Center for Embedded Computer Systems
University of California, Irvine

Compiler Assisted Out-Of-Order Instruction Commit

Nam Duong and Alex Veidenbaum

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

{nlduong, alexv}@ics.uci.edu

CECS Technical Report 10-11
November 18, 2010

Compiler Assisted Out-Of-Order Instruction Commit

Nam Duong and Alex Veidenbaum

Department of Computer Science
University of California, Irvine

{nlduong,alexv}@ics.uci.edu

Abstract

This paper proposes an out-of-order instruction commit mechanism using a novel compiler/architecture interface. The compiler provides information about instruction "blocks" and the processor uses the block information to decide which instructions can be committed out of order and when. Some blocks are guaranteed to be data independent blocks which allows instructions from different such blocks be committed simultaneously and out of order. Other blocks have data or control dependencies and require in-order execution and in-order commit. Micro-architectural support required for the new commit mode is made on top of the standard, ROB-based commit and includes out-of-order instruction commit, early register release, support for committing loads and stores out of order, and exception handling. All of these are driven by the block information which simplifies the hardware. Results for a 4-wide processor model based on the Alpha 21264 and a set of 6 SPEC2000 and 2006 benchmarks 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.

1 Introduction

Over the last two decades there has been a tremendous increase in processor performance due to increasing clock speeds and advanced architectural techniques, such as wide issue, out-of-order (OOO) execution and various forms of speculative execution. However, recently the growth in clock speed has slowed down and performance increases due to processor architecture improvements have reduced significantly leading to saturation of single thread performance on single core processors.

Single-core OOO processor performance growth is mainly limited by the long latency of cache misses,

especially in L2 or L3 caches. Under a long-latency miss the single core resources, such as Register File (RF), Load Store Queue (LSQ), Instruction Queue (IQ) and Reorder Buffer (ROB) entries, are exhausted leading to stalls and limiting the performance. Increasing the size of the instruction window and thus of the above resources and early resource release can help [15, 5, 1, 22] but has proven difficult to implement.

Alternatively, an out-of-order commit [2] can prevent long latency misses from stalling commit and resource release and improve processor performance. While the performance improvement achieved was smaller than that of large-window processors, it largely preserved the OOO processor architecture. One of the necessary conditions for OOO commit required delaying commit until all older instructions are exception free. This is a significant restriction under long-latency loads which stall the execution of dependent instructions. Such instructions have unknown exception status and therefore prevent all younger instructions from committing. In this work, we propose a different approach to OOO commit which relaxes this and other commit ordering conditions.

OOO commit can further improve performance and manage resources better in single-core, single-thread processors[1] if assisted by compilers. This paper proposes a compiler/architecture interface to achieve this by allowing information about data independence to be communicated from software to hardware at the granularity of "blocks". The main contributions of this work are that *a)* it augments the "standard" OOO processor architecture with a compiler initiated out-of-order commit mode; *b)* allows a simple software "marking" mechanism to identify data independent blocks from which instructions can be committed simultaneously and out of order, as well as blocks with dependencies that are executed in order and with in-order commit; and *c)* defines microarchitectural support necessary to achieve this, including OOO commit and register release, exception handling, and memory disambiguation.

The OOO micro-architecture is *augmented* with block management which enables the early release of ROB entries, LSQ entries, and registers. The knowledge of block independence is also used to deal with exceptions differently than prior proposals. In particular, in the architecture proposed here the OOO commit will proceed even if it is not known that all earlier instructions are exception free. The rest of the processor remains largely unchanged and, when the proposed OOO commit mode is not used, behaves as a standard OOO processor.

There are many naturally occurring code blocks in program loops in which there are no cross-iteration data dependencies, either through register and memory accesses. Such code blocks can be committed out of

[1] Non-speculative multi-threading is orthogonal to this work while thread-level speculation is largely redundant [12, 11].

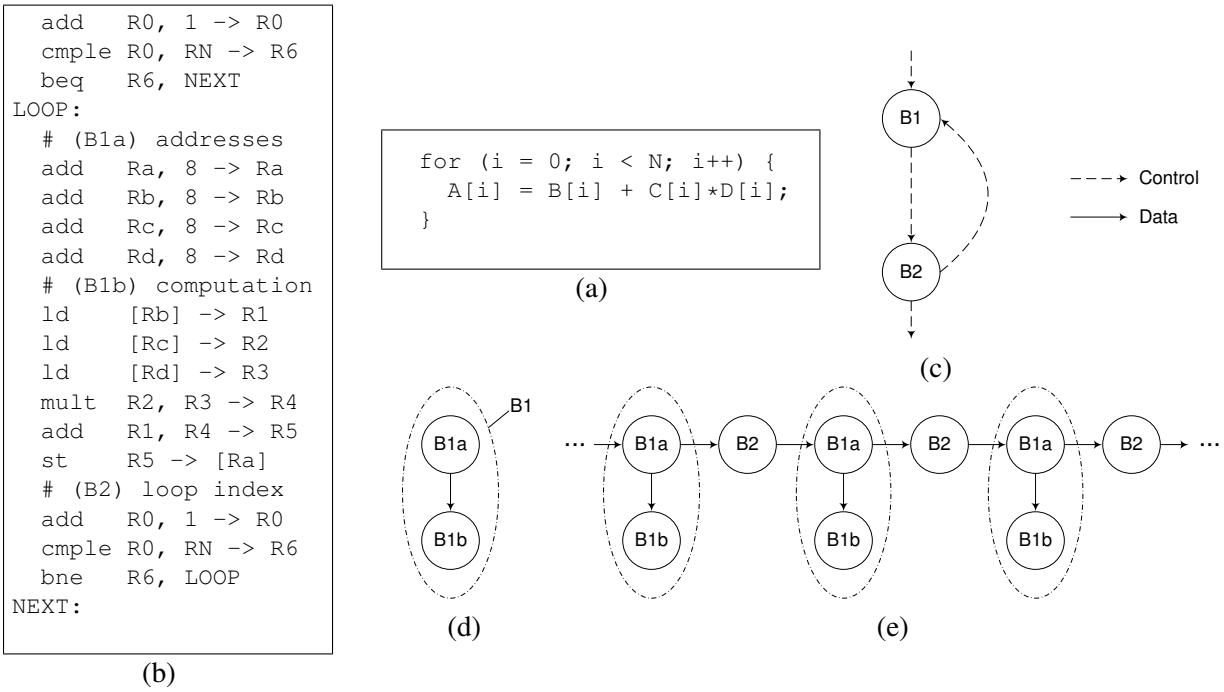


Figure 1. A loop example: (a) source code, (b) its assembly equivalent, (c) the control flow, (d) partitioned B1 consisting of B1a with cross-iteration dependencies and B1b without such dependencies, and (e) the dynamic instances of blocks in 3 iterations.

order and their resources released early, subject to exception handling and memory ordering. Furthermore, a compiler may be able to expose more such blocks through transformations and optimizations.

Example 1. Consider a loop shown in Figure 1(a). Its assembly equivalent (b) and its flow graph (c) are also shown. The latter consist of blocks B1 and B2. For our purposes, B1 is further partitioned into blocks B1a and B1b, such that B1a has cross iteration dependencies, e.g. a memory address calculation, and B1b has no such dependencies. A compiler for the proposed architecture can easily perform such partitioning as well as delineate block boundaries and types for the processor. Blocks are tracked in execution to enforce block dependencies as shown in (e), find independent instructions to commit, determine when block resources can be released, etc. One can think of this block management mechanism as somewhat similar to instruction group management in the IBM Power processors [23], but with group formation performed by compiler and thus allowing larger groups. Except that in our case "groups" are used only in the OOO commit mode and in combination with a standard ROB.

