Compiler-Assisted, Selective Out-Of-Order Commit

Nam Duong and Alexander V. Veidenbaum
Department of Computer Science
University of California, Irvine
{nduong, alexv}@ics.uci.edu

Abstract—This paper proposes an out-of-order instruction commit mechanism using a novel compiler/architecture interface. The compiler creates instruction “blocks” guaranteeing some commit conditions and the processor uses the block information to commit certain instructions out of order. Micro-architectural support for the new commit mode is made on top of the standard, ROB-based processor and includes out-of-order instruction commit with register and load queue entry release. The commit mode may be switched multiple times during execution. Initial results for a 4-wide processor show that, on average, 52% instructions are committed out of order resulting in 10% to 26% speedups over in-order commit, with minimal hardware overhead. The performance improvement is a result of an effectively larger instruction window that allows more cache misses to be overlapped for both L1 and L2 caches.

Keywords—Out-of-order commit, resource release, overlapping cache misses, architecture/compiler co-design

1 INTRODUCTION

The long latency of L2 and/or L3 cache misses has a major impact on a single-core, out-of-order (OOO) processor performance. Under a long-latency miss one or more of the core’s resources, such as a Register File (RF), a Load Queue (LQ), a Store Queue (SQ), an Instruction Queue (IQ) or a Reorder Buffer (ROB), become full. This leads to stalls and a loss of the core’s performance. Increasing the size of the individual resource would help [5], but large sizes are required to show significant performance improvements. The growth in resource sizes is constrained by their effect on cycle time and energy consumption/temperature.

The standard OOO architecture releases an instruction’s resources on instruction commit (or even later for registers) which occurs from the top of the ROB. When commit stalls on a long-latency cache miss, no resources can be released once the miss reaches the top of the ROB. This state can persist for tens to hundreds of cycles. An early instruction commit, i.e. prior to reaching the top of the ROB, or instruction resource release prior to commit can significantly improve performance for a given size of resources.

The early commit or resource release requires new ways to maintain precise exceptions and recover mis-speculation on branches or loads. Two approaches have been proposed. One approach [2] proposed to wait until no exception or mis-speculation could occur on older instructions before early commit and release of (some) resources. This approach, however, delays the time when the OOO commit and resource release can occur. Under a long-latency miss neither the exception status or the speculation status among younger instructions may be known until after the load miss service completes. Another approach [9] proposed to separate the exception recovery and mis-speculation on branches and loads. It used a checkpoint for exception recovery, while releasing instruction resources prior to ROB commit if all older instructions were mis-speculation free. The two approaches are thus similar in delaying commit/resource release to avoid mis-speculation. Finally, it should be mentioned that Kilo-instruction processors [5] were sometimes called OOO commit processors. However, such processors use multiple checkpoints without an ROB and commit only the oldest checkpoint after all its instructions have completed execution. All resources of the checkpoint are released on its commit. This is therefore an in-order commit and resource release.

This paper proposes to use a compiler to assist the OOO commit. It eliminates the wait for exception or load mis-speculation status of older instructions used by prior approaches by utilizing compile-time information. The compiler will designate which instructions may be committed out of order (hence the selective OOO commit). In particular, it will only designate loads for OOO commit if they would never need to be replayed. The compiler will also guarantee that any instruction committed early can be correctly re-executed if an exception was detected on older instructions, thus eliminating the need for checkpoints. Finally, the compiler will assist with in program order, non-speculative store commit. We believe that OOO commit of stores is too complex, e.g. buffering speculative stores in the cache [9], and avoid it. Overall, this approach simplifies the OOO commit micro-architecture while increasing performance.

The compilation approach is to define blocks of instructions with different commit properties with the micro-architecture deciding when it can start the commit of instructions in a block. Some blocks will be committed in program order while others may be committed out of order. The compiler described here focuses on finding instruction blocks in loops, which creates multiple instances of blocks in execution. The blocks that have a compile-time cross-iteration dependence commit in order, while the those that only have dependencies within an iteration may be committed out of order.

The OOO commit is built on top of the standard ROB-based micro-architecture. It uses compile-time information to make run-time decisions about commit. The proposed micro-architecture defines an OOO commit “mode” on top of the standard, in-order commit mode. The compiler determines when to switch to OOO commit mode and which blocks the hardware can start to commit out of order in this mode. Instructions commit in order within a block. The hardware uses the instruction blocks and their types in OOO commit, dependence tracking and register release.

