Pipelining: Hazards, Methods of Optimization, and a Potential Low-Power Alternative

Solomon Lutze
Senior Thesis, Haverford Computer Science Department
Dave Wonnacott, Advisor
May 4, 2011

Abstract

This paper surveys methods of microprocessor optimization, particularly pipelining, which is ubiquitous in modern chips. Pipelining is a method of executing instructions in stages, so multiple instructions can be operating in the pipeline simultaneously and allow the chip use its resources more efficiently. This system creates hazards, which are potential incorrect answers: these can be structural hazards (insufficient logical hardware to process all queued instructions), data hazards (data is read, written, and overwritten incorrectly), or branch hazards (the pipeline does not know whether to load target or fall-through instructions). These complexities slow down the pipeline, so in order to improve speed against all constraints, additional hardware (and therefore extra energy and heat) are required to detect potential hazards and resolve them. This work informs our study of an architecture, conceived of by Dave Wonnacott, that has a more complex and subdivided instruction set. This shifts much of the complexity from hardware to compiler design, which allows for smaller chips. Smaller chips have lower heat and energy costs, which is itself valuable but also creates the potential for running multiple chips at the same cost as one larger (pipelined) chip.

Contents

Introduction and Background – Single-Cycle Instructions and Microcoding

Before discussing the pipelined approach to architecture, we will briefly review less complicated architectures. These systems have the advantage of being simple to implement and to write a compiler for, but as we will see have several large holes in performance. Analyzing these holes will help emphasize the gains made by pipelining. The first of our simple hardwares, which we will call hardware "A," performs all operations in a single clock cycle, and has its clock slow enough that the longest instruction will fit in a single cycle. This is an extremely naïve implementation that is suitable for, say, an undergraduate hardware class, but is not used in modern architecture.

As an example of how this architecture performs, take the following machine code, which corresponds to a high-level expression such as integer x*y+z, with x, y, and z each occupying its own register:

R1 ← R1 * R2
R3 ← R1 + R3

In this code each instruction will simply execute in order; first the MULT will execute, and then the ADD will execute. Note that the clock cycle in this architecture is long enough to accommodate its longest instruction. Thus, the MULT and ADD both take the same amount of time to execute (one clock cycle) despite the fact that the MULT is more complicated and time-consuming operation.

The second, hardware "B," shortens clock cycles and permits an instruction to take multiple clock cycles to complete. This is called "microcoding."

Returning to the machine language example from hardware "A," we find that rather than taking one (unnecessarily long) clock cycle to complete, both the MULT and ADD consist of several shorter clock cycles – for example, the MULT might take (for example) ten clock cycles to complete, and the ADD might then take three cycles to complete. Again, each is allowed to complete before the next instruction is issued, so no problems arise from the dependency of the ADD on the MULT.

This architecture, though more complicated, is obviously preferable to architecture "A" because quick instructions (such as an "OR" operation, which takes little more time than a single gate delay and register load) need not take the same amount of time as the longest instructions. Thus, every instruction except the longest one executes more quickly.

Challenges to Correctness

One of the primary advantages of these naïve implementations is that there are very few challenges to correctness. Because all of the instructions execute in order and one at a time, we can always be assured, given correct clocking (clock cycle long enough to accommodate every instruction in "A," and instructions given enough clock cycles to complete in "B"), the hardware will execute the instructions it is given correctly. This means that essentially no time or hardware needs to be spent on ensuring correctness in execution, assuming of course that the logic inherent in each instruction is correct (that is, if each instruction runs correctly on its own, then any number of them in sequence will run correctly).

Challenges to Performance

Unfortunately, the sequential nature of these hardwares causes them to be very inefficient. For

hardware "A," where the clock cycle is long enough to accommodate the longest instruction, for any given instruction the difference between the instruction's execution time and the longest instruction's execution time is simply lost time that the entire processor spends idling. For a program running largely on very quick operations, such as "and" instructions, the processor could easily spend most of its time idling, which is very inefficient.

For the microcoded architecture, where each instruction is executed in multiple clock cycles, the time spent completely idle is much lower – the smaller the clock cycle, the more exactly the end of the final clock cycle can line up with the end of the instruction and so the less time is spent idle. However, even in this scenario, much of the chip is spent idle at any given time. When an instruction is running, it is first read, then decoded, then executed, then written back to a register – each stage of which takes a different piece of hardware. This means that, for example, while the instruction is executing, it is not being read, decoded, or written back to a register, which means that these parts of the chip are idle and thus performance is not ideal.

These performance issues are, within this architecture, insurmountable; it is impossible for an architecture that executes only one instruction at a time to eliminate this inefficiency in chip use. In order to ensure that as much of the chip's processing power as possible is in use at any given moment, pipelining is necessary.

Pipelined Architecture

The third architecture is a pipelined chip. Hennessy and Patterson explain pipelining as "an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction" [HP07] (A-2). Pipelining allows code to run much faster because, ideally, it permits a new instruction to be issued every clock cycle so that multiple instructions can be run simultaneously over the course of several clock cycles - it does not increase the speed with which an individual instruction can be completed, but performing steps from several instructions at once permits each instruction after the first to be completed sooner.

If, in our example, we were attempting to double the contents of several registers, our code might look like this:

R2 ← R2 + R2
R3 ← R3 + R3
R4 ← R4 + R4

(Notice that, in this example, no instruction is dependent on any other instruction. This will be important in our later analysis of pipelining optimization.)

