ATR: Out-of-Order Register Release Exploiting Atomic Regions

Yinyuan Zhao
University of California, Santa Cruz
Santa Cruz, CA, USA
yzhao321@ucsc.edu

Surim Oh
University of California, Santa Cruz
Santa Cruz, CA, USA
soh31@ucsc.edu

Mingsheng Xu
University of California, Santa Cruz
Santa Cruz, CA, USA
mxu61@ucsc.edu

Heiner Litz
University of California, Santa Cruz
Santa Cruz, CA, USA
hlitz@ucsc.edu

Abstract

Modern superscalar processors require large physical register files to support a high number of in-flight instructions, which is crucial for achieving higher ILP and IPC. Conventional register renaming techniques release physical registers conservatively, waiting until the instruction that redefines the same architectural register commits, which results in inefficient register utilization. In particular, registers frequently remain allocated for many cycles although they are no longer in-use. To address this deficiency, previous approaches have explored early register release, which aims to free registers as soon as they have been fully consumed. However, these techniques are either unsafe or conservative, limiting the benefits.

We observe that over 17% of all allocated registers in SPEC2017int and 13% in SPEC2017fp are located within *atomic commit regions*, sequences that do not include conditional branches nor exception-causing instructions. Instructions within such regions are guaranteed to atomically commit or flush together, allowing safe early release of the registers allocated by the first instruction of a region. In particular, registers can be released without waiting for the redefining instruction of the architectural register to commit. We propose a novel renaming technique that leverages this insight to reduce register file pressure. Our technique enables out-of-order register release by identifying atomic commit regions using a simple mechanism that requires no stacks, queues, extra memory, or shadow cells. We show that, for SPEC2017int benchmarks, the proposed register renaming scheme achieves an average speedup of 5.13% for a 64-entry, and 1.48% for a 224-entry register file.

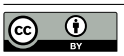
CCS Concepts

• **Computer systems organization → Superscalar architectures**.

Keywords

Out-of-Order Microarchitecture, Register Renaming

ACM Reference Format:
Yinyuan Zhao, Surim Oh, Mingsheng Xu, and Heiner Litz. 2025. ATR: Out-of-Order Register Release Exploiting Atomic Regions. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3725843.3756135



ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756135

1 Introduction

Modern processors seek to maximize instruction-level parallelism (ILP) by increasing the number of in-flight instructions. This is achieved by increasing the superscalar width and the depth of the processor, including the expansion of the reorder buffer (ROB), load-store queue (LSQ), and physical register file size. For instance, Intel's Golden Cove architecture supports a 6-wide pipeline with a 512-entry ROB and a physical register file of 280 entries. Scaling these structures improves performance, however, introduces significant challenges in terms of clock frequency and power. The physical register file storing renamed architectural operand values is most challenging to scale due to its large number or read and write ports. For instance, consider a modern architecture such as Apple's M4, which can fetch and rename up to 10 instructions per cycle. Assuming two source and one destination register per instruction, the register file needs to support 20 read and 10 write ports. A register file design based on Flip-Flops and multiplexers requires millions of transistors[1], consuming considerable area and power. SRAM designs can be more area-efficient; however, multi-ported designs support limited clock frequencies while multi-banked designs [32] suffer from bank conflicts. For this reason, processors like XiangShan [33] rely on port replication, effectively replicating the entire register file for each port, which is difficult to scale to wide machines due to space inefficiency.

Scaling the register file size in a wide superscalar design supporting high clock-frequencies is challenging but necessary for high performance. Figure 1 illustrates the impact of varying physical register file sizes across a range of integer Spec2017 [29] benchmarks on an Intel Golden Cove-like core. All IPC values are normalized, with 1.0 representing the performance under an ideal configuration with an infinite number of registers. As one can see, with 64 physical registers, the average IPC reaches only 37.7% of the ideal case. To maintain performance within 5% of the infinite-register baseline, a minimum of 280 physical registers is required.

Prior work has explored several techniques to scale the register file size in a more area and power-efficient way. Late allocation [7] only reserves a virtual register tag (vtag) at rename and delays

[1] A single 512:1 MUX using pass-gate logic consumes 1024 transistors. $1024 \times 64bits \times 20$ read ports = 1.3M transistors. Bit-storage introduces an additional $512 \times 64bit \times 20 = 650K$ transistors.

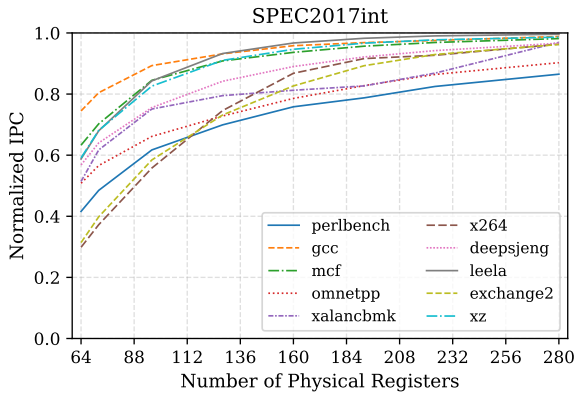


Figure 1: IPC improves with increasing register file size

allocation of the physical register to when the destination value is actually computed in the execution stage. Early register release [22] proposes to free physical registers as soon as the last consumer has read them. Therefore, the technique maintains a counter assigned to each physical register which is incremented when a consumer is renamed and decremented when a consumer executes. If the counter equals zero and the architectural register has been renamed by the next producer of the same architectural register, it is freed. Late allocation and early release effectively reduce the pressure on the register file, however, they introduce complex implementations that suffer from deadlock, misspeculation, and imprecise exception handling issues.

To address these challenges, we introduce ATomic register Release (ATR), a mechanism that releases registers out-of-order in a safe and non-speculative way. Our technique introduces *atomic commit regions*, defined as sequences of instructions that do not contain any conditional branches nor exception-causing instructions. All instructions within an atomic commit region are guaranteed to either flush or commit as a group which ATR exploits to safely release registers early. As a result, our mechanism does not jeopardize correctness by relaxing precise exception handling nor does it suffer from deadlock and misspeculation. Furthermore, ATR is orthogonal to previously proposed late allocation and early release mechanisms and hence can be combined for additional benefit. In summary, the main contributions of this paper are the following:

- We present a detailed analysis of benchmarks revealing that, on average, 17.04% of all allocated registers in SPECint and 13.14% in SPECfp are located within atomic commit regions.
- We propose a novel register renaming technique enabling out-of-order register release based on *atomic commit regions*.
- We demonstrate that our approach is safe providing precise exceptions while eliminating the need to handle mispredictions.
- We show that, for SPEC benchmarks, the proposed register renaming scheme achieves an average speedup of 5.13% for a given register file size, or an average reduction of over 27.1% in register file size while achieving less than 3% slow down over baseline.
- We demonstrate that our proposed technique can be integrated with other register renaming approaches to achieve improved performance.

