Journal of Signal Processing Systems (2019) 91:379–397
https://doi.org/10.1007/s11265-018-1369-4

Maximizing Limited Resources: a Limit-Based Study and Taxonomy of Out-of-Order Commit

Mehdi Alipour[1] · **Trevor E. Carlson**[2] · **David Black-Schaffer**[1] · **Stefanos Kaxiras**[1]

Received: 10 August 2017 / Revised: 11 March 2018 / Accepted: 16 April 2018 / Published online: 26 April 2018

Abstract

Out-of-order execution is essential for high performance, general-purpose computation, as it can find and execute useful work instead of stalling. However, it is typically limited by the requirement of visibly sequential, atomic instruction execution—in other words, *in-order instruction commit*. While in-order commit has a number of advantages, such as providing precise interrupts and avoiding complications with the memory consistency model, it requires the core to hold on to resources (reorder buffer entries, load/store queue entries, physical registers) until they are released in program order. In contrast, *out-of-order commit* can release some resources much earlier, yielding improved performance and/or lower resource requirements. *Non-speculative* out-of-order commit is limited in terms of correctness by the conditions described in the work of Bell and Lipasti (2004). In this paper we revisit out-of-order commit by examining the potential performance benefits of lifting these conditions one by one and in combination, for both non-speculative and speculative out-of-order commit. While correctly handling recovery for all out-of-order commit conditions currently requires complex tracking and expensive checkpointing, this work aims to demonstrate the potential for selective, speculative out-of-order commit using *an oracle implementation without speculative rollback costs*. Through this analysis of the potential of out-of-order commit, we learn that: a) there is significant untapped potential for aggressive variants of out-of-order commit; b) it is important to optimize the out-of-order commit depth for a balanced design, as smaller cores benefit from reduced depth while larger cores continue to benefit from deeper designs; c) the focus on implementing only a subset of the out-of-order commit conditions could lead to efficient implementations; d) the benefits of out-of-order commit increases with higher memory latency and in conjunction with prefetching; e) out-of-order commit exposes additional parallelism in the memory hierarchy.

Keywords Superscalar processors · Out-of-order commit · Performance evaluation · Memory hierarchy parallelism

1 Introduction

Typical dynamically-scheduled superscalar processors execute instructions out-of-order but *commit in-order* to present to the programmer the illusion that instructions execute *atomically* and *sequentially* as intended by the program. In this context, precise interrupts are easily provided as the processor verifies correct execution before each instruction is committed [25].

The disadvantage of in-order commit (IOC) is that it ties up resources (such as reorder buffer [ROB] entries, load-store queue [LSQ] entries, and physical registers) for a much longer time than is necessary for correct execution. In-order commit ties up resources until all instructions complete and commit in the correct sequential program order. This means that execution is halted when any of the resources are exhausted: when the ROB fills up, or when we run out of either LSQ entries or registers. To overcome this hurdle, designers size these structures so that they minimize the chances of any single resource becoming exhausted, creating a *balanced* microarchitecture. This, however, contributes to the power-inefficiency of the

✉ Mehdi Alipour
mehdi.alipour@it.uu.se

Trevor E. Carlson
tcarlson@comp.nus.edu.sg

David Black-Schaffer
david.black-schaffer@it.uu.se

Stefanos Kaxiras
stefanos.kaxiras@it.uu.se

[1] Department of Information Technology, Uppsala University, Uppsala, Sweden

[2] Department of Computer Science, National University of Singapore (NUS), Singapore, Singapore

OoO cores. At the very least, after a certain point, increasing the size of the LSQ (which is usually implemented as an expensive CAM) or the size of the register file results in an increase in energy consumption that far exceeds the performance benefit, a significant disadvantage for power-constrained processors. Thus, the incentive for pursuing *out-of-order commit* (OOC) lies in the promise of higher performance with fewer resources. A turning point in our understanding of out-of-order commit came with the work of Bell and Lipasti [5] who articulated the limiting factors for non-speculative OOC. The necessary conditions to allow an instruction to be committed range from the completion status of the instruction itself, to the branch prediction and exception state of intervening instructions. Several proposals for out-of-order commit, implicitly or explicitly, abide by these conditions, *potentially harming efficiency* to enforce them (Fig. 1).

1.1 Analyzes Performed

The question we explore in this paper is what could the performance gain be if we had the means to evade any one or even all of these conditions together. To the best of our knowledge, this work is the first to evaluate the potential performance contribution of relaxing each individual commit condition. For example, we investigate the potential performance gain if we could correctly commit past unresolved branches, stores with unresolved addresses, or instructions that can generate exceptions. We explore the interactions of out-of-order commit across a range of important processor design points, including variations in: prefetchers, MSHRs, memory latency, branch prediction, and the size of the dynamic instruction window. Answering these questions allows us to understand the most profitable conditions to address. Our work confirms previous studies [2, 5, 21] that the least aggressive core benefits the most from least aggressive out-of-order commit and in addition, by introducing a taxonomy, for the first time we show that more aggressive cores gain more, comparatively, from aggressive out-of-order commit. All in all, our study shows that as a future direction, not only safe but also unsafe out-of-order commit appears to be very promising. This is especially true if the potential benefits can be tapped with much more efficient and selective mechanisms to guarantee correctness, instead of bulk checkpointing and rollback. Already, such non-speculative mechanisms have been proposed [24].

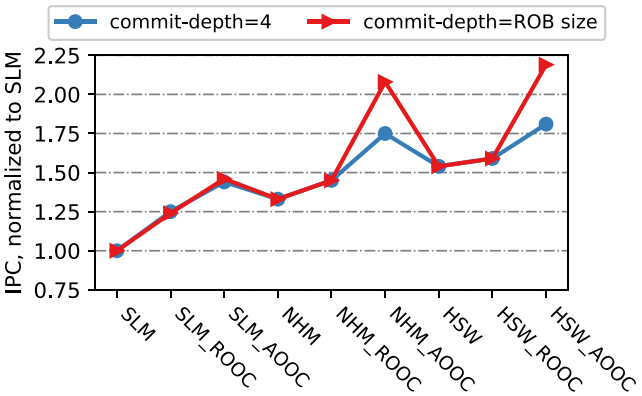


Figure 1 Performance comparison (harmonic mean of IPCs across SPEC CPU2006 [16]) of in-order and two types of safe out-of-order commit, reluctant (ROOC) and aggressive (AOOC), with a commit depth of 4 and ROB size for increasingly aggressive microarchitectures. These experiments respect all traditional commit conditions, and show that aggressive out-of-order commit can reach the performance of the next class of processor.

1.2 Why We Do it

Bell and Lipasti first articulated the conditions for "safe" non-speculative out of order commit [5] in an effort to tackle the problem of improving single-thread performance. This was at the same time that IC manufacturing broke through the 100nm technology node [3, 10]. However, the acceptance for OOC architectures has been slow. Today, significantly improving single-thread performance in an energy-efficient manner remains a challenge. The goal of this work is to help researchers to hone in on the most profitable aspects of OOC by offering: i) a detailed exploration of the limits of out-of-order commit conditions and ii) a taxonomy for out-of-order commit [2].

1.3 Analyzes Outside the Scope of this Work

However, in this paper we do not quantify the *cost* that would be required to guarantee *correctness* when committing past any or all of these conditions, as this would be tied to a specific hardware or software mechanism. Instead, we evaluate the potential performance benefits available as a means to gage the potential of future proposals in relation to their cost. In the same vein, we have not explored every possible core optimization that could be construed as extending the instruction window outside the core (e.g, run-ahead execution). In this work we chose to study three *common* architectures to provide baselines with understood costs.

1.4 Contribution

We study *safe*—non-speculative—out-of-order commit (refraining from committing until all conditions are met) and *unsafe*—speculative—out-of-order commit (committing even before all conditions are met assuming the potential for correct recovery). In addition to studying both safe and unsafe out-of-order commit as a minimum and

maximum potential performance improvement, we also study out-of-order commit in two additional dimensions: aggressiveness and degree of speculation:

- **Aggressiveness:** A dimension that determines the potential benefit is the aggressiveness of the out-of-order-commit implementation: how far into the instruction window we try to commit. Prior work [22] examines this aggressiveness in their implementation of checkpoint-based out-of-order commit, where they show a middle-ground in aggressiveness is needed to mitigate checkpoint-based penalties while still improving performance.

  In addition, work on non-speculative out-of-order commit [5] has recognized that a restricted commit window can achieve a good portion of the performance improvements compared to an unrestricted, unlimited out-of-order commit implementation.
- **Degree of speculation:** Apart from the aggressiveness of commit, we also consider varying degrees of speculation. The insight of this study is that selective speculation, preserving or relaxing of one more conditions necessary for out-of-order commit, could lead to more efficient implementations. In fact, recent work [15, 24] has shown that this might be the case for load instructions.

1.5 Results

This study aims to provide a guidepost for each major parameter of out-of-order commit to provide an upper bound on the performance benefits given a particular aggressiveness and degree of selective speculation. In this work, we show that

- there is significant untapped potential for *unsafe* out-of-order commit beyond traditional in-order and *safe* out-of-order commit; the gap widens in more powerful architectures;
- for energy-efficient cores with moderately-sized instruction windows, *reluctant*,[1] limited, out-of-order commit is sufficient to reap the most benefit, while aggressive versions of out-of-order commit become a requirement for larger cores;
- the order of importance of the commit conditions changes depending on the type of application, the architecture (limited or aggressive OoO processor), and the aggressiveness of out-of-order commit (commit depth);
- we unexpectedly found that a focus on specific out-of-order commit conditions could be an important future direction for high-performance, efficient out-of-order processors;
- the potential benefits of out-of-order commit increases with memory latency (relatively more for unsafe) while the benefits of the prefetching strategy that we picked are orthogonal to out-of-order commit benefits. This raises the enticing possibility of reducing system-wide silicon financial cost without compromising performance by coupling dense but higher-latency (slow-but-efficient) DRAM with out-of-order commit cores;
- out-of-order commit increases memory hierarchy parallelism [7];
- While it is generally acceptable that by releasing pipeline resources as early as possible, out-of-order commit improves performance in minor and small cores relatively more than in large cores, in this work we show that this is only true for reluctant out-of-order commit. In fact, performance improvement in large out-of-order cores can exceed that of smaller cores if aggressive out-of-order commit is employed;
- Our results show the potential for future systems that implement out-of-order commit, and indicate which are the most promising directions (safe vs. unsafe commit, and which of Bell and Lipasti's conditions [5] are most important to support) for future designs.