In this case, each instruction is loaded in turn into the pipeline, which is an abstraction representing the stages an instruction must pass through in order to be completed. The 5-stage pipeline described by [HP07] on page A-5 is specific to a particular RISC subset but is demonstrative of the way pipelining is enacted more broadly. To summarize Hennessy and Patterson, this example includes the following stages:

Instruction Fetch [IF] – The hardware fetches the instruction indicated by the program counter from memory.

Instruction Decode/Register Fetch [ID] – The instruction is decoded and required registers are read. This stage also includes performing equality tests on registers being read in the event of a potential branch. Branches can be executed at the end of this stage if the target address is written into

the PC at the end.

Execution [EX] – The ALU carries out the instruction as prepared by the previous cycle.

Memory Access [MEM] – The hardware accesses, if necessary, the memory cells affected by the instruction.

Writeback [WB] – The data produced by the instruction is placed, if necessary, into a register.

Between each of these stages may be registers to hold the data to be passed – for example, there may be an [IF]/[ID] register, an [ID]/[EX] register, etc.

So in our example, instruction 1 would be loaded into the pipeline in the [IF] stage in the first clock cycle. In the second clock cycle, instruction 1 would move to the [ID] stage of the pipeline and instruction 2 would move to the [IF] stage. In the third stage, instruction 1 would move to the [EX] stage, instruction 2 would move to the [ID] stage, and instruction 3 would move to the [IF] stage. If we take the metaphor of a pipeline – of some series of objects moving through a pipe – then pipelining involves putting a new object into the right after the one before it, rather than allowing each object to traverse the length of the pipe before putting in a new one.

Challenges to Correctness - Hazards

From this premise of allowing instructions to be issued while others are still completing comes a host of complexities and potential problems. Specifically, the potential for instructions to be run simultaneously presents the opportunity for them to execute incorrectly. Conditions that could result in the incorrect execution of an instruction are called hazards. There are three types of hazard: structural hazards (the hardware cannot support the instructions – for example, if multiple instructions need access to an adder but only one is available), data hazards (when an instruction depends on the result of a previous instruction that is not available or accurate), and control hazards (when branches or other instructions that alter the program counter must be completed in order to know which instruction to load). Managing these hazards is critical to an effective implementation of pipelining, so I will move now to a discussion of how each type of hazard arises and what measures are taken to counteract them.

A. Structural Hazards

Structural hazards occur whenever two instructions need access to the same piece of the hardware at the same time. As an example, suppose the following instructions occur:

R1 ← R1 * R1
R2 ← R2 * R2

(Highlighting is used to indicate the problematic concurrency). In this case, neither instruction relies on the other's result, so they can execute in any order or at the same time without affecting one another. However, if the architecture they are being executed on has only one multiplier, then they cannot both execute at the same time. This is referred to as a structural hazard: instructions cannot move forward in the pipeline due to the physical resource limitations of the hardware.

B. Data Hazards

Data hazards occur when some instruction relies on a piece of data that is not available due to a failure in scheduling, usually because it tried to access the data at the wrong time (too early, before it was available, or too late, after it had been overwritten). These hazards can occur for several reasons.

Hazards occur as a result of data dependencies, when one instruction relies on the result of a previous instruction to execute. In the MIPS example architecture provided by Hennessy and Patterson, an

example is provided of a pipeline that supports multiple (different) operations in the EX stage at once, where they may take more than a single clock cycle. For example, a floating-point addition spends 4 cycles in the EX stage (so it is an 8-cycle instruction), and a floating-point or integer multiply spends 7 cycles (an 11-cycle instruction). Take for example the following instructions, which are not dependant on one another:

R1 ← R2 * R2
R3 ← R3 + R3

In this case the multiply and addition could both execute as soon as they were issued, with no dependence on one another; the addition could move ahead using the adder as the multiply was occurring, and the add would finish two clock cycles earlier than the multiply despite being issued a cycle later. In this way, the architecture can allow two operations using unrelated parts of the chip to execute at the same time without the addition waiting for the slower multiply to finish executing.
Now examine the instructions:

R1 ← R1 * R2
R2 ← **R1** + R3

(The highlighting indicates the dependent operand and the result it is waiting for.)
In this case, our addition depends on the result of the multiply, meaning that the addition will check for one of its operands before that operand has been completely calculated – thus, the wrong operand may be used. Even though the addition is ordered after the multiply, the pipeline will cause them to be executing at the same time, risking an incorrect answer. This is an example of a data dependence: a correspondence between two values that demand they be executed in a particular order and, if violated, constitutes a data hazard.

The three types of data hazard are described on pages 71-2 of [HP07]. I will summarize here, relying on multiplication requiring more clock cycles than addition for my examples:

Read After Write (RAW): Instruction B tries to read a source before A writes it, so B gets the old (wrong) value. This corresponds to a true data dependence, represented in the above example: one instruction must read a value after it is written by another instruction.

Write After Write (WAW): Instruction B tries to write an operand before it is written by A, so the value written by A is incorrectly left in the destination. This corresponds to an output dependence: one instruction must be written to a destination after, not before, another.
For example:

R1 ← R2 * R3
R1 ← R4 + R5

Here, the addition – which by the ordering should have its result left after both instructions complete – will be overwritten when the multiplication finishes afterwards.

