Abstract

We introduce an alternative instruction commitment mechanism for a Reorder Buffer (ROB)-based out-of-order processor that commits a group of consecutive instructions atomically to support a larger instruction window. The proposed mechanism makes conservative use of the ROB, by only setting up entries for the instructions that perform the latest update to a register from that group. Further, the destination registers of instructions from a group that do not hold the most recent updates to architectural registers, can be released before the group containing these instructions is committed. The net result is an augmented ROB-based datapath, which increases the effective size of the ROB as well as the effective number of physical registers. The proposed design achieves an average performance gain of about 10% and 16% on the SPEC integer and floating point benchmarks, respectively, when compared to a traditional ROB-based design. The proposed design also achieves a performance gain of slightly over 5% when compared with an aggressive design that uses checkpoints and relatively complex hardware resources.

1. Introduction

Conventional out-of-order processors use a reorder buffer (ROB) and rename table information to rebuild the precise architectural state upon an exception or misprediction, by forcing all instructions to commit to the architectural state in the program order. This design encounters many performance problems. In-order commitment can stall the entire processor if the instruction at the head (commit end) of the ROB is stalled, even if all or most other instructions are ready to commit. Furthermore, the resources (physical registers, store buffers and so forth) used by each instruction must be retained until that instruction commits, even if no further consumers of the associated result are present.

To address these problems, checkpoint based commitment mechanisms [1, 5, 11] have been proposed to support large instruction windows. In a checkpointed processor, checkpoints are established at certain points in the instruction stream (typically at poorly predictable branches [1]). The precise state at each checkpoint is saved in a buffer associated with the checkpoint. The set of checkpoint buffers is arranged as a FIFO queue. The typical information stored in a checkpoint for an out of order processor typically contains pointers to registers and saved register mapping tables, or Speculative Rename Table (SRT). When the checkpoint at the head of this queue is committed, its associated resources are freed only after all instructions within the checkpoint have completed without exceptions. Any exception or branch misprediction that occurs between two checkpoints causes a rollback (restoration) of the architectural state to the last-known-good state at the previous checkpoint.

Most checkpoint based processors use reference counters or usage bitmaps to track when there are no remaining consumers of each physical register. A physical register can be reclaimed as soon as it is no longer referenced anywhere (i.e. by either the rename table or actual instructions still in the pipeline), rather than waiting for its instruction to commit. With checkpointing, the effective number of physical registers goes up compared to a processor using a ROB. This is a major advantage as the number of instructions in flight is limited only by the number of free physical registers and the load/store queue, rather than the size of the ROB.

In this paper, we introduce an alternative instruction commitment mechanism for a ROB-based out of order processor that commits a group of consecutive instructions atomically. The proposed mechanism makes conservative use of the ROB by only setting up entries for the instructions that perform the latest update to an architectural register from that group. In addition, the registers that are targeted by instructions that do not hold the most recent updates to architectural registers from the group can be released before the
group containing these instructions is committed. The net result is an augmented ROB-based datapath that increases the effective size of the ROB and the effective number of physical registers.

Our proposed design achieves an average performance gain of about 10% and 16% on the SPEC integer and floating point benchmarks, respectively, compared to a traditional ROB-based design. Our proposed design also achieves a noticeable performance gain over a design that uses checkpoints with similar hardware resources.

2. Motivation

Checkpoint based processors generally outperform traditional ROB-based by supporting larger instruction windows. When a checkpoint is created, all the entries in speculative rename table are copied in checkpoint buffer storage. If an architectural register is not renamed between consecutive checkpoints, its entry in the checkpointed table will be the same for both checkpoints, resulting in some redundancy and inefficiency in the use of the storage allocated for saving checkpoints. This effectively reduces the number of checkpoints that can be supported with a fixed hardware budget. When the storage for saving non-committed checkpoints are exhausted, the processors stalls till the earliest checkpoint in program order commits and releases its associated checkpoint storage locations. Also, as the checkpointing requirements of different applications vary widely in terms of the average number of instructions between checkpoints, the use of a fixed amount of storage for holding active checkpoints can turn out to be a performance limitation.

Some checkpoint based schemes [1] typically free a physical register as soon as the last consumer of that register has issued. These schemes associate a reference counter with each physical register to record the number of dispatched but not yet issued instructions that reference the register. The maximum counter value is decided by the sum of the number of instruction operands that use the register as a source register, plus the number of checkpoints that contain the register in their rename table maps. In the worst-case scenario, if the processor supports 128 physical registers, 64 architectural registers, 8 checkpoints and each instruction has 3 register operands, the counters should have $\log_2(128 \times 3 + 64 \times 8)$ or 10 bits each. In practice, this number is much lower, ranging from 4 to 8 bits. But a smaller counter size requires stalling the pipeline when the maximum count has been reached. Because most instructions will only have limited number of dependent instructions, the size of the reference counter is then determined by the number of checkpoints that need to be supported. If each rename map is stored only once, the number of updates to the counter are reduced and a smaller counter can be used.