2 Background and Related Work

This section introduces the basic concepts of register renaming and covers prior works on early-release that aim to improve register file utilization by reclaiming registers before the committing of redefining instructions.

2.1 Register Renaming

Register renaming is a fundamental technique to improve the performance of out-of-order processors. Its primary objective is to eliminate write-after-read (WAR) and write-after-write (WAW) hazards, by dynamically mapping architectural registers to physical registers. Therfore, processors maintain a register alias table (RAT) which maps each architectural register to its current physical register. When instructions are renamed (after decode, before issue) they first look up their source operands in the RAT, obtaining their corresponding physical source register numbers (ptag). Then each architectural destination register is renamed by allocating a new ptag and updating the RAT accordingly. Physical registers are allocated from a free list and returned (freed) if the next instruction with the same architectural destination register commits. Assume an older instruction A and a younger instruction B, where B is the next instruction that writes to the same destination register as A. We can only free the physical register of A when B commits because only then can it be guaranteed that no consumer of A's destination register is renamed. In particular, it is incorrect to free A's register when B is renamed because a mispredicted branch between A and B may cause B to flush and a new consumer of A's register may appear. If, at any time, the free list is empty, a processor needs to stall as no further instructions can be renamed.

2.2 Speculative Early Release

Prior work [6, 22] has proposed to speculatively early release registers as soon as they have been fully consumed. These mechanisms track the consumers of all physical registers by incrementing a counter when a new consumer is renamed and by decrementing the counter when a consumer reads its sources (at execute). A physical destination register is early-released when its consumer count is equal to zero, and it has been *redefined*, that is, another instruction with the same destination is renamed. However, these early-release techniques are speculative in that they may free-register values that are still required in the future. Figure 2 illustrates an example where a speculatively deallocated register value is later consumed. In this case, *I5* redefines *r1*, which is allocated in *I1*, causing register *p1* to be freed after *I2* consumes. However, since *I5* is off-path, misprediction recovery leads to *I6* attempting to consume *p1*, which has already been released. To address these problems, prior work checkpoints register values in a *shadow register file* on early-release. While the shadow register file may be less timing-critical and may require fewer read and write ports, the techniques are complex and effectively increase the number of required physical registers.

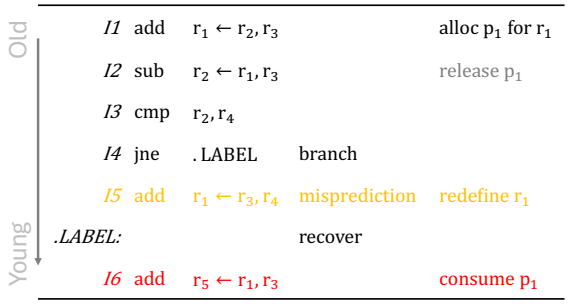


Figure 2: An example of a register value that is speculatively released and later reused.

2.3 Non-Speculative Early Release

To avoid checkpointing registers in a shadow register file, a non-speculative early release [16] mechanism can be designed to delay early-release until the redefining instruction becomes non-speculative. An instruction becomes non-speculative if all older instructions that may change the control flow are resolved. In particular, this means that (1) all older conditional branches and indirect jumps have to be resolved, (2) all older loads and stores are guaranteed not to cause an exception (e.g. page fault), and (3) all other instructions that may cause an exception, e.g. division by zero, need to be completed. We refer to an instruction that is no longer speculative as *precommitted* as it is guaranteed to commit eventually. Note that some prior works [19, 22] precommit instructions if condition (1) (all prior branches resolved) has been met without waiting for (2) and (3). These techniques, however, are unsafe in that they cannot support precise exceptions, which all modern ISAs require. For instance, they may release registers out-of-order before an older load causes a page-fault exception. Non-speculative early release enforcing (1)-(3) is safe, however, it is conservative, as instructions are required to commit in order, and often, an instruction precommits only a few cycles before committing. In this work, we introduce atomic commit regions, allowing to safely release registers that are not precommitted yet. Our key insight is that if we can guarantee that a producer instruction and all its consumer instructions are going to atomically flush or commit, we can early-release the producer's registers early without waiting for precommit.

3 Analysis

Section 2 introduced the techniques of speculative and non-speculative early release of physical registers. This section analyses the existing performance opportunities of prior works and motivates atomic commit regions for early register release.

3.1 Life-of-a-Register

While speculative early release frees registers aggressively, it is unsafe without a check-pointing shadow register file, defeating the purpose of actually reducing register file size. On the other hand, non-speculative early release is safe but conservative, as it releases registers in precommit-order. This section performs a register lifetime analysis of the two approaches to determine unmet opportunities that can be addressed with ATR. To perform our lifetime analysis, we introduce the following definitions. The state of a register is affected by events triggered by instructions. Note that different instructions affect the state of a register. For instance, as discussed in section 2 a register is allocated by one instruction I1 but it is released by the commit of the redefining instruction I2. The instruction number hereby defines an age order, i.e. I1 is older than I2 which is older than I3.

(1) **`I1` Renamed:** The time where an instruction I1 renames its architectural destination register A to a physical register P1.
(2) **`I2` Consumed:** The time where the last consumer instruction I2 of architectural register A respectively P1 executes.
(3) **`I3` Redefined:** The time where the next instruction I3 renames architectural destination register A mapping it to a new physical register P2.
(4) **`I3` Precommitted:** The time where instruction I3, which redefined A, precommits. Precommit requires that all older branches and exception causing instructions (div) have been completed and that all older loads/stores are guaranteed to not cause an exception or flush the pipeline.
(5) **`I3` Committed:** The time where instruction I3, which redefined A, commits, allowing to free P1.

As illustrated in Figure 3, there exists a partial order between these events: `I1` Renamed → { `I2` Consumed ↔ `I3` Redefined } → `I3` Precommitted → `I3` Committed. In particular, register A may be redefined first and then consumed or consumed and then redefined.