The rest of this paper is organized as follows. In Section 2 we first provide an overview of the conditions that need to be honored for in-order commit as well as provide an overview of out-of-order commit. Next, in Section 3 we present our evaluation methodology and simulated system configurations. Section 4 provides a detailed performance analysis for each out-of-order commit condition, both from the point of view of aggressive and reluctant out-of-order commit. In Sections 5 and 6 we compare the *early release of physical register* and *memory hierarchy parallelism*, respectively, with Out-of-Order commit; extending previous work [2]. Finally, the related works and conclusion are presented in Sections 7 and 8 respectively.

2 Out-of-Order Commit

The introduction of the reorder buffer (ROB) to provide in-order commit in an out-of-order scheduled superscalar pipeline was an important advance for computer architecture culminating an effort that started with Tomasulo's algorithm, and included techniques such as reservation stations and the register update unit (RUU) [26]. The reorder buffer maintains precise architectural state in the presence

[1] Reluctant out-of-order commit continues normal in-oredr commit and is only enabled when the hardware cannot continue to make forward progress. This mode switching happens early enough which is a cycle before the related queue (ROB, IQ, LSQ) is full so that the CPU stall is avoided

of interrupts, unknown memory dependencies, or memory re-orderings that can perturb the ordering required by a memory consistency model. Out-of-order commit, on the other hand, attempts to break this rigid updating of the architectural state either in a *safe* way (i.e., one that does not require *additional* speculation and rollback to revert changes to the architectural state) or in an *unsafe* way (i.e., one that does).

2.1 Safe vs. Unsafe OOC

A turning point in our understanding of out-of-order commit came with the work of Bell and Lipasti [5] in the form of a number of limiting conditions for *safe* out-of-order commit. The necessary conditions to allow an instruction to be committed out-of-order are:

1. The instruction is completed: instructions can commit only after their completion.
2. The instruction is not involved in memory replay traps. This condition simply says that we cannot commit speculative loads or their dependent instructions. This condition relates to unresolved memory dependencies or memory consistency enforcement. For example, total store order (TSO) requires a replay of speculative loads that violate load→load ordering when this reordering is detected by other cores.
3. Register WAR hazards are resolved (i.e., a write to a particular register cannot be permitted to commit before all prior reads of that architectural register have been completed).
4. Previous branches are successfully predicted. This condition simply says that we can commit only while on the correct path of execution.
5. No prior instruction in program order is going to raise an exception. This condition provides precise interrupts and is essential in easing handling of, e.g., page faults.

In the rest of this paper we will use the following convention to discuss how we *evade* the Bell/Lipasti conditions.

1. **Safe_OOC** where all out-of-order-commit conditions are preserved. This case provides the minimum potential performance improvement of out-of-order commit, but also the minimum hardware to implement as it does not rely on speculation and rollback beyond what is already available in the out-of-order core.
2. **Unsafe_OOC** where one or more (or all) of the out-of-order commit conditions are evaded (apart from true dependencies). Doing so, the maximum potential performance improvement of out-of-order commit is evaluated, but this may require extra support for speculation and rollback to be able to revert changes in the architectural state that were found to be incorrect after the commit.

2.2 Reluctant vs. Aggressive OOC

Aside from the limiting conditions described above, a separate dimension is the aggressiveness of committing out-of-order. Thus, concerning the mechanics of out-of-order commit, we distinguish two versions:

1. **Reluctant** out-of-order-commit (ROOC): where the out-of-order commit mechanisms are engaged only *when needed* and,
2. **Aggressive** out-of-order-commit (AOOC): where the out-of-order commit mechanisms are continuously active, looking for opportunities to commit instructions as early as possible.

While the *always on* nature of the aggressive out-of-order commit is obvious in its meaning, the *when needed* of the reluctant out-of-order-commit requires clarification. For this paper, reluctant out-of-order commit is engaged only when the core is in imminent danger of going into a complete stall. In other words, we engage reluctant out-of-order commit only when one of the critical resources (ROB entries, registers, load-store queue entries) is all but exhausted and cannot support the fetching of new instructions in the front end of the pipeline. As such, reluctant out-of-order commit acts as a safety valve to release the pressure on resources (just before this pressure reaches a critical point), rather than aggressively trying to keep resource pressure low.

In contrast, aggressive out-of-order commit releases resources more eagerly, but disregards the following issues:

1. it might prove wasteful as traditional in-order commit may still be able to provide sufficient resources for forward progress;
2. it may be futile as the chances of encountering an instruction that restricts further commit (e.g., an unresolved branch) tends to increase with aggressiveness.
3. it creates a significant management problem as out-of-order commit can create gaps in several structures, including the ROB and also the load queue and store queue (which is not completely addressed in prior works [5, 21, 22]).

2.2.1 Commit Width and Depth

The final parameter to explore within the context of out-of-order commit is the commit depth to scan for potential instructions to commit out-of-order. While commit width is the number of instructions that can be committed simultaneously per cycle, the commit depth is the measure of how far the core can scan looking for instructions to commit out-of-order in a given cycle.

2.3 OOC Conditions

In Section 3, we describe the methodology used to commit instructions out-of-order. With this methodology, we can selectively relax the commit conditions described in Section 2.1 (except completion) while still guaranteeing correct execution. For example, by relaxing the branch condition (Unsafe_BR): "committing only down a non-speculative path", we can continue to free resources past unresolved branches but effectively only commit from the correct path. In this paper we do not evaluate the implementation required to relax these conditions, but instead evaluate the potential for performance improvement. We evaluate the maximum potential performance improvement with a speculative out-of-order rollback cost of zero. In this section we provide details on the performance implications of the relaxation of a single and combination of conditions.

2.3.1 Instruction Complete

The core waits for an instruction to finish executing before commit can occur. We do not examine early commit of loads [14, 15] that miss in the cache and instead we consider them available for commit only after the data returns and is bound to the destination register.

2.3.2 Memory Replay Traps (safe_ST and safe_LD)

We describe two sub-cases for this condition:

Store-Load (safe_ST): This condition applies to same-thread memory dependencies involving a store and a load. In particular, we cannot commit a load out-of-order in the presence of a prior store with an unresolved address. If the store and the load prove to be dependent (the load should have taken the value of the store) the commit would have been incorrect. The LD condition disallows the commit of a load and its dependent instructions until all prior stores resolve their addresses and all the memory dependencies are correctly enforced. By relaxing this condition, we can commit loads and their dependent instructions even if prior non-aliasing stores have unresolved addresses.

Load-Load (safe_LD): This concerns memory consistency models that enforce load→load ordering (e.g., Sequential Consistency or TSO). Under this ordering constraint it is possible to allow loads out-of-order as long as this is *not observed in the memory system*. The safe_LD condition disallows the out-of-order commit of loads unless it is guaranteed that the correct order will be observed by the memory system. To relax this condition we allow load→load re-orderings that are not observed by other cores. A very specific case would be a memory mapped IO (MMIO) request that might change the order of memory operations. The MMIO case acts as a 'coprocessor', meaning that we have a multi-processor system here. We ignore memory requests from other cores (IO coprocessor).

2.3.3 WAR Hazards

WAR hazards are already handled by the out-of-order core within the ROB, and we assume a solution such as the Value Buffer [21] for committing out-of-order. Thus, we do not consider this condition further.

2.3.4 Unresolved Branches (safe_BR)

This condition guarantees that we commit only from the correct path of execution. Out-of-order commit should not proceed past unresolved branches until they are correctly resolved. We can relax this condition and commit past an unresolved branch if we are able to undo the commit. To evaluate maximum performance potential we assume a zero *rollback* cost for out-of-order commit mispredictions. However, the normal branch misprediction cost (10 cycles) is faithfully accounted (see Section 4.9 for more details.). In addition, we evaluate the rollback *count* for this condition in Section 4.9.

2.3.5 Exceptions (safe_EXC)

This condition caters to precise interrupts. Enforcing this condition requires that we do not commit past an instruction (floating-point, memory access, or any instruction that may cause an exception) unless we make sure that the instruction will not cause a exception. To relax this safe_EXC condition, we assume the code regions are exception free.

2.4 Safe and Unsafe Out-of-Order Commit

Normally, discussion of out-of-order commit tends to focus on the changes that occur in the back end of the pipeline. However, the purpose of out-of-order commit is to enable the *front end* to proceed. The perception that out-of-order commit increases performance can also be misleading: *instruction execution is not sped up*; it is the removal of conditions that stall the front end that increases performance. More specifically, in an *unconstrained* architecture (unrestricted ROB, registers, and load/store queue entries), *safe* out-of-order commit does not perform faster than in-order commit. For restricted, real-world implementations, Safe_OOC helps to ameliorate this problem.

In contrast, Unsafe_OOC has the capacity to exceed the performance of an unconstrained in-order commit

architecture as it removes penalties needed to guarantee correctness (e.g., correctly handling memory dependencies, correctly enforcing memory consistency ordering, etc.) in situations that the hardware does not have any other means of imposing such correctness. In this case, Unsafe_OOC has the potential to violate correctness and requires a means to revert back to a safe state if a violation occurs. This can be a good trade-off when the conditions that violate correctness are rare. We implement an oracle version of Unsafe_OOC, in that it violates the correctness conditions because it will be safe to do so (and therefore no correctness problems will occur). This removes the need to provide the mechanisms to revert to a safe state in the case of a misspeculation. This model provides a best-case speedup as recovery from misspeculation is *zero cost*.

2.5 Aggressive and Reluctant OOC

Orthogonal to the enforcement of the out-of-order commit conditions, the aggressiveness of out-of-order commit plays a significant role in the resulting performance and the cost it incurs. We introduced two approaches for out-of-order commit: Aggressive (AOOC) and Reluctant (ROOC) out-of-order commit. A good way to describe them is to contrast their main difference: *how often each mechanism is engaged*.

AOOC is engaged all the time, i.e., it constantly tries to find instructions to commit out-of-order if the opportunity arises. In this respect it aims to increase commit bandwidth.

In contrast, ROOC is only engaged when one of the critical resources in the core (ROB entries, registers, Load/store queue entries) is about to be exhausted. ROOC is concerned about front end stalls, not about commit bandwidth. This means that the number of instructions that ROOC needs to find that can commit out-of-order is *limited*: ROOC needs to provide enough free entries in the ROB/registers/LSQ so that the front end can dispatch as many instructions as possible (up to the dispatch width) to the out-of-order engine. Figure 2 shows this with an example. The reason that the commit stage of an in-order commit is blocked is because of an unresolved instruction at the head of ROB. For example, given a four-wide superscalar, shown in Fig. 2, tries to commit four instructions per cycle. While the first instruction at the head of ROB can commit, the second oldest instruction causes the commit stage to block and instead of four, only one instruction commits (in-order).