Figure 2 shows the proposed architecture with a *block table* and the ROB with two iterations of the above loop. The block table maintains all the information about blocks. It contains logic to track ready instructions

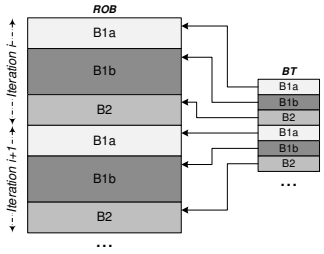
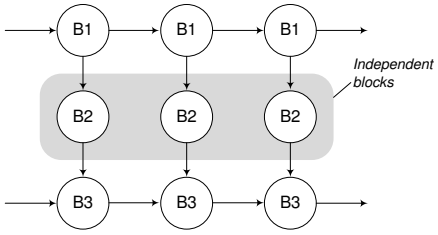


Figure 2. The Block Table (BT) and the Reorder Buffer (ROB) with two iterations of the loop in Figure 1.

```
(B1)  i = i + 1;
      Calc addrs of A[i-2], A[i];
      Calc addrs of B[i], C[i];
(B2)  T = B[i]/C[i];
(B3)  A[i] = A[i-2] + T;
```

(a)



(b)

Figure 3. Example of a loop with different types of dependencies.

from different blocks and, using block types/properties, may flag one or more instructions from a block for commit. The ROB commit logic takes over from here.

Instructions in blocks of type B1a and B2 are committed in order with respect to each other and may be committed OOO with respect to instructions in blocks of type B1b. This is because B1a and B2 contain instructions which are dependent of each other in different iterations, but they are not dependent on instructions in B1b. Thus the execution of the sequential and control dependent part(s) of the loop basically uses in-order commit. The processor may perform OOO commit of instructions from different blocks of type B1b, but only after the preceding block B1a (from the same iteration as the B1b) has committed. This restriction on the OOO commit allows it to be non-speculative and helps with resource release.

The above OOO commit mechanism can handle a block/loop code in which there are other cross-iteration data dependencies. Such dependencies or even dependence cycles may be tolerated as long as there is an iteration-independent component. The compiler just needs to place the instruction(s) causing a cross-iteration dependence into blocks which are committed in order. This is illustrated in the example below.

Example 2. Let us replace the body of the loop in Figure 1(a) with the following statement: A[i] = A[i-2] + B[i] / C[i], as seen in Figure 3(a) (we use the source code level and omit the loop controls in this example for simplicity). Block B1, which contains the index and address calculation as well as loop control instructions, corresponds to two blocks B1a and B2, and block B2 corresponds to the block B1b in

the Example 1. In this example, the store to A[i] goes into block B3, and will be serialized together as in Figure 3(b). It is thus guaranteed that block B3 in iteration *i-2* has committed before a load of A[i-2] is executed in block B3 of iteration *i*. Block B2 of different iterations, meanwhile, are independent of each other, hence they can be committed out of order. This moves enforcement of dependencies to the block level. The compiler can also apply this mechanism to nested loops which may further increase performance.

The approach proposed here requires modifications to a standard OOO core for OOO commit and early resource release. The main modifications are the addition of the block table and associated control logic, tracking block information for registers and instructions for early release, and removing ROB/LSQ entries not from the top. The in-order commit mechanism is fully preserved and is the default execution mode allowing backward compatibility. Another change required is the exception handling in OOO commit mode.

Memory disambiguation is also helped by the knowledge of data independence between blocks as it implies absence of cross-iteration dependencies between stores or between stores and loads in different iterations. Combined with the serialization of data-dependent block execution this guarantees that store-to-load replays are not possible across blocks. Other memory ordering issues are not considered here, but relaxed consistency models are preferred as they lead to higher OOO commit performance when stores can be committed out of order.

Similarly, the use of physical registers is also tracked at the block level and allows a register to be released after all the blocks that use it have committed. This identifies a range of blocks in which a physical register mapping is alive. No additional checkpointing of register maps is required given renaming based on physical to logical maps. Thus even with OOO handling of replays and exceptions, e.g. when an earlier block has an issue after younger blocks have committed OOO, the architecture guarantees that all required physical registers for an earlier block have not been released.

The proposed OOO commit mechanism has been implemented in the M5 simulator [3] and evaluated using a subset of six SPEC2000 and 2006 benchmarks. Optimized assembly code generated by a compiler was manually modified for a small number of loops in each program. The modifications consisted largely of inserting the appropriate block markers and moving instructions with dependencies into sequential blocks. Such manual modifications were necessary since we did not have a compiler and were quite time consuming. Thus we could only "compile" a small number of benchmarks and loops in each benchmark, selecting loops with a significant fraction of total execution time (compilation is planned for future work). The code remains a correct sequential code if these added marker instructions are ignored. The results show that, on

average, 52% instructions were committed out of order while in OOO commit mode. The percentage will be further increased by compiling and executing all loops in a benchmark in the OOO mode. The performance improvement from adding OOO commit to a reasonably standard 4-wide processor ranged from 10% to 26%. The hardware overhead required for the OOO commit is very small.

The rest of the paper is organized as follows. Section 2 discusses compiler support required. Section 3 describes the microarchitecture changes and OOO commit operations. Methodology is presented in Section 4 and experimental results are presented in Section 5. Related work is discussed in Section 6 and finally Section 7 presents conclusions.

2 Compilation

This section describes the compilation process envisioned for OOO commit. The goal is for a compiler to identify or create via program transformations independent blocks to which the OOO commit can be applied. The approach targets program loops since they typically take the bulk of execution time. But it can also be applied in other contexts, e.g. multiple independent blocks in one iteration. Recall also that within a given block standard in-order commit mechanism is used and all control and data dependencies are thus satisfied. Thus the compiler only needs to identify and pass to hardware the dependence information between blocks. Lastly, it is the dynamic instances of various blocks and their dependence relationship that is important for the OOO commit.

The simple loop example in Figure 1 represents one loop type that can be (automatically) compiled for OOO commit. Many other loop types can also be compiled, including "while" loops, reduction loops, nested loops, loops with reductions plus other code, etc. For instance, a complex nested loop from *equake* is shown in this section. It is a nested loop in which the inner loop is a while loop, and there are dependencies from the inner loop to the outer loop. It also uses pointers. Such loops may require user assertions to assist the compiler, similar in spirit to the parallel loop or section directives in OpenMP. Note also that, in general, independent blocks can be exposed without loops, i.e. the OOO commit mode can be entered with just a sequence of blocks that are independent, i.e. in loops whose iterations are too large to fit in the ROB.

Finally, note that traditional compiler optimization techniques such as loop unrolling or modulo scheduling would not be applicable in many of these types of loops. Thus such loops cannot obtain performance gains comparable to the OOO commit from traditional optimizations.