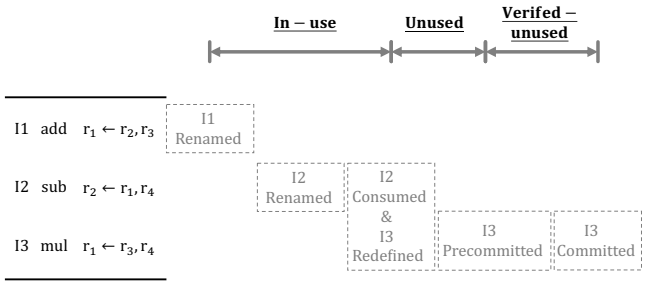


Figure 3: Partial events across the lifecycle of a register.

The events above determine the lifecycle of each physical register. During their lifecycle, registers advance through three states:

(1) **`In-use:`** The period in which the physical register P1 is actually in use. It starts with register allocation and ends when there are no pending consumers of P1 and the architectural register mapping of P1 has been redefined.
(2) **`Unused:`** The period after which physical register P1 is no longer required by any consumer, it has been redefined, but the redefining instruction has not yet precommitted. Determining the `unused`-state of a register requires Oracle information as a branch misprediction may cause flushing the redefining instruction, returning the register P1 back into the `In-use` state.
(3) **`Verified-unused:`** The time where physical register P1 is consumed, redefined, and the redefining instruction has precommitted. At this point, all older branches have been resolved, and register P1 is guaranteed to remain `verified-unused`.

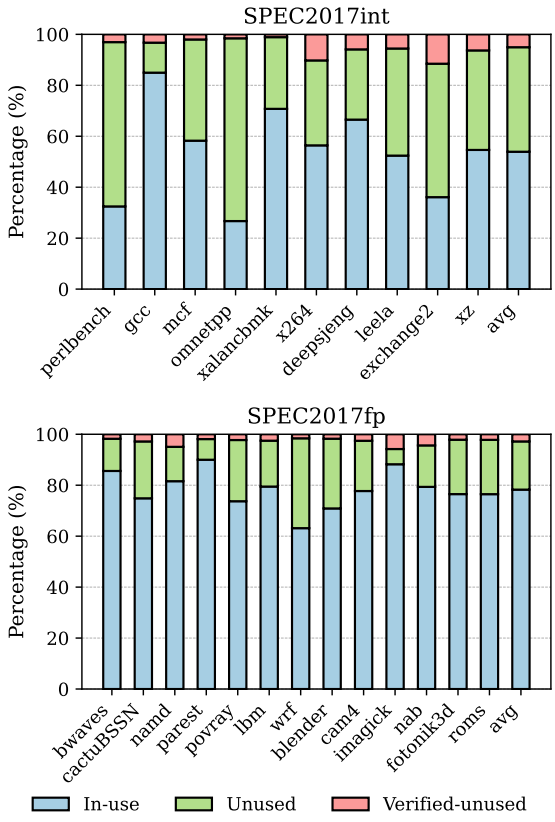


Figure 4: Cyclecount distribution across the register lifecycle.

To determine early-release opportunities, we now analyze the register states for applications of the Spec2017 benchmark suite using microarchitectural simulation. We examine an Intel Goldencove-like processor on the Scarab [24] microarchitectural simulator using the same configuration as in subsection 5.1. As shown in Figure 4, for SPEC2017int, registers are `in-use` for 53.52% of the time. 41.03% of their time, they remain allocated but unused. Speculative Early-Release [10, 31] releases such `unused` registers early, however, as we explored, this approach is unsafe due to branch mispredictions. Non-speculative Early-Release frees registers that are `verified-unused`. As can be seen, this allows to free registers only 5.05% earlier than the baseline which frees registers at commit of the redefining instruction. As can be seen there is a significant gap between Non-speculative Early-Release (5.05%) and Speculative Early-Release (41.03%). ATR leverages this performance opportunity to safely release additional registers early.

For the vector register file, we observe that registers are `in-use` for 78.27% of the time. For 18.91% of the time, registers remain `unused`, meaning they are neither consumed nor redefined during that period. Only 2.813% of the time are registers marked as `verified-unused`, representing cases where the register is confirmed to be safe for early release. This also highlights the performance opportunity of safely releasing extra registers early in order to approach the upper bound of Speculative Early-Release (18.91%) for SPEC2017fp.

3.2 Atomic Commit Regions

As shown by the analysis above, there exist significant opportunities beyond non-speculative early-release as shown by the speculative release upper bound. Our key insight is that we can early-release some of the still speculative registers without relying on precommit order. Consider the following example shown in the Figure 5, which presents a segment of instructions from SPEC2017int's Omnetpp, along with the cycle counts from a real-time simulation for the stages in which each instruction is renamed (Re), executed (Ex), completed (Cm), and precommitted (Pr). `I1` is a load instruction and `I2` a macro-fused conditional branch depending on the load. Both `I1` and `I2` prevent further instructions in the ROB to precommit as they can potentially cause an exception. The requirement of precommitting instructions in-order and the fact that `I1` has a high latency, delays the freeing of registers used by subsequent instructions such as RBX allocated by `I4`. In particular, at cycle 738, when `I5` renames RBX, the physical register allocated by `I4` can be safely freed although `I2` has not been resolved yet. In this case freeing the physical register of RBX is safe, even if I2 mispredicts and flushes all older instructions. In other scenarios, it is necessary to wait for precommit. For instance, I3 is not allowed to free the physical registers for RAX allocated by I1 because a mispredicted branch of I2 flushes I3. In summary, only a subset of registers can be freed early, in particular, those that are in an atomic commit region (there exists no branch nor exception-causing instruction between the renaming and redefining instruction).

			Re	Ex	Cm	Pr
I1	MOVE	RAX ← RAX	510	675	839	675
I2	TEST + JNZ	ZPS ← RAX	510	841	841	841
	...	...				
I3	LEA	RAX ← RDI	709	716	716	841
	...	...				
I4	LEA	RBX ← RAX	729	737	737	842
I5	SHR	RBX ← RBX, ZPS	729	738	738	842

Figure 5: An example segment of instructions from SPEC2017int omnetpp.

