Overview of Out of Order Execution (OoOoOE)

Jack LaSota
October 23, 2012
ECE Department, College of Engineering and Mines, University of Alaska Fairbanks

Abstract

One of the technologies used to allow superscalar Instruction Level Parallelism (ILP) is Out-Of-Order-Execution (OoOE). It is also used in single-issue machines with multiple clock-cycle long instructions. This paper explains common scheduling algorithms used in single issue implementations, their comparative advantages and some improvements suggested by other writers, the choice between precise and imprecise interrupts, and the options for dealing with the limit of the instruction window size. Finally, it remarks on the role and usefulness of OoOE in modern computers.

Introduction

One of the challenges of superscalar execution is to decide which instructions can be executed simultaneously. This decision can be made at runtime or compile time. In OoOE it is done in hardware at runtime. An alternative is Very Long Instruction Word (VLIW), where it is done at compile time. This saves greatly on hardware, but can only be done with the cooperation of the compiler, because VLIW gains its benefit from baking all of the control logic for the specific CPU into the instruction. This makes OoOE the more feasible option for superscalar personal computers because their processors must adapt to an existing body of software, rather than having it tailored to them.

OoOE can also be used in single-issue machines with multiple execution hardware to keep Instructions Per Cycle (IPC) close to 1 even with instructions that take more than one clock cycle. The methods of scheduling used in this simpler case are Tomasulo Scheduling, and the Thornton scoreboard approach. Tomasulo Scheduling is more complicated and allows for register renaming, which avoids some false dependencies. The Thornton algorithm is an older alternative that requires less transistors [3]. Both Thornton's algorithm [4] and Tomasulo's algorithm [8] can also be used in superscalar machines.

Tomasulo scheduling has problems because of the latency it creates in when it uses broadcasting on a bus to determine when dependencies have been resolved, which gets worse as more registers are used, and clock cycles increase [10]. An alternative that gets around this problem, which is a fatal flaw in modern processors where the clock speed is fast and the registers many, is the dependency matrix. This stores dependencies between instructions in a grid, so that readiness can be tested by a wide or gate on a row, and completion signaled by clearing a column.

OoOE introduces problems for interrupts because instructions that cause interrupts may execute after instructions that come after them in program order. This problem can be solved by abandoning precision of interrupts, but that causes problems for restarting the program after an interrupt [7]. There are several ways to get precise interrupts on an out-of-order machine, but each has downsides.

Part of the motivation for OoOE is so that the processor can keep making progress in the program even while waiting for slow instructions like cache misses. This is in competition for the same role of getting performance even when running programs with lots of cache misses with Simultaneous Multithreading (SMT). Simulations done in 1999 showed that OoOE and SMT together were no great improvement over SMT alone, as long as there were enough threads to use. However, OoOE is better software with single threads [12].

To have enough instructions to stay busy while waiting for cache misses, the processor must be able to consider possible dependencies across a wide range of instructions. The range of instructions it is considering executing is called the instruction window, and its size limits how far ahead the processor can get from the last incomplete instruction. There are a few ways to deal with this, one is keeping instructions dependent on known slow instructions in a special buffer that cuts down on the logic to check dependencies and can be bigger because of it called a Waiting Instruction Buffer. Another is running ahead with dummy data and restoring to a checkpoint when the slow instruction completes, to at least prefetch whatever memory the next part of the program needs.

Scheduling

The Thornton algorithm works by setting a valid bit for each register, which turns off when there is an active instruction that will write back to that register. Instructions are dispatched (sent to the functional units to actually be executed) when their source registers are valid. This produces the desired behavior for dependencies where a second instruction needs the result of the first, but because a completing instruction must set the valid flag for registers it updates, multiple instructions that will ultimately write to the same register cannot be active at once. In fact, decoding will stall if it finds an instruction that will write into a register already targeted by an active instruction [3]. This is an old algorithm, which was used in the CDC-6600 [3] and the Intel i960, which was the first processor to continuously execute more than one instruction per clock cycle [4]. In 1996, it was proven that the original scoreboard algorithm would run into deadlock, but this was fixed with a change to the valid flags, and the updated version proven correct later that year [5].

In the Tomasulo scheduling algorithm, instructions are assigned tags when they are issued. The register file is given a producer table, which contains both valid bits, and tags for each register. If they are supposed to write to registers, those registers have their valid bits cleared, and their tags set to the last instruction that will write into them. This allows multiple instructions that will write into the same register to be active at once, because completing instructions can check if their tag is the same as the one in the producer table, and if not, don't set the valid bit [1].