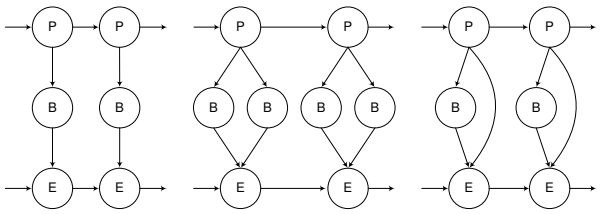


Figure 4. Examples of block types and their dependencies

Let us start by defining more precisely the various block types the compiler needs to identify to enable the OOO commit processing by the architecture.

2.1 Instruction Blocks

Recall that for the purpose of OOO commit a loop is divided into blocks of different type with specific dependence relations between the blocks. The commit behavior of an instruction is determined by the type of block it is in. Based on the dependence and compilation properties of blocks, three block types are defined: prologs, bodies and epilogs, as illustrated in Figure 4.

A *prolog* is a block dependent on previous prologs only. Prologs are used to resolve dependencies between two iterations, and all the blocks after a prolog can not be committed before it has been committed. Also, prologs are committed in order with respect to each other. They often contain instructions which compute loop index and memory addresses or branch instructions which determine loop termination (i.e. prologs are control dependent on each other).

A *body* is dependent on previous prolog(s) only. Any two bodies are independent of each other. Bodies can commit as soon as all previous prologs have been committed. A typical body often has one or more long latency load instructions followed by computation instructions which consume load results.

An *epilog* is dependent on previous epilogs and bodies. Similar to a prolog, an epilog is used to resolve dependencies between two iterations, but the difference is that no blocks other than future epilogs are dependent on the current epilog. Epilogs are also committed in order with respect to each other. Due to the fact an epilog can only be committed after all previous blocks have been committed, epilogs are committed only after they reach the top of the ROB. An epilog usually contains instructions with a loop-carried dependence.

Using the above definitions, blocks of the loop in Figure 1 are classified as following: B1a and B2 are prologs, B1b is a body (there is no epilog in this example); and blocks of the loop in Figure 3 can be classified as following: B1 is a prolog, B3 is an epilog, and B2 is a body.

Memory dependencies may not be known until execution time when the addresses are computed. Two *potentially dependent* memory instructions must be placed in the same body block (where in-order commit is used) or in blocks of the same type - a prolog or an epilog (which are committed in order). This guarantees that such instructions are committed in order with respect to each other. Note that if the above cannot be accomplished then the compiler does not generate OOO commit code for the loop.

2.2 Optimization

The block types defined above and their implied dependence relationships can be used to further optimize programs for the OOO commit mode. First, they may be used to compile nested loops, i.e. the OOO commit mode will start upon entering the outer of the nested loops. This reduces the overhead of switching between in-order and OOO commit modes that would occur if the OOO commit applied only to the innermost loop. This allows inner loops with very small bounds but nested to take advantage of OOO commit.

A loop iteration often contains multiple independent load instructions. Such loads and their dependencies have to be committed sequentially within the same body even in the OOO commit mode. A way to optimize this (by user or compiler) is to partition a body into sub-bodies to increase the chances of finding independent, ready instructions to commit. Thus a loop can be compiled to consist of a prolog, multiple bodies and an (optional) epilog. This is very effective in loops which contain multiple arrays. Each array in such a loop has its own memory hierarchy behavior. Loads which hit in caches are committed fast and their dependent (sub) block instructions can start committing while other blocks may still be waiting for data to arrive. When (sub) bodies have instructions which introduce dependencies between them, such instructions are moved to epilogs.

An example showing a complex nested loop and its blocks using the "full power" of the block types defined in this section is shown below.

2.3 Example: *equake*

Figure 5 is a simplified loop from *equake* that starts at line 1195 of *quake.c*. This is a shortened version of the loop in which the original three-dimensional vectors are represented as one-dimensional ones and only one sub-block is shown in the inner loop. Also, the loads and computations are replaced by "functions" F1 and F2 to simplify the figure.

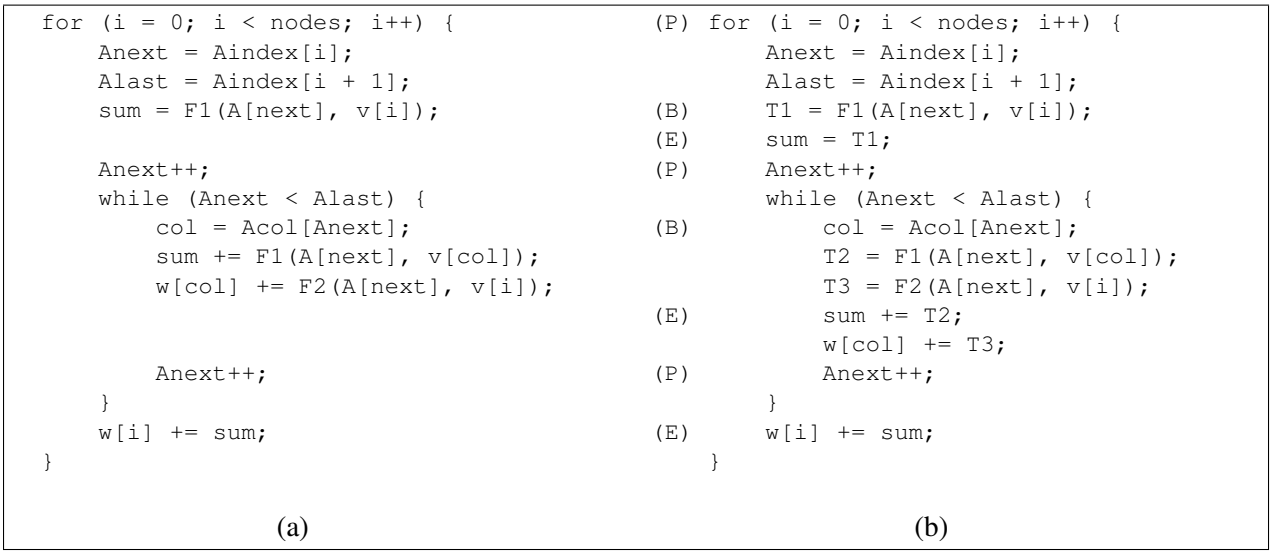


Figure 5. A (shortened) loop nest from equake (a) and its block types (b).

The loop is a nested loop, with a "while" inner loop. Its termination condition is computed in the body of the outer loop. The inner and outer loops contain reductions. There are also dependencies from the outer loop to the inner loop. Functions F1 and F2 are independent in terms of outputs and can be put into the bodies, as seen in Figure 5(b). The computation of loop indices and termination conditions (`i, Anext, Alast`) are serialized by putting them in the prologs. Note that `col` is the vector index, and it is a temporary variable, hence it is located in the body. The outputs of F1 and F2, which are written into T1, T2 and T3, are independent in different iterations, hence they are located in the body. The reduction computation of `sum` has the dependencies from the outer loop to the inner loops, therefore it is put in the epilogs. The stores of outputs to the vector `w` may introduce memory dependencies between different iterations of the inner loop and the outer loop. Hence they must be in the epilogs as well.

2.4 Relaxing Some Dependency Conditions

By the time instructions reach the commit stage some dependencies are already resolved during the execution. For example, if the results of two instructions A and B, which belong to different bodies, are consumed by instruction C. Due to this dependency, C can not be in the same body with A or B, but can only be in an epilog. This enforces that C can commit only after it reaches the top of the ROB. However, one can observe that the dependencies of C to A and B are resolved during instruction scheduling: C can only be issued after A and B are executed as it needs results from A and B. Thus it is safe to commit C after

it is executed, regardless the status of A or B in other iterations (C is still committed after A and B of the same iteration). This means that, it is safe to move C to the same body with A or B.