On the performance side, the checkpoint creation operation is on the critical path and can potentially block the instruction renaming step until all entries in rename table are copied over. In a typical X86-64 ISA implemented in the MARSSx86 simulator [20] a 64-entry rename table needs to be copied at each checkpointing step: these entries correspond to 16 integer, 16 SSE/FP low, 16 SSE/FP high, 16 control and microcode registers. Even with special hardware support in the renaming logic to shorten this delay, the rename stage still has to be stalled during the checkpointing operation.

The performance gain achieved by checkpoint-based processors comes partly from the out-of-order commitment which retires multiple instructions in a group in one shot. A checkpoint mechanism happens to be a relatively straightforward way of implementing such an approach, but the same benefits can also be realized in alternative implementations. This paper introduces and evaluates such an alternative solution, which provides higher performance on average than a very aggressive implementation of checkpointing but a lower implementation complexity.

3. Committing Instructions in Groups

In traditional designs for out-of-order superscalar processors, the granularity of committing results to the precise state is at the level of single instructions. This requires the results of all instructions that target registers or memory locations to be maintained in program order in a reorder buffer. We propose to commit results to a precise state only at discrete points along the control flow path, by increasing the granularity at which results are committed to the precise state. Specifically, instead of committing the result of one instruction at a time, we atomically commit the results produced by a series of consecutive instructions into a single step. Each such series of consecutive instructions that are co-committed atomically is called a commit group, and we therefore call our proposed technique a group-commit mechanism.

The ability to co-commit a series of instructions in a single atomic step allows greater efficiency in using physical registers and ROB entries. A group-commit approach uses only the results of the most recent instruction in program order within the group that targets the same architectural register, and saves only those results for commitment. For instance, consider
the scenario shown in Figure 1. The three instructions I0, I2 and I7, with the instruction indices corresponding to the program ordering of the instructions, are all within a single group and all target the same architectural register, r24. Using the group-commit scheme, only the result produced by I7, which is the most recent instance of r24, will need to be saved for commitment. Instructions that generate the latest instance (in program order) of an architectural register within a group are called the last writers. In contrast, the results produced by I0 and I2 will not need to be saved within the ROB for commitment, as marked with dashed back-slash ed lines in Figure 1 (a). Thus, the only entry that needs to be set up in the ROB will be the one for the last writer (Instruction I7) to r24. To deal with exceptions and mispredictions caused by all instructions in a single group, an alternative compact structure is used, as discussed in 3.6.

The group-commit structure leads to a very conservative and minimalist use of the ROB, as shown in Figure 1 (b). In the example shown, the ROB slots that would be occupied by I0, I1, I2 and I4 in a traditional design (Figure 1 (a)) can now be allocated to other instructions, thereby increasing the effective ROB size.

Our group-based instruction commitment mechanism has another advantage: the registers targeted by instructions that are not generating the latest instances on an architectural register within the group (such as instructions I0 and I2 Figure 1) can be released before the group commits, as long as their consumers have issued. This is a form of early register release that is not speculative in nature, and thus is easy to implement. The consequence of such early register releases is an effective increase in the number of physical registers, allowing a larger instruction window compared to what would be feasible in a traditional design that makes use of the same ROB and physical register file.

Although our proposed scheme appears to be superficially similar to some of the checkpointing schemes proposed for out-of-order processor designs [1, 5, 11], there are two important differences:

First, the primary goal of checkpoint-based instruction commitment schemes is to eliminate the ROB, as the ROB does not scale up well to support large instruction windows. In contrast, our goal is to simply make more effective use of the ROB, allowing it to support a larger instruction window. Second, in checkpoint-based instruction commitment schemes, even if the value of an architectural register remains unchanged from one checkpoint to the next, its value is stored as part of both checkpoints. Similarly, there are inefficiencies which result from the storing of unchanged rename table entries at both checkpoints. The existing checkpointing schemes are thus rather inefficient in using the storage artifacts for saving checkpointed states. Our proposed scheme does not have these inefficiencies.

We now describe our group-commit mechanism and its implementation.

3.1 Identifying the extent of groups

Increasing the size of the group, that is, the number of instructions in a group, increases the likelihood that multiple instructions from that group will write to a common architectural register. Under such conditions, a larger number of results from earlier instructions...
targeting a common architectural register can be discarded as soon as they have been consumed. This is because the commitment of the entire group of instructions requires only the value established by the latest write to an architectural register from instructions within the group. On the other hand, with smaller group sizes, the probability of finding multiple instructions that target a common architectural register is lower, leading to a commensurate drop in the efficiency of use of the ROB to hold only the results from the last writers within a group.