The rest of the paper describes a possible implementation to make the discussion concrete. It uses a certain division of responsibilities between a compiler and a micro-architecture, but alternative divisions and thus implementations are possible. Specifically, the proposed micro-architecture uses a two-level OOO commit mechanism: a block-level commit and an individual instruction commit. The latter is an almost unmodified standard ROB-based instruction commit, which commits instructions in the in-order commit mode. The block-level commit mechanism drives the OOO “mode” and a) decides when to start committing instructions in a block and b) tracks block dependencies and releases physical registers. The block-level commit hardware does not commit individual instructions, instead it “passes” them to the ROB for commit.

This implementation imposes the following constraints on the OOO commit in order to reduce the hardware complexity and preserve most of the standard OOO architecture:

- C1. Instructions in a block are committed in order even in the OOO commit “mode”, thus preserving all exception, speculation and resource management techniques used in the “standard mode”.
- C2. Instructions from different blocks that satisfy the OOO commit conditions may be committed out of program order.
- C3. Stores are committed in program order when it is safe to update the memory state.
- C4. The OOO commit of a block starts only after older branches are resolved.

The first three constraints are fundamental to the approach, the last one is not and is used for simplicity.

The proposed approach naturally leads to the use of weak or release consistency, even though stores are committed in order. This is due to the fact that loads may be retired out of order and removed from the
2 COMPILATION FOR OOO COMMIT

The compiler needs to identify and/or create via program transformation instruction blocks in a loop to which the OOO commit can be applied. Consider a loop example in Fig. 1. The loop body at the source level contains two statements, the first of which has a cross-iteration data dependence and the second one which does not. Let us create blocks for this example loop. Recall that all control and data dependencies within a block are guaranteed since the micro-architecture uses the standard in-order commit within a block.

The compiler creates two blocks in the loop body. An in-commit order block contains the two stores which need to be committed in order and a part of the first statement that has the data dependence. The second block contains loads and operations that can commit out of order. There are two other source blocks that correspond to the “for” statement. The first has the conditional statement and index computation, the second contains the loop’s unconditional branch back. The former has a dependence on “i” and requires in order commit. The latter block is also marked for in order commit.

All these blocks can be seen in the assembly code for the loop starting at labels P0, B0, B1, E and P1. An additional opportunity for OOO commit was created by partitioning the OOO commit block into two independent (sub-) blocks: Also note that address computations are in an in-commit block as they are incremented in every iteration. The figure shows a graph with dynamic instances of these blocks and their dependencies for two iterations of the loop (node labels correspond to block types defined below).

This construction of the blocks allows the effect of the OOO commit to be undone because such blocks do not have stores to memory. The micro-architectural state (registers) can be restored and the block re-executed at a later time. Also, note that the in-commit order blocks are not all the same: one of them contains stores. This will be exposed to the micro-architecture and create an additional OOO commit opportunity.

Instruction Block Types. Three types of instruction blocks are defined for a loop: a prolog, a body and an epilog. The blocks in Fig. 1 are labeled based on these names. The compiler will pass the block type information to the hardware. The micro-architecture will track blocks, their type and age and will enforce the specified commit order. Note that alternative block definitions are possible, leading to a different micro-architecture.

A prolog is a block that is data and/or control dependent only on previous prologs. There can be multiple prologs in a loop, as in the above example.

A body is a block dependent on previous prologs(s) only. Any two instances of a body block are independent of each other. There may be multiple body blocks in a single program loop.

Figure 1. A loop example.

The proposed OOO commit mechanism was implemented in the M5 simulator [3] and evaluated on six SPEC2000 and 2006 benchmarks. Only functions listed in Table 2 used OOO commit. The IPC improvement from the OOO commit over a reasonably standard 4-wide processor ranged from 10% to 26%. The differences are, in part, due to the percentage of instructions designated for the OOO committed in each benchmark. The main reasons for improved performance are the ability of the proposed micro-architecture to remove instructions from the ROB and the load queue faster and thus issue additional loads earlier than the standard mechanism would allow. The miss service for such loads is overlapped under a commit stall.

2.1 Commit Window

ROB and LQ. Thus program load order is not preserved and a standard commit time recovery can not be used.

The proposed OOO commit mechanism was implemented in the M5 simulator [3] and evaluated on six SPEC2000 and 2006 benchmarks. Only functions listed in Table 2 used OOO commit. The IPC improvement from the OOO commit over a reasonably standard 4-wide processor ranged from 10% to 26%. The differences are, in part, due to the percentage of instructions designated for the OOO committed in each benchmark. The main reasons for improved performance are the ability of the proposed micro-architecture to remove instructions from the ROB and the load queue faster and thus issue additional loads earlier than the standard mechanism would allow. The miss service for such loads is overlapped under a commit stall.