The above relaxation will help in different iterations. Let us consider two iterations *i* and *i+1*, and denote A_i as instruction A of iteration *i*. In the above example, if C is in the epilog, C_{i+1} must be committed after C_i and A_i and B_i. However, if C is moved to a body, C_{i+1} is allowed to commit before A_i, B_i and C_i.

The code in Figure 5 uses `sum` which is a scalar and is stored in a register. The assignment of `T1` to `sum` is computed before the accumulation of `T2` into that variable. Therefore, these two statements can be safely moved to the bodies.

By allowing more instructions to be moved into the bodies, size of the epilog can be minimized. This is very useful as instructions in the epilogs occupy the ROB and other resources longest, until they reach the top of the ROB. Another application is to break a body into sub-bodies. In both cases, the technique allows more instructions to be committed out of order.

However, this optimization may reduce the chance of committing instructions early due to the in-order commit of instructions in a block. For example, instruction I1 at the top of block A is dependent of instruction I2 at the end of block B will prevent later instructions in block A to commit until instruction I2 and instruction I1 are executed, and instruction I1 has been committed. This can be avoided by moving the instruction I1 down to the bottom of block A.

2.5 Creating Instruction Blocks

Compilation starts by analyzing dependencies between iterations. Instructions which have inter-iteration dependencies are placed into either a prolog or an epilog. The remaining instructions, which are independent, belong to bodies. An instruction producing inputs for instructions in the body is placed in the prolog, an instruction consuming results produced in bodies is placed in the epilog. The body is divided into sub-bodies by analyzing the load instructions. A body often starts with one or more load instructions which have the similar memory behavior (i.e., accessing to the same cache line), followed by dependent instructions. During this process, the techniques described in Section 2.2 and Section 2.4 may be applied. After type of each instruction is recognized, the instructions are rearranged into blocks, and the compiler inserts *markers* at the beginning of each block.

A set of additional instructions is assumed available for "marking" block boundaries. These "marker" instructions (or *markers*) are inserted into the binary. The markers do not need to consume registers during

renaming, require functional units during execution, or raise exceptions. One type of markers is used to indicate the start and the end of the OOO commit mode. There are two such markers called *loop markers* because they are typically associated with a loop, a *BLoop* marker and an *ELoop* marker. Loop markers are entered in the ROB and the OOO commit mode starts after the BLoop is committed and ends after the ELoop is committed.

The second type of markers are used to identify blocks and are called *block markers*. These markers are inserted into the binary before the first instruction of a block to indicate the start of the block and the type of the block. Unlike loop markers, block markers are discarded after the decode stage so that they do not use the bandwidth of later stages. When a block marker is seen in decode it causes the creation of a BT entry and all instructions until the next block marker are considered part of the same block.

The markers are additional instructions in the OOO commit binaries that are not present in the baseline version. They are overhead required by this architecture. However, our results show that this overhead is small and that performance improvement is achieved even in the presence of this overhead.

3 Architectural Support

This section describes the necessary architectural support to enable OOO commit while completely preserving the in-order commit capabilities. In fact, the changes described are additions to the in-order units. Most of the modifications required for OOO commit are in the commit stages of the pipeline. One change required in the front end is decoding/identifying the commit mode change and block marker instructions. Another change enables recording for each physical register the block number of the instruction that currently maps it and (eventually) the next block that remaps the same logical register. The ROB and the LSQ are also modified to support the OOO removal of instructions. Overall, the amount of additional hardware required is minimal.

3.1 The Block Table

A new structure, the *Block Table* (BT), shown in Figure 6 is added to store dynamic information about blocks and is used in conjunction with the ROB for OOO commit. Each BT entry corresponds to a block of consecutive instructions starting with a block marker. Similar to the ROB, which is used to manage the commit of instructions, the BT controls the commit of instruction blocks. Entries are inserted into the BT at

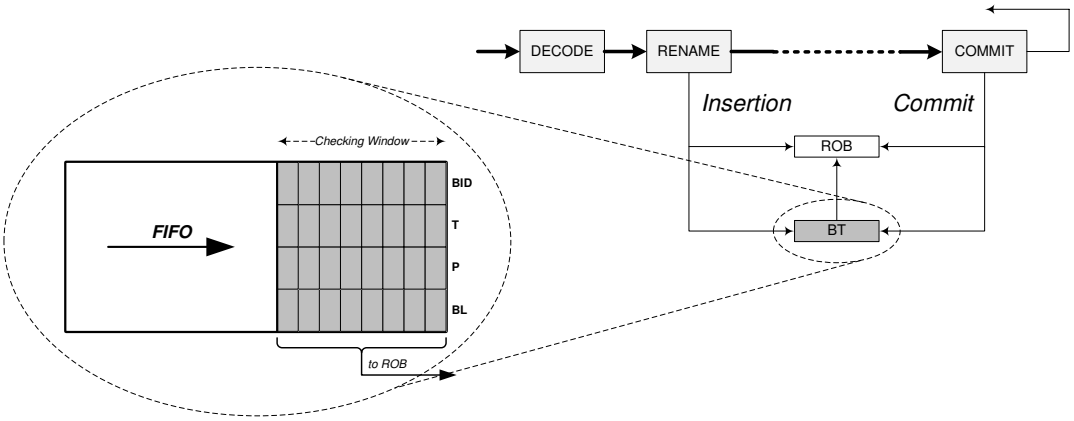


Figure 6. The Block Table.

the same time as the insertion of instructions into the ROB.

The BT is similar to the Group Completion Table (GCT) in the POWER4 processor [23] in that the GCT also holds information about a group of instructions and it controls the commit of instruction groups. However, our OOO commit architecture does not require the extra logic to detect instruction blocks, this is done by the compiler using the inserted markers. This reduces significant hardware overhead required to form the blocks from the instructions, and allows larger blocks to be processed.

A BT entry describes one instruction block and has the following fields:

- *Block ID* (**BID**), a unique ID used in releasing registers and entries from the LSQ;
- *Block Type* (**T**), e.g. a prolog, a body, or an epilog;
- *Position* (**P**), the location in the ROB of the first instruction of the block; and
- *Begin Loop* (**BL**) flag which is set to 1 if the entry is the first one of a loop. This flag is used to support more than one in-flight loops.

Instructions may be committed out of order in the backend if they are in different blocks. Instructions within a block are committed in order and an instruction at the top of the block, which is pointed by its BT entry, is checked for readiness. If the instruction satisfies the ready-to-commit conditions, it is committed regardless its age. The ROB and the BT are then updated at the end of the cycle. An entry in the BT is ready to be committed after all instructions in its block have been committed (subject to some additional conditions).

3.2 The BT Operation

Inserting entries into the BT. In the decode stage, when a block marker or a BLoop is decoded, a new BT entry is created (the ELoop marker is used by commit stage only) along with **T** and **BL**. A BLoop is passed to later pipeline stages while a block marker is discarded. A BT entry is inserted into the BT in parallel with the insertion of instructions into the ROB. **P** is set to position of the first instruction in the ROB.

