“Meltdown” in the Eyes of a Hardware Security Engineer

This article is written by Doctor Tbsoft of Antiy Micro–electronics and Embedded Technology R&D Center.

1. Modern Computer Architecture and CPU Microarchitecture

Modern computer architecture is basically based on von Neumann Architecture, i.e. "stored program system". That is, the program instruction code (instructions) and data are placed in memory. The runtime CPU fetches and executes instructions one by one from memory, and the instructions can also access memory data as necessary.

The Harvard Architecture is a variant of the von Neumann Architecture that separates the program (instruction) memory from the data memory with two separate memories, one for storing instructions and the other for storing data, without leaving the basic mode of "stored program system". Therefore, it still belongs to a variant or improvement of the von Neumann Architecture.

For programmers, even for assembly language, machine language programmers can only deal with memory instruction and data at most. What computers expose to programmers can only be von Neumann Architecture or Harvard Architecture, which is collectively called “Computer Architecture”.

However, the CPU itself is a hardware circuit composed of arithmetic units, controllers, registers and other components. After the memory instruction is taken into the CPU, complicated processes such as instruction decoding are also performed in the CPU to drive the corresponding components of the CPU hardware circuit, conduct corresponding actions in accordance with specific needs of the instruction, and finally complete the instruction execution. This process is completed by the CPU internal fetch instruction, instruction decoding, hardware drivers and other hardware components. There are great differences between different CPUs, and even for CPUs with the same instruction set, the circuit implementation methods may also be different. The structure of CPU internal hardware circuit is known as “CPU Microarchitecture”.

CPU microarchitecture is the core of the design of the CPU. For programmers, CPU microarchitecture is not exposed to programmers, the programmer at the bottom can only access to the CPU by CPU instruction. In other words, the minimum operational granularity for a programmer to use a CPU is an instruction. However, the CPU instruction may be parsed inside the CPU as a smaller execution unit, eventually driving a single hardware circuit component of the CPU. That is, there is smaller operational granularity in CPU microarchitecture, such as MicroOP, μOP or microinstruction, and they are all determined by the CPU microarchitecture.

In other words, the CPU is also a black box for programmers. A programmer only needs to write programs according to the CPU instruction set (if it is a high–level language, it will be eventually converted into CPU instructions by compiler or interpreter), and load the instruction program and the corresponding data into the memory (which is usually done by the operating system) to execute the program. The programmer only access to the computer architecture and how the instructions are further resolved and finally implemented within the CPU is determined by the CPU microarchitecture. Programmers do not, usually should not, and actually cannot care about whether it is open to them.

2. How to Improve CPU Speed

If CPU executes an instruction, in general, it will go through at least three steps, i.e. the instruction is fetched from memory, the instruction is decoded into micro–operation (μOP), and the hardware circuit components are ultimately driven by micro–operation (referred to as instruction fetch, decoding and execution). The next instruction cannot be executed until all three steps are completed. Thus, if it takes long for an instruction execution (in CPU hardware terms, it means “consuming multiple clock cycles”), CPU speed cannot be increased effectively.

When the first instruction is decoded, the second instruction is fetched at the same time. Then when the first instruction is executed, the second instruction is decoded, and the third instruction is fetched at the same time... Thus, instruction fetch, decoding and execution can be overlapped, just like the workers on the production line. After the first worker processes the first process for the first product, the product is passed on to the second worker for processing the second process, and meanwhile the first worker can process the first process for the second product... which will avoid workers’ unnecessary idle wait and the production efficiency will be greatly improved. This method of increasing CPU speed is also known as the "pipeline". Modern CPUs are using pipelining technique.

The pipeline effectively reduces the average execution time of a single instruction, but the instructions are still executed one by one. In fact, many instructions are irrelevant to each other. For example, two instructions for accessing two completely different registers can be executed in parallel. The order of execution does not affect the final execution result. For such irrelevant instructions, it can design multiple execution component circuits and place them directly into different executive components for parallel execution, which is called out–of–order execution, and can further improve the speed of instruction execution. Out–of–order execution is widely used in modern CPUs, and even for instructions that have dependencies, registers can be renamed to make them irrelevant instructions for out–of–order execution.