To analyze the opportunities of our proposed technique, we analyze the existence of atomic commit regions in Spec17. Figure 6 depicts the ratio of physical registers renamed as part of an atomic region and the total number of allocated physical registers. Our analysis explores three types of regions: *Non-branch* regions are defined as a sequence of instructions that starts with an instruction that renames architectural destination register A and ends with an instruction that redefines register A. The sequence can include an arbitrary number of instructions which may or may not consume A but no conditional branches and no indirect jumps. *Non-except* regions are defined as a sequence of instructions that do not include any memory instructions (loads, stores, and x86 ALU instructions

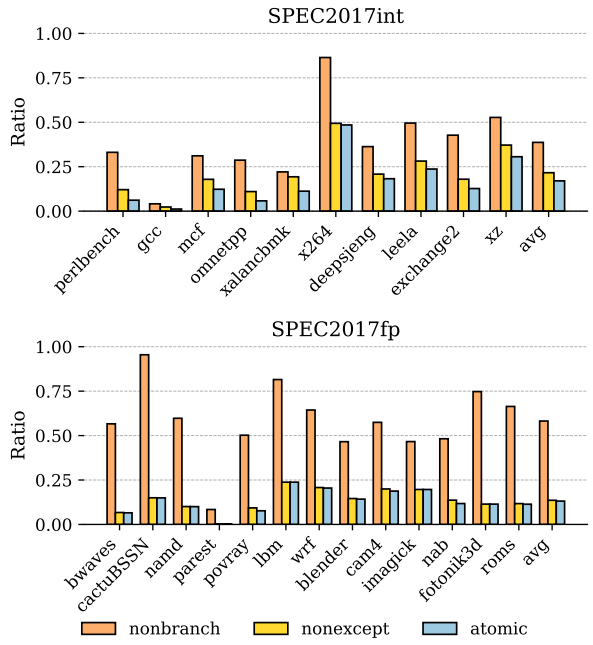


Figure 6: Atomic register ratio.

with memory operands) and exception-causing instructions (division). *Atomic* regions do not include any branches, indirect jumps, or exception-causing instructions. Note that regions are analyzed at rename in-order, however, the atomicity of a region is independent of the out-of-order execution of its instructions. As Figure 6 shows, on average, 17.04% of all allocated registers in SPEC2017int and 13.14% in SPEC2017fp are atomic. Physical registers allocated as part of an atomic region can be safely released as soon as they are *unused* even if the atomic region is executed on the wrong path. This is because we can guarantee that the producer of a destination and all of its consumers will flush.

4 Proposed Technique

We will now introduce the design of our proposed ATR technique shown in Figure 7 and describe its implementation in a modern superscalar out-of-order processor.

4.1 Design

ATR enables to safely early release registers allocated by instructions that are still speculative, as long as they are part of an atomic commit region. As shown in section 2, it is generally unsafe to free unused registers because of branches and exception-causing instructions. For instance, a mispredicted branch between the renaming and redefining instruction will flush the redefining instruction but not the renaming instruction. Due to a change in control flow and the fact that the register is no longer redefined, new consumers of the destination register may enter the rename stage. Prior non-speculative approaches [19, 22], hence, need to wait until the redefining instruction is no longer speculative (it is precommitted).

Atomic commit regions enable additional opportunities by early releasing registers allocated by still speculative instructions. Our insight is that this is possible as long as either all instructions (the producer, the redefining instruction, and all consumers) commit or flush atomically. Consider the following example illustrated in Figure 8: A branch instruction `I1` is followed by an atomic region starting with instructions `I2` (renamed event) and ending with instruction `I5` (redefined event). `I1` is resolved late because it depends on a long-latency load. When the redefining instruction (`I5`) and all potential consumers (`I3-I4`) complete, the architectural register of `I2` becomes *unused*. ATR now frees the physical register of `I2` although there exists an older pending unresolved branch `I1`. If `I1` was mispredicted, then all instructions `I2-I5` must flush. Flushed instructions need to free their physical registers anyways and hence ATR only needs to ensure that the physical register of `I2` is not double-freed. If I1 was predicted correctly, all instructions `I2-I5` are guaranteed to eventually commit, in which case it was also safe to release the unused physical register of `I2`. In summary, atomic commit regions ensure that no new consumers of a speculative unused register can appear, even after a branch (or indirect jump) misprediction.

Precise Exceptions require that if an instruction causes an exception, all older instructions must be committed, and all younger instructions must be flushed before it can be handled. The architectural state of the processor needs to reflect the precise state after completing the last committed instruction. Prior work [19] released registers if they were non-speculative (all prior branches resolved) without considering other exception-causing instructions. As all dominant instruction set architectures (x86, ARM, RISC-V) require precise exceptions we do not consider such approaches feasible and require atomic regions to also exclude exception causing instructions such as loads and stores.

Interrupts can be handled by either (a) stop fetching new instructions and draining the ROB or (b) flushing the ROB and re-executing instructions after the handler. When implementing (a), ATR does not require any modifications. If implementing option (b) is needed to reduce interrupt serving latency, ATR can be extended by adding a counter at the commit stage that tracks the number of active atomic regions. Particularly, the counter is incremented if an instruction with a consumer count of smaller no-early-release commits and it is decremented if an instruction with an invalid previous ptag (see subsection 4.2) commits. ATR continues to execute instructions until the counter is zero and then flushes the pipeline. In the unlikely worst case this can require to fully drain the ROB. This does not violate correctness as no ISA specifies the time that can elapse between the trigger of an interrupt and the time it needs to be handled.

4.2 Implementation

We first introduce the baseline register file architecture and then explain the design changes introduced by ATR.

4.2.1 Baseline. The register file consists of three main components: a Speculative Renaming Table (SRT), a physical register file, and a free list for managing physical register resources. The SRT, also referred to as the architectural or logical table or register alias table (RAT), maintains the mapping from architectural to physical

