CSC 565 - Computer System Architecture: Paper Summary #2

Due on February 15, 2023 @ 18:00, EST

Decisive Aspects in the Evolution of Microprocessors 2

Cason Konzer

February 15, 2023

Summary

In this paper summary we will discuss the work of [1] from section **IV.B** to section **V.B**. A summary of the previous sections is provided in the first summary (HW1) of this series and a condensed summary of the work in total is provided by the journal's prelog [2].

The paper is acronym (and variable) heavy, so let us first set the nomenclature which has not been previously defined in HW1.

General:

- CTIs: Control Transfer Instructions.
- BTA: Branch Target Address.
- CISC: Complex Instruction Set Computer.

....

Technical:

- n_{ir}: Issue Rate of the Processor.
  - ∗ *Maximum # of Instructions Issued in a Clock Cycle.*

With the terminology in place, let us dive into the sections.

IV.B: *Introduction and Evolution of Utilizing Temporal Parallelism : Implications of the Introduction of Pipelined Instruction Processing*

In this subsection on the resulting implications of the implementation of temporal parallelism, the work provides first an overview followed by technical deep dives. Resulting in the implementations, the memory subsystem required advancements (via caches), as well as the processing of CTIs (via branch prediction) [1].

The development of caches was driven with respect to demands for higher bandwidth from the memory subsystem. When processors break the *sequential* processing barrier and being processing with IPC near 1, the memory system needs to be enhanced in order to provide instructions and their data, as well as execute the instructions and their respective writes, in a similar frequency. The main issue with memory subsystems is that they operate slower than the processor, and thus caches work by implementing faster access to certain memory. One way to look at memory caches is to view them as an additional subsystem within the traditional memory subsystem. The key idea behind caches is that they store the memory which must be frequently accessed in an area which supports faster speeds. As a result, the frequented memory is less burdened by the slower traditional memory. For a more detailed view on caches we defer the reader to [3, Section 6.3] as Sima's focus is only to provide the insight into the driving factors of their development. What is mentioned by [1] is that processor performance has been growing faster than that of memory performance, and thus advancements in memory performance is an area of research requiring continued attention. In summary, the first bottleneck encountered outside of the processor is the memory.

Within the processor there are additional bottlenecks form serving useable instructions. The processor bottleneck is that of CTIs, which must be handled intelligently in order to minimize the number of flushed, or otherwise irrelevant, instructions served. CTIs come in two general flavors: unconditional and conditional branches. In both cases, a branch indicates a transition to new instructions within the program, specified by the BTA. Unconditional branches are always taken, while conditional branches are taken only if a specified condition is met. It is not uncommon to see nested branches within programs, and thus the proper handling of branches becomes even more critical due to branch dependencies.

As it may not be directly clear how branches cause bottlenecks, let us first take a more detailed view of the simpler unconditional branch. Within an unconditional branch, the processor traditionally will not know the BTA until the CTI reaches the end of the execute stage. When in the decode stage, and similarly the execute stage, the processor will load the next instructions following the CTI into the pipeline. Once the BTA is calculated, the processor will need to flush the previously loaded instructions and fetch the instruction located at the BTA, granted the BTA is not addressed to the instructions following the CTI. Within this framework, every unconditional branch will waste 2 clock cycles fetching and decoding instructions destine for a flush. Initially three concepts were introduced to help minimize this bottleneck, *adding dedicated hardware, redesigning the processor & delayed branching.* The most general of the implementations is *delayed branching*, which consists of inserting instructions into the previously 2 flushed slots, or bubbles, which are to be executed regardless of the branch. Less straightforward is the determination of which instructions satisfy such a constraint, although this is left to the compiler or programmer [1].

