Deconstructing Commit

Gordon B. Bell and Mikko H. Lipasti
Dept. of Elec. and Comp. Engr.
Univ. of Wisconsin-Madison
{gbell,mikko}@ece.wisc.edu

Abstract

Many modern processors execute instructions out of their original program order to exploit instruction-level parallelism and achieve higher performance. However even though instructions can execute in an arbitrary order, they must eventually commit, or retire from execution, in program order. This constraint provides a safety mechanism to ensure that mis-speculated instructions are not inadvertently committed, but can consume valuable processor resources and severely limit the degree of parallelism exposed in a program. We assert that such a constraint is overly conservative, and propose conditions under which it can be relaxed.

This paper deconstructs the notion of commit in an out-of-order processor, and examines the set of necessary conditions under which instructions can be permitted to retire out of program order. It provides a detailed analysis of the frequency and relative importance of these conditions, and discusses microarchitectural modifications that relax the in-order commit requirement. Overall, we found that for a given set of processor resources our technique achieves speedups of up to 68% and 8% for floating point and integer benchmarks, respectively. Conversely, because out-of-order commit allows more efficient utilization of cycle-time limiting resources, it can alternatively enable simpler designs with potentially higher clock frequencies.

1. Introduction

Dynamically scheduled processors execute instructions out of their original program order to exploit instruction-level parallelism and achieve higher performance. This allows independent computations to bypass long-latency events (e.g. cache misses, complex computational instructions, etc.) and improve instruction throughput. However even though instructions can execute in an arbitrary order, they eventually commit, or retire from the machine, in program order. This constraint provides a safety mechanism to ensure that mis-speculated instructions are not inadvertently committed and place the processor in an incorrect state that cannot readily be recovered from. For example, one reason that instructions typically commit in-order is to provide a mechanism to enforce precise exceptions. If an instruction is permitted to commit before an exception-causing instruction that appears earlier in program order, the processor will eventually transfer control from the exception handler back to the instruction that raised the exception. Precise exception semantics mandate that this point reflects all updates by preceding instructions and no updates by subsequent instructions. Because a subsequent instruction has already been committed and changed the architected state of the processor, these semantics are violated. Enforcing in-order commit prevents such situations, but can consume valuable processor resources and limit the degree of parallelism exposed in a program. In this case, no instructions can commit out of program order to protect against potential exceptions, even though such exceptions may be infrequent. Any long-latency event at the head of the commit queue will effectively stall the committing of instruction for the entire machine. Instructions will continue to dispatch until all available resources are consumed, at which point the entire processor will stall.

In this paper we assert that constraining instructions to commit in program order, while correct, is overly conservative. This work examines conditions under which this constraint can be relaxed and instructions can be permitted to commit out-of-order in a non-speculative manner, thereby eagerly freeing processor resources that can be allocated to future instructions waiting to dispatch. It also categorizes and analyzes different criteria required for instructions to commit out-of-order, provides ideal speedup measurements without implementation considerations, and finally discusses complexity issues and speedups with more realistic designs.

This paper is organized as follows: Section 2 discusses the methodology and simulation infrastructure used to collect results throughout the remainder of the paper. Section 3 presents a limit study intended to gauge the potential benefit of committing instructions out of program order. Section 4 provides a detailed discussion of what needs to occur at the commit stage in a dynamically scheduled processor and introduces the necessary conditions that instructions must fulfil in order to commit. Section 5 presents performance results of committing instructions out-of-order and provides a detailed analysis of microarchitectural modifications required to enable out-of-order commit. Section 6 presents related work, and Section 7 concludes this paper.

2. Methodology

This section discusses the methodology and simulation infrastructure used to collect results presented in the remainder of this paper. Uniprocessor results were collected on a simulator based on SimpleScalar 3.0 [3] that executes the Alpha EV6 instruction set. Modifications were made to model speculative instruction scheduling [13] as well as a decoupled issue queue. All no-ops are removed in the fetch stage of the pipeline and consume no execution resources. Additional machine parameters are presented in Table 1. A collection of SPEC2000 benchmarks were used in this paper, all of which were compiled with peak optimization using the DEC OSF compiler Each benchmark was fast-forwarded 400 million instructions before collecting timing information. Input sets were derived from the MinneSPEC workload suite [14].

3. Resource Sensitivity

This section presents a limit study that explores the potential

Table 1: Base Machine Configuration

Out-of-order execution	8-wide fetch/issue/commit, 10-cycle pipeline, 64 ROB, 32 LSQ, separate 32 integer / floating point issue queues, 64 rename registers, speculative scheduling, selective recovery for latency mispredictions
Functional Units	8 integer ALU, 4 FP ALU, 4 integer MULT/DIV, 4 FP MULT/DIV, 4 memory ports
Branch Prediction	Combined bimodal (16k entry) / gshare (16k entry) with selector (16k entry), 16 RAS, 1k entry 4-way BTB
Memory System (latency)	32KB 2-way 32B line IL1 (2), 8KB 4-way 16B line DL1 (2), 512 KB 4-way 64B line unified L2 (15), main memory (500)

benefit that committing instructions out-of-order may offer in terms of overall system performance. Typically, if the oldest in-flight instruction suffers a long-latency event it will prevent all newer instructions from committing before it. Furthermore, because all of these wedged instructions are consuming valuable processor resources, they may create structural hazards that prevent subsequent instructions from dispatching. Allowing newer instructions to commit before slower ones that have wedged the head of the reorder buffer utilizes resources more efficiently by eagerly freeing ROB slots, physical registers, and load/store queue (LSQ) entries, and allowing new instructions to be dispatched that would otherwise be stalled.

Applications that will likely benefit the most from committing instructions out-of-order are those whose performance scales with the availability of these resources. This section presents a limit study that measures the sensitivity of performance to the quantity of processor resources. We start by simulating a baseline processor with 32 ROB entries, 32 rename registers, and a 32-entry instruction window. Although we model an issue queue in simulations presented later this paper, in this study we allow any dispatched instruction to issue to isolate the effects of scaling the resources we wish to study. Figure 1 presents speedup as the size of each of these resources are increased by factors of two, from 32 to 2048. The LSQ contains half as many entries as the ROB in each case. In general, the floating-point benchmarks are more sensitive to the number of these resources compared to the integer benchmarks. This is at least partially due to the fact that these benchmarks contain many long-latency floating-point operations that consume processor resources for long amounts of time. Some of the benchmarks (such as *swim*, which exhibits a maximum speedup of over 20) show a large performance improvement, indicating that increasing the size of the window of instructions that are candidates for execution can expose significantly more instruction level parallelism (ILP). We hypothesize that allowing instructions to commit out-of-order will allow these benchmarks to utilize critical resources more efficiently and therefore exhibit the most speedup.