The **BID** is assigned using a *block counter* with a range much larger than the BT size so that the **BID** is a unique block identifier. The processor is stalled when the BT becomes full, similarly to when other structures (e.g., ROB, IQ) are full. The front end resumes the insertion of instructions into the ROB and blocks into the BT when there are free slots in the BT. Our results show that such stalls are an infrequent event, even with a small BT, because the number of entries in the BT is much smaller than the number of instructions in the ROB.

OOO Commit. The processor starts program execution in the conventional in-order commit mode (or sequential mode – *S-mode*). Instructions in the ROB are committed in the *S-mode* until the BLoop marker is committed from the top of the ROB. The commit mode then changes to OOO mode (*O-mode*) allowing instructions from different blocks to be committed. Only an instruction pointed to by the **P** field of a BT entry can be committed from this block after which the pointer is updated to point to the next instruction.

A small number of BT entries within a *Checking Window* of size *W* at the top of the BT are checked for ready to commit instructions. The **BL** flag is used during this process. W must be small to keep the BT logic simple and implementable and our results show that 96% of the ready instructions are covered with $W = 8$.

An instruction pointed to by the **P** field of a BT entry within the checking window can be committed OOO *if and only if*

1. The instruction has executed;
2. It is a memory access instruction and disambiguation has been performed (see Section 3.3 for details); and
3. The BT entry type is prolog or body and there is no older block in the BT of type prolog; or it is an epilog which is at the top of the ROB.

The logic required to check the above is not very complex and can be viewed as the part of the first

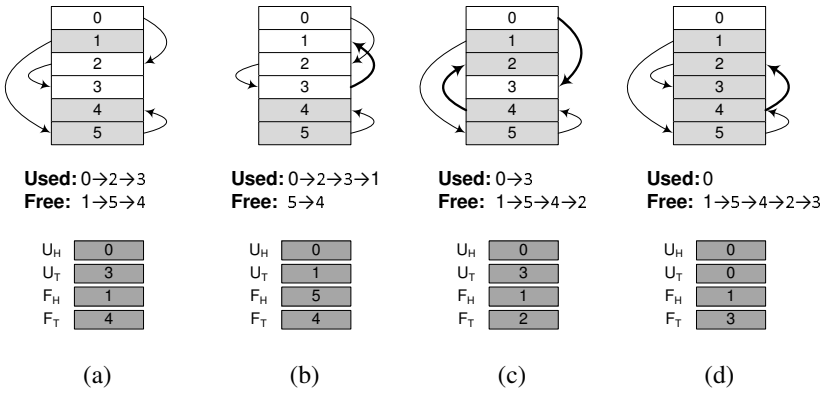


Figure 7. A linked-list ROB: (a) original state, (b) after insertion, (c) after commit, and (d) after squashing.

commit stage. Once "committable" instructions are thus identified via the BT checking window they are passed on to the ROB/LSQ for commit and resource release, subject to commit bandwidth. The **P** field is updated to point to the next instruction in the block. A block entry may need to be removed from the BT as a result of instruction commit. The BT and the ROB are "compressed" after the OOO removal of instructions and BT entries (as described below). The process continues until the ELoop is committed. The last entry is removed from the BT and the commit mode returns to *S-mode*. The above process can be easily pipelined.

Compressing the ROB and the BT. Instruction commit from the ROB and entry removal from the BT may leave empty slots in these structures. A mechanism to deal with this was described in [2] in which the ROB is collapsed by shifting instructions to fill the gaps. However, the compression requires complex hardware to move the ROB entries. Also, it prevents one from using the ROB position to identify an instruction. The OOO commit architecture proposed here uses a *linked-list ROB*, with two linked lists - a *used list* and a *free list*. Each ROB entry is augmented with a pointer and head/tail of each list are maintained.

Figure 7 shows a linked-list ROB with 6 entries. An ROB (the top structure of each figure) used list links in-flight instructions in the order of time. U_H and U_T are head and tail pointers of the used list, while F_H and F_T point to the head and tail of the free list. The initial state of the ROB is shown in Figure 7(a) and three different cases of list management are shown: insertion a new instruction to entry 1 in Figure (b), the OOO commit of an instruction in entry 2 in Figure (c), and the squashing due to mis-speculations, mispredictions or exceptions from entry 2 in Figure (d). In each case, the updates to the pointers are shown by the bold lines.

Insertion of a new instruction (Figure (b)) or removal of a committed instruction (Figure (c)) requires

only simple head and tail pointer manipulation of the free and used lists plus the update of one ROB entry pointer. BT entry update may also be required.

Squashing of an instruction removes all younger instructions till the tail of the used list. In this case the canceled instruction and all younger instructions in flight are added to the free list. Figure (d) shows this. The actions required for maintaining the linked list ROB are also not on the the critical path and should not affect processor clock. This is because insertion in the window and commit are already taking several cycles.

A similar mechanism can be applied to the BT where entries are also removed from the middle of the structure. However, fragmentation can only happen in the first W_{max} entries of the BT and an alternative design which exploits a collapsing mechanism can be used. In the latter case the collapsing only happens at the "top" of the BT while the later entries are organized as a FIFO (see Figure 6).

Squashing instructions and instruction blocks. When a misprediction or a misspeculation occurs, instructions are squashed from the ROB and blocks are squashed from the BT starting from the problem instruction and squashing younger instructions and younger blocks, as described above. The block counter is reset to the **BID** of the block which has the problem instruction. Register file update is described in Section 3.4.

3.3 OOO Memory Instruction Commit

There are three issues to consider when committing memory instructions out of order: store-to-load-forwarding, load ordering and store ordering. All of these are present in a standard LSQ of in-order commit processors, thus we only discuss the necessary changes. A replay mechanism is assumed for dealing with ordering violations.

For *store-to-load forwarding* the proposed architecture allows the queue tag search to be narrowed using the block information. A store and a load with a potential dependence which cannot be eliminated at compile time are placed by the compiler either in the same block or in prologs/epilogs where they are committed in order with respect to each other.

Each entry of the LQ and SQ is extended by two fields to describe its block ID **BID** and block type **T** (see Figure 8). They are set to the values in the BT when a load or a store instruction is inserted into the LSQ . Load disambiguation is performed such that *a)* for a load in a body, only older stores with the same **BID** are checked or *b)* for a load in a prolog or an epilog, only older stores in the same block type are checked.

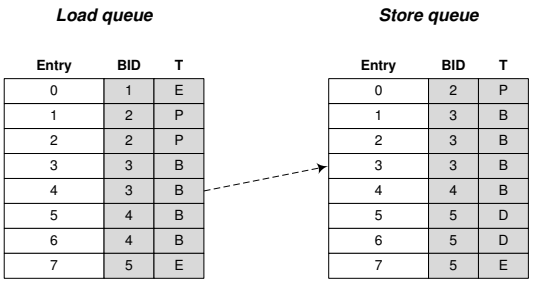


Figure 8. Support in the LSQ. Extensions to the queue are shown in the dark area.

In the figure, when entry 4 of the LQ (which is in a body) is checked for readiness, only addresses of the SQ entries 1, 2 and 3 need to be compared as they are in the same body, but not that of entry 0 as it belongs to another block (prolog in this case). By using this information from the additional fields as provided by the compiler, the CAM search can be faster for entries that cannot match.

The store and load ordering constrains depend on the memory consistency model. We do not model multiple processors in this work but assume a weak consistency model and support for memory barriers.

