Chapter 4 Literature Review

[4.3.4] Delayed Control Transfers
[4.3.5] Memory Accesses: Delayed Loads & Stores
[4.3.6] Data Dependencies & Hazards
[4.3.7] Controlling Instruction Pipelines

Justin Clark

Introduction

In chapter 4 of [1], the overarching discussion is about increasing the performance of the CPU. The primary method of increasing CPU performance is through instruction pipelining. However, some issues can arise when it comes to pipelining and various strategies are required to handle the different types of resulting issues. How can we maximize the throughput of a pipelined processor while accounting for these issues? Should we deal with pipelining strategies at the hardware or software level? When is the most popular strategy not necessarily the optimal strategy? These are a couple of the questions that will be answered in subsequent sections.

The primary topics of consideration are as follows: In the first section, we will discuss *delay control transfers*, which are (simply put) a clever way of getting the most out of a given clock cycle [1§4.3.4]. In the following section, we will cover the the topic of memory access in pipelined architectures (specifically delayed loads and stores) [1§4.3.5]. Next, the discussion will move to the different types of data hazards that may arise from pipelined architectures [1§4.3.6]. Finally, we will cover some strategies for dealing with potential data hazards [1§4.3.7]. The discussions will briefly overview the topics of the corresponding sections of [1] and then move onto some related literature to extend the topics.

Delay Control Transfers

In section 4.3.4 of [1], the discussion is an extension of the idea of branch instructions. Branch instructions are typically executed immediately when a branch condition is true or if the branch is unconditional, but what about the instruction sequentially following the branch? This instruction is always fetched whether or not the branch succeeds or fails, but if the branch succeeds, this instruction is flushed from the pipeline (in processor with vanilla pipelined branching). This is poor use of a clock cycle and where *delayed branches* shine. The concept of delayed branching is for the processor to execute the instruction regardless of whether the branch succeeds or fails, which allows more productive use of the clock cycle. The practical implementation of this feature uses something called a *delay slot*. A delay slot is where the sequential instruction(s) following a branch is (are) stored. In delayed branching, it is necessary for the compiler to find instructions following branches that do not depend on whether the loop succeeds or fails. Thus, delayed branching is one of the strategies where it makes more sense to be implemented at the software level.

Another thing discussed in [1] is how determining which instruction(s) should be put in the delay slot(s) can be a tricky problem. To make things even more complicated, sometimes it is necessary to translate a program with delayed branches from one architecture to another. This is the focus of the authors of [2]. Their objective is to be able to analyze the effect of delayed branches and also propose a method for translating compiled programs with delayed branches from a source machine to a destination machine. As mentioned in the previous paragraph and in [1], the problem of delayed branching is taken care of by the compiler. The discussion starts with the importance of binary translation (translating compiled code on a source machine to compiled code on a destination machine). Then the authors propose a basic strategy of binary translation, which is as follows: make the translated program rely on only the program counter (*PC*) as much as possible while eliminating the next program counter (*nPC*) and the annul status (*annul*) as much as possible. *Annul* specifies whether the instruction at PC should be executed. The reason the authors of [2] propose eliminating *nPC* and *annul* as much as possible is because more software overhead is required to update *nPC* and *annul* on the target machine. The authors also provide pseudo code for implementing such a translation.

As a counter-argument to delayed branching with delayed slots, in [3], the authors discuss a strategy to overcome the software overhead of managing delay slots called *Prophetic Branches*. The basic idea of prophetic branches is as follows (directly from [3]): "*Likely branches, in the prophetic branch mechanism, allow the control transfer to take effect* *before the evaluation of the condition code; i.e. the branch is specified and predicted before the condition that controls its outcome.*" Figure 1 below (Figure 7 of [3]) compares a few different branching techniques in regards to number of instructions and number of cycles.