Write After Read (WAR): Instruction B tries to write an operand before it is read by A, so A gets the new (wrong) value. This corresponds to an antidependence: one instruction must read a value before it is overwritten by another instruction.

For example:

R1 ← R1 * **R2**
R2 ← R3 + R3

In this example, if there is any chance that the addition might complete before the multiplication, the multiplication will have the wrong result.

The hazard names may be somewhat counterintuitive, since for example a Read After Write hazard occurs when a read fails to happen after a write – that is, a write incorrectly happens before a read. The names should be thought of as naming the action that should have occurred, rather than the problem itself – for example, a "walking around the hole" hazard versus a "falling into the hole" hazard. Simply, the name of the hazard corresponds to the data dependency being violated.

Data hazards occur whenever dependent instructions are near enough to one another that being in the pipeline simultaneously could result in incorrect order of access, and therefore the wrong answer. They also form obstacles in reordering.

C. Control Hazards
A control hazard can sometimes be created when an instruction changes the program counter. This is obviously problematic because if a branch is encountered in the middle of the pipeline, then instructions after the branch may need to be discarded, in which case the pipeline would occur in the pipeline while the instructions indicated by the new program counter value were issued.

Take, for example, the following instructions:

R1 ← R1 * R2
BZ(LABEL)
R3 ← R1 + R3
R4 ← R4 - R2

(BZ(LABEL) here represents a branch to the target LABEL if the previous operation calculates to zero). In this case, the branch depends on the result of a time-consuming instruction, and while it is calculating whether the branch occurs it must either stall or make a prediction about whether the branch will be taken. In this case, it must have a method for undoing the operations it has begun in the event that it predicted incorrectly. The amount of time taken to correct a misprediction, including undoing the results of mistakenly executed instructions and to catch the correct instructions up in the pipeline varies with the architecture, particularly according to the length of the pipeline. (Note that this is one of the limiting upper bounds on pipeline length. As with all hazards, penalties for control hazards become steeper as pipeline length increases – see "The Optimum Pipeline Depth for a Microprocessor," [HarP02])

Challenges to performance – Stalling to avoid hazards

For all of these hazards, the simplest solution to implement is to stall. Stalling involves having the hardware introduce a delay, or bubble, into the pipeline when a hazard is encountered until that hazard is resolved. For the structural hazard above – two multiplies both trying to use the same

multiplier – the second multiply, and everything after it, would be delayed until the first multiply had finished using the multiplier. For a data hazard, such as the example of an addition that requires the result of a multiply to execute, the addition would stall before execution until the result of the multiply had been written back into its register. For a control hazard, no instructions after the branch would be loaded into the pipeline at all until the result of the branch condition had been determined.

In our simple pipelined architecture, the only way to avoid hazards is to stall, which requires hardware to detect the need to stall and to implement it. When the hardware stalls and a bubble is introduced, that bubble "floats" through the rest of the pipeline, causing idleness in the processor. For example, suppose we have the following queue of instructions:

R2 ← R3 * R4
R4 ← R2 + R2
R3 ← 1
R2 ← 2

This represents a data hazard, since the value of z (R2) will not be ready by the time the addition needs to use it. Suppose each stage of a four-stage pipeline (omitting memory access for this example) takes one clock cycle. These instructions would be loaded as followed into the pipeline across several cycles

In Cycle 1 the multiply is fetched:

Fetch: R2 ← R3 * R4
Decode: -
Execute: -
Write-Back: -

In Cycle 2 the multiply is decoded and the addition is fetched:
Fetch: R4 ← R2 + R2
Decode: R2 ← R3 * R4
Execute: -
Write-Back: -

In Cycle 3 the multiply is executed, the addition is decoded, and the first assignment is fetched:
Fetch: R3 ← 1
Decode: R4 ← R2 + R2
Execute: R2 ← R3 * R4
Write-Back: -

Now, however, we have a problem: the addition cannot execute until the value for z has been written back. Thus in Cycle 4, a bubble occurs while the addition waits for the multiply to finish its write-back stage:
Fetch: R3 ← 1
Decode: R4 ← R2 + R2
Execute: BUBBLE;
Write-Back: R2 ← R3 * R4

Once z has been written back, the hazard is averted, and the program can continue with every

operation behind the multiply running behind. For Cycle 5, we see:

Fetch: R2 ← 2
Decode: R3 ← 1
Execute: R4 ← R2 + R2
Write-Back: BUBBLE;

A bubble, we can see, acts like a blank instruction after it is created, since it forces the hardware for each stage to be bound up doing nothing. In more complicated pipelines, such as pipelines with multiple-cycle executions for some operations (seven cycles for a multiply, for example), stalling could cause delays longer than a cycle, and if stalls occur frequently, stalling costs can add up quickly. Stalling is the main reason why long pipelines become problematic; although a longer pipeline increases throughput because more instructions can be in the queue at once, it also increase the opportunity for hazards and the costs for stalls, necessitating a balance for optimal efficiency. Hartstein and Puzak identified these tradeoffs in "The Optimal Pipeline Depth for a Microprocessor" using 35 different workloads including legacy, modern, and SPEC95 and SPEC2000 workloads simulated on pipelines ranging from 10 to 35 stages. Obviously, there is no consistent ideal pipeline depth; what works best varies with the architecture's clock cycle, issue width, and specific workload [HarP02].