Conversely, some of the benchmarks show little or no speedup when simulated on processors with more aggressive resource configurations. This can generally be attributed to one of two situations. In one case, the application may exhibit a high degree of ILP and the processor is able achieve a high throughput of instructions committed per cycle, even with minimal resources. The program's inherent parallelism requires only a small number of instructions to be examined in order to select several whose execution can be overlapped. Because significant overlap can be achieved with few instructions in flight, adding resources to allow a larger set to be examined will not likely yield speedup. Alternatively, if the application has exceedingly little inherent ILP (for example, due to long chains of dependent instructions), then marginally increasing the size of the instruction window may not be sufficient to expose parallelism. In this case, adding resources allows more instructions to be examined, but will not result in detecting many whose execution can be overlapped with others. In either case, the lack of resources is not a performance bottleneck, and adding more will not achieve significant speedup. Therefore we expect that committing instructions out-of-order will not help performance for these benchmarks, and we omit them from results presented in the remainder of this paper. Specifically, we use the benchmarks that exhibit a speedup of at least 10% between the second and third bars in Figure 1.

Although Figure 1 indicates that not all benchmarks presented are constrained by resource availability, we expect that future trends will place increasing strain on the number of ROB entries and physical registers. As processor cycle time continues to outpace DRAM memory access time, an outstanding cache miss will require that more instructions issue to overlap execution. Similarly, techniques such as simultaneous multithreading will also enable more instructions to execute concurrently. More instructions in-flight will require more physical registers, and building large, monolithic physical register files has already proved to be problematic due to circuit constraints [24,26,22,3,25,6,5,7]. Therefore techniques such as out-of-order commit that utilize physical registers more efficiently may become increasingly valuable in future processors. Finally, the fact that many commercial workloads exhibit poor cache performance can also place added strain on the ROB and physical register file.

We note that these results are consistent with other experiments that have been performed in a variety of similar studies [15].

4. Deconstructing Commit

In order to explore conditions under which the in-order commit constraint can be relaxed, it is first necessary to define what exactly it means for an instruction to commit in the context of a dynamically scheduled processor. We feel that various industrial designs and academic research have caused this architectural term to become somewhat overloaded -- here we attempt to deconstruct what actually needs to happen at commit.

Commit is typically the last stage an instruction passes through in a pipelined processor. At this point the instruction exits the machine and irrevocably modifies the architected state of the system. The commit stage also frees any allocated resources for consumption by subsequent instructions. This can include physical registers, register map entries, reorder buffer (ROB) entries, and load/store queue (LSQ) slots. Many of these resources are scarce and their size and number are often constrained by cycle-time and power consumption requirements, thus holding them longer than necessary can prevent future instructions from issuing and limit overall throughput.

Commit also represents the point at which it is known that an instruction will not be squashed from execution due to a taken exception, mispredicted branch, speculative scheduling, memory replay trap, or other microarchitectural event. While committing instructions in-order can overconsume processor resources, it pro-

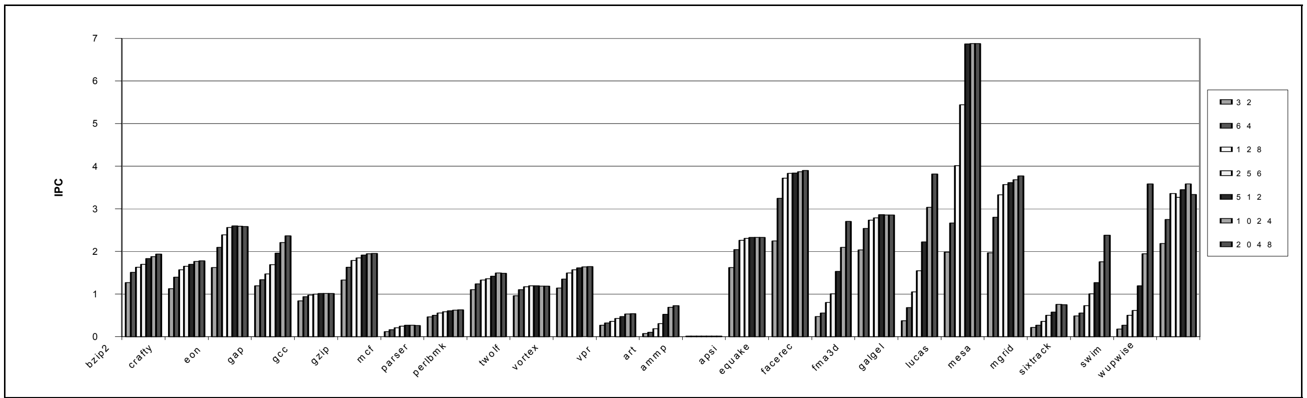

FIGURE 1. Sensitivity of performance to resource availability. Each bar corresponds to an equal-sized ROB, issue queue, and rename register file. The LSQ is sized to half of the ROB size**.**

vides a clear and intuitive way to guarantee that an instruction will not incorrectly modify the architected state under one of these conditions. In this manner, in addition to updating processor state and freeing processor resources, commit also provides a safety mechanism by which incorrect state changes are avoided.

We assert that the constraint that instructions must commit in program order is over-conservative. We provide a set of necessary, rather than sufficient, conditions under which in can be guaranteed that an instruction will not be squashed and can thus can be permitted to commit out-of-order. These conditions provide an equivalent, non-speculative safety mechanism to in-order commit that utilizes resources considerably more efficiently and yields higher performance.

4.1 Necessary Commit Conditions

This section presents a series of necessary conditions required for commit. If an instruction fulfills all of them, it can be permitted to commit regardless of whether it is the oldest instruction in the machine or not. The microarchitectural modifications required to enable mechanisms that allow instructions to commit out-of-order are detailed in Section 5.

4.1.1 Not Finished

In order to commit, an instruction must finish execution. It needs to complete its access of any functional units allocated to it as well as produce any results dictated by the instruction set semantics. Because dynamically scheduled processors decouple execution from retirement, instructions can finish long before they leave the machine. This represents the opportunity for committing them out-of-order, and identifying such instructions is one of the primary contributions of this work.

4.1.2 WAR Hazard

A write to a particular register cannot be permitted to commit before all prior reads of that architected register have completed. Thus all older instructions with the same source operand(s) as the later instruction's destination register must have accessed the register file and obtained their correct input values. This prevents the earlier, incomplete instruction from obtaining incorrect operands due to write-after-read (WAR) hazards in a processor that does not implement register renaming.