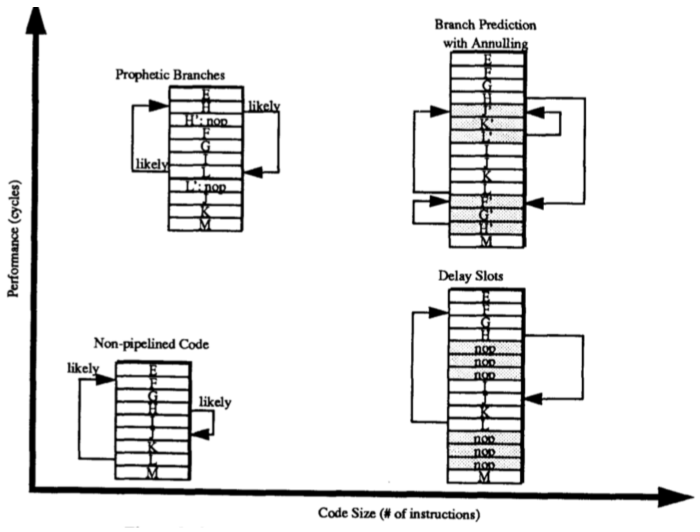


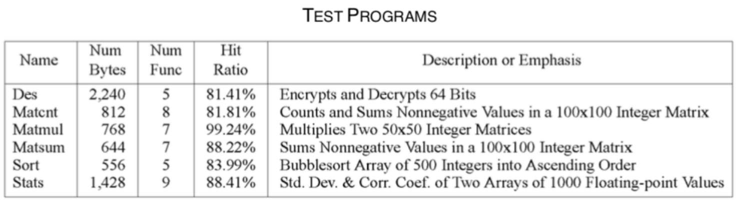
Figure 1: Quantitative comparison of different computing methods.

Memory Accesses: Delayed Loads & Stores

Section 4.3.5 of [1] addresses the following question: what happens when there is an instruction that is dependent on data fetched by the previous operation, but the data takes one or more clock cycles to fetch? In the most basic hardware implementation, this would cause the second instruction to operate on garbage data (that is, data that was previously in the register in question). This problem is solved by delayed memory accesses (particularly LOADs and STOREs). Since memory access can be an extremely slow process, even if the data is in cache, it is essential to handle this in some way. The simple way to solve this at a hardware level is by freezing the instructions that depend on the memory access until the data has been fetched. However, this isn't necessarily the best solution for all cases. Instead, this problem can be handled at the software level by making developers aware of the behavior of the hardware in these cases for them to handle on their own.

The question remains: how much time do memory accesses take and how does that affect the overall program performance? The authors of [4] propose a quantitative method for determining best and worst case bounds for program performance on computers that implement instruction caching and pipelining (program performance is defined as the number of cycles). There are four steps to determining the bounds of program performance: 1) a static cache simulator categorizes caching behavior of each instruction, 2) a timing analyzer will use these categorizations to determine performance of a path (sequence) of instructions, 3) combine these paths of instructions to create a single tree of instruction paths, and 4) use the tree to estimate the overall performance of the program. More details on methodology and pseudocode are provided for these steps in [4], but is omitted for brevity. Results of of their analysis and test programs used are in the following two tables in [4]:

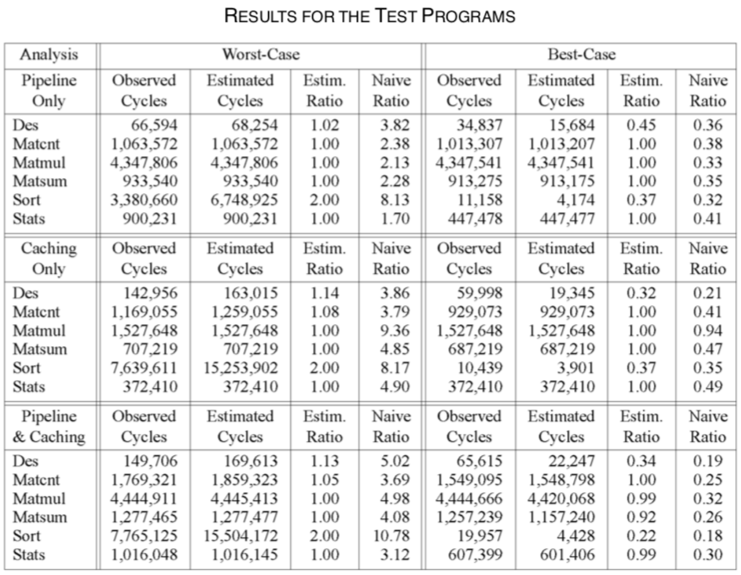
Table 1



TEST PROGRAMS

Name	Num Bytes	Num Func	Hit Ratio	Description or Emphasis
Des	2,240	5	81.41%	Encrypts and Decrypts 64 Bits
Matcnt	812	8	81.81%	Counts and Sums Nonnegative Values in a 100x100 Integer Matrix
Matmul	768	7	99.24%	Multiplies Two 50x50 Integer Matrices
Matsum	644	7	88.22%	Sums Nonnegative Values in a 100x100 Integer Matrix
Sort	556	5	83.99%	Bubblesort Array of 500 Integers into Ascending Order
Stats	1,428	9	88.41%	Std. Dev. & Corr. Coef. of Two Arrays of 1000 Floating-point Values

