain-the-working-of-the-instruction-pipelining-with-the-help-of-a-diagram-5m-dec2005
- fig 4: https://www.researchgate.net/figure/The-effect-of-bottleneck-stages-on-the-performance-of-a-pipeline-Here-we-have-a_fig3_222539867
- fig 5 - 11: https://link.springer.com/content/pdf/10.1007%2F3-540-55984-1_19.pdf

.