3.4 OOO Register Release

During in-order commit when an instruction that writes a logical register commits, the previous physical register that mapped the same logical register is released. Under in-order commit this guarantees that *a)* a new value is created for the logical register and *b)* the previous value in this logical register has been read by all dependent instructions. This technique does not work with OOO commit and/or early register release. However, register release may not have to wait until a logical register which maps a physical register is renamed to another physical register (assuming it is not the current renaming) if one uses block type information. Two problems need to be solved first. One is tracking the physical register uses and the other one is restoring register state on mis-speculation or exceptions which occur out of order.

Tracking register use by dependent instructions via reference counters [1] or a dependency matrix [9] has been proposed, each with its own issues. The architecture proposed here uses block commit information to drive the register release.

The main problem is that a physical register P may be released out of order and before it was read by all dependents. P can then be mapped to a new logical register and a new value written into the register. For this reason early release of physical registers was not allowed in [2]. Again, using the block properties, registers can be released after they are no longer used.

The architecture proposed here associates two block ID's with each physical register. The first one is the **BID** of a block which *maps* the physical register, and the second one contains the **BID** of a block which *re-maps* the same logical register. The register is *written* and may be used by all the blocks between these two blocks. The register is safe to release only after all these blocks have been committed. One should also note that block types are not taken into account when releasing the registers.

However, blocks may be committed and removed from the BT out of order requiring a mechanism to determine whether all the consumer blocks have been committed. This is accomplished here by using a block reference count via a mechanism similar to [7]. Each physical register uses a counter which counts the blocks between the allocating block and deallocating block. In the above example, the initial value of the counter to the register P3 is set to 0 when it is mapped to the logical register R2 at the instruction I2. The counter is incremented when a new block is inserted after that, which are block C and D (assume that no instructions in block B consume this register, the other counter is incremented as well). The counting stops when R2 is renamed again at instruction I4. When block C or block D are committed, the counter is decremented. When the counter reaches 0 again, the mappings R2 to P3 can be released.

Misspeculation or exception handling is different for OOO commit only if it occurs in an older block A and after younger blocks 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]. OOO (or in-order) register release in this case simply removes its mapping. This does not affect any older live mappings. A reference counter for each register is updated as per [1] as canceled instructions are removed from the ROB. Note that blocks that commit out of order both increment and decrement the reference counters and thus do not affect the mechanism.

3.5 Exceptions

Standard OOO processors commit instructions in program order. This is required to maintain sequential program semantics and, in particular, maintain precise exceptions with relative simplicity. Specifically, in-order commit allows all instructions after an exception to be canceled and later re-executed starting at the exception PC. OOO commit makes maintaining precise exceptions more difficult.

Consider a case when an exception occurs in an older iteration i while (part of) a younger iteration $i+1$ has

already been committed and its (ROB, LSQ, RF) resources released. A restartable exception should cause cancelation of all instructions in the iteration *i+1* and their re-execution later on. But results of iteration *i+1* were already committed and thus re-execution may lead to an error. For instance, the following code cannot be re-executed as it will result in an incorrect value in memory:

```
ld  R4, 4(R8)
add R4, R4, R2
st  4(R8), R4
```

However, the loop and block properties again help solve the problem. Recall that our execution model allows the OOO commit in the "data independent" blocks of an iteration but commits instructions in the sequential part of all iterations in sequential order. This means that exceptions in the sequential part of all iterations remain precise. Furthermore, the register state is also precise in a sense that all registers needed by an instruction in iteration i have not been released.

Exceptions in the data independent blocks are made precise by the architecture and compiler. The approach proposed here for achieving this is to handle the exception and then restart execution. This will involve potentially re-executing all instructions from data independent blocks committed out of order. Such blocks are dependent only on prologue blocks and thus can be re-executed and produce the same results subject to solving the problem of re-execution described above. The source registers used in such blocks either come from outside of a loop (e.g. globals) or from prologs. None of these registers could have been released.

This approach results in "partially precise" exceptions, i.e. the values in all registers and memory locations produced by all instructions in a loop prior to an exception are precise. But other memory locations may have been updated by later instructions. However, this is sufficient to allow software or hardware exception handlers to deal with the exception and guarantees that the state will become completely precise again. For instance, the virtual memory or f.p. exceptions can be handled this way. A software trap would be decoded before later iterations and force in-order commit. A trap from a keyboard interrupt generated for debug can be delayed until in-order commit mode is entered.

The problem of re-executing memory operations boils down to stores with a WAR dependence to any load within the same iteration, e.g. A[i] = A[i] + 4. Such dependencies can not exist between data-independent blocks. The approach proposed here is to simplify the hardware and rely on the compiler to detect such stores (anti-dependencies within an iteration) and move them to the epilog of an iteration.

Parameter	Configuration
IQ (Total FP and INT units)	96
ROB	128
LQ/SQ	32/32
Registers (Int/Float)	160/160
BTB entries	4K
Frontend/Execution/Backend widths	4/6/4
DCache	64KB, Assoc 8, Line Size 128, 2 cycles
ICache	32KB, Assoc 8, Line Size 128, 2 cycles
L2Cache	4MB, Assoc 16, Line Size 128, 20 cycles
Memory latency	300 cycles

Table 1. Processor configuration.

Precise exceptions can be enforced on stores with compiler assistance (using a compiler flag). This would require bodies of type B1b to not contain stores, instead all the stores would be moved to blocks of type B2 (epilogs). This will enforce their sequential execution across iterations. Finally, stores may be forced to commit in order by hardware. This may not even affect the performance much, as some programs (such as *mgrid*) have very few stores. But other programs may see lower performance.

4 Methodology

We used the M5 simulator [3] to study the OOO commit technique. M5 is a detailed execution-driven simulator for system-level architecture and processor microarchitecture. The simulator was configured to target Alpha 21264 architecture [13, 18]. Selected parameter settings are listed in Table 1.

A set of six benchmarks from SPEC2000 and 2006 benchmark suites was used in our study. As mentioned above, the choice was dictated by our ability to hand-compile a small number of loops in each application such that the speedup from OOO commit in these loops would noticeably reduce the application's execution time. This researched focused on the OOO commit architecture and its potential, compilation for OOO commit architecture is left for future work. A compiler will enable us to compile all benchmarks in the suite and also every profitable loop in a benchmark, leading to even higher performance gains.

Benchmark description is given in Table 2. Profiling was used to identify the loops with significant percentage of overall execution time and these are shown in the last column of the table.

A baseline and an OOO code versions were created for each benchmark. Both the baseline and modified benchmarks used software prefetching in all loops executed in the OOO commit mode. All benchmarks were

Benchmark	Suite	Type	Language	Loops (line number)
art	SPEC2K	Floating point	C	584 in *scanner.c*
equake	SPEC2K	Floating point	C	1195 in *quake.c*
mgrid	SPEC2K	Floating point	Fortran	149, 189, 230, 270, 291 in *mgrid.f*
swim	SPEC2K	Floating point	Fortran	261, 315, 397 in *swim.f*
libquantum	SPEC2K6	Integer	C	89 in *gates.c*
bwaves	SPEC2K6	Floating point	Fortran	168 in *block_solver.f*

Table 2. Benchmark information.

compiled using GNU Alpha compilers with optimization flag `-O3`. A selected source loop was identified in the assembly and loop and block "markers" were manually added to it. Instructions were re-ordered in some cases to satisfy OOO constraints, e.g. to move dependent instructions into epilogs. The modified assembly was then assembled for use by the simulator.