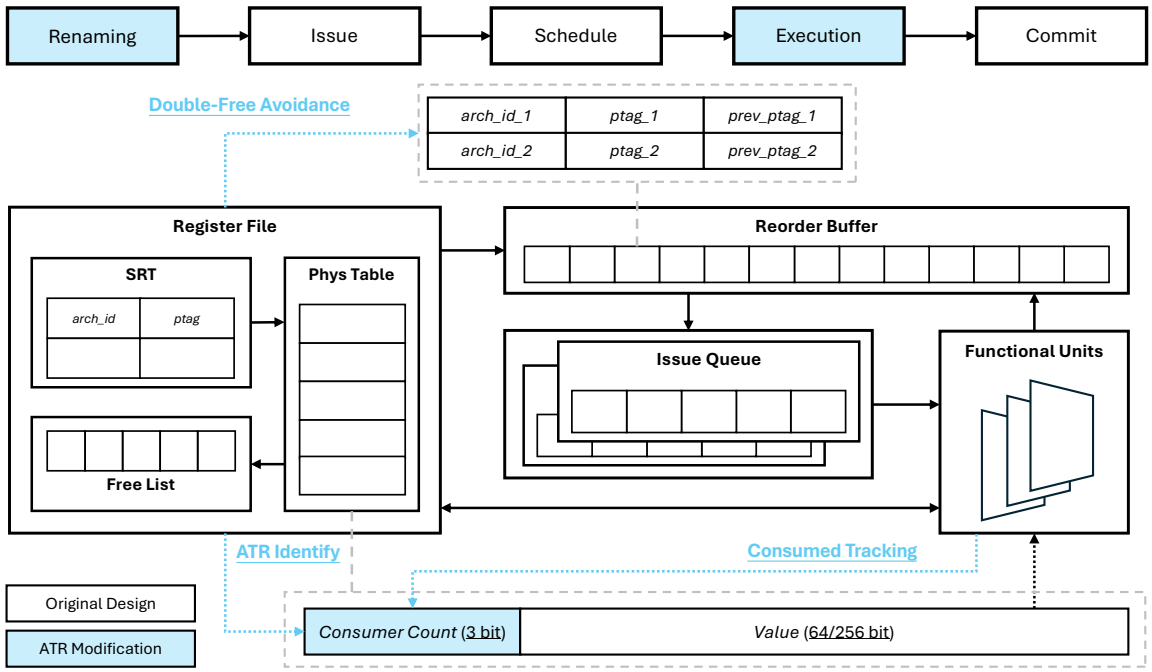


Figure 7: The proposed ATR design.

I1	jne	.LABEL	branch
I2	add	$r_1 \leftarrow r_2, r_3$	rename r_1
I3	sub	$r_2 \leftarrow r_1, r_4$	consume r_1
I4	mul	$r_3 \leftarrow r_1, r_5$	consume r_1
I5	mul	$r_1 \leftarrow r_4, r_5$	redefine r_1

Figure 8: An example of aggressively releasing within ATR.

registers. The SRT is checkpointed on low-confidence branches, to enable restoring the SRT on a flush. If a non-checkpointed branch is flushed, we first restore the most recent checkpoint and then walk the ROB from the checkpoint to the flush point, re-applying register mappings to restore the correct final SRT.

The physical register file stores the actual register values. The free-list contains physical registers (ptags) that are currently unused. During register renaming, a new physical register (ptag) is allocated from the free list, and the SRT is updated with this new mapping. The *previous ptag* (the ptag that was mapped to the architectural register before the rename) is saved in the instruction's metadata to support future release or recovery. At the commit of an instruction, it releases the physical register associated with the previous ptag, which is then returned to the free list. The rename stage is stalled when the number of available entries in the free list falls below $MAX_{DEST} \times WIDTH_{STAGE}$, where MAX_{DEST} represents the maximum number of destinations per instruction and $WIDTH_{STAGE}$ represents the number of instructions that can be renamed per cycle. In x86, MAX_{DEST} equals to four as `CPUID` has four destination operands. The rename stage remains stalled until sufficient entries become available in the free list. When a mispredicted branch is resolved, ptags allocated by flushed instructions are reclaimed by walking the ROB from the tail to the flush point, releasing ptags back to the free list. While checkpointing-based bulk-reclamation techniques [35] are potentially more efficient, to our best knowledge, state-of-the art superscalar CPUs implement walk-based ptag reclamation to support allocation and reclamation of a large and variable number of ptags per cycle. A potential implementation consists of multiple, small, per-superscalar-way, register-based FIFOs that contain a few ptags to enable allocation and reclamation of 0 to N ptags within a single cycle, where N is the superscalar width. The FIFOs allocate ptags from a shared pool (free-list). This pipelined approach enables high ptag bandwidth, high clock frequencies, and high ptag utilization. The reclamation bandwidth on a flush does not need to exceed the commit, respectively, allocation bandwidth, and can be performed across multiple cycles. Such a pipelined implementation provides the opportunity to support out-of-order reclamation techniques, including move elimination [11, 21, 26, 27], register packing [5], and ATR.

We now discuss the required hardware changes to implement our proposed technique. We assume a state-of-the-art baseline architecture that utilizes split register files for scalar and vector registers. This is a common implementation, as vector registers are considerably larger than scalar registers (256 or 512 bits in size). We are assuming separate SRTs and physical register tables for scalar and vector registers, however, the following discussion applies to both, i.e., ATR introduces the same modifications for the scalar and vector register file.

4.2.2 Atomic Region Detection. ATR needs to determine atomic commit regions at runtime and provide the capability to track consumers of physical registers. Therefore, we extend the physical register table (PRT) containing the 64-bit physical register values with a 3-bit consumer count. We modify the register renaming stage of the processor as follows: When the destination of an instruction is renamed, we obtain a new physical register (ptag) from the free list and initialize the consumer count to zero. Whenever a consumer of that destination is renamed, we increment the consumer counter. We reserve the counter value of 7 as *no-early-release*. In particular, if more than 6 consumers register for the same ptag, ATR does not allow to early release that register, instead it will be conventionally released when the redefine instruction commits. Whenever a branch or potentially exception-causing instruction is renamed, ATR sets the counter value to no-early-release for all ptags currently referenced by an architectural register in the SRT. As in a superscalar design, multiple exception-causing instructions can be renamed per cycle, this process needs to be repeated multiple times per cycle. As shown in Figure 9, the ATR logic reads all (old) ptags from the SRT and all (new) ptags from the N instructions being renamed this cycle. This adds one additional read port to the SRT in addition to the existing N per-superscalar-way ports. ATR's SRT reads can be immediately registered to minimize the capacitive load on the SRT. For each renamed instruction, the logic checks if it is a branch or exception-causing instruction, and then it sets all current ptags to no-early-release. For instance, in an 8-wide, x86 design, there are 16 + 7 = 23 ptags that potentially need to be set to no-early-release. We use a parallelized logic design to compute the 23 no-early-release signals simultaneously for all ptags. The bulk no-early-release logic can be N-stage pipelined to further alleviate timing issues. This requires delaying the *redefined* signal of a ptag by the same N number of cycles. This guarantees that a ptag does not appear as redefined before its no-early-release status is computed. The output generated by the logic in Figure 9 is a set of 23 signals for bulk marking ptags as no-early-release eligible and the pipelined SRT ptags used by ATR to determine register redefinition. Section 4.4 provides additional timing results. As we will show in Figure 14, pipelining generally does not delay early atomic release, as ptag redefinition almost always happens before ptag consumption. Particularly, Figure 13 shows that a pipeline delay of 2 has a minimal effect on performance.

4.2.3 Early Atomic Release. Early release of a ptag is performed if (1) the ptag has been redefined (potentially with a delay) and (2) the ptag's consumer count is zero. The consumer count is decremented at issue, unless the count was equal to no-early-release. Decrementing the consumer count does not require accessing the SRT, as each issuing consumer maintains its ptags. ATR guarantees that consumers either atomically commit or flush with their corresponding producer. As a result, there is no need to restore consumer counts on a flush.