The behavior of out-of-order commit depends on its aggressiveness:

AOOC: AOOC attempts to find up to *commit-width* instructions so it can satisfy the need to commit at the highest possible commit bandwidth. In our example it aims to find four instructions. However, since only one more is needed to fill the gap in the head group of four leading instructions, AOOC produces an excess of commit-ready instructions that can potentially be exploited in the near future.

ROOC: ROOC on the other hand, aims to find the minimum number of instructions needed to commit (out-of-order) so that no resource is exhausted and the front end can continue to issue instructions at its peak bandwidth. The reason for seeking the *minimum* number of instructions to commit out-of-order is that this minimizes the perturbations in instruction order. This could potentially lead to more efficient hardware implementations. While Fig. 3 only considers the ROB, in the general case all dispatch-related resources are considered in the same way. In our particular example, ROOC needs to find just one instruction to commit out-of-order. The ROB already contains two empty slots, and the in-order commit mechanisms can find one instruction to commit, leaving one left for ROOC.

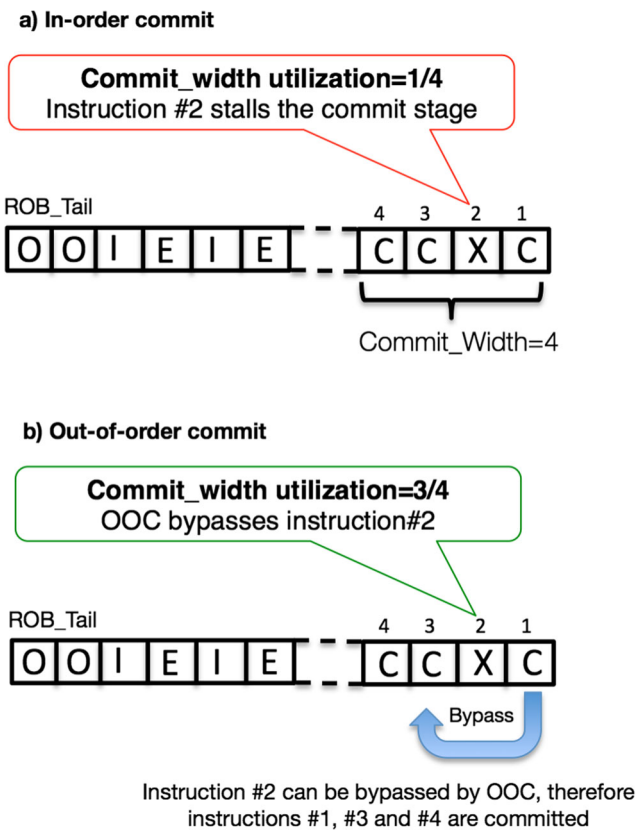


Figure 2 Conceptual comparison of in-order and out-of-order commit (commit depth = 4). **C:** Ready to commit, **X:** Not ready to commit, **E:** Executing, **I:** Issued, **O:** Empty entry.

In contrast, in AOOC mode, there are in total three instructions that commit (instructions #1, #3 and #4 in

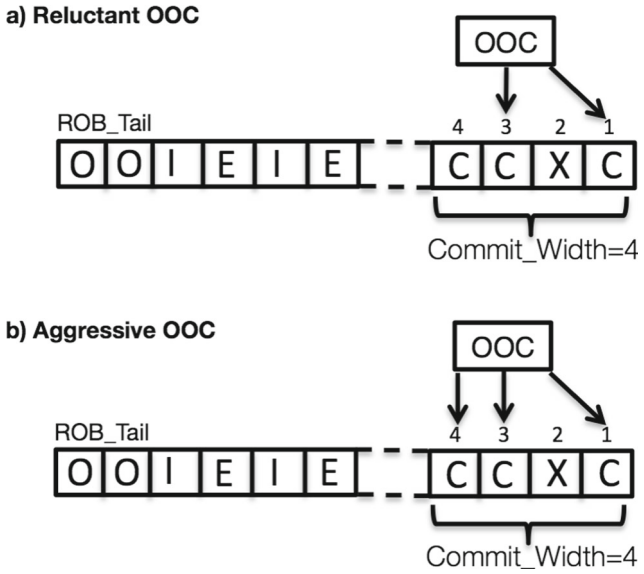


Figure 3 Functionality of AOOC and ROOC. In this example AOOC commits one more instruction than ROOC (commit depth=4). **C:** Ready to commit, **X:** Not ready to commit, **E:** Executing, **I:** Issued, **O:** Empty entry.

Fig. 3). The result is that AOOC needs to scan deeper than ROOC.

3 Methodology

We use the gem5 [6] simulator in *full system* mode to simulate an x86-64 target with a frequency of 3.4GHz. To test our models, we use the SPEC CPU2006 benchmark suite. We use ten uniformly distributed checkpoints from each benchmark. Each simulation checkpoint has three phases that begin with 250M instructions of cache warm-up, followed by 100k instructions of detailed pipeline warm-up and ends with a detailed simulation of 100M instructions. Furthermore, three different configurations, similar to three conventional out-of-order processors, are used: Intel's SLM, NHM, and HSW [9] microarchitectures. Tables 1 and 2 list the detailed configuration of the simulation environment.

To implement our out-of-order commit model, we first configure a simulated machine with a very large number of core resources. We then monitor the number of committed and non-committed instructions that appear in the pipeline, and control, at dispatch, whether we can support additional instructions in the back end of the processor. In this way, we dynamically determine the resource availability in the processor for each cycle based on the out-of-order commit conditions.

Table 1 Baseline core parameters.

Full system simulation	3.4 GHz, x86-64
L1i/d	32KiB, 8-way, 4clk
L2	256KiB, 8-way, 12clk
L3	1MiB, 8-way, 36clk
DRAM	200clk
Branch Predictor	Tournament, front end penalty 10clk
Prefetcher OFF(default)/ON	Stride, degree 8

Table 2 Microarchitecture configuration with reorder buffer (ROB), instruction queue (IQ), load and store queues (LQ/SQ) and integer and floating-point register file (RF) details.

Microarchitecture	D/CW/CD	ROB	IQ	LQ/SQ	RF(INT,FP)
Silvermont-like (SLM)	4/4/8	32	32	10/16	32,32
Nehalem-like (NHM)	4/4/8	128	56	48/36	68,68
Haswell-like (HSW)	4/4/8	192	60	72/42	130,130

Dispatch width (D), commit width (CW) and Out-of-Order commit depth (CD) is the same to enable a fair comparison (SLM hardware has a D/CW of 2). Register values are additional physical registers above architectural state

4 Out-of-Order Commit Evaluation

In this section we analyze the benefits of out-of-order commit on the performance of a number of applications. We look at how the commit bandwidth changes with out-of-order commit, and how the effective resource size of each critical component changes as we enable different out-of-order commit conditions. Next, we show how out-of-order commit conditions affect performance. Safe_OOC and Unsafe_OOC are two extreme points that are defined by either enabling (respecting) or disabling all of the out-of-order commit conditions. This results in a minimum and maximum potential performance improvement across the benchmark suite. To understand the effect of each condition in isolation, we study the effect of each one, both in the presence and absence of other conditions. These studies on Safe_OOC and Unsafe_OOC conditions were conducted for both Aggressive (AOOC) and Reluctant (ROOC) out-of-order commit.

4.1 Microarchitecture Aggressiveness

We target three microarchitectures resembling Intel's Silvermont (SLM), Nehalem (NHM) and Haswell (HSW) as small, medium and large cores (See Table 2 for details). As an overview, Fig. 1 shows the performance improvement for each microarchitecture assuming Safe-OOC (all conditions respected) for all benchmarks on average across SPEC CPU2006 [16]. We can see that in the case of narrow commit depth (four in this figure), a relatively small

out-of-order processor (SLM), has more potential for relative improvement compared to the medium and aggressive microarchitectures. The reason is that the smaller processor (with a shorter instruction reach and, given a balanced design, smaller hardware structures) will more likely stall as it exposes a smaller amount of the potential ILP in an application. In case of a larger commit depth, the more aggressive cores (NHM and HSW) have higher potential performance improvement (See Section 4.4 for a detailed overview). Out-of-order commit frees the processor from the traditional limits, reducing the number of times the processor experiences exhausted resources. In medium and large aggressive cores, thanks to a larger ROB as well as other hardware resources, more intrinsic ILP is extracted by traditional in-order commit, leaving less potential for out-of-order commit with a narrow commit depth.

4.2 The Effect on Commit Bandwidth

Figure 4 shows the distribution of the number of committed instructions per cycle for three different microarchitectures, for both in-order and out-of-order commit, across all SPEC CPU2006 benchmarks. Although an in-order commit, 4-wide commit HSW microarchitecture can retire up to four instructions per cycle, this occurs, on average, less than 20% of the time. In practice we see a large number of commit stage stalls (zero instructions committed per cycle). For out-of-order commit, the distribution shifts toward four instructions per cycle reflecting the improved commit performance (and the resulting improvement in overall performance). Finally this figure shows that for smaller, less aggressive cores (such as SLM, see Table 2), out-of-order commit provides a relatively larger improvement compared to the other microarchitectures because of the short commit depth of four. Larger cores improve more with more aggressive commit depths and out-of-order commit parameters (see Section 4.4 for details).

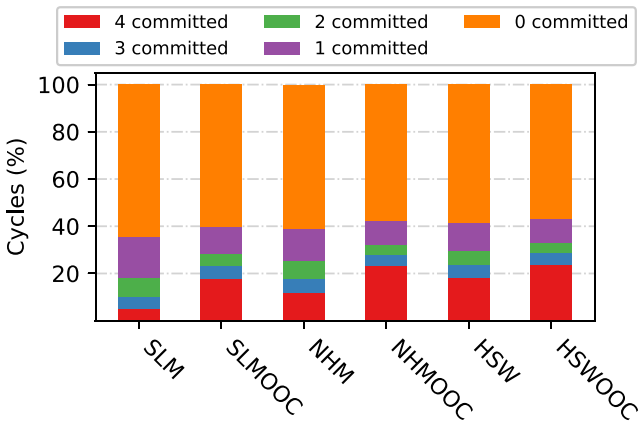


Figure 4 Commit bandwidth distribution for the SPEC CPU2006 benchmarks of a 4-wide core with in-order commit and out-of-order commit respecting all commit conditions (Safe_OOC). Out-of-order commit increases commit pressure even without aggressive speculation.