In each experiment, the caches were warmed up for 500M instructions after skipping initialization. After that each benchmark was run for at least 500M instructions. To make a fair comparison between the two code versions, the benchmarks were run until they reached the same point (i.e. after a number of iterations) instead of until they committed the same instructions as the number of instructions is different for the OOO version.

5 Results and Analysis

This section presents simulation results for the OOO commit architecture. The following statistics are presented and discussed: the percentage of instructions committed in the *S-mode*, instructions committed from prologs, bodies and epilogs, registers released early inthe *O-mode* and statistics on BT operations. Finally, speedups over the baseline OOO processor are presented.

Figure 9 shows the percentage of committed instructions in the standard commit mode (the *S-mode*) and from prologs, bodies or epilogs in the *O-mode*. On average, 29% of instructions are committed in *S-mode*. The number is larger for three benchmarks, *art* (39%), *equake* (68%) and *libquantum* (35%). Note that this very dependent on the number of loops processed for the OOO commit.

The average number of instructions committed in the *O-mode* in prologs, bodies and epilogs are 20%, 48% and 3%, respectively. The OOO commit is successful because it commits instructions from prologs and bodies out of order and releases their resources (LQ, SQ, RF and ROB entries). Note that the number of epilogs is small compared to prologs and bodies because epilogs only contain instructions which might

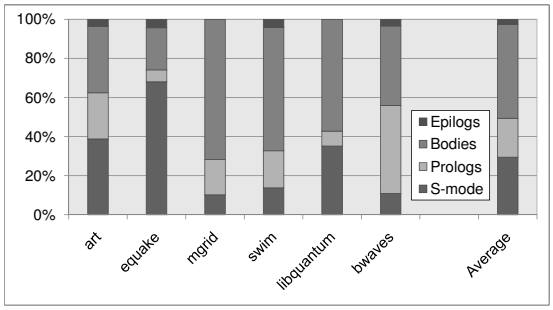


Figure 9. Distribution of committed Instructions.

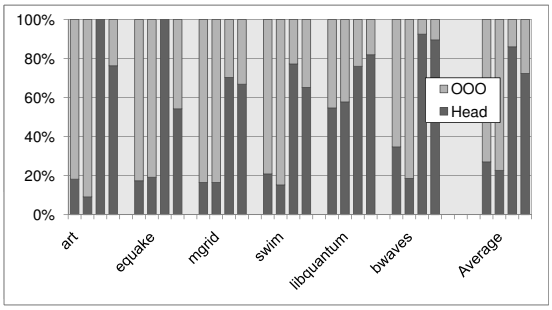


Figure 10. *From left:* **instructions, loads, and stores committed and registers released in the O-mode.**

have inter-cross dependencies. In some benchmarks, e.g. *mgrid* and *libquantum*, there are no epilogs as loop iterations are completely independent.

Statistics about instructions committed in *O-mode* are shown in Figure 10, the leftmost bar for each benchmark. Such instructions may release the resources faster and result in program speedup. On average, the percentage of instructions committed out of order is 73% while in *O-mode*. The remaining 27% are committed from the top of the ROB ("Head"). Overall, 52% of all committed instructions (accounting for the *S-mode* commit) are committed out of order.

The second and third bar for each benchmark in Figure 10 show the percentage of loads and stores committed out of order vs from the top of the ROB ("Head"). The percentage of store instructions committed out of order is small for several reasons. First, stores in the epilogs are committed in order. Second, store instructions are often the last instruction of a body block and the last instruction is often committed from the top of the ROB. This happens when a block becomes ready to commit after waiting for data on a cache miss and reaching the top of ROB.

The last bar of the figure shows the percentage of registers released out of order. Overall, the number released out of order is smaller than for other categories. OOO commit increases register pressure and OOO release would be very helpful. Of the registers released out of order, many are renamed multiple times within a block (around 65% of OOO released registers). Registers used across multiple blocks take longer to release, for instance registers in prologs. Short loops may require more registers.

Figure 11 shows BT statistics. The leftmost bar (*occupancy*) is the average number of entries in the BT while in the *O-mode*. This indicates that the BT can be rather small. The next bar (*search size*) shows the average position in the BT past which there are no blocks with instructions that can be committed out of

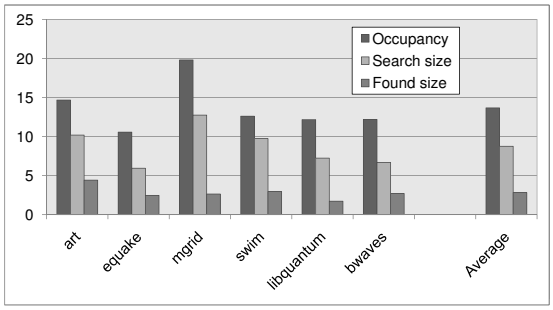


Figure 11. BT operation statistics.

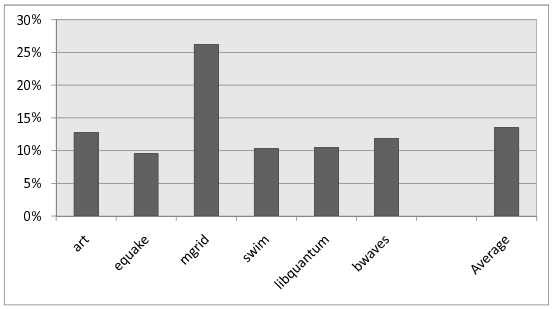


Figure 12. Speedup.

order. This indicates that the size of the BT search window can be small. The last bar (*found size*) shows the BT position where the first block with an instruction ready to commit is found and such instruction is committed. This confirms that most of the OOO commit happens in the top BT entries.

The benchmark *mgrid* has a large BT occupancy because its loop body was broken into sub-blocks. Multiple blocks of type B1b expose more instructions ready to commit while in a single block instructions commit is in order.

Figure 12 shows the speedup from the OOO commit over the baseline architecture or, more precisely, over the unmodified benchmarks. One should keep in mind that, due to manual insertion of marker instructions into the assembly code, the modified loop code in the benchmarks is no longer highly optimized. Results show that the minimum speedup is 10% and the maximum is 26%. The speedup clearly depends on the number of instructions committed in the OOO commit mode. The lowest speedup is in *equake* which commits only 32% of instructions in *O-mode* (see Figure 9). *mgrid* has the highest speedup as the percentage of instructions in the *O-mode* is high Also, *mgrid* does not have epilogs, which are committed in order and thus occupy the ROB and other structures for longer than prologs and bodies.

In summary, the proposed OOO commit mechanism benefits from (1) the number of instructions committed in the *O-mode* (and the number of loops compiled for OOO commit), (2) the number of outstanding commit pointers, and (3) different behavior of memory instructions belonging to different instruction blocks, e.g. hits vs misses.

6 Related Work

OOO commit has been studied [2]. It examined constraints under which instructions can be committed regardless their age and proposed a mechanism to commit instructions early. The constraints limit the

number of instructions which can be committed out-of-order. Extra hardware was proposed to detect WAR dependencies in order to release registers early. Second, load instructions can become committable only after all previous stores have resolved addresses. Third, the constraints in exception handling in their work reduces the number of instructions committed out of order significantly. Our work employs block properties to relax these constraints and reduce the hardware overhead needed to support OOO commit.