Instructions are issued into reservation stations, which hold them, their tag, and perhaps copies of the source data they need, with a valid bit to tell whether it is with them, or whether it is not ready. If not ready, they also store the tags of the instructions that will generate the data. They are dispatched as soon as their source data is ready. The reservation stations are connected to a bus called the CDB (Common Data Bus), and completed instructions are broadcast on it so that waiting instructions can dispatch [1].

Between the functional units the instructions are dispatched to and the CDB are producers, which are just latches there to store the results and tags of the instructions until the CDB is available. An optional reorder buffer to allow for precise interrupts may be between the CDB and the register file. This is covered further below [1].

Scheduling can also be done with a dependency matrix, patented in 2001 by Merchant et al [10]. In this scheme, instructions are issued into an array, which holds them until they are dispatched. There is a square matrix of bits, with its main diagonal missing, with a row for each entry in the array. These bits are set when the instruction whose row they belong to depends on the instruction whose column they belong to. The main diagonal is missing because an instruction cannot depend on itself. When an instruction is issued, it is checked against all active instructions prior to it in the program order and the proper bits are set in the dependency matrix [10].

Every clock cycle, fast domino logic will OR together all of the bits in the row of every instruction, to find which are ready to dispatch. When they have no active dependencies, they are ready. Instructions remain in the array and matrix until they have completed, and then their row is cleared in the matrix, and their spot in the array deallocated. This has an advantage over Tomasulo scheduling in computers with lots of memory and fast clock cycles, because the comparisons to determine dependencies are much faster [10]. One advantage Tomasulo scheduling as described in [1] has over this is that multiple instructions that will write to the same register can be active at the same time.

Interrupts

One way to get precise interrupts in an out-of-order machine is to use checkpoints. In this scheme, which can also help with branch mispredicts, the state of the processor is periodically saved, and upon an exception, the address of the instruction is saved, the rest of the state of the processor reset to the checkpoint. Then the processor performs a "repair," and re-executes each instruction between the checkpoint and the offending instruction [6]. This causes a lot of overhead, and introduces a tradeoff based on how frequent the checkpoints will be. If they are too frequent, the overhead will get worse. If they are too sparse, repairing will take longer.

Another way is to use a reorder buffer. This is a circular buffer that holds the results of completed instructions temporarily, which is big enough to hold as many instructions as the processor could possibly execute ahead. These results are then committed to the register file in order. If the results of an instruction include that an exception was triggered, future instructions in the program order can be discarded. The reorder buffer can also include tags of the instructions, which allow the results to be used in dependent instructions. A correctness proof of this scheme is detailed in [2], and a hypothetical DLX implementation is given in [1].

The IBM 360/91 allowed precise interrupts by halting decoding at an interrupt until all instructions are complete, but also allowed imprecise interrupts which discarded the state of the processor [7].

Instruction Window

In OoOE, the instruction window is the range of instructions that are being checked for the possibility of execution. Part of the motivation of OoOE is to deal with slow instructions such as memory access. For this to work, the processor has to be able to reach far enough ahead to find instructions it can execute while waiting. As normal instructions speed up faster than memory accesses, this is a problem, and longer instruction windows are needed [9]. Unfortunately, longer instruction windows demand more dependency determination hardware, because there are more possible dependencies to check when deciding whether to dispatch an instruction.

One way to deal with this is to add a Waiting Instruction Buffer (WIB), which is a queue much larger than the instruction window, where instructions dependent on the slow instruction are stored. This saves time because instructions in the WIB don't have to be continuously rechecked for readiness. They only have to be rechecked when the slow instruction that put them there finishes. They might return if they are still dependent on slow instructions. Instructions still have to be checked against those in the WIB for dependencies when they are fetched, but because the continuous dependency checking was the critical path, this still allows the WIB to grow much larger. When the authors of this idea simulated it, they were able to get a 20%-84% speedup on the benchmarks they used, using a 2000-entry WIB and a 32-entry instruction window [11].

Their algorithm decides whether to hold instructions in the ordinary instruction window, dispatch them, or put them in the WIB by keeping "wait bits" next to every ready bit. A wait-bit indicates that a register is "pretend ready," which means that instructions depending on it that are in the issue queue holding the instructions in the instruction window will be sent to the WIB when they would otherwise be dispatched, and that they will set the wait-bits of the registers they write to instead of clearing the ready bits. The bit is initially set by a cache miss [11]. The authors note that this allows for instructions in the WIB to be dependent on multiple waiting cache misses, which allows loops containing many cache misses to be unrolled, and the misses stacked together.