Table 2



RESULTS FOR THE TEST PROGRAMS

Analysis	Worst-Case				Best-Case			
Pipeline Only	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	66,594	68,254	1.02	3.82	34,837	15,684	0.45	0.36
Matcnt	1,063,572	1,063,572	1.00	2.38	1,013,307	1,013,207	1.00	0.38
Matmul	4,347,806	4,347,806	1.00	2.13	4,347,541	4,347,541	1.00	0.33
Matsum	933,540	933,540	1.00	2.28	913,275	913,175	1.00	0.35
Sort	3,380,660	6,748,925	2.00	8.13	11,158	4,174	0.37	0.32
Stats	900,231	900,231	1.00	1.70	447,478	447,477	1.00	0.41
Caching Only	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	142,956	163,015	1.14	3.86	59,998	19,345	0.32	0.21
Matcnt	1,169,055	1,259,055	1.08	3.79	929,073	929,073	1.00	0.41
Matmul	1,527,648	1,527,648	1.00	9.36	1,527,648	1,527,648	1.00	0.94
Matsum	707,219	707,219	1.00	4.85	687,219	687,219	1.00	0.47
Sort	7,639,611	15,253,902	2.00	8.17	10,439	3,901	0.37	0.35
Stats	372,410	372,410	1.00	4.90	372,410	372,410	1.00	0.49
Pipeline & Caching	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio	Observed Cycles	Estimated Cycles	Estim. Ratio	Naive Ratio
Des	149,706	169,613	1.13	5.02	65,615	22,247	0.34	0.19
Matcnt	1,769,321	1,859,323	1.05	3.69	1,549,095	1,548,798	1.00	0.25
Matmul	4,444,911	4,445,413	1.00	4.98	4,444,666	4,420,068	0.99	0.32
Matsum	1,277,465	1,277,477	1.00	4.08	1,257,239	1,157,240	0.92	0.26
Sort	7,765,125	15,504,172	2.00	10.78	19,957	4,428	0.22	0.18
Stats	1,016,048	1,016,145	1.00	3.12	607,399	601,406	0.99	0.30

The "Estimated Cycles" and "Observed Cycles" columns in Table 2 are compared with a metric called "Estimated Ratio". If the upper and lower bound estimates are to hold, the

"Worst-Case" table should have an $EstimatedRatio \geq 1$ and the "Best-Case" table should have $EstimatedRatio \leq 1$, which does, in fact, hold up for their test cases.

Data Dependencies & Hazards

In the previous section, we discussed what happens if memory accesses take too long. Section 4.3.6 of [1] is an extension of memory access issues. Now that we can assume the correct memory is fetched in the right amount of *time*, we have to figure out what happens when memory is accessed in the wrong *order*. In [1], the following three-instruction example is given:

```
I1: ADD     R1, R2, R3
I2: SUB     R3, R4, R6
I3: XOR     R1, R5, R3
```

When I_2 uses `R3` as an operand, we need to make sure it is using the stored value from I_1. If unhandled, this can lead to a *read-after-write* (RAW) data hazard. When I_1 and I_3 both write to `R3`, we need to make sure these are written in the correct order. Not doing so could result in a *write-after-write* (WAW) data hazard. Finally, we must ensure that I_3 writes to `R3` only after I_2 has read from `R3`. If out of order, the resulting hazard is called a *write-after-read* (WAR). The most common data dependency is the RAW hazard and is the only possible hazard on single pipelined machines. However, on processors with multiple pipelines, the other hazards must be dealt with.

In the example above, it is not necessarily obvious which data hazards could be present. A valuable piece of knowledge to have about a program is all the *possible* data hazards that may arise between instructions. In [5], the author describes a simple method for systematically enumerating all possible data hazards. The basic methodology is as follows: try all possible pairs of instructions and all possible execution sequences. Well - this seems quite straightforward. Why is it useful? The author points out that a systematic method like this can make it easier on students to learn the concepts of data hazards and gain an intuition of where they may arise. The author of [5] also mentions that there are many automated methods that bypass doing this by hand. However, the author makes the point that it is detrimental to learning if the process is automated outright.