It should be noted that there are stalls which can be predicted by the compiler and stalls which cannot. As an example of the latter, imagine a variable load immediately followed by an addition involving that variable:

R1 ← LOAD(R3)
R1 ← R1 + R2

Without any guarantee as to when the load will finish (since it might miss in several layers of cache and seek out the hard drive before it finally completes), there is no way for the compiler to predict in advance what the delay will be. In other instances, however, delays can be determined more exactly – take for example the following pseudocode:

```
x=x*y;
z=x+5;
```

which might translate to:

R1 ← R1 * R2
R3 ← R1 INC 5,

with an increment instruction being used instead of a regular addition. In this case the compiler, which necessarily has an intimate knowledge of the workings of the hardware, would know exactly how many cycles were needed for a multiply to complete and thus would be able to determine ahead of time exactly how long the increment instruction would need to be delayed before its operands would be ready. Importantly, it can determine this without any additional information from the hardware, and could trust that the hardware would know to stall in this instance.

Methods of Optimizing Against Hazards – Compiler-Level

While stalling is a universal remedy in that it can be used to resolve any pipelining hazard, the costs are high, and stalls impact a chip's ability to perform efficiently. However, there are other

methods available to resolve hazards that help retain efficiency. The first we will examine are all performed in the compiler; no additional hardware need be added to implement them because the improvements are made to the code itself, not to the machine running it. Implemented correctly, compiler-level optimizations, as opposed to hardware-level optimizations, can provide a solution to hazards that does not require extra power and can be performed on any hardware that can implement the simple pipeline described above.

Resolving Structural Hazards

One approach to structural hazards is to reorder operations such that two instructions are never so close to one another that this sort of hazard occurs. The extent to which this is possible depends largely on the hardware; a string of successive multiplications could be difficult to organize such that a structural hazard never occurred. However, barring that, reordering operations to prevent this kind of hazard is often viable during compiler time. Because the compiler has knowledge of how long a particular instruction will take – it might know, for example, that a multiply instruction will begin using the multiplier two cycles after it is issued and stop using it seven cycles later – it can organize instructions such that they do not overlap in this way using only information available at compile time.

This technique is called *static instruction reordering*, and is used to avoid data hazards due to dependencies. Take, for example, the sequence of instructions:

R1 ← R1 + R2
R3 ← R3 + R3
R4 ← 1
R5 ← 2
R6 ← 3
R7 ← 4

If the microprocessor were to execute the operations in the order given (and if it did not have a second adder unit), it would be forced to wait until the first add instruction had left the adder to begin the second add. However, the resources required by the SET instructions are not involved in the ADD instruction (that is, there is no structural hazard present there); thus, these instructions could be carried out after the first add had been started so that by the time the second was issued the architecture would be able to handle it:

R1 ← R1 + R2
R4 ← 1
R5 ← 2
R6 ← 3
R7 ← 4
R3 ← R3 + R3

In the above example, as with many potential structural hazards, it is possible to detect the hazard statically at compile time, meaning that significant out-of-order execution optimization may be performed at the compiler level. Because my performance analysis for this paper is limited to resources concerned at the hardware level (time, chip space, and power consumption), being able to move this optimization to the compiler level is optimal because resources need not be spent on it by the chip at runtime. In some cases, however, enough instructions may depend on the same hardware that

scheduling cannot fix this problem, so some attention will be given to means of averting structural hazards available at the hardware level.

Resolving Data Hazards

At the compiler level, the only real way to resolve data hazards, as with structural hazards, is the reordering of instructions. Because the compiler has intimate knowledge of the amount of time spent in each stage of the pipeline, it can order instructions to prevent some of these hazards from occurring. However, there are not always enough non-dependent instructions to completely clear up these problems by scheduling. Due to this, there are extensive methods of resolving data hazards available to the hardware, as will be discussed in depth later.

An earlier example used for stalls:
R1 ← R2 * R2
R2 ← R1 + R3
R4 ← R5 – R6

could be reordered to:

R1 ← R2 * R2
R4 ← R5 – R6
R2 ← R1 + R3

This would reduce the delay incurred by the addition's dependence on the multiply. Moreover, the compiler's awareness of the length of the multiply instruction would enable it to schedule exactly as many instructions between the multiply and the dependent instruction as needed to prevent a stall, provided enough intermediate instructions were available.

Obviously, reordering can sometimes be made more difficult by data hazards; reordering to avoid one hazard can result in others, as in the following example:

R1 ← R2 * R2
R2 ← R1 + R3
R4 ← R5 – R2

In this case, the subtraction's use of R2 as an operand violates an antidependence between it and the addition, meaning that the two instructions are not independent. Normally, we might want to put an independent instruction between the multiplication and the addition, since multiplications take longer than additions and thus the addition would be delayed. However, an attempt to hide the dependency between the multiplication and the addition cannot use the subtraction to do so, since moving the subtraction above the addition would constitute another data hazard.

Resolving Control Hazards

Managing control hazards at the compiler level is related to distancing the logical operation on which the branch is based from the branch itself and on limiting the number of branches. Both of these are accomplished by loop unrolling.