4.3 The Effect on Resources

One of the main issues that out-of-order commit can help resolve is the early release (and subsequent reuse) of hardware resources that otherwise would still be required to maintain the in-order state in an in-order commit processor. Therefore, with a small ROB (and other appropriately sized structures) given in-order commit, the core can more easily stall when it runs out of resources. For example, consider a ROB size of 32 from a SLM microarchitecture. When the ROB is full it contains 32 micro-operations in flight and the CPU's back end will no longer accept additional micro-operations from the front end. Increasing the physical size of the ROB along with other resources is one potential, but rather expensive, solution to this issue. An alternative solution, and one of the benefits of using out-of-order commit, is the early release of resources. This early release increases the *effective* size of a resource (compared to an in-order commit core) improving performance of the core.

Benchmark Specific Analysis The `xalan` benchmark is one of the top 5 benchmarks with a large number of CPU stalls caused by exhausted resources. Both AOOC and ROOC are very effective for the `xalan` benchmark. OOC is able to provide additional free entries in the ROB, RF and LSQ (see Fig. 7). On the other hand, `leslie3d` has the lowest number of CPU stalls based on exhausted resources, which limits the potential for improvement with OOC.

Figure 5 compares the effective size of the SLM microarchitecture between in-order commit and both safe and unsafe aggressive out-of-order commit (AOOC) models for the SPEC CPU2006 benchmarks. The results are normalized to a fixed size SLM_IOC microarchitecture (see Table 2). An effective size of 1.0 translates to frequent

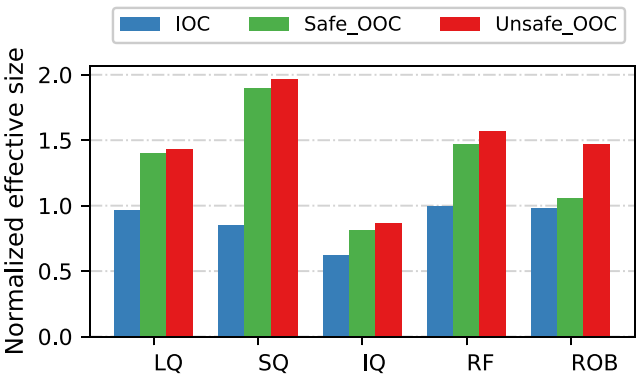


Figure 5 Effective resource use in in-order and Aggressive out-of-order commit across SPEC CPU2006.

stalls due to resource exhaustion, while sizes greater than 1.0 shows the effective resource size increases due to OOC.

For aggressive out-of-order commit, the larger effective sizes show the reach of this technique. We see that the utilization of all structures except for the ROB is almost the same for safe and unsafe out-of-order commit, which allows Safe_OOC to achieve most of the performance of the unsafe version. Nevertheless, the unsafe core is much better at improving the reach of the ROB, allowing applications like `hmmer` continue to show a benefit when moving from safe to unsafe out-of-order commit. See Section 4.5.2 for more details on `hmmer`.

4.4 Evaluation of Commit Depth

To gage the effect of the commit depth (i.e., how far we scan the ROB to find instructions that can commit out-of-order) we impose a hard limit on it and evaluate the effects on the resulting performance. The strictest limit is the commit-width itself: starting to commit out-of-order from the first *commit-width* instructions. We then relax this to the immediate vicinity (e.g., double the commit-width) and progressively relax until we reach the size of the ROB.

In Fig. 6 we see that a commit depth of 4 (equal to commit width) provides the smallest benefit, but also the smallest difference between Safe_OOC and Unsafe_OOC. In addition, the SLM core benefits more from OOC compared to NHM and HSW when commit depth is smaller than 8. At a commit depth of 8 and above, the large aggressive cores benefit much more than the smaller core type with the maximum improvement of Unsafe_OOC at 80%, 119% and 129% for SLM, NHM and HSW, respectively. For aggressive cores, the larger commit depth allows for continued performance improvements, and will be necessary for the design of a balanced, aggressive out-of-order commit design.

4.5 Out-of-Order Commit Performance

In this section we analyze the out-of-order commit conditions to determine the minimum (and maximum) performance improvement potential. The minimum and maximum improvement is provided by Safe_OOC and Unsafe_OOC, respectively. In Fig. 7, we show the amount of improvement provided by both safe and unsafe out-of-order commit, for both aggressive and reluctant modes, for all three microarchitectures.

4.5.1 Safe_OOC

Honoring all out-of-order commit conditions results in a modest performance improvement. One implication for future processors is that Safe_OOC does not require additional support for speculative out-of-order rollback recovery mechanisms; it requires support for the commit of instructions out-of-order and the freeing of structures for use by future instructions.

Safe_AOOC When Safe_AOOC is evaluated on the SLM microarchitecture (see Fig. 7a), the range of improvement spans from a low of 3% (`leslie3d`) up to 82% (`xalan`), with an average of 44%. In the NHM microarchitecture, the improvement ranges from 9% to 108% for `milc` and `xalan`, with an average improvement of 57%, and for HSW, we see an improvement of 1% for `mcf` to 110% for `wrf`, with an average of 55% (See Section 4.4 for more details).

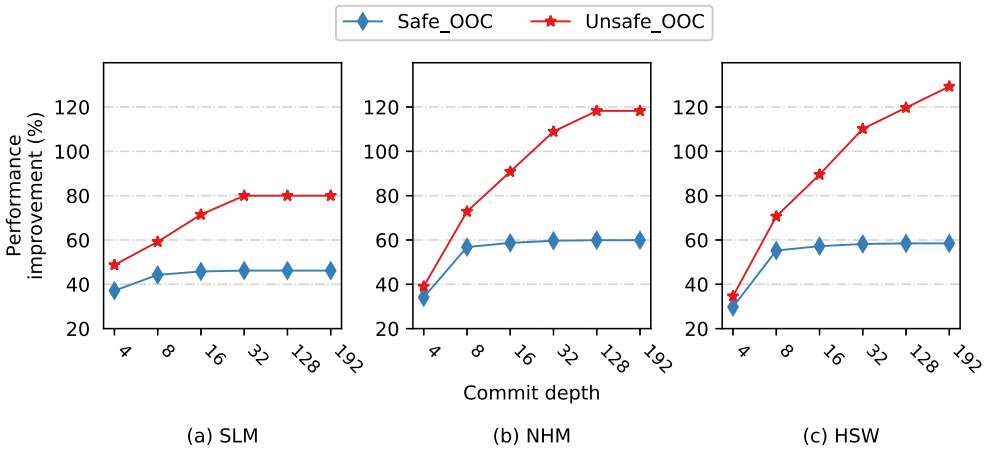


Figure 6 Effect of commit depth on performance improvement normalized to their respective in-order commit performance across SPEC CPU2006. The smaller core receives less benefit from increasing the depth of commit. Safe_OOC saturates at a commit depth of 8, 16 and 32 for SLM, NHM and HSW respectively. (the distance to the first unresolved instruction from the head of the ROB). There is no saturation in performance improvement of Unsafe_OOC while the commit depth is increased.

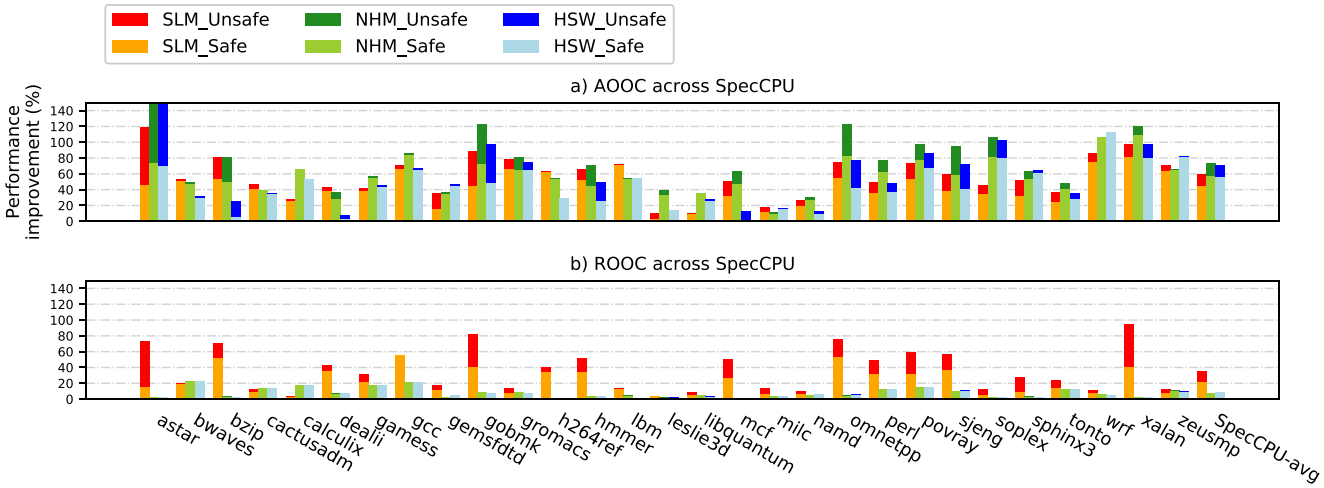


Figure 7 IPC improvement of safe and unsafe out-of-order commit relative to in-order commit as a baseline for both reluctant and aggressive versions applied on SPEC CPU2006 benchmarks on three microarchitectures.

Safe_ROOC For Safe_ROOC, the performance improvement is lower for all three microarchitectures compared to AOOC (See Section 2.5 for additional details). In the SLM microarchitecture, the range of performance improvement of safe ROOC is from 2% to 55% for `calculix` and `gcc` respectively, with an average improvement of 20%. In the NHM microarchitecture, we see performance improvements that range from 1% (`mcf`) to 22% (`bwaves`) with an average improvement of 7% (8% for HSW). Because NHM and HSM have fewer CPU stalls, ROOC has less of an effect on performance compared with the smaller, more efficient core.

4.5.2 Unsafe_OOC

By relaxing all conditions, Unsafe_OOC provides the maximum potential for performance improvement. Unsafe_OOC will require recovery mechanisms for these techniques, which can reduce the performance potential because of recovery costs.

To understand the effectiveness of all conditions together we consider zero cost for recovery for any misspeculated out-of-order commit condition.