Most dynamically scheduled processors implement register renaming, which utilizes more physical registers than defined by the architecture. They dynamically allocate result storage locations to individual instructions, and require a mechanism to detect when these locations are no longer live and can be re-allocated to new instructions. The point at which a register is marked dead and added to the register free list is typically when a subsequent instruction is committed that will overwrite its corresponding architected register. The same WAR requirement exists in this case. Although the younger instruction's architected destination register may be a different physical register than the older instruction's, it still cannot be permitted to commit before the younger one. Doing so would free the previous definition of that architected register and cause the physical register still required to be read by the prior instruction to be prematurely placed on the free list. It could then be re-allocated and written before it is has been read.

Consider the code example in Figure 2. Instruction 3 cannot commit before instruction 2 has read logical register r4 (physical register p16) from the register file. This is because committing 3 will add p16 to the register free list, where it could be allocated to subsequent instructions and overwritten, as it is by instruction 4. In this case the previous value of p16 (written by 1) is lost even though it is still required by 2. The hardware mechanisms required to detect such WAR dependencies are presented in Section 5.

```
①  p16 | addi r4,r4,1
②  p11 | addi r3,r4,1     // read p16
③  p12 | add  r4,r7,r6   // free p16
④  p16 | add  r3,r8,r4
⑤      | beq  r10, L0    // taken
⑥  p15 | addi r3,r8,4    // free p16
⑦  p17 | subi r3,r9,1    // free p15
⑧  p16 | addi r2,r1,8
    L0: |
⑨  p18 | add  r11,r3,r4
```

FIGURE 2. Code example. Physical destination registers are indicated left of the dashed line. Dynamic instruction order is indicated with circled numbers**.**

4.1.3 Unresolved Branch

In general, instructions cannot commit unless it is known that they are not on a mis-predicted branch path. Therefore we cannot allow an instruction to commit until all previous branch directions and targets have been resolved and are known to be correct. In Figure 2, instruction 6 follows the branch (predicted not-taken) and writes to architected register r3, which has been mapped to physical register p15. This write cannot be committed until the branch is resolved because doing so would incorrectly free the previous definition of r3 (p16) that is still needed on the correct path (instruction 9).

An exception to this constraint allows instructions to commit beyond unresolved or mispredicted branches if the changes made to the architected state by those instructions are masked during re-execution. One example of such a case occurs if there is a register write-after-write (WAW) condition beyond a potentially mispredicted branch. Consider what would happen if instructions 6, 7, and 8 were permitted to commit before the branch is resolved. If the branch was mis-predicted, then 9 will obtain the incorrect value for r3 (it will obtain the value written by instruction 8 rather than instruction 4). However the second write to register r3 (instruction 7), while still on a potential wrong path of execution, can be committed before the branch is resolved as long as all pending reads of r3 have completed. This is because committing that instruction will free the previous definition of r3, which in this case p15. Freeing p15 will not affect the code along the correct branch path because p15 was already on the free list prior to the branch. In practice such a condition rarely occurs, and we mention it only for completeness; we do not consider it in the detection mechanisms discussed later in the paper.

4.1.4 Exceptions

One of the primary motivations for forcing in-order commit is that it provides a relatively simple mechanism to maintain precise exceptions. Allowing an instruction to commit only when it is the oldest satisfies the constraint that no younger instructions beyond the point of the exception have updated the architectural state of the machine. However it is still possible to maintain these semantics if instructions are permitted to commit out-of-order. When an instruction commits early, it only needs to guarantee that no older instructions will raise an exception. Many dynamically scheduled processors implement precise exceptions by setting a flag in an excepting instruction's ROB entry to indicate than an exception should be raised when that instruction commits [23]. For an instruction to commit early in such an implementation, it needs to check every older ROB entry to verify that this flag is not set nor will be set.

Because many types of exceptions are identified early in the pipeline (such as I-TLB misses, invalid opcodes, etc.), this task is straightforward for these types because their flag is marked when a ROB entry is allocated to that instruction. Other types of exceptions occur later in the pipeline and require waiting a longer period of time before it can determine if it can commit early. Examples of this type are page faults and D-TLB misses. If there are outstanding memory operations that occur earlier in program order than a particular instruction, then that instruction cannot commit before those memory operations generate their addresses and access the TLB. This is because if the addresses are not known, it cannot be guaranteed that neither a page fault nor TLB miss will not occur after the younger instruction has committed.

A second example of exceptions that occur later in the pipeline and can therefore potentially hinder the benefit of out-of-order commit are arithmetic operations. An instruction cannot be permitted to commit before an older unfinished arithmetic operation that could raise an exception. However many instruction sets support non-excepting arithmetic instructions that do not trap to the operating system on overflow or underflow conditions. In this case an instruction can commit before older arithmetic operations of this type regardless of whether they have completed or not. A similar case exists for floating point operations. Because many architectures permit any floating point instruction to raise an exception, it may initially appear that no instruction can commit before all older floating point operations have completed. Scientific applications with a large number of long-latency floating point instructions would therefore benefit very little from out-of-order commit. However just as arithmetic instructions exist that cannot raise exceptions, many architectures support a mode of operation that disables floating point exceptions or causes them to be imprecise [20]. For performance reasons many applications are compiled such that they set the processor state in this mode, as enabling precise floating-point exceptions can result in a slowdown of over 10x [9].

One last case of late-occurring exceptions are those corresponding to data-related errors, such as uncorrectable ECC errors. Simply waiting for older memory accesses to generate their addresses is not sufficient to guarantee that exceptions of this type will not occur, and waiting for all memory operations to complete before allowing younger instructions to commit would forfeit much of the benefit of out-of-order commit. However this type of error leads to a machine check in many architectures, and maintaining precise interrupts for a machine check does not make sense in many cases because the application often simply terminates when it encounters such a condition. If precise recovery for data-related errors is required, then the machine can fall back on committing instructions conventionally in program order, or a checkpointing scheme similar to that proposed in [19] can be added.

The previous paragraphs implied that no instruction could commit before any other that can potentially raise an exception. However the WAW condition that allowed certain instructions to commit before unresolved branches can also be applied to precise exceptions. Because an exception changes control flow to the operating system's exception handling routines, they behave similarly to branches. Therefore instructions that adhere to the WAW conditions proposed for branches can also be allowed past potential exceptions.

Asynchronous interrupts are not required to be associated with a specific program counter value and thus can be delayed indefinitely. Therefore the simplest way to deal with them with out-of-order commit is to simply fall back on conventional in-order commit when an interrupt is detected. After all instructions older than those that may have committed early have committed, the machine is in a consistent state and the interrupt can be processed normally.