Loop unrolling essentially expands the body of a loop so that fewer branches are necessary. Hennessey and Patterson have this code as an example of loop unrolling (75):

```
for (i=1000; i>0; i=i-1)
        x[i] = x[i] + s;
```

When this is converted into MIPS assembly language, with r1 holding the highest address in the array and F2 holding the scalar ("s" in the above code) to be added, it would look like this:

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result
        DADDUI  R1,R1,#-8   ;decrement pointer 8 bytes (to the next double)
        BNE     R1,R2,Loop  ;branch R1!=R2
```

(Note that the above code is taken verbatim from [HP07], p. 76.). This code both results in a number of stalls due to data dependencies (the ADD.D, for example, must be stalled while waiting for the L.D to complete) and potentially in stalls at the branch due to either branch misprediction or a lack of prediction altogether. Instead of simply suffering these stalls in each iteration of the loop, the loop can be unrolled – that is, copies of the loop body can be made inside the loop, so that fewer branches need to be followed and so that more instructions exist to exploit parallelism from out-of-order instructions. An example of code unrolled four times (four copies of the loop body) and reordered to prevent stalls is below, again taken directly from [HP07] (77):

```
Loop:   L.D     F0,0(R1)    ;
        ADD.D   F4,F0,F2    ;
        S.D     F4,0(R1)    ;drop duplicated DADDUI and BNE
        L.D     F6,-8(R1)   ;
        ADD.D   F8,F6,F2    ;
        S.D     F8,-8(R1)   ;drop duplicated DADDUI and BNE
        L.D     F12,-16(R1) ;
        ADD.D   F12,F10,F2  ;
        S.D     F12,-16(R1) ;drop duplicated DADDUI and BNE
        L.D     F14,-24(R1) ;
        ADD.D   F16,F14,F2  ;
        S.D     F16,-24(R1) ;
        DADDUI  R1,R1,#-32  ;
        BNE     R1,R2,Loop
```

Now, the loop has removed three decrements and branches, and clever scheduling has eliminated all of its stalls. Note that if we do not know the upper bound, which is likely, then when executing *n* iterations of a loop whose body has been unrolled *k* times, two loops can be used: one is the original loop, and executes *n* mod *k* times, and the other is the unrolled loop, which executes floor(*n*/*k*) times. Limits on loop unrolling include a potential increase in the cache miss rate (due to the increased code size) and running out of registers. For this reason, a very sophisticated compiler is required to determine what the optimal unrolling of a given loop is.

Costs and Challenges of Optimization – Compiler-Level

Compiler-level hazard avoidance is, when successful, the best kind of pipeline optimization because it costs nothing at runtime and requires no special hardware. For any given program, being able to identify and rearrange hazards at compile time is preferable to identifying them at runtime. First, compiler-level resolutions scale well: the program may be compiled once, during which optimization occurs, and then run again and again at no additional cost of time. Second, resolving these issues requires no extra heat or power out of the chip. Third, and perhaps most importantly, these resolutions do not require any additional chip hardware or space, a point to which we will return later. Essentially, in terms of hardware, the costs of compiler-level optimization are "none," which is of course ideal.

Unfortunately, compiler-level optimization is also extremely limited in the *types* of hazards it can detect, because many hazards are not detectable until runtime. For example, loops, which are responsible for many control hazards due to the large number of branches they include, often involve a branch that is taken many times and then not taken once, and those values are difficult to predict at compile time. Ultimately, compiler-level optimization, while extremely cost-effective, has severe limitations in terms of the kinds of hazards it can address. In order to optimize so as to avoid as many stalls as possible, hardware-level optimization is necessary.

Methods of Optimizing Against Hazards – Hardware-Level

Resolving Structural Hazards

The obvious way of dealing with structural hazards is to add more of the requisite hardware – two multipliers, for example, or more. Obviously this may not scale well, since having enough of each logical unit (adder, etc.) for every instruction that could be in the pipeline at once takes a lot of space, particularly if there is no guarantee that they will all be in use enough to justify them – thus, this solution is, by itself, not necessarily sufficient to resolve structural hazards. However, if our addition takes, for example, two clock cycles, then we may never need more than two adders (since more than two additions in the pipeline would not be in the execution stage, when they would need the adders, at the same time). Thus in this case two adders may solve addition-related structural hazards and be relatively cheap to implement.

Resolving Data Hazards

Assuming that instructions cannot be reordered at compile time to prevent the hazard, the hardware has several tools for dealing with these hazards.

Forwarding [PH07] (A-17)
Forwarding, also called bypassing, is a hardware technique that eliminates some of the problems generated by proximity of dependent instructions in the pipeline. Essentially, it permits an instruction to transfer its result as soon as it is available to the dependent instruction. Hennessey and Patterson have as an example a subtract dependent on an add:

R1 ← R2 + R3
R4 ← R1 – R5

Obviously the above instructions are dependent; in an architecture without forwarding, the FPSUB

would need to be stalled until it was able to read the FPADD's result out of R1. However, FPADD's answer is available before the register writeback stage. If we are still considering the five-stage pipeline detailed above, then we will see that the FPADD instruction will be leaving the execution stage at the same time as the FPSUB is entering the execution stage. Thus, with the appropriate hardware, the FPADD's answer may be given directly to the FPSUB even before it has been written into a register. Quoting [HP07] directly:

> Forwarding works as follows:
> 1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.
> 2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file. (A-17)

In other words, the ALU has the results from the execute/memory access register and memory/writeback registers available as inputs; it uses them if it determines that the register those are going to write is used as an input in the instruction currently in the ALU.