Controlling Instruction Pipelines

In section 4.3.7 of [1], the discussion continues from section 4.3.6. How should one handle the possible data hazards mentioned in the previous section (RAW, WAW, and WAR)? There are a few common approaches. The first is called *data forwarding*. This is where the processor forwards operands to the ALU to be used by next instruction while simultaneously being sent to the destination register. Data forwarding requires more hardware, but is an effective hardware solution. The next approach is called the *scoreboard method*. The scoreboard method is a mixed hardware and software solution that stores data about instructions in order to effectively manage their operation. A few examples of information stored in the scoreboard tables are whether or not the instruction has been issued, whether or not operands have been read, and whether or not the result has been written. The third popular strategy discussed in [1] is *Tomasulo's method*. This is really just an extension of the scoreboard method but has a few distinctions. The most obvious difference is that Tomasulo's method manages instructions in a distributed fashion in an attempt to handle higher concurrency of operations. Additionally, this strategy of managing instructions requires the use of a shared bus. This can cause a serious bottleneck, but Tomasulo's method is still effective enough to be used as a baseline in terms of pipeline management.

To extend the previous paragraph, the authors of [6] provide a proof of correctness of Tomasulo's method for out-of-order execution. The authors base their proof on a system used in [8], which uses the PVS system to determine the correctness of Tomasulo's method in general. The PVS system is an automatic theorem prover and is also a research prototype[1]. The authors of both [6] and [8] show a step-by-step method for arriving at the proof of correctness for Tomasulo's method in general and specifically for out-of-order execution.

For a less mainstream solution to managing potential data hazards, we might be interested in [7] which discusses a solution for dealing with data hazards in time-sensitive financial data. Much of [7] is beyond the scope of this paper, but the basic idea of the authors is to use a specific architecture that can deal with financial data protocols with lower latency. They propose a bus-based architecture to reduce the complexity of the design (and thus lower the potential for possible data hazards). One major point here is that this is a more hardware-heavy approach than the scoreboard method and the Tomasulo's method. At the same time, parallels can be drawn between the method in [7] and Tomasulo's method in that they are both bus-dependent strategies.

Conclusion

The preceding four sections have discussed major topics in simple pipelined architectures. Specifically, we discussed various software and hardware approaches to solving issues related to pipelining and when one solution might outperform another and vice versa. We also introduced methods for analyzing the performance of certain architectures; e.g. upper and lower bounds on performance of a pipelined program while accounting for slow memory access. The major takeaway here is that it is necessary for programmers and computer engineers to weigh the tradeoffs between different pipelining strategies. Sometimes it makes sense for solutions to be implemented in hardware and sometimes it makes more sense for strategies to be implemented in software. Further, not all hardware solutions are made equal, likewise for software solutions. The implemented strategy should depend heavily on the use case of the system in question.

[1] https://github.com/SRI-CSL/PVS

References

[1] Dumas, Joseph D. II. Computer Architecture: Fundamentals and Principles of Computer Design, Second Edition, CRC/Taylor & Francis, 2017. ISBN 978-1-4987-7271-6.

[2] Ramsey, N., & Cifuentes, C. (2003). A transformational approach to binary translation of delayed branches. ACM Transactions on Programming Languages and Systems (TOPLAS), 25(2), 210–224. doi:10.1145/641888.641890

[3] Srivastava, A., & Despain, A. (1993). Prophetic branches: a branch architecture for code compaction and efficient execution. Microarchitecture, 1993., Proceedings of the 26th Annual International Symposium on (pp. 94–99). USA: IEEE Comput. Soc. Press. doi:10.1109/MICRO.1993.282745

[4] Healy, C., Arnold, R., Mueller, F., Whalley, D., & Harmon, M. (1999). Bounding pipeline and instruction cache performance. Computers, IEEE Transactions on, 48(1), 53–70. doi:10.1109/12.743411

[5] Mahran, A. (2012). A handy systematic method for data hazards detection in an instruction set of a pipelined microprocessor.

[6] Arons, T., & Pnueli, A. (1999). Verifying Tomasulo's algorithm by refinement. VLSI Design, 1999. Proceedings. Twelfth International Conference On (pp. 306–309). IEEE Publishing. doi:10.1109/ICVD.1999.745165

[7] Tang, Q., Jiang, L., Su, M., & Dai, Q. (2016). A pipelined market data processing architecture to overcome financial data dependency. Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International (pp. 1–8). IEEE. doi:10.1109/PCCC.2016.7820632

[8] T. Arons and A. Pnueli. Verifying Tomasulo's algorithm by refinement. Technical report, Dept. of Comp. Sci., Weizmann Institute, Oct 1998.