4.2.4 Double-Free Avoidance. We prevent double freeing of ptags on commit, respectively, precommit by invalidating the *previous ptag* field. In particular, if a redefining instruction is detected at rename without a prior branch, ld, or store, the allocating instruction of the same architectural register is early-release-eligible. To ensure that only ATR releases that register, the previous ptag is assigned to `invalid`. In contrast, when the previous ptag is valid, the ptag will be freed by the commit or precommit logic instead. This ensures that each ptag is freed by only one mechanism.

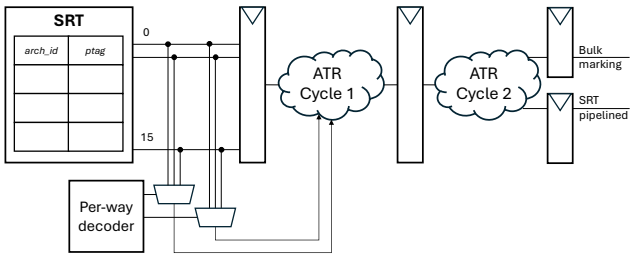


Figure 9: Renaming stage modifications for bulk setting of no-early-release

During a flush operation, preventing double frees is more complex because three cases must be considered: (1) ptags that are not eligible for early release, (2) ptags that are eligible but have not yet been early released, and (3) ptags that have already been early released. To handle this, ATR introduces two bits of storage for each architectural register ID (32 total for x86). The *redefined bit* indicates that the corresponding architectural register has been overwritten, and the *consumed bit* indicates that all consumer instructions have been issued. During the flush walk, ATR examines the source and destination registers of each instruction. If the previous ptag of a destination register is invalid (i.e., ATR-releasable), ATR sets both the redefined and consumed bits for said architectural register. If a source register of a subsequent younger instruction (because we walk from oldest to the flush point) matches a redefined register and the instruction has not yet been issued, the consumed bit is cleared.

For each instruction, if its destination register has the redefined bit set and the consumed bit still set, ATR skips releasing the corresponding ptag, as it was already early released by ATR. After performing this check, both flags are cleared for the corresponding architectural register. All remaining ptags, those that are either non-atomic or have not yet been released, are reclaimed and returned to the free list. For the algorithm above to work properly, we perform the actions in the following order. For each instruction, we first check whether its ptag needs to be freed, second, we potentially clear the consumed bits for its source registers, third, we potentially set the redefined bit for its architectural destination register.

4.3 Combination with Non-Speculative Early Release

ATR can be combined with prior work on non-speculative early release. In this case, registers can be early released in two cases. (1) ATR releases physical registers if they are allocated as part of an atomic commit region, if their consumer count is zero, and if the register is redefined. (2) Non-speculative early release frees registers if they have a consumer count of zero and if the redefining instruction precommits. The two approaches are synergistic. ATR

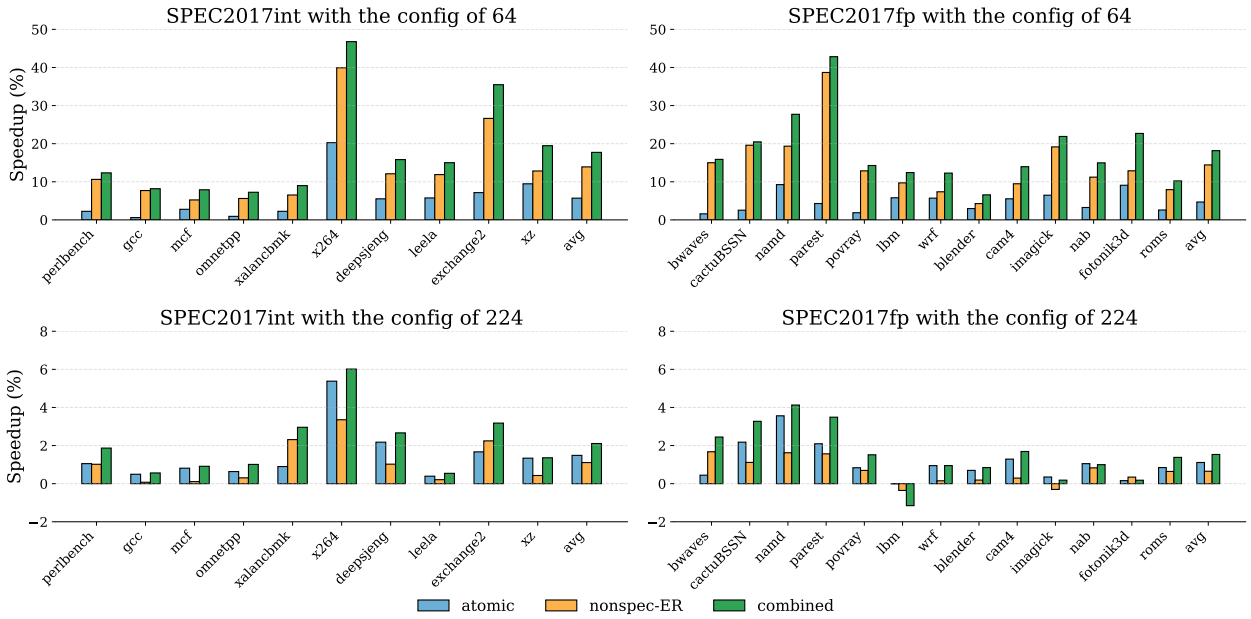


Figure 10: IPC speedup over the baseline with 64 and 224 physical registers.

releases registers earlier but is limited to atomic commit regions. Non-speculative Early Release frees registers of non-atomic regions but needs to wait until the redefining instruction precommits.

4.4 Hardware Overheads

ATR introduces a storage cost of three bits per physical register to maintain the consumer counter. This leads to an overhead of $3/64 = 4.6\%$ for the scalar integer register file. For the vector register file the overhead is $3/256 = 1.1\%$. Note that these calculations are conservative. Physical register files often include additional metadata, such as a duplicate count, to enable move elimination [11, 21, 26, 27] in the register renaming stage or a bit vector to encode the state of the register. As a result, the reported overheads are likely lower in existing implementations. Furthermore, if ATR is implemented on top of Non-speculative Early Release, the 3-bit consumer counter can be shared, resulting in effectively no storage overhead.

We implemented the bulk no-early-release logic in System Verilog and synthesized it into a netlist with Yosys 0.37. The worst-case path of the design has 42 logic levels. Assuming a 4.5 ps FO4 delay at 5nm [23] and a 100% margin due to wire delay and gate fan-in (NAND can be 1.4 FO4 [30]), this results in a delay of 378ps or 2.6GHz. Adding two additional pipeline stages should enable clock frequencies of beyond 4GHz. The logic circuit consists of 2,960 gates.