2.2 Block Creation.

The compiler analyzes dependencies (typically at the IR level) for statements in a loop body using array subscript and pointer analysis. A high-level algorithm to create blocks is shown next for a singly-nested for loop. Multiply-nested loops are handled similarly.

1. Loop control statements (including the loop terminating branch) form a prolog (e.g. P0 above).
2. Address computations in a loop are moved (up) into the prolog.
3. Stores and minimal statement dependence cycles are moved (down) into an epilog.
4. An unconditional branch at the end forms a second “prolog” (e.g. P1 above).
5. All remaining instructions form a body block, which may be split into multiple bodies to expose independent loads.

The advantage for nested loops is that the commit mode does not change between instances of the inner-most loop. The outer for loop(s) each get their own prologs P0 and P1 and, if necessary, an epilog.

Block Markers and Commit Mode Instructions.

Two new instructions are generated by the compiler to 1) mark block boundaries and identify block type (markers) and 2) switch between the in-order and the OOO commit modes. A block marker instruction indicates to hardware that all instructions until the next block marker belong to the same block as well as the block type.

3 ARCHITECTURAL SUPPORT

Several changes are made to the micro-architecture to support OOO commit. The rename records for each physical register the block id of the instruction that mapped it and (eventually) the id of the block in which it is remapped. The ROB and the LQ are modified to support the OOO removal of instructions. Other changes are described next.

The Block Table. A new structure, the Block Table (BT), shown in Fig. 2 is added to store dynamic information about instruction blocks. The BT controls the commit of instruction blocks and identifies instructions that the ROB can commit. A BT entry is created when a block marker is decoded and contains the following fields: a unique Block ID (BID); a Block Type (T), e.g., a prolog, a body, or an epilog; and a Position (P) in the ROB of the next instruction to commit in this block.

Instructions may be committed out of order from 2 the block is a committed in order. An instruction at position P is checked for ready-to-commit conditions and is committed regardless of its age if the conditions are met. The ROB performs the actual instruction commit and resource release.

The unique BID is assigned using a block counter with a large range. The decode stalls when the counter overflows or when the BT is full.

OOO Commit. The processor starts execution in the standard mode (S-mode). The commit mode changes to OOO mode (O-mode) when the mode switch instruction is retired. In this mode all instructions pointed to by P in each ready-to-commit block in the BT are passed to the ROB for commit.

A ready-to-commit block is defined as 1) a prolog or a body such that any older block of type prolog has committed all of its instructions or 2) an epilog that has reached the top of the BT. The first condition above guarantees that a loop terminating branch and an induction variable computation for the current and all prior iterations have executed. The second condition guarantees that stores are committed non-speculatively and in order. Note that the ready to commit condition uses only block type and BT order information for dependence tracking. Also, a prolog may commit before older body blocks thus allowing younger blocks to become ready to commit.

1. Note that it is possible to commit several consecutive instructions starting at P, but this implementation does not.
A small number of entries, W, at the top of the BT form a Commit Window.
In this window are checked blocks younger than A for availability for committing instructions. Using a small W simplifies the BT logic and makes it implementable, the rest of the BT is just a FIFO. For instance, our results show that 96% of all ready instructions are covered with W = 8.

The Commit Window logic can be viewed as part of the first commit stage. The “commitable” instructions identified by it are passed on to the ROB, which constitutes the second commit stage. In this stage, each block entry is committed and removed from the BT after all its instruction have committed. The BT and the ROB are “compressed” after the OOO removal of instructions and BT entries.

Compressing the ROB and the BT. A mechanism for such compression was described in [2] which collapsed the ROB by shifting instructions to fill the gaps. Such compression requires complex hardware to move the ROB entries. It also prevents one from using the ROB position to identify an instruction. The OOO commit architecture proposed here uses a linked-list ROB2 by adding pointers to each ROB entry to create two linked lists - a used list and a free list. Insertion into or removal from the ROB thus requires only pointer manipulation. A BT entry update may also be required. Squashing of an instruction moves it and all younger instructions to the free list.

OOO Load Instruction Commit. OOO load commit is crucial to the performance of the proposed architecture. Only loads in body blocks may be committed out of order i.e. a load in a younger block before a load in an older block. Two issues stand in the way of OOO load commit: store-to-load forwarding and multi-processor load ordering. Both of these require a load to be available so that it can be replayed upon such an ordering violation. A standard Load/Store Queue (LSQ) is already a part of the baseline in-order commit processor, thus only the necessary changes for OOO load commit are discussed. A standard replay mechanism is used for dealing with ordering violations.