Other studies have been proposed to early release of resources [17, 5, 21]. Cherry [17] decouples instruction commit and resource recycling. Resources are released and put back in use when they are no longer needed. The work distinguished the reversible instructions and irreversible instruction using *Point of No Return* (PNR). The PNR points to the youngest instructions which are no younger than the oldest instructions which might cause misprediction or misspeculation. Reversible instructions, which are no older than the PNR, are handled as in conventional mechanism. Irreversible instructions can release resources early. If exceptions happen, the whole processor rolls back to a dedicated checkpoint. The LSQ and the RF are also redesigned to support the scheme. The work in [5] employs checkpointing mechanism. Register release and store retirement, which would change the processor state, are not done until checkpoints are committed. If exceptions occur, the processor state rolls back to a previous checkpoint and restarts execution from that point. *Validation Buffer* (VB) was proposed in [21] as a replacement for the ROB. Instructions are inserted into the VB in program order and leave when they are no longer speculative. To support this, a program is divided into *epochs* and each has an *epoch initiator*, which contains instructions which might cause problem. Instructions in an epoch are allowed to leave the VB only after instructions in the epoch initiator are completed. The concept of epoch initiators is somewhat similar to prologs in our work. A major difference between the above prior work and ours is that, it used hardware to support their techniques, while in our work compiler support is employed. Also, our approach does not require checkpointing.

Mechanisms to release registers early have also been studied [19, 7, 10] for in-order commit processors. In these architectures a register is released as soon as the last consumer has read data from it. In [19], a register is attached to the last instruction which reads it in the program order. The register can be released when this instruction is committed. However, it is not easy to know which instruction is the last consumer of a register due to possible mispredictions or misspeculations. The work in [7] used a counter to keep track of the number of instructions which read a register. A new consumer is recognized in rename incrementing the counter and a consumer commit decrements it. The register can be released when it has the value of 0. Misprediction/mispeculation make maintaining such reference counters difficult. In [10], the authors

observed that many registers are read only once after being written. Using the compiler to identify these registers, the pipeline is modified to support the their early release after they are read.

There has been a large body of work on exploiting loop properties. Loop processor architecture [8] detects and captures the loops in front-end stages, and keeps them in the loop window after the first iteration, allowing loop instructions to be fetched, decoded and renamed once. Later iterations are fetched from the loop window, hence saving bandwidths of frontend stages. This work is orthogonal to our work, as we aim to improve the commit stage. In the compilation domain, the work in [24] explored parallelism in DOALL loops using optimizations during the compilation process. The optimizations include loop fission, prematerialization, and isolation of infrequent dependencies. While the focus of this work is multicore domain, some of these techniques can be applied in the compilation process for our proposed architecture.

Long-latency misses are one of the critical issues which have been studied for a long time [15, 14, 20, 6]. To overcome this problem, the load instruction and its dependencies can be removed from the main pipeline to a buffer [15] or speculatively retired [14], allowing other instructions to proceed. In [15], the load and its dependencies are moved to an aside buffer and are moved back to the main pipeline once the load has its data. In speculative retirement [14], value prediction [16] is employed to predict the result of the load early. If the prediction is recognized to be wrong later, the pipeline is rolled back to a dedicated checkpoint. Another proposal [20] aims to warm the caches while the whole pipeline is blocked by long-latency misses by entering *runahead execution* mode. Instructions are then executed, but their results are not committed. The approach in [6] enlarges effective sizes of critical structures, increasing the number in-flight instructions in order to hide the miss latency. Our work is orthogonal to these proposals as well, as they emphasized execution stage, while we focused on commit stage.

7 Conclusions

The in-order commit mechanism is overly conservative as instructions are committed only when they reach the top of the ROB. After a long-latency miss instruction reaches the top of the ROB the processor stalls and performance is significantly reduced. This paper proposed an out-of-order commit mechanism using compiler support. The work is based on the observation that loops are very common in programs and they can be used to support out-of-order commit mechanism by tracking loop code blocks. The paper defined three types of blocks that allow iteration-independent code to be separated as well as to be able to

process complex multiply-nested loops. These blocks are ”marked” by the compiler and imply certain types of dependencies that hardware can infer. Blocks are tracked in the processor to determine relaxed commit conditions. In-order commit is completely preserved and can be used for all or part of a program. The compiler also assists in exception processing by guaranteeing certain conditions are met.

The architectural extensions include block-based early register release, independent multiple commit points in different blocks, OOO commit of memory instructions, and compiler-assisted exception processing. The results show that on average 52% of instructions can be committed out of order and the resulting speedup ranges from 10% to 26%. The additional hardware requirements are minimal.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 423, Washington, DC, USA, 2003. IEEE Computer Society.

[2] G. B. Bell and M. H. Lipasti. Deconstructing commit. In *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 68–77, Washington, DC, USA, 2004. IEEE Computer Society.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[4] T. N. Buti, R. G. McDonald, Z. Khwaja, A. Ambekar, H. Q. Le, W. E. Burky, and B. Williams. Organization and implementation of the register-renaming mapper for out-of-order ibm power4 processors. *IBM J. Res. Dev.*, 49(1):167–188, 2005.

[5] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 48, Washington, DC, USA, 2004. IEEE Computer Society.

[6] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward kilo-instruction processors. *ACM Trans. Archit. Code Optim.*, 1(4):389–417, 2004.

[7] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing processor performance through early register release. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*, pages 480–487, Washington, DC, USA, 2004. IEEE Computer Society.

[8] A. Garca, O. Santana, E. Fernndez, P. Medina, and M. Valero. LPA: A first approach to the loop processor architecture. 4917:273–287, 2008.

[9] I. Gonzalez, M. Galluzzi, A. Veidenbaum, M. A. Ramirez, A. Cristal, and M. Valero. A distributed processor state management architecture for large-window processors. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 11–22, Washington, DC, USA, 2008. IEEE Computer Society.

[10] T. M. Jones, M. F. P. O'Boyle, J. Abella, A. Gonzalez, and O. Ergin. Compiler directed early register release. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 110–122, Washington, DC, USA, 2005. IEEE Computer Society.

[11] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using SPEC CPU2006. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 215–225, New York, NY, USA, 2007. ACM.

[12] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, page 24, New York, NY, USA, 2006. ACM.

[13] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[14] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Checkpointed early load retirement. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 16–27, Washington, DC, USA, 2005. IEEE Computer Society.

[15] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 59–70, Washington, DC, USA, 2002. IEEE Computer Society.

[16] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society.

[17] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 3–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[18] E. J. McLellan and D. A. Webb. The Alpha 21264 microprocessor architecture. In *ICCD '98: Proceedings of the International Conference on Computer Design*, page 90, Washington, DC, USA, 1998. IEEE Computer Society.

[19] T. Monreal, V. Vi nals, A. González, and M. Valero. Hardware schemes for early register release. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing*, page 5, Washington, DC, USA, 2002. IEEE Computer Society.

[20] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 129, Washington, DC, USA, 2003. IEEE Computer Society.

[21] S. Petit, J. Sahuquillo, P. Lopez, R. Ubal, and J. Duato. A complexity-effective out-of-order retirement microarchitecture. *IEEE Trans. Comput.*, 58(12):1626–1639, 2009.

[22] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 107–119, New York, NY, USA, 2004. ACM.

[23] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM J. Res. Dev.*, 46(1):5–25, 2002.

[24] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 290 –301, 16-20 2008.