However, there will be a problem in both the pipeline and out–of–order execution. In principle, they only apply to purely sequential execution instructions. In the event of a branch, i.e. the conditional jump instruction, where the procedure turns will be unknown because it does not execute the conditional jump instruction itself. That is, the next instruction after the conditional branch instruction is uncertain until it is executed, so it cannot obtain subsequent instructions for the conditional branch instruction, and the pipeline and out–of–order execution will be invalid because their premise is to obtain subsequent instructions in advance.

In order to solve this problem as far as possible, modern CPUs widely use the "branch prediction" means that is to predict which branch the conditional jump instruction will jump to, and then prefetch subsequent instructions for this branch. The common strategy of branch prediction is that if a condition jumps to a fixed branch within a certain period of time, it can be predicted that this conditional jump instruction is likely to go to this branch next time.

A typical example: the loop structure in the program is generally cycled for many times before the end. Thus, before the loop ends, the conditional jump instructions that determine the loop condition obviously move to branch which continues the loop.

Branch prediction cooperated with pipelining and out–of–order execution, can greatly improve the CPU speed, so modern CPU microarchitectures basically use this design.

3. Problems that the Branch Prediction Brings – "Rollback" of Instruction Execution

The branch prediction does not guarantee a 100% success. Once the prediction fails, that is, if when the conditional jump instruction is finally executed, it is found that the jump target is not the branch direction previously predicted, the subsequent instructions prefetched according to the branch prediction will actually fail. The work done on these instructions must be "undone", otherwise the instruction will be wrongly executed.

With a programmer–friendly metaphor: Subsequent instruction execution of branch prediction is like a "transaction". If the branch prediction is correct, then the "transaction" can be "submitted" and the subsequent instructions actually work. If when the conditional jump instruction is finally executed, it is found that the jump target is not the branch direction previously predicted, the "transaction" must be "rolled back". Even if the subsequent instructions have been executed, or even out of order, all the work that has been completed must also be "undone". The subsequent instructions should appear to have no effects, and be redirected to the correct branch and the new instruction will be fetched and executed.

Theoretically, no matter whether the instruction execution is "submitted" or "rolled back", it is only related to

the CPU microarchitecture, which the process programmers should not see. The programmer can only see macro instructions executed in accordance with the program flow. What is inside the CPU internal program should still be a black box to programmers.

4. To Overcome the Difference Between CPU Speed and Memory Access Speed – Cache

Currently, the main frequency of CPU has reached 3GHz (or more), and multi–core parallel is generally used. Although the main frequency of the main memory (DDR SDRAM) has reached 2GHz–3GHz or higher, it cannot fully meet the needs of multi–core CPU speed, because the instruction execution must fetch instruction from the memory. If the memory access speed is not enough, CPU speed will be limited by memory access speed.

In order to overcome this problem, a method of inserting a multi–level cache between CPU and the main memory is adopted. Cache is a memory with extremely high access speed and can even be integrated inside the CPU and becomes a part of the CPU microarchitecture. Between the main memory and cache, data is exchanged in blocks, the length of which is generally dozens of bytes.

When the CPU needs to access memory, for example, fetching instructions from memory, it is needed to read the corresponding memory block into a spare Cache block for the first time and then the CPU directly accesses the Cache block. The memory access speed will be slower at this time because there is a time between the main memory and the cache for transferring into the block data; when the CPU accesses the same block of memory for the second time, the block can be directly accessed without accessing the main memory, and the memory access speed will be much faster.

Main memory– Cache system constitutes a modern CPU memory system, the principle of which is very similar to that of the hard disk in the operating system – memory system constitutes a virtual memory.

5. "Rollback" of Instruction Execution in Main Memory – "Traces" Left by Cache System

As described above, if the branch prediction fails, the subsequent instructions that the branch prediction prefetches must be "rolled back" even if a plurality of instructions have been executed out of order, and what have already been completed in the CPU microarchitecture by instructions must all be "undone".