A store-to-load forwarding for a load in a body block can only happen, per definition of the body block, from stores outside the loop or from within the block. The former stores are all retired by the time OOO commit mode is entered. Stores within a block are moved by the compiler to an epilog block. Thus a load may be OOO committed from a body block because it cannot need to be replayed due to store-to-load forwarding. The prolog and epilogs are committed in order and thus have no issues.

A “per-prolog” load mapping is required by some memory consistency models. The proposed architecture OOO commits some loads and thus the program order cannot be guaranteed. This is why a weak or release consistency model is assumed. Recall that the stores do commit in program order.

OOO Register Release. For the in-order commit, when an instruction writing a logical register commits, the previous physical register that mapped the same logical register is released. This guarantees that a new value is created for the logical register and

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Loop line numbers</th>
<th>FF</th>
<th>W</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar</td>
<td>354 in smoker.c</td>
<td>8.1B</td>
<td>0.3B</td>
<td>1.3B</td>
</tr>
<tr>
<td>equake</td>
<td>1195 in equake.c</td>
<td>2.3B</td>
<td>0.2B</td>
<td>1.7B</td>
</tr>
<tr>
<td>mgrid</td>
<td>149, 189, 230, 270, 291 in mgrid.f</td>
<td>0.9B</td>
<td>0.3B</td>
<td>1.7B</td>
</tr>
<tr>
<td>swim</td>
<td>261, 315, 397 in swim.f</td>
<td>0.2B</td>
<td>0.3B</td>
<td>1.1B</td>
</tr>
<tr>
<td>libquantum</td>
<td>89 in gates.c</td>
<td>2.2B</td>
<td>0.3B</td>
<td>0.7B</td>
</tr>
<tr>
<td>bwaves</td>
<td>168 in block_solver.f</td>
<td>1.9B</td>
<td>0.5B</td>
<td>1.8B</td>
</tr>
</tbody>
</table>

blocks younger than A have been committed. Restarting execution in block A requires that the register map be restored to its state at that time and that none of the registers alive then have been released. The latter is guaranteed by the block reference count because block A has not committed. As for restoring the register map, let us assume that a physical to logical register map is maintained, such as the CAM-based renamer in [4]. The OOO (or in-order) register release in this case removes a register mapping without affecting any other live mappings. A reference counter for each register is updated as per [1] as canceled blocks are removed from the BT. Note that blocks that committed out of order both increment and decrement the reference counters and thus do not affect the mechanism.

Exceptions. Standard OOO processors commit instructions in program order to maintain sequential program semantics and, in particular, maintain precise exceptions with relative simplicity as exceptions are processed at commit. This is possible because the stores commit from the top of the ROB and the register state can be restored.

The proposed OOO commit mechanism commits stores in-order, thus only the register state needs to be restored. The OOO register release mechanism described above guarantees that the register mappings are not released upon OOO commit and are available for replays.

4 RESULTS AND ANALYSIS

Methodology. The M5 simulator targeting the Alpha 21264 architecture [8] was used to evaluate the OOO commit. The OOO commit architecture was implemented on top of the baseline processor and compared to three in-order commit processors shown in Table 1. The LargeA configuration is similar, in terms of resource sizes, to the Intel’s Sandy Bridge processor [7]. The LargeB configuration was chosen so as to achieve a similar average performance to the OOO commit processor.

Six SPEC2000 and SPEC2006 benchmarks were used (see Table 2). The OOO commit was only applied in loops (shown in the table with a “loop” flag and generated software prefetch instructions. The table also shows the number of Fast-Forewarded (FF), Cache Warmup (W), and Committed (C) instructions in simulation. Instruction blocks were created by manually inserting the markers (as no-ops) in the assembly code for the selected loops.

Instruction Commit. Fig. 3 shows the percentage of committed instructions in the standard commit mode (the S-loop) and in the O-mode, with the latter further classified as coming from prologs, bodies or epilogs. On average, 71% of the instructions are committed in the O-mode, distributed as 20% from prologs, 48% from bodies, and 3% from epilogs. These are very dependent on the number of loops compiled for OOO commit.