The other two concepts require additional knowledge of conditional branches, and thus we now give their overview. In the most basic implementation, a conditional branch will undergo 2 execution stages, the first to check the condition, and the second to calculate the BTA. Given the condition is false (e.g. do not branch), none of the instructions fetched after the CTI need be flushed, and the processor can continue normal operations. If though the condition is true, the processor need additionally calculate the BTA, causing a total of 3 bubbles in the pipeline. In a similar manner, using *adding dedicated hardware*, one can conceive implementations in which the BTA can be calculated, and the condition can be checked, during the decode stage. But, in order to check a condition, the condition itself must be known. As it may be the case the condition is not known until completion of the decode stage, attempts at *redesigning the processor* have been made in order to make the condition known earlier. When *redesigning the processor*, engineers have implemented a half cycle decode stage to make the condition known, and utilized the first half of the execute stage to preform condition checking. A downside to this implementation is that now the condition need be evaluated in only half of the execution cycle, making the implementation infeasible in highly clocked processors [1]. In any case, *delayed branching* can be still used to fill these bubbles, or delay slots, with useful instructions.

The remaining content in this subsection provides a brief overview of branch prediction, and emphasizes that it helps to avoid the bottlenecks of unknown conditions and restricted execution times. First introduced in CISC processors, branch prediction is a technique in which the processor uses a heuristic to decide whether or not to take conditional branches early on in the pipeline, and then cross checks the assumption once the condition becomes known [1]. In the case the heuristic is correct, the processor need flush none of the instruction following the branch, although if upon checking the heuristic the conditions has been falsely predicted, the processor need flush all of the instructions following the CTI and pick back up at the correct instruction address. The simplest of heuristics is *fixed prediction*, in which case the processor decides either always to take, or otherwise not take, the branch.

Some of the nuances of branch prediction are exposed, in such a way to emphasize the importance of intelligently designed heuristics. Specifically, when CTIs are placed in the pipeline following complex instructions, the time to resolve conditional branches increases. That is, in the case of *unresolved conditional branches*, many bubbles will be inserted in the pipeline until the execution of the prior complex instruction is resolved and the branch condition can be evaluated. The advancements in caches and branch prediction bring the evolution of *temporal parallelism* to exhaustion for traditional *pipelined* processors, causing the need to exploit *issue & intrainstruction parallelism* for further gains in microprocessor architecture [1].

V.A: *Introduction and Evolution of Issue Parallelism : Introduction of Issue Parallelism*

Advancements in *issue parallelism* mark the onset of first generation *superscalar processors.* In general *superscalars* are classified as processors which satisfy the inequality IPII > 1. Upon study one realizes that this inequality is infeasible when $n_{ir} \leq 1$, and thus the evolution of *superscalars* leverages increases in n_{ir}. These processors began production around the early 1990s, utilizing the issue of multiple instructions per clock cycle [1].

V.B: *Introduction and Evolution of Issue Parallelism : Overall Implications of Introducing Superscalar Issue*

Once we begin to exploit *issue parallelism*, our pervious exploitations of *temporal parallelism* become increasingly important. Bottlenecks imposed by the memory subsystem, and processing of CTIs, become stressed, or *aggregated*, and require continued attention. Sima proposes some generalizations, assuming fixed f_c, which help to frame the aggravation of the microprocessor when comparing *superscalars* with *pipelined* counterparts:

1. Required Instruction/Memory Fetching and Storing Scales with n_{ir}.
   - ∗ *Memory Bandwidth Scales with* n_{ir}.
2. Branches per Clock Cycle Scales with n_{ir}.
3. Flushed Instructions per Misprediction Scales with n_{ir}.

What continues to hold is that as f_c increases, regardless of architecture, the memory bandwidth is additionally stressed. This point, in conjunction with generalization (1), drove further developments in the memory subsystem through caches.

Generalizations (2) and (3) apply specifically to smart processing of CTIs through *branch prediction.* The evolution prompted by these was focused within three domains:

- ⋄ Devising Better *Branch Predictor* Heuristics.
- ⋄ Producing the BTA earlier within the Pipeline.
- ⋄ Handling *Unresolved Conditional Branches.*

Each of these points requests additional discussion, but in general they are all handled by the fetch subsystem, and discussion is deferred to later sections and cited works [1].

This concludes section **V.B** in the work, and this summary within the series.

References

[1] D. Sima, "Decisive aspects in the evolution of microprocessors," *Proceedings of the IEEE*, vol. 92, no. 12, pp. 1896–1926, 2004.

[2] H. Falk, "Prelog to decisive aspects in the evolution of microprocessors," *Proceedings of the IEEE*, vol. 92, no. 12, pp. 1895–1895, 2004.

[3] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Pearson, 3rd, global ed., 2019.