However, large group sizes also create a potential performance penalty. Misspeculations of branches, or misspeculation of other events that support speculative execution, or exceptions will all cause execution to roll back to the beginning of the group, undoing a significant amount of useful work. Another tradeoff to consider here, from an implementation perspective, is the instruction commit rate. A larger group size demands a wider commit width, to permit the commit rate to be balanced with activities elsewhere in the pipeline. In particular, if the commit rate slows down, then the load/store queue (LSQ) size can go up which will have an adverse effect on performance. Overall, this suggests that there is likely to be an optimum group size that maximizes the overall performance gains achieved with the use of group commitment.

Conceptually, the problem with large group sizes is equivalent to increasing the number of instructions between checkpoints in checkpointing schemes. As branch mispredictions are likely to occur with a relatively much higher frequency than exceptions, it is useful to use branch confidence estimators [2] to identify branches that are more likely to be mispredicted. The checkpointing scheme of [1] uses a branch confidence estimator to place checkpoints just prior to low confidence branches.

Our group-commit mechanism also uses a branch confidence estimator to end a group at the instruction, which occurs just prior to a low confidence branch. In our scheme, we also impose an absolute limit on the size of a group, so that a group ends when we reach this limit or when we encounter a low confidence branch, whichever occurs earlier. For the x86 ISA, where an x86 instruction is decoded into multiple RISC-like micro-ops (uops), the group boundaries are made to coincide with the instruction boundaries.

The proposed processor can also be run in the single instruction mode. The single instruction mode can be enabled explicitly by setting the processor in the debugging mode or implicitly after exceptions as explained in Section 3.6.

3.2 Associating instructions with groups

In our design, instructions belonging to a group need to be identified as belonging to a specific group. The group id, the address of the instruction starting the group, and the count of instructions in the group are all entered in the entry at the tail of a FIFO queue (the Group Order Queue, GOQ) that records the program order of the groups. The GOQ is capable of holding a fixed number of entries, and the offset of an entry in the GOQ thus serves as a unique id for the group that is assigned to the entry. In addition to the three items named above, there are three other additional fields in a GOQ entry. The first of these fields contains a pointer to the ROB slot where entries for the group are started, known as the SOG (start of group) index. The second contains the ROB index of the last entry set up in the ROB for this group, known as the EOG (end of group) index. The EOG is updated as instructions group are dispatched. And the final field in a GOQ entry is a single bit field (EXC) that indicates if an exception occurred during the processing of the group.

The entry at the head of the GOQ corresponds to the earliest group in program order that has yet to be committed. Somewhere between the head and tail entries in the GOQ (in program order) there will be a pointer marked “current” which points to the GOQ entry of the group that is being currently dispatched.

All instructions in the group are tagged with the group id as they are dispatched. As we will see in 3.6, when the execution of some or all instructions within a group needs to be aborted due to exceptions or branch mispredictions, this group id is used to locate and annul instructions from the group. As instructions from a group dispatch, the dispatch stage(s) will also tag instructions with a relative sequence number within the group, starting with a sequence number of zero for the first instruction in the group.

3.3 Identifying the latest writers

A key requirement of our scheme is the ability to identify, and save for commitment, only the result from the very last update to every architectural register that are destinations of instructions in the group. The last instruction in the group to write to an architectural register (that is, the last writer) is identified and marked in the front-end stages of the pipeline as described below.

Figure 2 depicts a representative pipeline implementing the X86 ISA where each of the native X86 instructions are decomposed by the hardware into RISC-like equivalents call uops. This decomposition requires a complex multi-stage decoder (a 3-stage version is shown in Figure 2). Many X86 pipelines also use trace caches to avoid the repeated decoding overhead of recently encountered instruction sequences
based on an independent trace predictor (which was initially employed in the Intel P4 Netburst implementation [10]). The trace cache access functions are also shown in Figure 2. When a trace is predicted to be executed, the decoding steps are aborted to improve the throughput. If a trace cache hit does not occur, entries are set up in the trace cache for the current sequence of uops that make up a trace.

The grayed out stages (two delay stages) and the grayed out functions shown as part of the existing stages in Figure 2 are required by our technique to correctly identify the last uop to write a specific architectural register within a group. We employ a rename table like structure, called the Last Writer Index Table (LWIT) to identify these last writers. A LWIT is indexed by an architectural register id and holds the index (that is relative position of the last writer within a group from the very first uop in the group). A single bit flag in each LWIT entry indicates if the corresponding architectural register was written to by any instruction within the group. An example of a LWIT for the instruction group is shown to the right of Figure 1(a), at a point when the very last instruction in the group has been generated.