Unsafe_AOOC This technique provides the highest performance improvement for all three different architectures. In SLM cores, the improvement ranges from 9% to 120% for `libquantum` and `astar` applications respectively and the average is 59%. In NHM architecture, the average is 72% and the range is between `milc` and `astar` respectively with 11% and 196% improvement. In HSW cores, similar to NHM cores, `astar` has the maximum benefit from Unsafe_ooc with 192% improvement, while the minimum improvement is for `dealii`, at 7%. The average improvement for this class of architecture is 70%.

Unsafe_ROOC Reluctant out-of-order commit is lower performing because it is not continuously looking to commit additional instructions (See Section 2.5 for details). In the case of SLM, the improvement ranges from 3% to 93% respectively for `leselie3d` and `xalan` with an average of a 38% improvement. In NHM, `mcf` and `bwaves` with 1% and 22% show the maximum and the minimum improvement with an average of 7% (HSW is similar with an average of 8%). Between the three microarchitectures, the limited SLM benefits the most from ROOC because of the large number of stalls seen by this core. Therefore ROOC, especially Unsafe_ROOC, is an interesting methodology to improve the performance of relatively small but energy efficient CPUs as we see a relatively high performance improvement for a less aggressive commit implementation.

Benchmark-Specific Analysis The `hmmer` benchmark is a particularly strong case for the benefits of out-of-order commit for SLM. This application is L1-cache resident, and exhibits very few last-level cache misses. Nevertheless, we still see a very strong improvement in performance, from 33% to 52% for Safe_OOC, increasing to 51% to 66% for Unsafe_OOC. Looking ahead to Figs. 8 and 11, we can see that evading the branch condition provides the most benefit for this application. Making room for additional instructions to allow the hardware to expose additional ILP works well even for those applications without a significant number of LLC misses. The `mcf` benchmark contains load-dependent branches, has the highest MPKI (misses per kilo instructions) among the benchmarks and therefore, it has a rather low IPC when it is executed on an in-order commit CPU. This results in a good opportunity to improve performance as it is extremely limited by these misses.

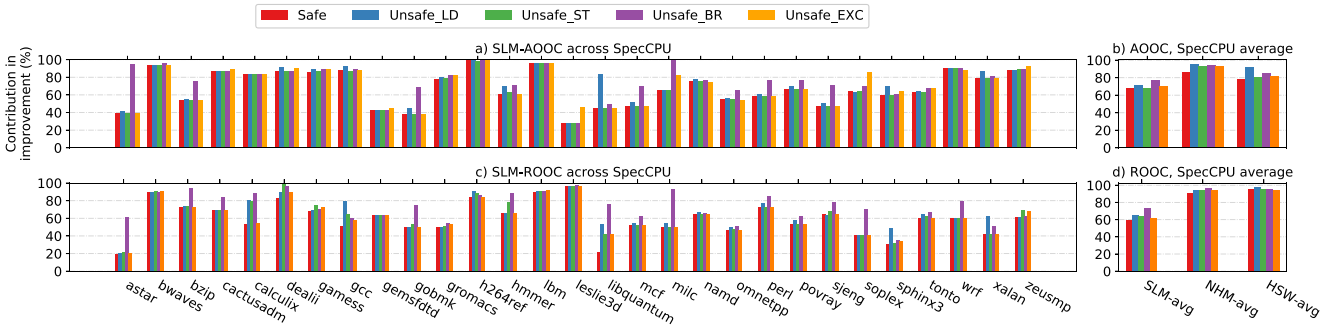


Figure 8 Contribution of safe and selectively unsafe out-of-order commit on three different microarchitectures. Unsafe_XX is equivalent to activating (enforcing) all out-of-order commit conditions except XX (the XX condition is relaxed). By relaxing the specific XX condition, the dependence between other conditions is also observed.

4.6 Performance Effects of Commit Conditions

In the previous section, by analyzing safe and unsafe out-of-order commit, we observe that there is a large gap between the performance improvement of these two implementations. Understanding the cause of this performance improvement (by looking at individual commit conditions in isolation), allows us to better understand where to focus future hardware efforts.

4.6.1 Positive Contribution of Out-of-Order Commit Conditions

To study the gap between safe and unsafe out-of-order commit (Fig. 7), we analyze the effect of relaxing each condition in the presence of the other preserved conditions in Fig. 8. We analyze the SLM microarchitecture in detail and provide averages across all microarchitectures for both AOOC and ROOC. Each out-of-order commit condition in analyzed in isolation, and we consider Unsafe_OOC (all relaxed conditions) as the 100% *potential improvement budget*. In the case of the `mcf` benchmark in Fig. 7a, the safe and unsafe OOC performance improvement is 33% and 71% respectively (46% of the potential improvement budget is provided by Safe_OOC). We also observe that by relaxing the LD condition (unsafe_LD), 52% of potential improvement budget is achievable (see Fig. 8a). In Fig. 8, we can see in some applications (like `namd` in AOOC mode and `leslie3d` in ROOC mode) that relaxing just a single condition is not sufficient to fill the gap between safe and unsafe OOC. This does not mean that a single condition is not important, but rather that other preserved conditions are preventing out-of-order commit from achieving its full potential.

AOOC We observe that for most of applications Unsafe_BR and Unsafe_LD are the most interesting conditions (Fig. 8a). Additionally, the more aggressive the core, the more important the Unsafe_LD condition becomes. In SLM, NHM and HSW CPUs, Unsafe_LD respectively fills 4%, 10% and 12%, and Unsafe_BR fills 9%, 8% and 7% of the gap between safe and unsafe OOC. Unsafe_ST is not very effective because of the rarity of this condition and the conservative memory dependence predictor used. Unsafe_EXC or relaxing exceptions are not that effective because they are very rare, especially in integer benchmarks.

ROOC Relaxing OOC conditions in ROOC has less effect in reducing the gap between safe and unsafe OOC and this is because of the nature of this on-demand OOC mode which is enabled and needed more in SLM and much less in NHM and HSW (see Fig. 7b).

Benchmark-Specific Analysis The `astar` and `gobmk` benchmarks have the highest number of mispredicted branches per 1000 instructions. Therefore relaxing the branch condition (unsafe_BR) improves the performance of these two benchmarks by 68% and 94% respectively. On the other hand, benchmarks such as `cactusadm` and `lbm` have a low branch misprediction rate (0.5%) and therefore relaxing the branch condition does not show significant improvement for these benchmarks.

The `sphinx` benchmark has high degree of intrinsic ILP,[2] thus an increase in the number of effective resources allows this benchmark to improve performance. As most of its load instructions are L2-cache misses, relaxing the load condition (Unsafe_LD) improves performance of this benchmark (Fig. 9).

4.6.2 Negative Contribution of OOC Conditions

In this section we analyze the gap between safe and unsafe out-of-order commit from a different angle.

[2]The `sphinx` benchmark continues to show performance improvements as the in-order commit processor aggressiveness increases from SLM to HSW.

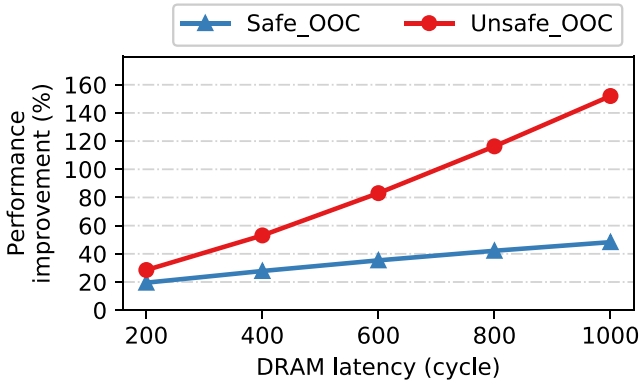


Figure 9 A comparison between safe and unsafe out-of-order commit across DRAM latencies for SLM and AOOC on SPEC CPU2006. OOC improvement increases with a higher DRAM latency. Unsafe_OOC outperforms Safe_OOC. This study uses caches that are four times smaller compared to Table 1 to put additional stress on the DRAM. This has been done for both in-order and out-of-order commit configurations.

We relax all of the out-of-order commit conditions except for one. For example safe_LD means the LD condition is preserved but ST, BR and EXC conditions are relaxed. By preserving one of the conditions we look at the negative effect (performance reduction) of the activated condition compared to Unsafe_OOC.

Figure 11 depicts the effect of each condition on performance. For most of the benchmarks, the BR and the LD conditions are the most effective ones. Among floating-point benchmarks, LD and EXC conditions have a large impact on performance. Therefore, relaxing the EXC condition, as it is rare, could lead to significant performance improvements at relatively low cost, especially if recovery mechanisms in software are used. ST has the least effect among out-of-order commit conditions when it is preserved in isolation from other conditions. This is valid between all three microarchitectures.

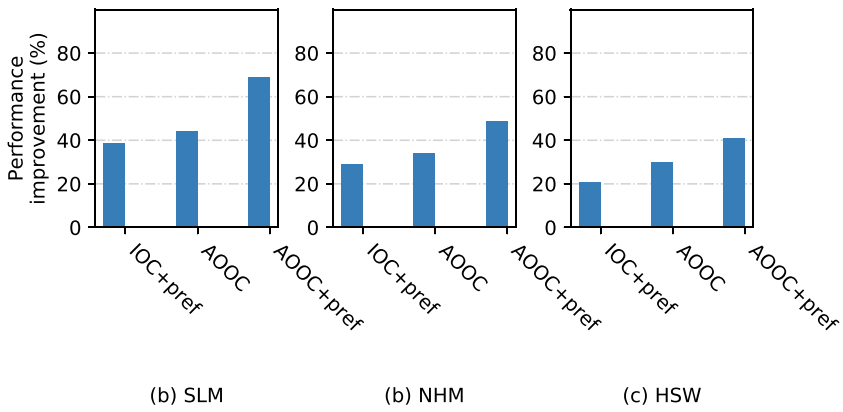
4.7 Memory Latency Evaluation

Current DRAM cells are optimized for cost, and not for access latency [20]. The potential to use more affordable, but higher latency memory could have a large impact on allocation of memory in datacenter servers as high density DRAM modules cost much more than those that are 4× smaller (1.75× per GB [4]). To evaluate the potential of out-of-order commit to handle higher DRAM latency, we evaluate memory latency from 200 cycles to 1000 cycles in 200 cycle increments. In addition, we reduce the size of all caches by 4 times when compared to Table 1, to put additional stress on the DRAM subsystem. Here we see an increasing performance improvement for the Unsafe_OOC condition, while Safe_OOC increases linearly compared to the in-order commit core. For memory-intensive applications, an efficient implementation of Unsafe_OOC could allow the use of denser, higher-latency DRAM that could potentially cost much less.