Resource Release in the O-mode. Even in the OOO mode instructions may not commit out of order, as seen in Fig. 4. It shows that, on average, 73% of instructions were really committed out of order. Of these, just

Table 1. Processor configuration.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>IQ (total FP and INT)</td>
<td>Baseline</td>
</tr>
<tr>
<td>ROB</td>
<td>128</td>
</tr>
<tr>
<td>IQ/QQ</td>
<td>32/32</td>
</tr>
<tr>
<td>Registers (Int/Float)</td>
<td>160/160</td>
</tr>
<tr>
<td>BTB entries</td>
<td>4K</td>
</tr>
</tbody>
</table>

Table 2. Benchmark info.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Loop line numbers</th>
<th>FF</th>
<th>W</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar</td>
<td>354 in smoker.c</td>
<td>8.1B</td>
<td>0.3B</td>
<td>1.3B</td>
</tr>
<tr>
<td>equake</td>
<td>1195 in equake.c</td>
<td>2.3B</td>
<td>0.2B</td>
<td>1.7B</td>
</tr>
<tr>
<td>mgrid</td>
<td>149, 189, 230, 270, 291 in mgrid.f</td>
<td>0.9B</td>
<td>0.3B</td>
<td>1.7B</td>
</tr>
<tr>
<td>swim</td>
<td>261, 315, 397 in swim.f</td>
<td>0.2B</td>
<td>0.3B</td>
<td>1.1B</td>
</tr>
<tr>
<td>libquantum</td>
<td>89 in gates.c</td>
<td>2.2B</td>
<td>0.3B</td>
<td>0.7B</td>
</tr>
<tr>
<td>bwaves</td>
<td>168 in block_solver.f</td>
<td>1.9B</td>
<td>0.5B</td>
<td>1.8B</td>
</tr>
</tbody>
</table>

2. Note that the linked list part of the ROB is used only in the OOO commit mode.
under a third comes from prologs, which shows their importance to OOO commit. Fig. 5 shows that (leftmost bar), on average, slightly over 50% of all BT block entries were committed before reaching the BT top. The second bar shows that, on average, 76% of loads were committed out of order from the LQ and the ROB. The rightmost bar shows that the OOO release of registers is less frequent. A more aggressive OOO release is being investigated.

**BT Operation.** The leftmost bar (occupancy) in Fig. 6 shows the average number of entries in the block table BT. It indicates that the BT can be quite small. The second bar (search size) shows the average position in the BT past which there are no blocks with instructions that can be committed out of order. This number indicates that the size of the BT search window can be small, allowing fast search. The last bar (found size) shows the BT position of the first block that has a ready to commit instruction and the instruction is committed. This confirms that most of the OOO commit happens in the top BT entries. Combined, these results indicate that the BT is not expensive and its fast implementation is possible.

**Performance Improvement.** Fig. 7 shows a speedup over the baseline for three processor configurations: OOO commit, LargeA and LargeB. The leftmost bar is the speedup of OOO commit architecture with the same size of the ROB, LQ, SQ, and register files as the baseline. The speedup is 10% or larger, mgrid is highest with 26%. Again, the speedup depends on the loops modified and the number of instructions committed in the OOO commit mode. The lowest speedup is in equake which commits only 32% of instructions in the O-mode (see Fig. 3). mgrid has the highest speedup as the percentage of instructions in the O-mode is high.

The OOO commit architecture also outperforms the in-order LargeA architecture which has approximately 50% more resources. It takes a 768-entry instruction window with very large LQ and SQ for the in-order commit processor (LargeB) to reach the same performance, on average. The OOO commit can improve performance without such a large increase in resources.

**Source of Speedup.** The cache misses are the main reason for performance loss in the in-order baseline. Thus the cache behavior is analyzed to better understand how the OOO commit improves performance. Fig. 8 shows the distribution of allocated L2 cache MSHRs measured at the time a new MSHR is allocated. This statistic indicates an overlap of miss service for L2 misses. On average, the OOO commit has more concurrently allocated MSHRs than in-order commit baseline for all of the benchmarks. The OOO commit has fewer instances when only one miss is outstanding and noticeably more instances with more than four outstanding. LargeB has even more concurrently allocated MSHRs than the OOO commit. This statistic correlates well with speedups in Fig. 7. The L1 cache misses that hit in the L2 have a 20 cycle latency and cause stalls, albeit lower than those due to the L2 misses. OOO commit also increases the overlap of such L1 misses (not shown here), which further improves performance. Here the increase comes from more concurrent misses per an L1 MSHR rather than from the number of MSHRs.

In summary, the proposed OOO commit mechanism clearly improves performance. It is achieved by freeing the ROB, the LQ and the register file space early and allowing new instructions to enter the window, especially loads which are thus issued earlier. The amount of improvement depends on (1) the number of instructions committed in the O-mode (in turn dependent on the number of loops compiled for OOO commit), (2) the number of blocks available for concurrent commit, and (3) the amount of L1 and L2 miss overlap, which is the main reason for performance improvement.

**REFERENCES**