There are some instances where forwarding is unable to prevent a stall, for example when a load instruction immediately precedes an arithmetic instruction that requires the result of the load as an input. In this case, the result of the load would not be available until the end of the cycle it was first needed in (when the arithmetic instruction entered its EX stage) ([HP07] A-19).
Forwarding also has advantages from an energy perspective. Balfour, Harting and Dally note that via forwarding and operand register files (small files that extend pipeline operand registers and permit an increased number of variable references without significantly increasing energy costs), it is possible to reduce energy usage in the registers and forwarding network by 62%. This is accomplished by keeping a greater amount of communication local, thereby decreasing costs in communication (an area that sees far less benefit from changes in technology than does computation). These gains were assessed using a simulator and focused on operand access statistics and the energy-consumption statistics derived from them, and were benchmarked using seven kernels that included file compression, encryption, and Fast Fourier Transform algorithms [BHD09] (62).

Dynamic Scheduling
Dynamic scheduling is the name given to instructions reordered by the hardware at runtime, used when there is a data dependence that cannot be hidden using forwarding or bypassing ([PH07], 89). Compiler-time reordering presents valuable gains, however there are many runtime delays that cannot be predicted by the compiler, including cache misses, unforeseen memory references, and branch delays.

Register Renaming
Register renaming involves the elimination of apparent, but false, dependencies that occur in instructions, and is comparable to the compiler-level operation called Single Static Assignment (SSA). In SSA, variable names are changed when two variables that have the same name (usually out of convenience for either the coder or user) are actually unrelated. Take the following piece of pseudocode:

1. A=10
2. B=A+2
3. A=45
4. C=A-15

5. D=B+C

In this case, the variable "A" instantiated in step 1 and used in step 2 is wholly distinct from the variable "A" instantiated in step 3 and used in step 4. Thus, we could simply give the second instance a new name, and the program would play out the same:

1. A=10
2. B=A+2
3. A_1=45
4. C=A_1-15
5. D=B+C

with the assumption that "A" in future steps would refer to A_1 (unless, of course, "A" was redefined later).

Register renaming is a similar process that occurs at the hardware level. When the compiler suggests (as is likely with a very simple algorithm for register allocation) that a single register be reused many times in rapid succession, this saves on the number of registers used at any given time but can cause slowdown if multiple operations are waiting for a single register to be free. Thus, the hardware seeks places where a new register can be used instead of waiting for the one originally requested to be freed up.
Take for example:

1. R1 ← R2 * R3
2. R5 ← R1 OR R4
3. R1 ← R2 + R7
4. R6 ← R1 * R3

Since the OR instruction is dependent on the result of the first multiply instruction, it might be reasonable to reorder these instructions as follows, with the addition and second multiply instructions each moved up by one space to allow the OR sufficient time to resolve:

1. R1 ← R2 * R3
2. R1 ← R2 + R7
3. R6 ← R1 * R3
4. R5 ← R1 OR R4

The problem with this new order should be clear: all four instructions use R1, but instruction 4 uses the result of instruction 1, which was overwritten by instruction 2 (and subsequently used by instruction 3). Thus we have a write after write hazard. However, this hazard only exists because the register numbers used are leftovers from the previous ordering, and is thus easily corrected by renumbering the registers used, in this case by changing the ADD instruction's use of R1 (and the subsequent use of R1 in the second MULT instruction) to, for example, R8:

1. R1 ← R2 * R3
2. R8 ← R2 + R7
3. R6 ← R8 * R3
4. R5 ← R1 OR R4

With the above renaming, instructions 2 and 3 can be in the pipeline before (or more likely, at the same time as) instructions 1 and 4 without as much slowdown and with the write-after-write hazard avoided. Because even a moderately clever optimizing compiler could resolve similar dependencies at compile time, it appears that register renaming is only useful in instances when the compiler must work with a

set of registers that is smaller than the number of registers available on the hardware itself, as for compilers designed to work for older hardware. More importantly, it is triggered by reordering performed at runtime; if any other optimization has caused a reordering to occur, it can create a WAW hazard that can be resolved by register renaming.
Register renaming is useful for antidependencies and for output dependence, where there is no true data dependence.

Resolving Control Hazards

In pipelining, branch prediction is used to keep the odds of selecting the correct branch high, which prevents stalls and limits incorrect guesses. Limited branch prediction can be attained in the compiler using certain simple heuristics detailed in Ball and Larus' "Branch Prediction For Free" [BL93]. For two-way branches with static destinations (that is, destinations known at compile time rather than determined at runtime), it is possible to come up with useful predictions for each branch that remain constant throughout execution. How useful this is depends on the branch. If a branch in execution tends to go one direction much more often than the other direction, then a perfect static predictor (one that always guessed correctly which was the more frequently-chosen destination) would miss only infrequently; it would only miss in those rare instances where the less-chosen path was taken. If a branch took both paths with relatively even probability, then even a perfect static predictor could miss up to half the time.

Ball and Larus found that their heuristics could predict loop branches with a mean miss rate of 12%(±10%) (3) and could predict non-loop branches with an average miss rate of 26%. This is substantially better than random guessing (which in theory misses 50% of the time), and, importantly, can be completed outside of hardware. Their calculations are based on SPEC benchmarks (as well as several others including compression, an interpreter, and a compiler), and their tracing tool examines every procedure in each benchmarking program [BL93].