4.8 Prefetching Evaluation

Prefetching is essential for performance in modern systems and interacts tightly with the pipeline and out-of-order commit. We do not consider all prefetching options, but instead focus on the potential benefits. Therefore, to evaluate this interaction, we configure the L1-D cache in gem5 with a stride prefetcher of degree 8. Figure 10 compares the relative performance gains of prefetchers for in-order commit, out-of-order commit, and prefetchers for out-of-order commit. Across all three architectures, both out-of-order commit and prefetching on their own provide roughly 40% improvement in performance, with out-of-order commit being slightly better. However, the combination of out-of-order commit and prefetching delivers nearly 70% better performance, nearly the sum of the two independent contributions.

Figure 10 The effect of aggressive out-of-order commit on the performance with and without prefetchers in the L1 data cache across SPEC CPU2006. Improvement is relative to the baseline IOC architectures without prefetchers.



In fact, combining aggressive out-of-order commit with prefetchers allows us to reach 84% of the ideal (sum of both) improvement for SLM, 77% for NHM, and 83% for HSW, showing that these techniques work well together (Fig. 11).

4.9 Rollback Costs for Unsafe Branches

While this work does not evaluate hardware costs, we are able to evaluate the number of rollbacks caused by committing past a mispredicted branch for each of the evaluated configurations. For this evaluation, we use a commit depth of 8, with aggressive out-of-order commit (AOOC) and Unsafe_BR (only the branch condition is not respected for out-of-order commit). In Table 3, we list the number of executed branches, mispredicted branches, the number of rollbacks caused by speculatively committing a mispredicted branch, and the number of instructions that were rolled back due to this rollback. While the number of executed branches increases with the aggressiveness of the core (because these cores can more aggressively speculate past branches), the average number of rollbacks (and instructions rolled back) decreases slightly with the aggressiveness of the core. These aggressive cores resolve speculative state quickly, reducing the number of instructions that need to be committed out-of-order. This results in a similar number of rollbacks per thousand architecturally committed instructions, around 3, for all configurations.

Table 3 Out-of-order commit costs for unsafe branches.

Metric-PKI	SLM		NHM		HSW	
	Avg	Max	Avg	Max	Avg	Max
Branches	146.2	373.3	153.3	460.8	156.8	495.7
Mispred. br.	7.8	50.9	8.1	54.5	8.2	56.0
Num rollbacks	3.2	24.3	3.1	25.6	3.0	26.5
Instrs rollback	8.5	52.9	6.6	43.0	5.4	34.3

5 Memory Parallelism

Overlapping cache misses to service them in parallel, in particular long-latency accesses to DRAM, but also lower-latency accesses to the LLC, can result deliver significant performance benefits [8]. This memory parallelism is typically achieved through the use of multiple Miss Status Holding Registers (MSHRs) [19], which track outstanding memory requests, and allow them to execute in parallel. In this section, we compare in-order commit and out-of-order commit in terms of memory parallelism (both to DRAM (MLP) and within the cache hierarchy (MHP) [7]) by changing the number of L1 MSHRs and observing the effect on performance. To explore these effects, we select three applications that are highly memory-bound [18] (mcf), medium memory-bound (lbm), and largely not memory-bound (gcc) in Fig. 12. One key observation is that out-of-order commit, in both reluctant and aggressive modes, is much better than in-order commit in exposing intrinsic application memory parallelism. Figure 12 shows that the gap between in-order and out-of-order commit is much larger in the case of HSW which means that the more aggressive is the microarchitectures, the more MLP is covered if we apply out-of-order instruction commit. In case of *lbm*, comparing the SLM, NHM and HSW we observed that the rate of performance improvement in the range of $(1 <= MSHRSize <= 5)$ is higher than the rate in the range of $(5 < MSHRSize <= 9)$. In the range of $(MSHRSize > 9)$), HSW and NHM have continued improvement. This is most likely the result of an additional loop iteration containing DRAM accesses that is being covered by out-of-order instruction commit.

Figure 12 also allows us to explore the impact of memory-boundedness by comparing across the three

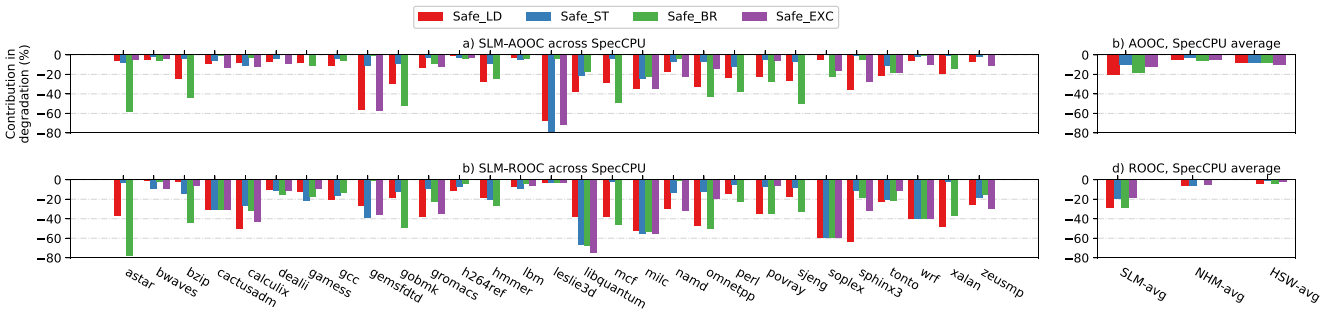


Figure 11 Maximum potential performance improvement is provided by Unsafe_OOC in which all conditions are unsafe. By preserving all conditions, maximum performance is reduced. This figure shows the negative effect of preserving (or making safe) a single condition compared to the maximum potential performance improvement. Safe_XX is equivalent to disabling all out-of-order commit conditions (all unsafe, highest performance) where XX indicates the only safe and preserved condition.

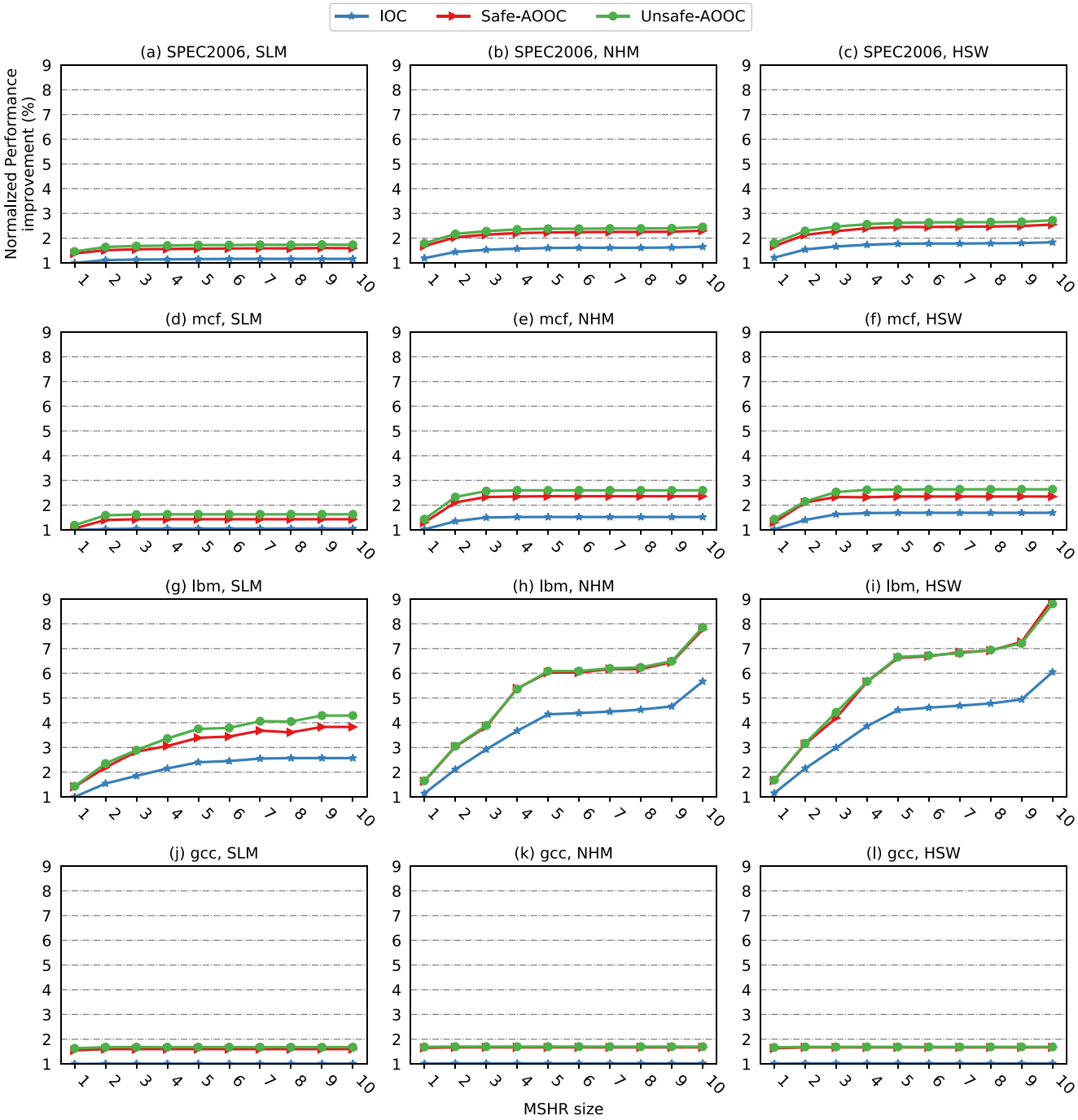


Figure 12 Memory hierarchy parallelism comparison between in-order commit and safe and unsafe out-of-order commit in MSHRs size of 1 to 10. The results have been normalized to in-order commit with MSHR size of one. Subgraph (**a**), (**b**) and (**c**) are based on harmonic mean across SPEC CPU2006. The mcf, lbm and gcc benchmarks are respectively representative of categories with high, medium and low memory boundedness.

applications (mcf is most memory bound, gcc least). Although *mcf* (Fig. 12d, e and f) is more memory-bound than *lbm*, (Fig. 12g, h and i), it exposes less memory parallelism when the number of MSHRs is increased (the gap between in-order commit and safe aggressive out-of-order commit is larger in case of *lbm* compared to *mcf*). This is because *mcf* has more isolated cache misses, that are misses that cannot be serviced at the same time to provide memory level parallelism. Also, *mcf* has branch instructions based on memory accesses (dependent loads) that miss in the cache hierarchy, thereby reducing the number of instructions that can potentially be committed

out-of-order. *lbm* has medium level of memory-boundedness [18], and therefore exposes much more memory parallelism as the number of MSHRs is increased. Its performance improvement is therefore much better when the CPU commits instructions out-of-order. *gcc* benchmark, Fig. 12j, k and l, is one of the least memory-bound benchmarks. As a result, increasing the number of MSHRs does not improve its performance, even in the case of unsafe out-of-order commit. Overall, out-of-order commit outperforms in-order commit by exposing additional memory parallelism (both to DRAM and in the hierarchy). In summary, in case of MLP, out-of-order commit provides more benefit for bigger architecture and more memory-bound applications.