4.1.5 Replay Traps

For a memory operation to commit early it must be known that an instruction is free from store-load and load-load replay traps. Specifically, it must guarantee that a load's address will not match that of any older unresolved stores (uniprocessor or multiprocessor) or loads (multiprocessor) issued by the same processor. This is because if the older unresolved load or store eventually generates the same address as the younger load, the younger load may have received incorrect data and will need to be squashed and re-executed. Such squashing and re-execution is not possible if that load has committed early and updated the architected state of the machine. Therefore in order to avoid violating this replay trap requirement, all memory operations that are candidates to commit out-of-order must not be younger than any memory operations to unresolved physical addresses. Note that this requirement is already fulfilled by the previous section, which mandates that no instruction can commit before outstanding memory operations to unresolved addresses in order to prevent imprecise memory-access exceptions.

4.1.6 Memory Consistency Model

When a memory operation commits, it exposes itself to the memory hierarchy and to other processors in the system. Therefore instructions that commit out of program order must be careful that they adhere to the defined memory consistency model, which dictates the order in which memory operations must appear to other processors in a system. Memory operations become visible at the point that they become globally ordered with respect to other memory operations executed by all processors in the system. Operations can become ordered before they finish execution, and the sequence that they become ordered is not necessarily the same as the sequence in which they eventually execute. Therefore committing instructions out-of-order requires only that the global order is not perturbed in a manner that violates the consistency model.

This section considers the implications of committing instructions out of program order for both sequential consistency as well as weaker models. In a sequentially consistent system, all instructions must appear as if they were executed in their specified program order. Therefore, memory operations that appear later in program order cannot be globally ordered before those that appear earlier in program order. Maintaining a sequentially consistent execution requires that no memory operations can commit out-of-order until all of that processor's earlier memory operations are ordered with respect to the rest of the system.

Unlike stronger consistency models such as sequential consistency, a weakly-ordered model does not require that instructions appear in program order. We can therefore allow memory operations to commit out-of-order as long as two constraints are upheld. First, memory operations cannot commit out-of-order beyond an unordered memory barrier instruction. Because the processor pipeline is typically flushed on the infrequent occurrence of a memory barrier (and thus there are no younger instructions that are candidates to commit early), adhering to this constraint does not significantly impede the effectiveness of out-of-order commit. Second, we cannot allow a memory operation to commit before a prior unordered reference to the same address.

In practice, memory operations usually become ordered much sooner than they complete. Ordering occurs when an instruction observes its own reference by snooping the memory bus, which is usually shortly after it issues. In this manner instructions can commit before older pending instructions that have suffered cache misses and have been ordered yet may still not complete for a long time. In a weakly ordered system, it is unlikely that older accesses to the same address have not been ordered and we do not expect adhering to this restraint to significantly hamper the effectiveness of out-of-order commit.

4.2 Reasons Instructions Cannot Commit

In order to gauge the potential benefit of out-of-order commit, we first characterize the fundamental reasons why instructions cannot commit early. After each commit cycle we examine each remaining entry in the ROB and assign to it one, several, or none of the necessary conditions discussed in Section 4.1. In this way we can identify the cases in which an instruction could commit out-of-order in a processor that supports such a mechanism. The height of each stacked bar in Figure 3 represents the total number of simulated cycles multiplied by the average ROB occupancy for each benchmark (normalized to 100%). In this manner the graph can be thought of as a profile of the ROB averaged over the duration of the simulation. The bottom segment labeled *none* indicates that none of the conditions in Section 4.1 occurred and the only reason that the instruction did not commit was that it was not the oldest in the machine (provided sufficient commit bandwidth still exists). This portion represents the opportunity lost by restricting commit to be in-order. The stack labeled *finished* are cycles in which instructions in the ROB could not be committed only because they have not finished execution. They may be waiting for input operands to become ready, ALU operations to complete, effective address calculation to complete, or other microarchitectural event to occur. *branch* indicates that an instruction in the ROB cannot commit because an older branch has not been resolved, therefore it is not known whether this instruction is on a mis-speculated path. Similarly, *agen* are those instructions that are behind a memory operation that has not generated its physical address. The memory address may generate a page fault or TLB miss exception and so the earlier instruction cannot commit. *WAR* indicates that an instruction cannot commit because an older instruction has not finished reading that instruction's logical destination register. Note that we omit the final condition relating to the memory consistency model because it is only possible to measure this metric in a simulator that models a multiprocessor coherence protocol (which SimpleScalar does not). These conditions are not mutually exclusive, and any or all of them may apply to a single ROB entry. Because the intersections of them are too numerous to clearly present in graph form (and none of the combinations are uniformly dominant across benchmarks), we simply use the last bar, *combination*, to represent more than one condition applied.

All of the benchmarks exhibit a non-trivial portion (in two cases over 80%) of ROB entries in which the only reason an instruction cannot commit is that it is not the oldest in the machine. Therefore if we allow the instructions corresponding to these entries to commit out of program order, a large number of ROB slots can be freed for consumption by later instructions, potentially yielding higher performance.

4.3 Results

This section presents results collected by modifying our simulator to commit instructions out of their original program order. During each cycle, any in-flight instruction which is unconstrained by all of the conditions introduced in Section 4.1 is allowed to commit regardless of whether it is the oldest in-flight instruction. Because adhering to these constraints is sufficient to guarantee that this instruction will not be squashed or re-executed, committing instructions in this manner is non-speculative and requires no checkpointing or recovery code.

As previously discussed, eagerly committing instructions out of program order yields better resource utilization and allows additional instructions to dispatch that may not have been able to otherwise. If any of these newly issued instructions can also commit out-of-order, then their resources are freed for yet more instructions to issue. In this manner out-of-order commit can allow execution to

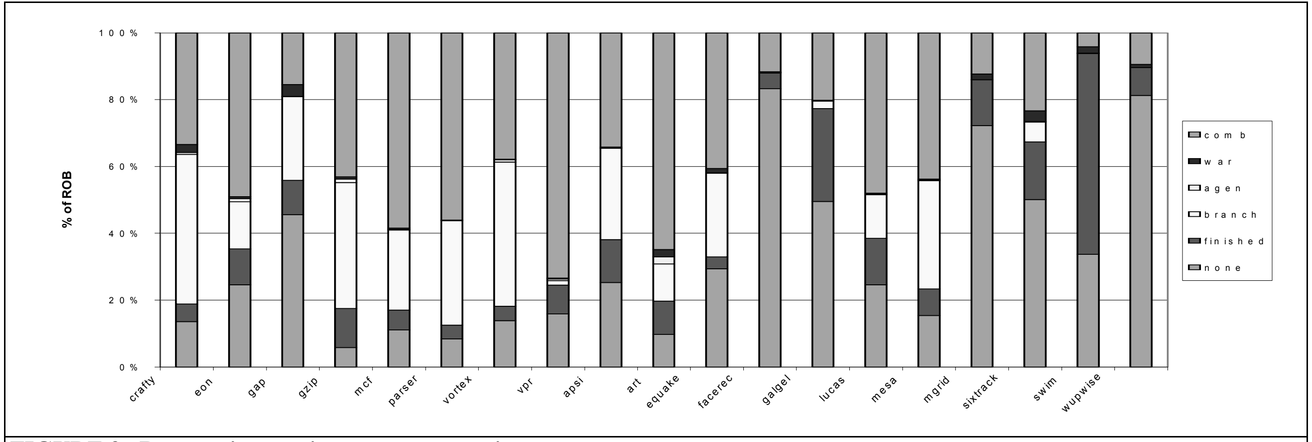

FIGURE 3. Reasons instructions cannot commit.

run ahead of traditional execution and expose more instruction-level parallelism.