Branches are by their very nature dependent on information that only exists at runtime, meaning that hardware is able to detect information about which way a branch might go that would not be available to a compiler. Thus some methods of branch prediction involve recording information about the tendencies of branches over time, making predictions easier once patterns have been established.

A significant part of modern branch prediction is the two-bit branch prediction counter. The two-bit counter involves a branch-prediction buffer, keeping two bits of memory per branch (indexed by part of the branch instruction). These bits indicate a value from 0 to 3. If they indicate a 0 or 1, the branch is predicted not-taken; if they indicate a 2 or 3, it is predicted taken. The counter is incremented (up to 3) on a taken branch and decremented (down to 0) on a not-taken branch.

Suppose we have a branch that is almost always taken. The first time it is taken, its value is set at 2. The second time it is taken, it increases to 3. Thereafter the number remains at 3 regardless of how many taken branches it correctly predicts. However, in the event of a not-taken branch, the hardware will guess incorrectly and reduce its value to 2. If it again mispredicts, it will reduce its value to 1 and begin predicting not-taken instead of taken. This hardware exists in contrast to the one-bit counter. The one-bit counter instead records only whether the branch was taken (1) or not taken (0) the last time it was run. The reason a two-bit counter offers higher performance is that for a branch that almost always predicts either taken or not taken, a one-bit counter will usually mispredict twice if it mispredicts at all (once when changes its value to match the unusual branch, and again to return to the usual branch). In contrast, an occasional exception only results in one misprediction for the two-bit counter. This is one of the simplest forms of branch prediction at the hardware level. [HP07] suggest that, using the SPEC89 benchmarks, a 4k entry buffer (small in 2005) could expect between 82% and 99% accuracy, with accuracy increasing for larger buffers (82).

Costs of Optimization – Hardware-level

Hardware-level optimizations, obviously, require extra hardware, which actually ends up being quite costly in terms of space. The additional hardware required for all of these optimizations is not insignificant; looking at chip diagrams from processors over a range of several implementations, we can see that the FPU – the portion of the chip that controls actual floating-point operations – is getting smaller in proportion to the rest of the chip, eventually taking less chip area than any one of the branch target buffer, register renaming, or out of order logic pieces of the chip, with this last being at least triple or quadruble the size of the FPU. [HH07] (451). This means that, with a chip heavily optimized at the hardware level, a huge amount of energy is being spent at runtime to ensure the pipeline is running efficiently – presumably far more, in fact, than is being spent on the actual calculations. This type of architecture, then, is effectively optimized for speed at the expense of more or less every other technical aspect, consuming much more energy and space than is actually required for any individual instruction in order to maximize throughput. That most sources are willing to provide specific information about speedup while giving little to no explanation of heat costs suggests that speedup is prioritized. This suggests that a chip that was optimized to maintain performance as much as possible but with an emphasis on heat reduction – with the reduced energy and cooling costs associated with low heat – could be a valuable research topic.

Controlling Energy Costs

Dating from as far back as 1999 [HS99], concerns have been raised about energy costs in pipelining and its optimizations. One approach to managing those costs is to reduce the optimizations used in pipelining or to attempt to shift them out of hardware and into the compiler. To this end, I will turn to a description of preliminary work on Wonnacott's "Duck Pond" project, which rethinks ISA from the ground up for the purposes of optimizing for energy efficiency, rather than speed (as is done in traditional architectures).

The Duck Pond Architecture

We will now move into a discussion about another design for future investigation tentatively called the Duck Pond Architecture. This architecture is in a way similar to both our initial naïve architecture, where each instruction executes in a single cycle, and to the pipelined architecture, where the chip is able to exploit parallelism by running multiple operations simultaneously in different parts of the chip. This is apparently contradictory, because it suggests both that each one-cycle instruction completes before the next instruction is issued and that computation from two instructions can be performed simultaneously. Both statements are correct. Duck Pond reframes the meaning of "completing" an instruction so that an instruction is completed not when an answer is returned, but when it has begun processing on the chip and data from the instruction is no longer needed. This architecture manages more time-consuming instructions, such as additions, by treating each instruction as though it is simply triggering the operation, rather than executing it in its entirety. For example, an "addition" would begin the "addition" operation, and would expect the program to not try to retrieve the results of that operation until after it was completed. This section will explore some of the differences between the Duck Pond architecture and the conventional pipelined architecture.

One of the primary features of the Duck Pond architecture is that it attempts to have as little hardware-level optimization as possible, which makes it simpler, lower-power, and smaller, while at the same time sacrificing some of the gains available with such optimization. Because pipelining introduces a lot

of complexities not present in the Duck Pond architecture, some optimizations are unnecessary; others, we believe, could be approximated or made up for through optimizations made by the compiler. Even in the event that Duck Pond is in some instances not comparable in terms of speed to a pipelined architecture (which is probable), we believe its smaller size, which could mean two (or perhaps more) cores could fit on a chip the size of a single-core pipelined chip, could make up for that loss, or at least provide a cooler, lower-power unit.

Challenges to Correctness versus Pipelining