6 Early Release of Physical Registers vs. Out-of-Order Commit

Register renaming enables the avoidance of false dependencies through the register file, thereby improving core performance. It does this by translating architectural registers to larger set of physical registers. As this renaming is done early in the pipeline, and the physical registers are not released until commit, which may happen long after the last consumer reads the data, the physical register entries may be kept alive for longer than what the dependencies alone require. To address this resource constraint, techniques such as delayed allocation and Rarly Release of the Physical Register (ERPR) [23] have been developed. In this section, we compare ERPR with the OOC taxonomy [2].

OOC and ERPR are similar as they release physical register as early as possible. Aggressive OOC outperforms ERPR in general because it releases ROB and LSQ entries early, as well as physical registers. Unfortunately, neither ERPR nor OOC can release IQ entries earlier than usual because OOC only address processor state after the instructions have left the IQ and ERPR is effective before instructions are inserted tot the IQ. From another point of view OOC put additional pressure on the IQ and provides free entries in one or all of the resource of superscalar processors.

6.1 Analysis of CPU Aggressiveness

CPU aggressiveness analysis in this context is based on microarchitecture configurations (see Table 2) and commit-depth. The overall trend of Fig. 13 shows that AOOC outperforms ROOC and ERPR in all configurations. This is not surprising, as AOOC is always enabled, while ROOC is enabled only when a resources is exhausted (see Section 2.5).

ERPR is essentially a subset of AOOC that only focuses on the register file (RF). Therefore, the higher is the program sensitivity to the RF capacity, the higher is the effect of ERPR. To better understand this behavior, we analyzed the reasons for CPU stalls across different microarchitectures (Table 4). The data show that increasing the effective size of RF (e.g., what ERPR does conceptually) is less effective in SLM compare to NHM and HSW barbecue is SLM, the CPU stalls are more often due to other resources, such as the ROB size. As a result since in SLM architecture the size of RF is less effective in CPU stalls, ROOC outperforms ERPR in terms of performance improvement (ROOC can virtually extend the effective size of ROB, LSQ as well as RF although ERPR only does this on RF). RF in NHM and HSW architectures is more effective on CPU stall than in SLM architecture therefore, ERPR outperforms ROOC regarding performance improvement in these two architecture.

By focusing only on ERPR, we observe that the wider the core the more effective is the is increasing commit-depth (wider commit-depth covers more ready instructions to commit). For example, in the case of SLM, although widening the commit-depth releases additional physical registers earlier than general, these freed registers are not effective anymore because the CPU stalls are due to limits in other resources (like ROB, IQ or LSQ). As NHM and HSW have more resources, increasing the commit-depth is more effective in those architectures. In the case of AOOC, increasing the commit-depth is more effective than ROOC and ERPR because it enables additional instructions to be committed out-of-order. (because it is active on every cycle compare to ROOC and also affects all of the resources compare to ERPR that only affects the RF) In addition, we can see that the more aggressive the CPU, the more effective is increasing the commit-depth. This is because wider cores execute and complete instructions that are far from the head of ROB earlier with their additional resources(either provided by the baseline or released by the OOOC), and they can therefore provide more instructions to commit if the commit-depth is increased.

6.2 Analysis of Out-of-Order Conditions

Figure 13 shows the potential for performance improvement from relaxing the out-of-order commit conditions across the three architectures for both safe and unsafe out-of-order-commit. As has been seen previously [2, 5, 21], the least aggressive out-of-order commit configuration (safe_OOC with a commit depth of 4) is most beneficial for the least aggressive out-of-order processor, SLM (compare the left-most bars of Fig. 13a–c). This is expected, as the SLM processor has the least execution hardware and therefore suffers from more stalls than the more aggressive processors.

Increasing the commit depth from 4 to 8 (second group of bars in Fig. 13a–c) shows that the more aggressive

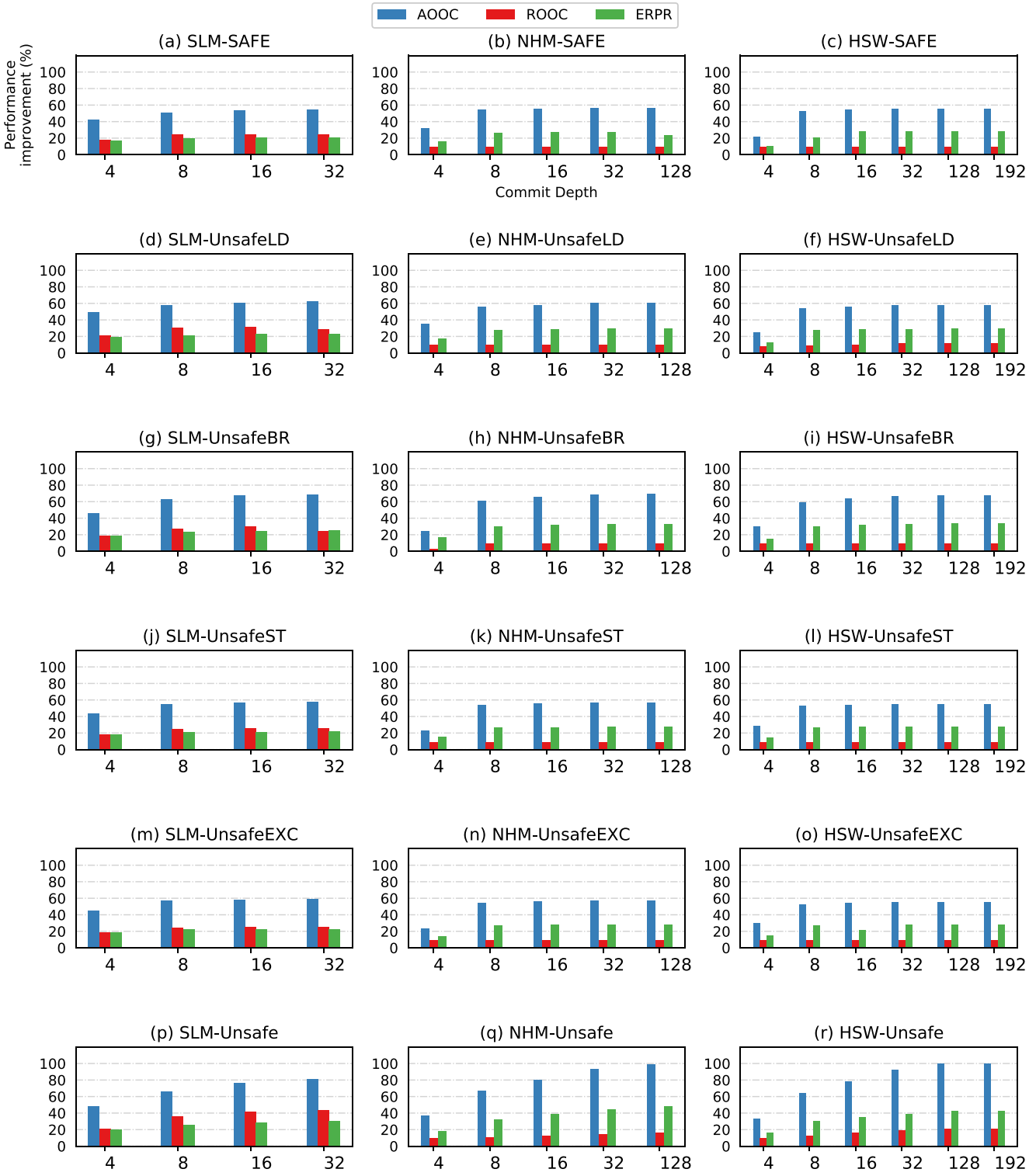


Figure 13 Comparison between AOOC, ROOC and ERPR in different commit-depth and microarchitecture setup in six different modes. UnsafeXX is equivalent to (enforcing) all out-of-order commit conditions except condition XX . By relaxing the specific XX condition, the dependence between other conditions is also observed.

out-of-order processors (NHM and HSW) now benefit more from out-of-order commit than the simpler SLM architecture. This new result shows the interaction between the aggressiveness of the out-of-order execution and out-of-order commit: for more aggressive out-of-order execution, there is more benefit from more aggressive out-of-order

Table 4 The Contribution of exhausted (full) register file (RF) and reorder buffer (ROB) on CPU stalls.

Microarchitecture	RF%			ROB%			Other%		
	min	avg	max	min	avg	max	min	avg	max
SLM	6	59	86	13	36	92	0	5	27
NHM	2	68	91	8	30	98	0	20	7
HSW	2	69	91	8	29	98	0	20	7

Other reasons to the CPU stalls could be the instruction queue (IQ), the load-store queue (LSQ), instructions cache misses or not having a free functional unit to allocate to the ready-to-dispatch instructions

commit. The reason is that aggressive architectures appear to have more unused resources available for computation. This trend continues through the more aggressive out-of-order commit modes as well, with *unsafe* optimizations (Fig. 13d–r) benefiting the more aggressive out-of-order execution designs (NHM and HSW) more than the simpler one (SLM). The reasons for this are similar to the safe case, as the more aggressive processors are faster to achieve the first out-of-order condition, instruction completion, and therefore providing greater commit depths (e.g changing the commit depth from 4 to 8 in Fig. 13a–c) results in a larger likelihood of finding instructions to commit out-of-order. Table 5 uses this analysis to rank the relative importance of each out-of-order commit condition for the three architectures. This information provides a guideline for which areas to work on to achieve the best performance improvement depending on the baseline out-of-order execution.

Table 5 Ranking of the benefits of out-of-order commit conditions in different microarchitecture based on the depth of commit.