As soon as a uop is generated by the last decoding stage, and if that uop writes an architectural register, the index of the uop relative to the first uop in the group is written to the architectural register’s entry in the LWIT, overwriting an existing entry. When all of the uops in the group are decoded (in sequence), the LWIT contains indices that identify the relative positions of the last writers within a group. The LWIT entries are, of course, cleared before any entry is made into it for any uop in the group. At this point, the LWIT entries are used to tag the last writers for the group within the trace cache. If the newly formed trace is subsequently executed, there is no need to identify the last writers again.

The pipeline maintains multiple LWIT, one for identifying the last writers for every instruction group that has been demarcated by the branch predictor and branch confidence estimator. A LWIT is released as soon as its contents are used to tag the last writer within a group and all existing entries within that LWIT are cleared. We found that 4 LWITs are sufficient across all of the benchmarks we had simulated for all of the datapath organization we used. In a pathological case, when we run out of LWITs all instructions that write registers are marked as last writers. The last writers are identified in the trace cache, as mentioned in the previous paragraph, as well as in the uop stream that was just decoded. The tagging of the newly decoded uop stream is done as follows.

Every uop that is decoded is tagged with a group id and a sequence number. The uops carry these tags as they move through the pipeline till the final dispatch stage. As a uop is processed by the last dispatch stage, the LWIT for this uop’s group is probed to see if the uop was identified as the last writer. If the uop is the last writer, it is tagged as such when it’s entry is set up in the issue queue. A subtlety in tagging the last writers within a group needs to be mentioned. It is possible that as a uop that writes an architectural register from a group to exit the final dispatch stage before the next uop in the group has been decoded. This dispatched uop may thus be identified incorrectly as a last writer (as the groups LWIT has not yet been finalized). To reduce this possibility, the two extra delay stages are added as shown in Figure 2. If we assume that each stage following the last decode stage handles 4 uops, by the time the first uop leaves the final dispatch stage, 24 uops from the group have already been decoded (and corresponding updates, as needed, have been made to the groups LWIT). In reality, as shown in Section 5, a group with a group size limit of 16 (as used) contains about 10 uops on the average – thus, in practice the last writers within a group will be all identified correctly on the average following the initial decoding. For long groups, some redundant last writers will be identified following the initial decoding, but these last writers will always be
identified correctly on executions that subsequently hit the trace cache.

3.4 Instruction Dispatching and Commitment

During instruction dispatching, instructions are tagged with the id of the group that they belong to, as described earlier. A ROB entry is set up only for the instructions marked as the last writers.

Instructions issue and execute as in a traditional design, and write their results to the allocated destination. As each instruction completes, the group instruction count pointer in the corresponding GOQ entry is decremented. Results are bypassed to an instruction as it moves to the required function unit. If an exception is generated by an instruction, it locates the relevant GOQ entry using its group id tag (which is the index into the GOQ for the entry) and also sets the EXC flag for the entry. The processing of exceptions and branch mispredictions are described in Section 3.6.

Groups are committed in program order. A group can be committed if instructions within the group did not generate any exception (in other words, the EXC field is not set in the GOQ entry) and when the instruction count field is zero indicating that all instructions have completed. The actual commitment step consists of updating the back-end register map with the addresses of the registers in the ROB entries for the group, and the release of the physical registers associated with the updated entries in the map, as in a traditional design.

Note that the store queue by itself cannot support the atomic commitment of memory writes from a group, as some of these writes can propagate to the memory before the group is committed. To prevent such premature memory updates, the stores from a group are held in locked cached lines till the group commits, as in implementations of transactional memory systems in some current and announced product offerings [12]. Checkpointing schemes, such as the one in [1] rely on similar support.

3.5 Early Register Releases

Our group-commit mechanism supports early register releases: registers not targeted by the last writers in a group can be released as soon as the last consumer of each such register has read the register. We implement this mechanism by using a 2-bit consumer counter for each register, and a 1-bit flag (release) that enables or disables early releases. The count field is incremented on renames, and decremented on instruction issues. We permit the early release of registers with up to three consumers. Registers with four or more consumers cannot be early-released, and this is implemented by setting the early release flag to disable such releases. By permitting the early release of registers with up to three consumers, we cover nearly 94% of cases where early release was possible, on average across all benchmarks. One key difference between the operations of our counter for early register release, as compared with previous checkpoint schemes, is that our method does not require that each group increment/decrement the consumer counters of its referenced registers, as discussed in Section 2, neither does it require wide counters.

In the x86 implementations that translate X86 instructions to uops, temporary architectural registers (not visible at the ISA level) are used internally in translating the x86 instructions to a bundle of RISC-like uops. These registers are also early-released in our technique.