This section evaluates how far ahead out-of-order commit allows execution to continue. Figure 4 presents several metrics for each benchmark. *rob_base* indicates the average dynamic distance in terms of issued instructions between the oldest and youngest instructions in the machine. It is simply the average ROB occupancy (average number of in-flight instructions) for the duration of the benchmark. Notice that many of the benchmarks have an average ROB occupancy close to the maximum ROB size of 64 entries. This corroborates the limit study in Section 3.0, in that these benchmarks were selected because their overall performance is sensitive to the degree of available resources. *rob_ooc* indicates the same metric in a processor that can commit instructions out of program order according to the constraints in Section 4.1. Note that in this case, *rob_occ* does not strictly represent the ROB occupancy because instructions between the oldest and youngest may have already retired early from execution. For example, *mgrid* indicates that in the base case there is an average distance of 63 instructions issued between the oldest and youngest instruction in the machine. Because none of these can commit before the oldest instruction, the average ROB occupancy is also 63 instructions. The bar labeled *rob_ooc* indicates that in the case of out-of-order commit, there is an average distance of 138 instructions between the oldest and youngest. On average 75 of them have committed early, and there are still an average of 63 instructions in-flight. In order to expose the equivalent degree of instruction-level parallelism as the machine with out-of-order commit, a traditional processor would need to support an ROB size that could contain (on average) 138 instructions. This metric can therefore be thought of as the average ROB occupancy that would need to be sustained in a machine that is limited to committing instructions in-order to achieve the equivalent performance of out-of-order commit. It is the effective average ROB occupancy when out-of-order commit is employed. The following bar, labeled *rob_ooc_full*, is the same metric as *rob_ooc*, however it only represents the average distance between oldest and youngest instruction when the ROB is full. Because out-of-order commit is only beneficial when there are no free ROB slots, this indicates how far execution can run ahead when it would be advantageous to do so. Clearly, these last two metrics would have similar values for benchmarks and processor configurations that cause the ROB to usually be full. Finally, the number that appears over each of the bars in Figure 4 indicates the maximum distance between oldest and youngest instruction at any time during a benchmark's execution. Many benchmarks experience points where the execution runs ahead over 1000 dynamic instructions farther than it would in the base configuration, and over 2000 instructions in one case. Figure 4 also presents occupancy statistics corresponding to LSQ occupancy.

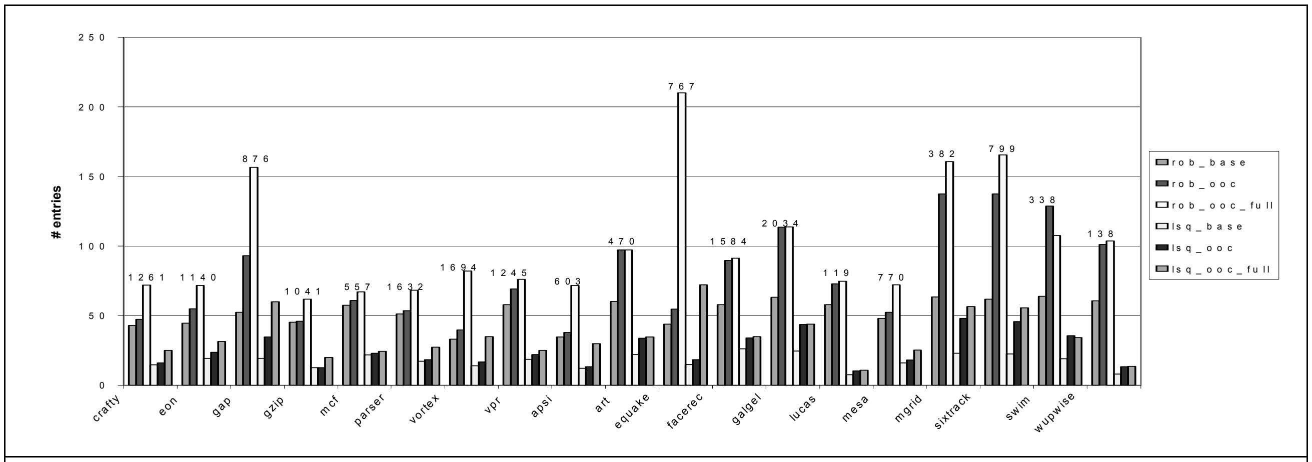

FIGURE 4. ROB (64-entry) and LSQ (32-entry) effective occupancy.

5. Implementation Issues

At this point we have described and characterized the performance benefits of out-of-order commit, but have said nothing of the actual mechanism required to commit instruction out of program order. Typical processors implement the ROB as a circular FIFO queue in which entries are inserted at one end and removed at the other. This leads to a straightforward implementation at the circuit level, as entries remain stationary and only the head and tail pointers move. However permitting instructions to commit from the ROB in an arbitrary order raises issues as to the feasibility towards a real implementation. In this case instead of managing the ROB as a circular queue in which entries are inserted and removed at the head and tail, we need to support removing entries from the queue's center and either collapsing the ROB to fill the void just created, or manage the gaps on a free list that determines where new entries can be added. This section explores the implication out-of-order commit has on microarchitectural design.

5.1 Reorder Buffer Fragmentation

As just mentioned, committing instructions from positions other than the head of ROB has the problem that it creates gaps within the queue. We envision three design alternatives for dealing with such gaps. The first alternative removes these gaps with a collapsing ROB that is similar to collapsing issue queue designs. When an instruction is committed and removed from the center of the queue, all entries above it shift down to fill the newly created gap. The second alternative does not immediately remove the gaps, but rather manages them via a free list structure similar to those used to manage free physical registers. Instead of inserting new instructions into the tail of the queue, new entries are inserted directly into gaps pointed to by the free list. While this design avoids the added complexity of making the ROB collapsible, it loses the desirable property that all of its entries are in program order. The commit logic presented in the next section relies on this order, and it must be determined through the addition of some sort of pointer-based structure. A final option does not reclaim ROB entries at all, and instead only reclaims physical registers and Load/Store slots. The next section shows that this alternative requires associatively searching through a larger number of entries to identify candidate instructions to commit early. Collapsing the ROB does not add significant complexity, therefore the remainder of this section only considers this first design. Exploration of other design alternatives is left to future work.