"Undo" instruction is easy. The final task completed by the instruction is nothing less than the modification of the register or the memory. The "modification" can be temporarily "cached". For "undo" instruction, the register or the memory cannot be really modified.

However, for memory read write instructions (commonly referred to as Load/ Store instructions or LD/ ST

instructions in CPU design), taking a memory instruction as an example, if the final "commit" is done, the actual memory address must be read. If the corresponding memory block has not been read into the Cache block, the read speed will be affected, so before the final "commit" or "rollback" of reading the memory instruction, CPU microarchitecture will usually prepare Cache block in advance. That is, if the corresponding memory block has not been read into the Cache block, it shall be read in advance.

That is, even if the branch prediction fails, as long as there are read memory instructions in the multiple prefetched instructions that have been executed out of order, it will be considered as "rollback" in the end, which has an impact on the generalized CPU microarchitecture– the corresponding memory block has already been read into the Cache block.

However, for whether the memory block has been read into the Cache block, there is a certain difference in access speed, which is equivalent to the "trace" that the "rollback" read memory instruction left in CPU microarchitecture. The "trace" can be used as "side-channel"

6. Simple and Concise Explanation for Meltdown Attack Principle

Kernel data of operating system is protected by the CPU microarchitecture, and the user-mode application cannot be accessed if the access is to trigger CPU error exception.

To construct a branch, first detect whether the address for reading the memory is legal. If the address is legal, read the corresponding memory bytes of the address, then according to the value of the memory bytes, make the value of the memory byte corresponding to the memory blocks mapped to different Cache blocks, and then deliberately read the memory blocks mapped to different Cache blocks; if the address for accessing to memory is illegal, such as the kernel data address of operating system, it will not read directly.

Obviously, such a branch, no matter whether it reads legal or illegal addresses, it will not go wrong.

This branch is executed for multiple times in a large loop, and reading the memory address is legal for several earlier times. "Training" branch prediction of CPU makes CPU microarchitecture consider that for the next time it should also move to read this branch of the memory.

Then, it will suddenly execute a kernel data address read for an illegal operating system.

Logically speaking, if the memory address read is illegal, it should move to not read this branch, but the CPU branch prediction has been "trained" as a "conditional reflex". CPU carelessly prefetches instructions to read the illegal address, execute them out of order and parallelly, and only does not "commit". Due to no "commit", even if

it reads the illegal address memory, it will not lead to CPU anomalies; and at this time according to the value of the illegal address memory bytes, intentionally read the block instructions mapped to different Cache memory blocks, and the corresponding Cache blocks have been prepared by CPU microarchitecture (because it is parallel out- of-order execution). That is, the corresponding memory blocks have been read into the Cache block.

Of course, in the end, the CPU will find that the branch prediction is wrong, and it does not matter that the prefetched instructions execute "rollback" out of order and parallelly. The instructions that read the illegal addresses are not executed at all, and the CPU exception is not triggered.

However, "traces" have been quietly left behind, and the memory block corresponding to the illegal address memory bytes has intentionally been read into the Cache block.

Use the program to traverse all possible blocks of memory to see who visits the fastest, who is likely in the Cache, and illegal address corresponding to the operating system kernel data byte value is intentionally corresponding to the memory block. Thus, the kernel data byte value of the operating system that is originally inaccessible in the user mode can be reckoned.

This is called "side channel leakage."

7. One-sentence Explanation of This Attack

It is a leakage of internal information from CPU microarchitecture to macro-computer architecture through side channel.

8. Lesson

It is taken granted that CPUs are designed to pursue speed, but there must be a balance between speed and security. No matter how the microarchitecture is to pursue high-speed optimization, the messy situation should be cleared up, and must not leak internal information to the macro architecture.

Gordon Moore, one of the founders of Intel in 1965 and an electronics engineer at Fairchild Semiconductor, came up with Moore's Law and predicted the rapid progress of human computing. In the past two decades, mankind has been driving the development of information technology under the guidance of the Internet. To some extent, what is overdrawn in the pursuit of speed is security. Perhaps this is exactly the moment to redefine the balance.