Because of the parallelism exploited by Duck Pond, which is similar to that of pipelining, Duck Pond is still subject to hazards. This section will parallel the section on pipelining hazards, indicating the ways in which Duck Pond can deal with particular hazards. Again, the lack of hardware-level optimization severely limits the ways in which Duck Pond can avoid hazards, so solutions to hazards (other than stalling) must be creative, but will in many ways be similar to the compiler-level optimizations for pipelining.

Structural Hazards

Structural hazards will not be significantly easier to resolve in Duck Pond than in a pipelined architecture; once again, reordering and including redundant hardware are the best ways to ensure that two operations do not need to have access to hardware at the same time. Duck Pond's instruction set, upon completion, will either include specific "wait-until-instruction-occupying-this-part-of-the-chip-is-completed" versions of instructions or have this functionality inherent in all instructions. This will indicate to the hardware that it should not issue an instruction that needs a particular piece of hardware – the adder, for instance – until instructions currently using that hardware have finished executing. The end result is an implicit or explicit stall without any hardware-level reordering.

Data Hazards

Data hazards, arising from dependencies of data values upon one another, behave similarly in Duck Pond as in pipelined architecture. However, because there are fewer hardware-level optimizations, some data hazards do not occur, or occur less frequently.

One of the primary changes between pipelining and Duck Pond is the absence of dynamic scheduling; the hardware will not reorder instructions at runtime, implying a performance loss due to stalls when, for example, an unexpected load takes place. However, this also means that certain hazards cannot occur. An example offered earlier – of the hardware reordering instructions to avoid one type of data hazard and in the process creating another hazard, which in turn it solved with register renaming – is also an example of one of the hazards that could not occur in Duck Pond. Because reordering could not occur at the hardware level, it could not generate a problem at hardware level that needed to be resolved with register renaming.

Control Hazards

Again, a control hazard arises when an instruction, often a branch, needs to change the program counter, thereby requiring the program to either halt until it is known which way the branch will go, or to predict and be able to self-correct in the event of a false prediction.

If the Duck Pond architecture chooses to do without branch prediction, then it must necessarily stall whenever a branch is predicted until it is able to tell for certain whether to branch or not. It is

possible that this might be done earlier than one might expect, however. If each branch is performed based on a value from an addition being zero, then as that value is stabilizing in the ALU, bits would be determined to be 0 or non-zero from right to left. As soon as one could be determined to be non-zero, the branch would be known to not be taken. Research into this as a possibility is continuing.

Challenges to Performance versus Pipelining

Without an abundance of available data on current chips or a working Duck Pond simulation to benchmark, our ability to compare the two structures is limited to a hypothetical analysis based on the differences between the designs. The most notable difference is that the Duck Pond architecture will attempt to do away with most of the hardware-level design implementations, making it significantly lower-energy but potentially lower-speed as well. What we believe is that in some instances Duck Pond will run as quickly as pipelined architecture, in a few instances it will more slowly, and that in all instances it will run at significantly reduced heat and energy costs. Instances where the hardware will perform well are those where its architecture actually improves throughput; for a long chain of additions to the same variable, we believe our hardware could perform as or more quickly than a standard architecture by continuously feeding values into the same adder, where they would be immediately available upon completion without a register writeback phase. Instances where the hardware will perform less well are those where it would lose without the kinds of hardware-level optimization that makes pipelining efficient; for example, a program that required a lot of branches with difficult-to-predict results could cause a lot of stalls in our architecture that a pipelines architecture might be more able to resolve. Although our current understanding of Duck Pond's potential does not suggest that it would necessarily be able to match or outpace a pipelined architecture on average, we believe that its other, lower costs may have applications in certain specialized systems.

Conclusion and Further Research

Pipelining, though an extraordinarily effective means of boosting throughput, introduces a substantial amount of complexity to the hardware, especially with regards to maximizing performance while retaining correctness. The trend for computers thus far appears to have been maximizing for speed against all other constraints, for which pipelining is the best available option. However, we believe that for situations where other constraints, particularly energy costs, are more pressing, an architecture that reduced hardware-level optimization and increased compiler-level optimization might be a viable alternative. Pursuing this research would involve investigating more specifically the costs of existing microprocessors (where available) and constructing and simulating a working Duck Pond architecture and instruction set. If it seems that the optimizations of pipelining can be scaled back or obviated through the use of clever compiler design and unconventional instruction sets, it may be possible to create an architecture that, in certain circumstances, is a viable alternative to pipelining.

References

[BHD09] J. Balfour, R. C. Harting, and W. J. Dally. "Operand Registers and Explicit Operand Forwarding." *IEEE Computer Architecture Letters*, Vol. 8, No. 2, July-December 2009.

[BL93] T. Ball and J. R. Larus. "Branch Prediction For Free." In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 1993.

[HH07] D. M. Harris and S. L. Harris. *Digital Design and Computer Architecture,* 2007.

[HarP02] A. Hartstein and T. R. Puzak. “The Optimum Pipeline Depth for a Microprocessor.” In *Proceedings of the 29th annual International Symposium on Computer Architecture,* 2002.

[HP07] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Fourth Edition,* 2007.

[HS99] S. Hily and A. Seznec. “Out-of-Order Execution May Not Be Cost-Effective on Architectures Containing Simultaneous Multithreading.” In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.