In effect, the proposed scheme implements a technique that promotes the efficient use of the ROB and, as a byproduct, offers a simple way of relieving register pressure through the early release of registers.

3.6 Branch Mispredictions and Exceptions

The most common case for a branch misprediction is due to the misprediction of a low confidence branch that terminates a group. In this case, we flush all instructions that follow this branch along the wrong path, including instructions that belong to all groups that follow this group. Such groups are identified from the GOQ, and the ids of these groups are used to locate and annul all instructions belonging to these groups in the pipeline, including ROB, Issue Queue and Function Units, etc. The resources allocated for each annulled micro-op, such as LSQ and physical register, are also released during the annulments. Execution then resumes on the correct path.

For the misprediction of branches that don’t occur at the end of the group, the execution is rolled back to the beginning of the group G in which the event occurred. This requires the annulment of all instructions in group G, and all the other groups that follow it, in the same manner as before. In this case, the executions will resume in a mode where the ROB is used exactly as in a traditional design, so that an entry will be established in the ROB for each and every instruction that is processed. The traditional mode execution then continues till we reach the offending instruction, in other words the mispredicted branch. After performing the due processing for the offending instruction, execution then resumes with the group...
commit mode, where ROB entries are established only for the last writers.

Table 1: Simulated processor configuration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine width</td>
<td>4-wide fetch, 4-wide issue, 4-wide commit</td>
</tr>
<tr>
<td>Window size</td>
<td>96 entry load/store queue, 128–entry ROB or 8 checkpoints</td>
</tr>
<tr>
<td>Function Units and Lat (total/issue)</td>
<td>4 Int Add (1/1), 2 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 2 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)</td>
</tr>
<tr>
<td>Physical Register</td>
<td>196 (including architectural registers)</td>
</tr>
<tr>
<td>L1 I–cache</td>
<td>32 KB, 8–way set–associative, 64 byte line, 4 cycles hit time</td>
</tr>
<tr>
<td>L1 D–cache</td>
<td>32 KB, 2–way set–associative, 64 byte line, 4 cycles hit time</td>
</tr>
<tr>
<td>L2 Cache unified</td>
<td>2048 KB, 8–way set–associative, 64 byte line, 6 cycles hit time</td>
</tr>
<tr>
<td>BTB, branch penalty</td>
<td>1024 entry, 4–way set–associative, 10 cycles</td>
</tr>
<tr>
<td>Branch Pred.</td>
<td>16K entry gShare, 12-bit global history</td>
</tr>
<tr>
<td>Branch confidence Predictor</td>
<td>16K entry gShare, 12-bit global history</td>
</tr>
<tr>
<td>Memory latency</td>
<td>120 cycle</td>
</tr>
<tr>
<td>TLB</td>
<td>32 entry (I), 32 entry (D), fully associative</td>
</tr>
</tbody>
</table>

Exceptions are handled in a slightly different manner. As described above, when an exception occurs, the EXC flag in the GOQ of the group is set. This exception, as well as any other exception within the same group, is recognized when the group is being committed. The annulments and processing that are then implemented are similar to the situation when a branch that was not at the end of a group was mispredicted. When multiple exceptions or multiple page faults occur within a single group, execution is rolled back to the beginning of a group and a traditional execution mode is entered as in the case of branch mispredictions to commit instructions individually and to identify the source of the exception/page fault.

In all the situations mentioned above, the front end rename table can be restored by walking forward from the oldest ROB entry indicated by the head of the ROB, in parallel with other pipeline flushing activities. The rename table is restored, with the content of the back-end register mapping table which points to committed register values as the start point. Then each ROB entry encountered during the walk will be updated with the architectural to physical register mappings stored in these ROB entries. This process is similar to the rename table recovery process in a traditional design, except that fewer ROB entries are processed - only those for the last writers.

3.7 Area overhead

The group commit mechanism requires new hardware structures whose area can be roughly estimated as follows. Four LWITs are used; each is like a rename table holding a 5-bit offset in each entry. The GOQ is a two-ported register file with 4-bit entries (to support 16 active groups), so the total number of bitcells needed by these structures is (4*5*64 + 16 * 4), that is, 1344. The decoders/drivers, group tags and pass transistors needed to support additional ports will, very conservatively, double the equivalent area, so the total area need is the equivalent of about 2800 bitcells. Using 8 transistors per bitcell, including the pass devices to support two single-ended read ports, this translates to 22400 transistors, which is a miniscule part of the overall transistor budget. This additional area is equivalent to the silicon real estate occupied by 18 ROB entries (164 bits per entry, no address compression), ignoring the transistors needed for multiple ports on the ROB as well as additional devices needed in the decoders and drivers in the ROB. As shown later in Section 5, the performance gains realized by the group commit mechanism far exceeds the performance gains one would get with 18 additional ROB entries added to a baseline design.
accommodate the uops from this last X86 instruction.