5.2 Temporal Locality

This section examines the number of sequential ROB entries that need to be examined in order to identify instructions that can commit out-of-order. Clearly we wish to avoid the case in which the ROB needs to be serially scanned from head to tail every cycle in order to identify candidates to commit out-of-order. We begin with the observation that the maximum commit width that needs to be supported need not exceed the dispatch width of the machine. Committing additional instructions offers no performance advantage because it will not result in the ability to dispatch any more instructions. We model an 8-wide superscalar machine, therefore we only need to be able to commit up to a maximum of eight instructions per cycle. Figure 5 graphs the cumulative distribution of the number of consecutive entries that need to be examined in order to identify instructions that can be committed for the benchmark *wupwise*. The y-axis indicates the percentage of all out-of-order commit opportunities that can be captured by looking forward the number of entries on the x-axis from the head of the ROB. *all* indicates the average number of entries to look ahead to capture the maximum benefit of out-of-order commit (up to the dispatch width of eight). *any* indicates the number of entries required to look ahead to capture any benefit (i.e. to identify at least one instruction that can commit out-of-order). *any_NC* and *all_NC* represent the same two measurements for the third design alternative presented above, in which ROB entries are not reclaimed and the ROB is not compacted (*NC*). The number of instructions required to be examined is considerably higher to capture the same opportunity in this case because instructions whose physical registers have been already been reclaimed are not removed from the ROB. The existence of this "dead space" requires serially looking forward a larger number of entries. We observe that if ROB entries are reclaimed, examining the first 20 ROB entries will capture over 90% of the cases in which the maximum benefit of out-of-order commit is achieved (i.e. the point at which there are no other instructions that can commit or a dispatch width's worth of instructions has already committed), and 100% of the cases in which at least one instruction was found to commit out-of-order. However if ROB entries are not removed, then looking ahead 20 ROB entries will never capture all of the benefit and only 18% of the time capture any benefit. This represents a potential drawback of techniques that do not recycle ROB entries. We choose *wupwise* because it clearly illustrated this point, however the remainder of the benchmarks exhibit similar trends.

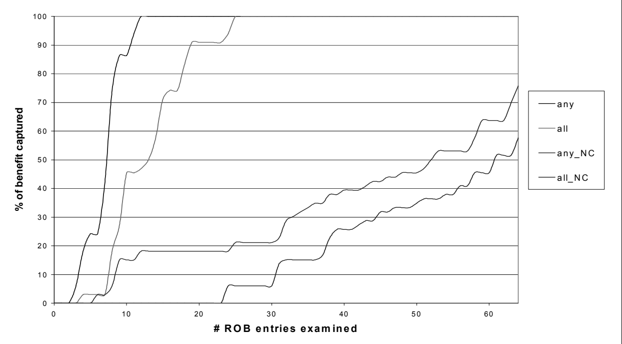

FIGURE 5. Cumulative distribution of distances of instructions that can commit for *wupwise*.

5.3 Performance

Section 4 examined the increase in resource efficiency enabled by committing instructions out-of-order. This section quantifies how this efficiency translates to overall system performance. Figure 6 presents speedup for a variety of processor configurations. We measured IPC with and without out-of-order commit enabled for a 64-entry ROB, 32-entry LSQ (normalized to the base configuration for each benchmark), as well as a 128-entry ROB, 64-entry LSQ. The left-most bar is the base configuration that commits instructions in program order. We also compare against a similar technique (second bar) implemented in the DEC Alpha 21164. Although the 21164 is a mostly in-order machine, it allows loads that miss the L1 cache to commit before their data has been returned by the memory system [8]. In this manner younger

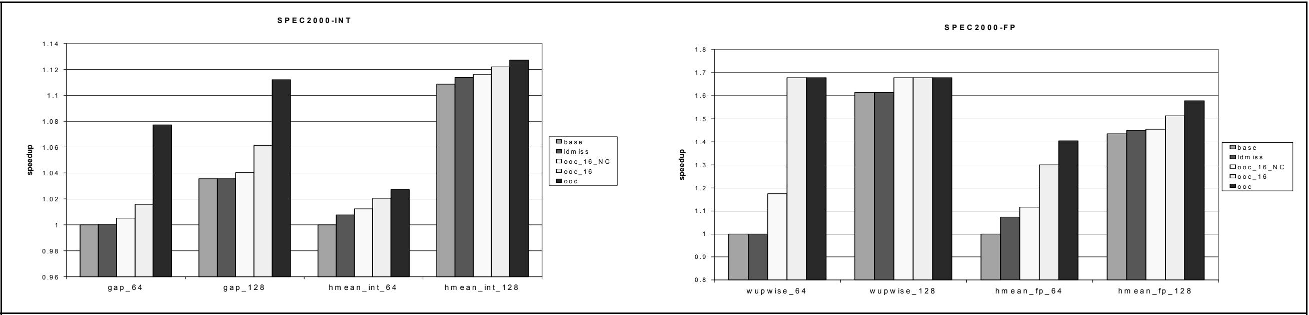

FIGURE 6. Speedup due to out-of-order commit. Each graph displays the harmonic mean of all benchmarks in the set (integer or floating-point), as well the benchmark that exhibited the most speedup (*gap* and *wupwise*).

instructions can continue to commit up until another condition blocks commit or a use of the value loaded from memory is encountered. In most cases this technique achieves only a small fraction of the benefit of our more general technique. The third and fourth bars reflect processor configurations that commit instructions out of program order, but limit the window in which candidates are selected to the 16 oldest instructions. The third bar removes instructions that commit early from the ROB and performs compaction, while the fourth bar does not and only frees other resources, such as physical registers and LSQ slots. The final bar corresponds to committing instructions out of order with compacting, and places no restrictions on which ROB slots candidate instructions reside.

Due to space considerations, Figure 6 presents average (harmonic mean) speedups for SPEC2000 integer and floating-point benchmarks, as well as data corresponding to the benchmarks that exhibit the most speedup.