They also duplicate instructions that aren't really waiting in the WIB so that the WIB contains instructions in program order and branch mispredict recovery is simpler. The WIB also has a set of bit vectors, arranged as columns in a matrix. Each column is for a particular cache miss, and each row is for an instruction. A set bit at an intersection means that that instruction is waiting for that cache miss. For each cache miss it stores a pointer to the corresponding row in the matrix, so that instructions dependent on it can be sent back into the issue queue. Even if they are dependent on another cache miss, they are still sent into the issue queue, which makes the wakeup logic in the WIB simpler, but can result in instructions being sent back and forth between the WIB and the issue queue.

Another solution to this described in [9] is runahead execution. This works by saving a checkpoint, then recording a dummy value for the result of the instruction, and putting the processor in a speculative mode as soon as it starts to hang on a slow instruction. This will let the instruction window move far ahead, executing and "pseudo-retiring" instructions, which means that they do not make any real architectural changes, but are allowed to leave the instruction window. When the instruction completes, the speculative results are dumped. But they still have the effect of preparing the caches to provide what they will need, so when they really execute, they do it faster [9].

This solution was modeled in 2002, and improved the performance of a virtual Pentium 4 processor with a 128-entry instruction window by 22% for memory intensive programs, which was within 1% of the performance of a machine with a 384-entry instruction window.

Conclusion

Out of order execution is most useful for tolerating cache misses in single-threaded applications, which it can do by executing instructions later in program order than the offending ones. To do this well in modern computers without interfering with clock speed, modifications such as a WIB or runahead execution are best.

Modern computers also have enough registers to demand that for a fast clock speed, scheduling algorithms cache some part of the information that determines when there are dependencies, instead of recomputing from scratch at every clock. Techniques like using dependency matrices can solve this problem. Register renaming is also useful to eliminate Write-After-Read and Write-After-Write dependencies so that they don't prevent the processor from working far ahead of the slowest instruction.

References

1: *Out of Order Execution*. (n. d.). Retrieved October 11, 2012 from: http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur2/ss03/literature/chapter6.pdf

2: Kroening, Daniel, Mueller, Silvia M., and Paul, Wolfgang J. *A Rigorous Correctness Proof of a Tomasulo Scheduler Supporting Precise Interrupts*. (1999). Retrieved October 11, 2012, from: http://www-wjp.cs.uni-saarland.de/forschung/projekte/Computer_Architecture/papers/sci99.pdf

3: Johnson, William M. *Super-Scalar Processor Design, pp 13*. (1989). Retrieved October 11, 2012, from: ftp://reports.stanford.edu/pub/cstr/reports/csl/tr/89/383/CSL-TR-89-383.pdf

4: McGeady, S. *The i960CA SuperScalar Implementation of the 80960 Architecture*. (1990). Retrieved October 11, 2012, from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=63681

5: Müller, Silvia M, and Paul, Wolfgang J. *Making the Original Scoreboard Mechanism Deadlock Free*. (1996). Retrieved October 11, 2012, from: http://docis.info/docis/lib/sisl/rclis/dbl/istcsi/(1996)%253C92%253AMTOSMD%253E/www-wjp.cs.uni-sb.de%252F~smueller%252Fistcs.ps.gz

6: Hwu, Wen-Mei W, and Patt, Yale N. *Checkpoint Repair for High-Performance Out-of-Order Execution Machines*. (1987). Retrieved October 11, 2012, from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5009500

7: Torng, H. C. and Day, Martin. *Interrupt Handling for Out-of-Order Execution Processors*. (1993). Retrieved October 11, 2012, from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=192223&tag=1

8: Önder, Soner, and Gupta, Rajiv. *Superscalar Execution with Dynamic Data Forwarding*. (1998). Retrieved October 11, 2012, from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=727183

9: Mutlu Onur, Stark, Jared, Wilkerson, Chris, and Patt, Yale N. *Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors*. (2003). Retrieved October 23, 2012, from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1183532

10: Merchant et al. *Scheduling Operations Using a Dependency Matrix.* (2001). Retrieved October 23, 2012, from: http://www.google.com/patents?hl=en&lr=&vid=USPAT6334182&id=MFkIAAAAEBAJ&oi=fnd&dq=Tomasulo+Scheduling+Patent&printsec=abstract#v=onepage&q&f=false

11: Lebeck, Alvin R., Koppanalil, Jinson, Li, Tong, Patwardhan, Jaidev, and Rotenberg, Eric. *A Large, Fast Instruction Window for Tolerating Cache Misses.* (2002). Retrieved October 23, 2012, from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1003562

12: Hily, Sébastian and Seznec, André. *Out of Order Execution May Not Be Cost-Effective on Processors Featuring Simultaneous Multithreading.* (1999). Retrieved October 23, 2012, from: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=744331