The actual group size will consist of 19 uops to
the next X86 instruction can translate to 5 uops, t he
cases, a group may also have more uops than dictate d
confidence branch is encountered. In some other
groups end prematurely with a low confidence branch.

The initial set of X86 instructions in the group, a nd if
limit is 16 uops and 14 uops have been generated fo r
of an X86 instruction. For instance, if the group size
may end earlier before this limit is reached if a l ow
unknown. The baseline machine configuration is summarized in Table 1. The proposed
group-commit mechanism also uses the same basic
configuration as described in the table.

In our experimental evaluations, we use benchmarks from the SPEC 2006 suite. For each
benchmark, we ran each benchmark for 500 million
instructions. The number of instruction and all
the results in this paper is based on user level
instructions only; kernel level instructions are excluding. All benchmarks were compiled using gcc
4.4, with the SPEC-recommended optimizations to
target the 64-bit x86-64 instruction set.

5. Experimental Results

To evaluate the effect of group size limit on
performance we run SPEC2006 with three different
group size limits in uops. This is a soft limit; a group
may end earlier before this limit is reached if a low
confidence branch is encountered. In some other
cases, a group may also have more uops than dictated by
this limit, as we need to end a group at the boundary
of an X86 instruction. For instance, if the group size
limit is 16 uops and 14 uops have been generated for
the initial set of X86 instructions in the group, and if
the next X86 instruction can translate to 5 uops, the
actual group size will consist of 19 uops to
accommodate the uops from this last X86 instruction.

Figure 3 shows the average number of instructions
per group. In general, the average number of uops in a
group is lower than the group size limit as many
groups end prematurely with a low confidence branch.

The influence of the group size limit on the average
group size is much bigger on floating point
benchmarks. The reason behind this is that the integer
benchmarks typically have more branches and higher
mispredictions than floating point benchmarks.
Figures 3 through 8 have the floating point benchmark
on the left side of the graphs (bwaves – zeusmp) and the integer benchmarks on the right side (astar –
alanc). The reason behind this ordering of the
benchmarks in the charts is to emphasize the different
impact the proposed technique has on floating point
and integer benchmarks.

Figure 4 depicts the impact of the group size limit
on the performance (IPC) realized by our technique.
The results of Figure 4 indicate that the performance
realized by the proposed scheme is not sensitive to the
group size. This is because the penalty with a higher
group size is compensated by the performance gain due
to early release and higher chance to find instructions
with same last writer. In some cases, a reduced IPC
is realized with a higher group size – this happens when
mispredictions cause a long group to be rolled back,
because of the atomic nature of group commitments.
For the rest of the results that we present, we selected a
group size limit of 16 uops. This size was selected
because it ensures that all the benchmarks have an
actual size less than 24 uops and all of the last writers
will be identified correctly during the initial decoding
(Section 3.3).

Figure 5 shows number of uops per last writer
within a group, that is the actual number of uops in the
group divided by the number of last writers. Since
ROB entries are allocated in our technique for the last
writers and jump instructions, this ratio indicates the
degree to which the number of ROB entries are reduced in our scheme compared to a traditional (base)
design. Although the size of the group can affect
number of instructions per last writer, the primary
dependence of this ratio is on the code itself. As Figure
6 shows, Hmm has the highest ratio but its group size
is not in the top 5 benchmarks. The average number of
instruction per last writer over the integer benchmarks
is 1.5 and 1.3 over the floating point benchmarks.
The proposed group commit design realized an average IPC gain of 12% on floating point benchmarks and an average IPC gain of 8% on integer benchmarks. In fact all of the floating benchmarks performed better than all the integer benchmarks with the exception of Calculix, which only had a 8% performance gain. There are multiple reasons why the floating point benchmarks showed better results. The first reason is that the integer benchmarks have higher misprediction rate, which not only affect the group size but also cause degradation in performance because the group commit design rolls back execution to the previous inter-group boundary, discarding more instructions than the traditional design on branch mispredictions. The second reason is that the floating point benchmarks have a higher number of instructions per last writer than the integer benchmarks as shown in Figure 5. Figure 7 depicts the hit rate on the trace cache, that is, the trace prediction accuracy. The hit rate depends not only on the locality of the program instructions but also on the repetitive behavior of the branches. A trace cache hit implies that the uops in the group are identical to what is stored in the trace cache to guarantee that the group has the same last writers. The trace prediction rates are less than 50%.