5.3.1 Unrestricted Commit Window

We observe that committing instructions out-of-order achieves an average speedup of 40% and maximum speedup of 68% (*wupwise*) across the floating-point benchmarks executed with a 64-entry ROB, and a 10% average and 16% maximum (*wupwise*) speedup for a more aggressive resource configuration with a 128-entry ROB and 64-entry LSQ. Some of the these benchmarks achieved dramatic speedups, with three benchmarks over 50% and one at 68% (some results not shown). Integer benchmarks achieved an average speedup of 3% and a maximum speedup of 8% (*gap*) for a 64-entry ROB, and 2% average and 7% maximum (*gap*) speedup for a 128-entry ROB.

Not surprisingly, the benchmarks that benefit the most from a larger number of ROB / LSQ resources (that is enabled by either larger structures in the earlier limit study, or higher efficiency through out-of-order commit) are those whose ROB / LSQ is full a large percentage of the time. For example Figure 1 shows that *mgrid* is highly sensitive to these resources, achieving a speedup of over 200% when the ROB and LSQ sizes are increased. This makes sense, given that the ROB in the base case of *mgrid* has an average occupancy of 63.4, which is near the maximum ROB size of 64 entries (suggesting that the lack of these resources is limiting performance). When out-of-order commit is enabled *mgrid* achieves an effective ROB occupancy of 138 instructions and the percentage of time that it is full drops dramatically, leading to a performance boost of 41% (some results not shown). In this case out-of-order commit with a 64 ROB entries and rename registers, and 32 LSQ entries outperforms a machine model equipped without out-of-order commit but with twice as many physical resources.

5.3.2 Restricted Commit Window

The above analysis compared the base configuration with one that could commit any instruction that met the conditions presented in Section 4.1 regardless of that instruction's location in the ROB (i.e. the first and last bars of Figure 6). However, as discussed in Section 5.2, committing instructions that are towards the tail of the ROB when many younger instructions are in-flight requires associatively searching through a large number of ROB entries. Because such a search may exceed cycle time constraints, Figure 6 also presents performance data when only the 16 oldest ROB entries are examined as commit candidates. This technique achieved between 75% and 100% of the speedup of an unrestricted out-of-order commit configuration for all of the benchmarks (the difference between the last two bars in Figure 6). In fact, some of the benchmarks that benefited the most from out-of-order commit achieved all of this benefit from examining only the first 16 entries, such as *wupwise*. However if committed instructions are not removed from the ROB, corresponding to the middle bar in Figure 6, much of this benefit is lost. In the case of *wupwise,* looking forward 16 entries in the ROB will capture only 25% of the benefit.

5.4 Microarchitecture Design

This section describes the hardware modifications to the ROB required to support committing instructions out-of-order. The previous section contended that only a limited window corresponding to the oldest instructions need to be examined to identify those that can commit. Here we denote this commit window size as *n*.

Figure 7a shows a high level diagram illustrating the additional hardware required. ROB entries are separated by *w*-input multiplexors that implement the collapsing functionality described in Section 5.1 (where *w* denotes the dispatch width of the machine). A conventional ROB is typically implemented as a circular FIFO queue where instructions are inserted at the tail and removed from the head. The head and tail pointers are continually updated by adding or subtracting modulo the size of the queue. Because our design moves the entries themselves rather than the pointers, the location of the head is fixed at the first entry, and the tail is fixed at the last entry. When instructions within the last *n* entries commit and are removed, all younger entries shift down to fill the gaps. New instructions are inserted into the top of the ROB in the top *w* locations and "trickle" down to fill the holes created by the recently committed instructions until all instructions form a contiguous set located at the bottom the ROB.

In a typical processor a number of status flags in the ROB

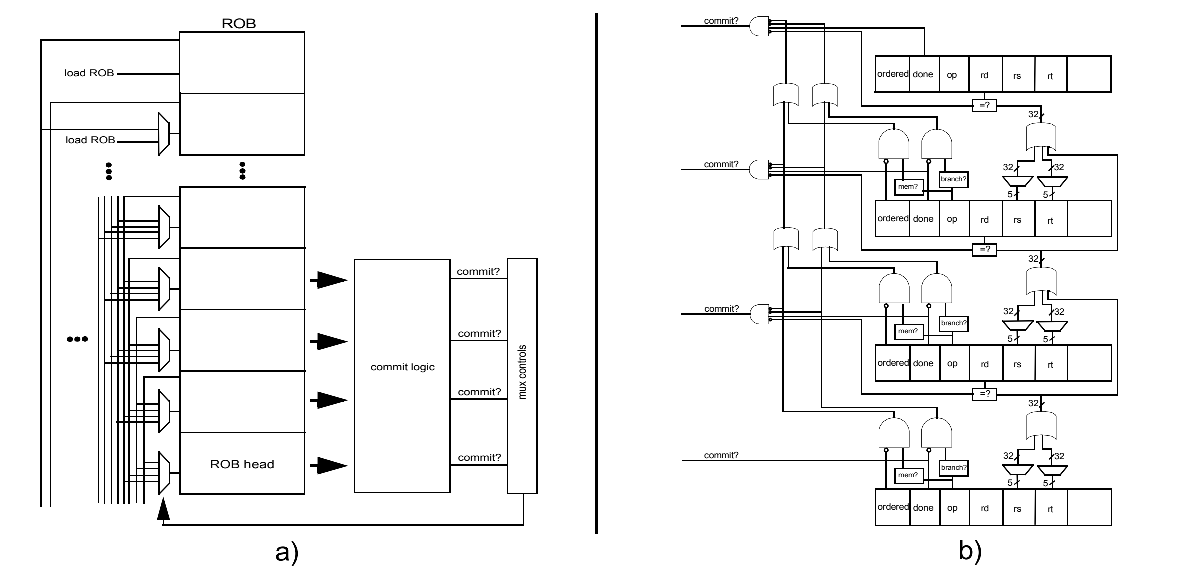

FIGURE 7. Microarchitecture of out-of-order commit (n=4, w=4). a) high level diagram. b) expanded (mirror) view of commit logic (sequential consistency).

need to be continually updated as an instruction passes through various stages of execution. Indexing the ROB to locate instructions is traditionally straightforward, as instructions remain in the same physical location throughout their lifetime. However in a collapsing ROB indexing is complicated by the fact that instructions' locations can change by shifting down. To deal with this situation, instructions are assigned a unique identifier when they are dispatched and status updates are performed by associatively indexing the ROB by this ID. Previous designs have leveraged clever circuit techniques to achieve highly associative indexing (e.g. the 128-way set-associate L1 cache in the IBM Power3 processor [11]). We therefore do not believe that this represents a fundamental circuit design limitation.