5 Evaluation

We first describe our evaluation methodology and then present the performance results for the proposed register renaming scheme.

5.1 Experimental Methodology

Simulation Environment. We leverage the open-source, cycle-level Scarab simulator [24] to perform our analysis. Scarab models a detailed superscalar out-of-order processor with a decoupled frontend, modern branch predictors, register renaming, diversified functional units, wrong-path simulation, and a detailed memory hierarchy. We configure Scarab to match a recent Intel Golden Cove processor in terms of width and depth, as summarized in Table 1. Scarab supports both execution-driven and trace-based frontends for microarchitectural simulation. In execution-driven mode, it leverages Intel's PIN [15] to dynamically resteer the binary onto incorrect control paths, enabling accurate simulation of wrong-path instruction execution. In trace-based mode, Scarab captures every executed program counter (PC) and its corresponding instruction to replay the instruction stream, including wrong-path execution.

Workloads We use the SPEC2017int and SPEC2017fp benchmarks [29] for our experiments, as shown in Table 2. For each application, we simulate representative 10M instruction simpoints [8] aggregated according to their weights. The number of simpoints varies by application, ranging from 8 to over 100. Each simpoint is warmed up with 10M instructions. The traces are collected using DynamoRIO [3] and Intel PT [15], capturing a precise, continuous sequence of dynamically executed basic blocks along with their corresponding memory addresses. These traces are then analyzed to identify and extract representative steady-state regions for further evaluation.

5.2 Performance

Figure 10 presents a comparison of normalized IPC across SPEC2017 integer and floating-point benchmarks using fixed configurations of 64 and 224 physical registers. Four schemes are evaluated: the

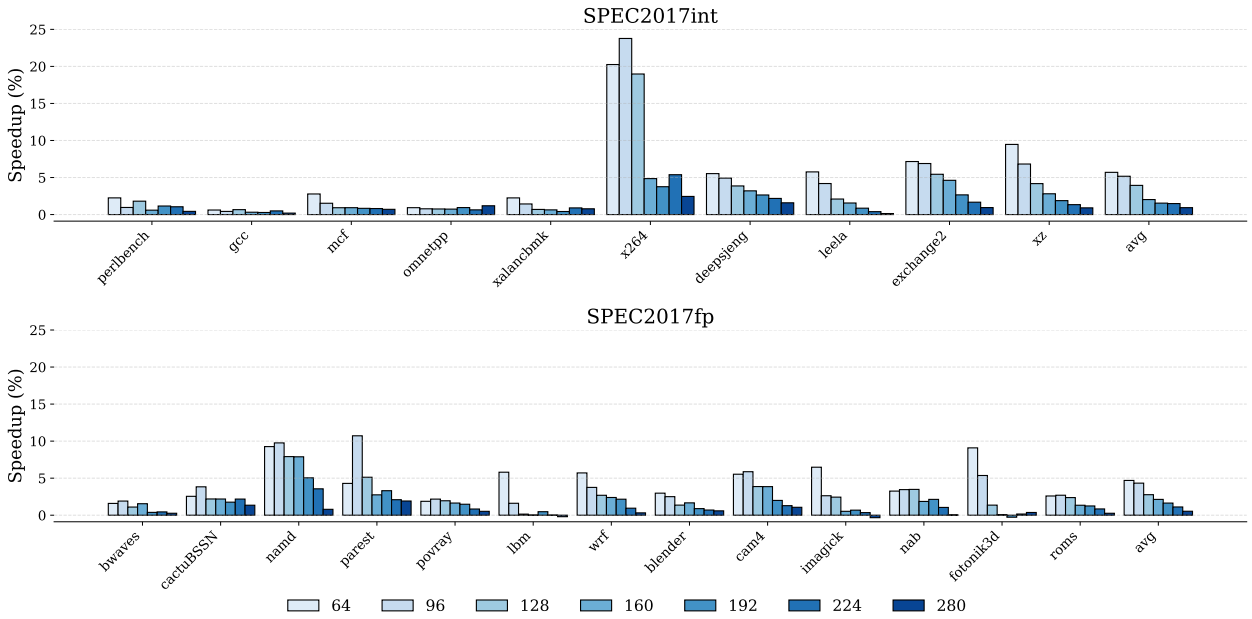


Figure 11: IPC speedup of the atomic scheme over the baseline with different RF Size.

Table 1: Processor Configuration

Parameter	Value
Core	
CPU	Golden Cove
All-core turbo frequency	3.0 GHz
Frontend width and retirement	6-wide fetch/decode, 8-wide retirement
Functional Units	5 ALU, 3 Load, 2 Store
Branch Predictor	TAGE-SC-L + BPU enhancements
Branch Target Buffer (BTB)	12K entries
Indirect Branch Target Buffer	3K entries
ROB	512 entries
Reservation Station	160 entries
Data Prefetcher	Stream, Spatial
Instruction Prefetcher	FDIP with deeper prefetch window
Load Buffer	96 entries
Store Buffer	64 entries
Frontend Fetch targets (FT) per cycle	2
FT block size	64 B
Caches & Memory	
L1 instruction cache	32 KiB, 8-way
L1 data cache	48 KiB, 12-way
L2 unified cache	1.25 MiB, 10-way
LLC unified cache	Shared 3 MiB/core, 12-way
L1 D-cache latency	3 cycles
L1 I-cache latency	3 cycles
L2 latency	14 cycles
LLC latency	40 cycles
Memory	DDR4-3200 (2 channels)

Table 2: SPEC CPU 2017 Benchmarks

Integer benchmarks (SPEC2017int)			
500.perlbench_r	502.gcc_r	505.mcf_r	520.omnetpp_r
523.xalancbmk_r	525.x264_r	531.deepsjeng_r	541.leela_r
548.exchange2_r	557.xz_r		
Floating-point benchmarks (SPEC2017fp)			
503.bwaves_r	507.cactuBSSN_r	508.namd_r	510.parest_r
511.povray_r	519.lbm_r	521.wrf_r	526.blender_r
527.cam4_r	538.imagick_r	544.nab_r	549.fotonik3d_r
554.roms_r			

baseline, non-speculative early release (nonspec-ER), the proposed atomic technique, and the combined approach integrating both techniques. For the configuration of 64 physical registers, the atomic scheme achieves average speedups of 5.70% and 4.69% over the baseline for SPEC2017int and SPEC2017fp, respectively, while the nonspec-ER scheme improves IPC by 13.91% for SPEC2017int and 14.43% for SPEC2017fp compared to the baseline. Furthermore, the combined technique provides an additional performance gain over nonspec-ER, with speedups of 3.23% and 3.27% for SPEC2017int and SPEC2017fp, respectively. With a configuration of 224 physical registers, the atomic scheme shows average improvements of 1.48% for SPEC2017int and 1.11% for SPEC2017fp over the baseline, and further outperforms the nonspec-ER scheme by 0.37% and 0.46%, respectively.