The performance gain that the trace prediction mechanism adds to the proposed technique is also shown in Figure 6. The IDEAL_TP bars represent the performance realized with a 100% trace cache hit ratio. Figure 6 shows that even with a less than 50% trace cache hit, the performance realized is close to that of the ideal case.

Figure 8 compares the performance achieved by checkpoint based design of [1] against the group commit design with both trace prediction (TP) and early register release (ER). The group commit design realized a higher performance gain over the base case compared to the checkpointing scheme across all of the benchmarks. Overall, the average IPC gain over a traditional ROB based design (the base case) is 14% across all benchmarks (integer and floating point) with a maximum IPC gain of 22% over the base case. Across all the SPEC 2006 benchmarks, our design achieves an average IPC gain of 5% and a maximum IPC gain of 14% over an aggressive checkpointing scheme.

To understand the impact of the group commit scheme, we analyzed the detailed instrumented data for the benchmarks presented earlier, as well as the branch misprediction rates for the benchmarks (Figure 9). Benchmarks that realized the highest performance gain (for example, Milc, Figure 8, about 18%) have a high uop to last writer ratio and a low branch misprediction rate. The results for the benchmark Hmm is representative of the adverse effect of the branch misprediction: although it has the highest instructions per last writer ratio, the performance gain realized for Hmm is below the overall average, as significant rollbacks due to misprediction eat into the benefits provided by our scheme.

Figure 10 shows the average IPC realized across all the benchmarks with gradually increasing register file sizes, assuming that all other resources area are scaled appropriately to match the register file size. For the results shown in Figure 10, we have chosen the sizes of other resources based on the physical register size and not on the ROB size, because the checkpoint mechanism does not have any ROB. From Figure 10, we can see that the group-commit scheme consistently outperforms both the base case and checkpoint design with 8 checkpoints and its performance is almost identical to a checkpoint-based design with 32 checkpoints. As expected, checkpoint-based designs need additional checkpoints to provide the performance gains that come with other additional datapath resources. The most important takeaway from Figure 10 is that our group-commit based design with 128 registers provides an average performance that is very close to that achieved by a baseline machine with significantly more resources: a 256-entry ROB and a 256-entry register file. Similarly, with 192
6. Related Work

The classic paper by Smith and Pleszkun [22] formalizes the notion of a precise state in out-of-order processors and presents three broad mechanisms for implementing a precise state to permit the processor to resume correctly following an interrupt. Most real out of order processors implement two of the three schemes presented and evaluated in this paper - the

Figure 8: Performance comparison between base design, checkpointing and group commit with both TP and ER

registers, the group commit scheme exceeds the performance realized by a baseline scheme with 512 registers. In effect, the proposed mechanism has thus increased the efficiency of using the ROB and the register file quite dramatically.

Figure 11 shows the power consumed by the group commit scheme compared to the base design. We used McPAT [17] to measure the power dissipation in the base case (for a 65 nm implementation) and CACTI to estimate the power dissipated in the LWITs and the GOQ (which are essentially multi-ported RAMs). Our design has a very small power overhead. One source of extra power consumption is the extra hardware structure added for detecting the last writers and the smaller ROB dimensions.

Another source of additional energy consumption is due to re-rolls triggered by a branch misprediction and the branch in the middle of the group. On the other hand, the proposed technique uses fewer ROB entries and that saves power due to the lower number of ROB accesses and the smaller ROB dimensions.

A number of techniques for reducing the complexity of a traditional ROB, which is normally a heavily ported structure, is described in [15], including

the use of a distributed ROB structure. However, the goal of the work of [15] is not to enhance the performance compared to a traditional ROB design, but instead to maintain a performance level which is close to that of traditional designs, albeit with “simplified” structures. In [3], Akl and Moshovos propose a “Turbo-ROB” mechanism that is used to complement a traditional ROB to support fast state restoration. The Turbo-ROB provides fast recovery for the common case of mispredicted branches, whereas state recoveries in all other instances make use the normal ROB. The key idea behind the Turbo-ROB is to save only the minimum necessary state information related to the rename table at each “recovery” point, with each such recovery point at a normal branch or at low confidence branches in a design that conserves the hardware requirements of the scheme. The Turbo-ROB mechanism can also replace a traditional ROB and can also complement a normal checkpointing scheme.