Condition/commit-depth	4	8	16	32	128	192
(a) SLM						
Unsafe_LD	1	2	2	2	NA	NA
Unsafe_BR	2	1	1	1	NA	NA
Unsafe_ST	4	4	4	4	NA	NA
Unsafe_EXC	3	3	3	3	NA	NA
(b) NHM						
Unsafe_LD	1	2	2	2	2	NA
Unsafe_BR	3	1	1	1	1	NA
Unsafe_ST	4	4	4	4	4	NA
Unsafe_EXC	2	3	3	3	3	NA
(c) HSW						
Unsafe_LD	4	2	2	2	2	2
Unsafe_BR	1	1	1	1	1	1
Unsafe_ST	3	3	4	4	4	4
Unsafe_EXC	2	4	3	3	3	3

7 Related Work

The goal of this work is to provide a detailed understanding into the potential for performance benefits across different out-of-order commit conditions and levels of aggressiveness. While this work aims to describe the maximum potential benefit for each individual condition, there have been many previous works that describe hardware solutions for early release of hardware structures and out-of-order commit strategies.Below, we provide an overview of these works, and how they fit into the categories described in this work.

7.1 Speculative Release of Hardware Structures

A number of implementations require register and processor state checkpointing support to speculatively retire or release hardware structures [1, 11, 12, 17, 22].

Processor state checkpointing, especially with the advent of very large SIMD registers such as AVX-512 registers which now support up to 32 registers with up to 512 bits per register, can require a significant amount of state to be saved when speculation is aggressive.

7.2 Non-speculative Structure Release

Non-speculative early release of hardware structures before commit requires knowledge that no older instruction (that has come earlier in the instruction stream) can cause the program to abort, raise an exception, or require exposure of the architected state at that time. Non-speculative solutions have the potential to be the most energy efficient, a necessity in an era of the end of Dennard Scaling for power-limited platforms. A range of solutions, from hardware-only to software-assisted solutions are described below.

Compiler support. Two previous works [1, 13] allow for early commit or reclaim of resources based on compiler knowledge, but require software recompilation.

Checkpoint-free. A number of solutions do not use checkpoints [5, 13, 21] and therefore do not require speculation (they respect the all commit conditions) or used software help to extend knowledge to the hardware.

Selective early commit. Early commit of loads [14, 15] allows for loads that have not yet received data from

the memory hierarchy to become part of the committed state of a processor. This allows the processor to continue to process instructions past normally blocked structures, improving performance for memory-bound workloads. To accomplish this, the authors [15] decouple page faults that can occur from the fetching of data. While recompilation is not strictly required for this technique, the authors evaluate their work using a compilation strategy to expose loads early. Late allocation and early release of physical registers is proposed as a register renaming scheme in [23]. They explored the effect of late allocation and early release of physical register on the performance of an 8-way superscalar processor without the need for additional speculation and maintains precise exceptions. The out-of-order commit methods evaluated in this work overlap with the early release of registers while also evaluating the early release of other structures (like the LSQ).

Evading out-of-order commit conditions. [24] provides a coherency protocol level solution such that the load→load reordering can be evaded non speculatively under TSO. In their design, it is no longer necessary to squash and re-execute speculatively reordered loads if their reordering can be hidden by the coherency protocol. This allows speculatively reordered loads, that otherwise adhere to the rest of the Bell-Lipasti conditions, to be committed out-of-order without affecting the memory model.

8 Conclusion

To obtain higher performance, extending the reach of the processor core has been a focus in much of microarchitecture research. One promising direction is the use of out-of-order commit, which releases precious processor resources early to allow the processor to extend its reach past typical hardware limits. In this work, we present a limit study for out-of-order commit through the introduction of reluctant and aggressive out-of-order commit modes. We show how smaller processors, even with a limited commit scan depth, can benefit from out-of-order commit strategies, but that larger, aggressive cores require deeper commit scan depths to achieve improved performance. In addition, we provide a detailed breakdown of the contributions for each out-of-order commit condition for the SPEC CPU2006 benchmark suite, and compare against similar works such as early release of physical registers. Our results show a very high potential for performance improvement, above 2.25x for some benchmarks, and believe that out-of-order commit strategies can play an important role for future energy-efficient and high-performance processor designs.

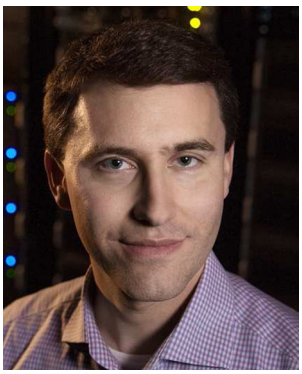
References

1. Afram, F., Zeng, H., Ghose, K. (2013). A group-commit mechanism for ROB-based processors implementing the x86 ISA. In *HPCA* (pp. 47–58).
2. Alipour, M., Carlson, T.E., Kaxiras, S. (2017). Exploring the performance limits of out-of-order commit. In *CF* (pp. 211–220).
3. Allan, A., Edenfeld, D., Joyner Jr. W.H., Kahng, A.B., Rodgers, M., Zorian, Y. (2002). 2001 technology roadmap for semiconductors. *Computer*, *35*(1), 42–53.
4. Badalone, R. (2015). Dram's surprising role in the cost of data centers. http://www.datacenterknowledge.com/archives/2015/11/12/dont/.
5. Bell, G.B., & Lipasti, M.H. (2004). Deconstructing commit. In *ISPASS* (pp. 68–77).
6. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A. (2011). The gem5 simulator. *SIGARCH Computer Architecture News*, *39*(2), 1–7.
7. Carlson, T.E., Heirman, W., Allam, O., Kaxiras, S., Eeckhout, L. (2015). The load slice core microarchitecture. In *ISCA* (pp. 272–284).
8. Chou, Y., Fahs, B., Abraham, S. (2004). Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA* (pp. 76–88).
9. Corporation, I. (2016). Intel® 64 and ia-32 architectures optimization reference manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.
10. Corporation, I. (2016). Intel® intel's 'tick-tock' seemingly dead, becomes 'process-architecture-optimization'.http://www.anandtech.com/show/10183/intels-tick-tock-seemingly-dead-becomes-process-architecture-optimization.
11. Cristal, A., Ortega, D., Llosa, J., Valero, M. (2004). Outof- order commit processors. In *HPCA* (pp. 48–59).
12. Cristal, A., Santana, O.J., Valero, M., Martínez, J.F. (2004). Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization*, *1*(4), 389–417.
13. Duong, N., & Veidenbaum, A.V. (2013). Compiler-assisted, selective out-of-order commit. *IEEE Computer Architecture Letters*, *12*(1), 21–24.
14. Gwennap, L. (1994). Digital leads the pack with 21164. *Microprocessor Report*, *8*(12), 249–260.
15. Ham, T., Aragón, J.L., Martonosi, M. (2015). Desc: decoupled supply-compute communication management for heterogeneous architectures. In *MICRO* (pp. 191–203).
16. Henning, J.L. (2006). SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, *34*(4), 1–17.
17. Hilton, A., & Roth, A. (2010). BOLT: energy-efficient out-of-order latency-tolerant execution. In *HPCA* (pp. 1–12).
18. Jaleel, A. (2010). Memory characterization of workloads using instrumentation driven simulation. http://www.glue.umd.edu/ajaleel/workload.

19. Kroft, D. (1981). Lockup-free instruction fetch/prefetch cache organization. In *ISCA* (pp. 81–87).
20. Lee, D., Kim, Y., Seshadri, V., Liu, J., Subramanian, L., Mutlu, O. (2013). Tiered-latency dram: a low latency and low cost dram architecture. In *HPCA* (pp. 615–626).
21. Marti, S., Borras, J., Rodriguez, P., Tena, R., Marin, J. (2009). A complexity-effective out-of-order retirement microarchitecture. *IEEE Transactions on Computers*, *58*(12), 1626–1639.
22. Martinez, J.F., Renau, J., Huang, M.C., Prvulovic, M. (2002). Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *MICRO* (pp. 3–14).
23. Monreal, T., Vinals, V., Gonzalez, J., Gonzalez, A., Valero, M. (2004). Late allocation and early release of physical registers. *IEEE Transactions on Computers*, *53*(10), 1244–1259.
24. Ros, A., Carlson, T.E., Alipour, M., Kaxiras, S. (2017). Non-speculative load-load reordering in TSO. In *ISCA* (pp. 187–200).
25. Smith, J.E., & Pleszkun, A.R. (1988). Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, *37*(5), 562–573.
26. Sohi, G.S., & Vajapeyam, S. (1987). Instruction issue logic for high-performance, interruptable pipelined processors. In *ISCA* (pp. 27–34).

Mehdi Alipour received his bachelor's and master's degree in the field of computer hardware and architecture in 2008 and 2011 respectively. Since 2014 he is pursuing his Ph.D. in the field of computer architecture at Uppsala Architecture Research Team, Sweden. His research interest is focused on microarchitecture specifically resource allocation and deallocation in superscalar processors.

Trevor E. Carlson is an Assistant Professor at the National University of Singapore. He received his B.S. and M.S. degrees from Carnegie Mellon University in 2002 and 2003, and his Ph.D. from Ghent University in 2014. While a staff engineer at IBM he helped to author 4 issued patents between 2003 and 2007. He has over 13 years of computer architecture experience covering both industry and academia, with publications at leading publications in computer architecture. His work on simulation, sampling, modeling and performance analysis has seen three Best Paper Awards and three nominations for Best Paper. He co-developed the Sniper Multi-core Simulator which is being used by hundreds of researchers to evaluate the performance and power-efficiency of next generation systems. His research interests include highly-efficient microarchitectures, hardware/software co-design, performance modeling and fast and scalable simulation methodologies.



Dr. David Black-Schaffer is a Professor in the Department of Information Technology at Uppsala University. His research focuses on approaches for moving data more efficiently in heterogeneous computer systems, using both software and hardware techniques. His results have led to multiple patents and a startup. Prior to joining the faculty of Uppsala University, he contributed to the design and development of the OpenCL standard for heterogeneous computation at Apple. He received his PhD in Electrical Engineering from Stanford University in 2008 for work on power-efficient embedded processing systems. Dr. Black-Schaffer is also an award-winning teacher, and his educational tools are used by tens of thousands of students.

Stefanos Kaxiras is a full professor at Uppsala University, Sweden. He holds a PhD degree in Computer Science from the University of Wisconsin. He was a member of the Computing Sciences Center at Bell Labs and of the faculty of the ECE Department of the University of Patras. Kaxiras' research interests are in the areas of memory systems, multiprocessor/multicore systems, with a focus on power efficiency. He is a Distinguished ACM Scientist and IEEE Senior Member.