Determining which instructions can commit entails reading the *n* oldest entries from the ROB into the *commit logic* which applies the restrictions presented in Section 4.1. The *n*-bit output of the *commit logic* indicates which instructions can commit. This information is passed to the *mux control logic*, which determines how the ROB entries need to shift in order to remove committed instructions.

Figure 7b provides a more detailed view of the *commit logic* block in Figure 7a that determines which, if any, instructions can commit. Each entry needs to determine if it is eligible to commit based on the conditions described in Section 4.1. Determining if an instruction has completed is completely a local decision. However the remaining conditions require information concerning lower (older) entries in the ROB -- for example whether there are older unresolved branches, older unresolved memory addresses, older unordered instructions, or older instructions that have pending register reads. This information needs to be propagated upwards to younger entries to determine which instructions can commit, resulting in several long chains of serialized logic. The fact that propagating these signals upwards may take prohibitively long is the primary reason why the out-of-order commit window needs to be limited to a subset of the ROB. Sections 5.2 and 5.3.2 show that we can achieve adequate performance with only limited complexity.

6. Related Work

There is a long history of published work in the area of support for precise exceptions and maintenance of architected state in the presence of out-of-order execution (e.g. [23,10]). Similarly, many researchers have focused on scalable, realizable approaches for building large register files and other structures needed to support deep speculative execution [24,26,22,3,25,6,5,7]. Other studies have focused on efficient use of a limited number of storage resources by exploiting program semantics, dynamic behavior, or the register values themselves [18,16,12,17,21]. Two recent studies have demonstrated that very large instruction windows are beneficial for tolerating long memory latencies, and can in fact be filled with useful instructions. The first focused on exploiting distant ILP by allowing speculative pre-execution in a small portion of the processor's window [2]. The other concentrated on complexity-effective techniques for expanding the research of the scheduling logic by removing instructions that are dependent on pending cache misses and allowing newer independent instructions to issue [15]. Finally, two recent studies that build large virtual windows using checkpointing also lend credibility to the idea that very large windows are worth pursuing [19,4,1].

Moudgil et. al [21] described several of the same conditions necessary to free physical registers as presented in Section 4. Specifically, it proposed using register map checkpoints to speculate beyond unresolved branches, and noted that registers cannot be freed if they are mapped by any saved mapping tables corresponding to pending branch resolutions. However this technique did not commit instructions in that they still occupied ROB slots. Their work also relies on a history buffer (or similar mechanism) that retains previous register values in order to recover from precise exceptions.

Our work differs fundamentally from all of this prior work. To our knowledge, ours is the first study to systematically characterize reasons for deferring commit of individual instructions in a processor's window, and the first to promote true, non-speculative, out-of-order commit of instructions beyond the oldest unfinished instruction in the window. To emphasize the novelty of our work: we have shown that a nontrivial portion of an instruction window can be deallocated by finding instructions that are perfectly safe to commit, committing them, freeing the resources they occupy, and enabling deeper execution past the oldest uncommitted instruction. We have also demonstrated that this technique allows execution to run ahead on the order of hundreds and even thousands of instructions with a relatively modest hardware budget.

7. Conclusions

This work contends that the typical convention of committing instructions in program order is overly conservative. We provide a detailed analysis of what the term commit really means in a dynamically scheduled processor as well as describe the necessary conditions that need to be met before instructions commit. We then propose mechanisms by which instructions can commit out-of-order and present a variety of results that indicate the effectiveness of such a technique towards increased performance. Finally we illustrate the required microarchitectural modifications to existing hardware required to realize out-of-order commit and discuss performance trade-offs. We believe that existing trends in high performance processor design will increase resource contention, and techniques such as these will be increasingly important in future designs.

References

[1] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 423. IEEE Computer Society, 2003.

[2] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically allocating processor resources between nearby and distant ILP. In *28th International Symposium on Computer Architecture*, July 2001.

[3] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *International Symposium on High-Performance Computer Architecture*, February 2002.

[4] A. Cristal, M. Valero, J.-L. Llosa, and A. Gonzalez. Large virtual ROBs by processor checkpointing. Technical Report UPC-DAC-2002-39, Univ. Pol. de Catalunya, July 2002.

[5] Jose-Lorenzo Cruz, Antonio Gonzalez, Mateo Valero, and Nigel T. Topham. Multiple-banked register file architectures. In *International Symposium on Computer Architecture*, 2000.

[6] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register file design considerations in dynamically scheduled processors. In *International Symposium on High-Performance Computer Architecture*, pages 40–51, 1996.

[7] Antonio Gonzalez, Jose Gonzalez, and Mateo Valero. Virtual-physical registers. In *International Symposium on High-Performance Computer Architecture*, pages 175–184, 1998.

[8] Linley Gwennap. Digital leads the pack with 21164. *Microprocessor Report*, 8(12), September 1994.

[9] J.L Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[10] W.M. Hwu and Y.N. Patt. Checkpoint repair for out-of-order execution machines. In *International Symposium on Computer Architecture*, pages 18–26, June 1987.

[11] IBM Microelectronics Division. *PowerPC 604 RISC Microprocessor User's Manual*, 1994.

[12] Stephan Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *International Symposium on Microarchitecture*, pages 216–225, 1998.

[13] Ilhyun Kim and Mikko H. Lipasti. Half-price architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-01)*, June 2003.

[14] AJ KleinOsowski and David J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 12, June 2002.

[15] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *International Symposium on Computer Architecture*, pages 59–70, 2002.

[16] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9), 1999.

[17] Luis A. Lozano and Guang R. Gao. Exploiting short-lived variables in superscalar processors. In *International Symposium on Microarchitecture*, pages 292–302, 1995.

[18] Milo M. Martin, Amir Roth, and Charles N. Fischer. Exploiting dead value information. In *International Symposium on Microarchitecture*, pages 125–135, 1997.

[19] Jose Martinez, Jose Renau, Michael Huang, MIlos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *International Symposium on Microarchitecture (MICRO)*, November 2002.

[20] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors, 2nd edition*. Morgan Kauffman, San Francisco, California, 1994.

[21] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. Technical Report TR93-1379, 1993.

[22] Matt Postiff, David Greene, Steven Raasch, and Trevor N. Mudge. Integrating superscalar processor components to implement register caching. In *International Conference on Supercomputing*, pages 348–357, 2001.

[23] J.E. Smith and A.R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[24] John A. Swenson and Yale N. Patt. Hierarchical registers for scientific computers. In *International Conference on Supercomputing*, pages 346–353, 1988.

[25] Steven Wallace and Nader Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1996.

[26] Javier Zalamea, Josep Llosa, Eduard Ayguade, and Mateo Valero. Two-level hierarchical register file organization for VLIW processors. In *International Symposium on Microarchitecture*, pages 137–146, 2000.