In an earlier work [19] Moshovos proposed a scheme that supports out of order checkpoint release in ROB based processor that can free a checkpoint once...
corresponding branch is issued without misprediction. With the use of a reorder buffer, in-order commitment of instructions delays the release of many resources critical to performance. In [16] Latorre proposed a compressed ROB (CROB). CROB releases the ROB segment if all the entries in that segment are ready to commit and can be treated as one atomic unit. CROB uses a logical ROB structures to track the released segments in case of exception. Exceptions cannot happen in the middle of released segment since the CROB releases only the exception-free segments. In [2] Akl et al used a novel checkpoint-aware speculation strategy that temporarily throttles speculation to reduce misspeculation costs. Bell and Lipasti established conditions for out-of-order commitment and propose mechanisms that are augmented to a reorder buffer to support out of order commitment in [4]. Further, in the technique of [4], entries are set up for every instruction that writes a ROB. The Cherry scheme [18] supports aggressive early resource releases in using checkpoints in a ROB-based datapath but restricts “early” resource releases to ones that are not needed to support branch misprediction handling and memory replays. There is considerable complexity in the implementation of the Cherry scheme, particularly in identifying resources for early release; however, the performance gains stemming from early resource releases are impressive.

In the past, other alternatives to the traditional ROB designs have also been proposed. The use of checkpointing to establish precise states at intervals of multiple instructions along the control flow path has been a relatively pervasive alternative to ROBs. However, there are some fundamental differences between checkpoint-based state restoration schemes and our technique (Section 3).

Hwu and Pat and [11] introduced a number of checkpoint-based techniques to help out-of-order processors to recover from exceptions and branch mispredictions, using the concept of checkpoint repair ranges and precise repair points. Checkpoints have also been proposed for out-of-order processors in some recent work [1, 5]. In [1], the authors introduced a checkpointing scheme called “CPR” to support large instruction windows in a scalable manner, including a hierarchical store buffer to hold on to memory writes pending commitment in program order. This mechanism attaches a separate speculative copy of the complete rename table to each checkpoint, decided by a branch confidence predictor. The front-end rename table is updated as micro-ops are renamed, and when a checkpoint is made, a fresh checkpoint is allocated and the rename table is copied into the checkpoint. One source of complexity in the CPR scheme is the use of wide counters for tracking register consumers and the overhead associated with concurrent updates to wide counters in a practical implementation.

In [5], Cristal et al introduced a clever associative structure for mapping architectural registers to physical registers (reminiscent of similar structures used in Alpha 21264 and the HAL/Fujitsu SPARC) and augmented it with two associatively addressed P-bit vectors to implement checkpointing, where P is the number of physical registers. These two bit vectors are used to indicate the physical registers that belong to a checkpoint and to indicate when a physical register is to be freed. Cristal’s technique delays the release of a register compared to traditional ROB-based schemes. Such prolongation of the register lifetime exerts considerable pressure on the register file and Cristal et al suggest the use of additional techniques [6] for easing this pressure.

Block-structured architectures [7, 23] also permit instruction to be committed in groups, but rely on the compiler to identify instructions that belong to a group. IBM POWER4’s commits a group of consecutive instructions atomically, but their concept of a group [24] is different from ours: in POWER4, all instructions in a group are co-dispatched and this limits the group size (to 5 in the POWER4). Furthermore, in a single cycle only one group of instructions can be dispatched - we don’t have this limitation either. We also identify and eliminate redundant writes in our larger groups, unlike the POWER 4. Another alternative for replacing the ROB is introduced in a recent work [9], which is based on a distributed register management mechanism. This design does not rely on either ROB or checkpointing for precise state restoration. Instead, a distributed register allocation and deallocation mechanism is used, and a distributed state is used for implementing precise states. In contrast, our proposed technique is designed to augment a conventional ROB-based design to provide reasonable performance enhancements.

Techniques for early register releases have been explored well in the past [8, 14, 21, 16]. Our approach for identifying registers to be released earlier-than-usual is different from any of these schemes. We use a simple hardware to identify the last writers (as defined in Section 3) and release the registers targeted by all other instructions within a group, as long as these registers have been consumed. We also target the early release of all temporary registers used in translating x86 instructions to RISC-like uops.

7. Conclusions

The proposed group-commit mechanism makes conservative use of the ROB, by only setting up entries
for the instructions that perform the latest update to an architectural register from that group. In addition, the registers that are targeted by instructions which do not hold the most recent updates to architectural registers from the group can be released earlier, before the group containing these instructions is committed. The net result is an augmented ROB-based datapath, which increases the effective size of the ROB and the effective number of physical registers. Our proposed design achieves an average performance gain of about 10% and 16% on the SPEC integer and floating point benchmarks, respectively, and a maximum IPC gain of 22% compared to a traditional ROB-based design. Our proposed design also achieves an average performance gain of 5%, and a maximum performance gain of 14% on the SPEC benchmarks over an aggressive design that uses checkpoints and similar hardware resources. It thus appears that our proposed technique is an attractive approach for further harnessing instruction level parallelism.

8. Acknowledgements

This work is supported in part by the NSF through award numbers 0958501 and 1040666 and awards from Intel Corporation.

9. References