5.3 RF Size Sensitivity

Figure 11 studies the sensitivity of performance to register file size by comparing the atomic scheme against the baseline across register counts ranging from 64 to 280. On average, the speedup decreases as the register size increases, with the highest gains observed at 64 registers. At this configuration, the atomic scheme demonstrates average speedups of 5.70% and 4.69% over the baseline for SPEC2017int and SPEC2017fp, respectively. At the 280-register configuration, where register pressure is minimal, the atomic scheme yields the smallest average gains, with performance improvements of 0.93% for SPEC2017int and 0.53% for SPEC2017fp, respectively.

5.4 Consumer Counter Width Sensitivity

ATR utilizes a 3-bit consumer counter, which allows for up to six consumers, as one value is reserved for no-early-release. Figure 12 shows the number of average consumers per atomic region. As

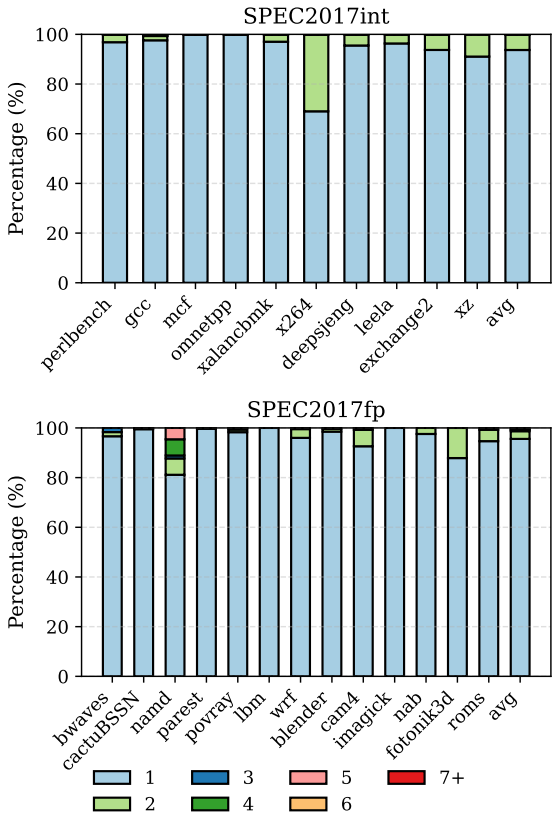


Figure 12: Consumer count distribution.

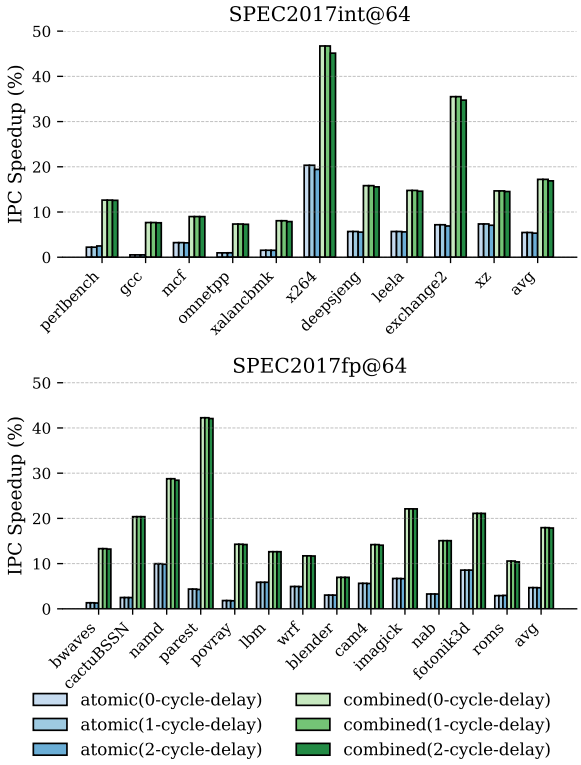


Figure 13: Performance effect of pipelining the register redefinition logic.

can be seen, for most workloads, regions only have 1-2 consumers in average. Only namd shows a considerable number of regions with up to five consumers. As a result, the performance difference between ATR with an infinitely sized and a 3-bit-sized counter is negligible. Note that atomic regions are generally small in terms of instructions, as they cannot contain any branches or exception-causing instructions. Mechanisms such as non-speculative early release require a larger consumer counter, as they need to cover regions (instructions between allocating and redefining instruction) that are generally larger.

5.5 Pipeline Delay Sensitivity

Section 4.2.2 introduces the modifications introduced to the renaming stage by ATR. As the logic for bulk-setting ptags to no-early-release may need to be pipelined, Figure 13 explores designs in which the register redefinition logic of ATR is delayed by 1, respectively 2, cycles. As can be seen the impact on performance is negligible. To provide further insight, Figure 14 analyzes the average time between (1) a register is renamed and redefined, (2) a register is renamed and consumed, and (3) a register is renamed and the redefining instruction commits within atomic commit regions. It is observable that in average, the consumption of a register happens significantly later than its redefinition. As ATR early releases only if both conditions are met, delaying redefinition by 1 to 2 cycles does not significantly affect performance. Consumption generally happens later because it needs to consider data dependencies, whereas redefinition happens at rename regardless of any data dependencies. ATR improves register utilization as it holds registers for a significantly shorter time (2) than the baseline, releasing at commit of the redefining instruction (3).

5.6 Overhead Optimization Study

Figure 15 shows the effectiveness of each scheme in optimizing the overhead required to keep IPC within 3% of the baseline configuration with 280 registers. The atomic scheme requires 204 registers, reducing the register demand by 27.1%. The nonspec-ER approach lowers the requirement to 212 registers, yielding a 24.3% reduction. The combined scheme, which integrates both atomic and nonspec-ER techniques, delivers the most significant improvement, requiring only 196 registers and achieving a 30% reduction in register file size. In addition, we perform a detailed power and area analysis using McPAT [13] to assess the architectural efficiency of our approach. The atomic scheme achieves a 5.5% reduction in runtime power and a 2.7% reduction in core area, while the combined scheme provides a 5.5% power saving and leads in area reduction with a 2.9% decrease.

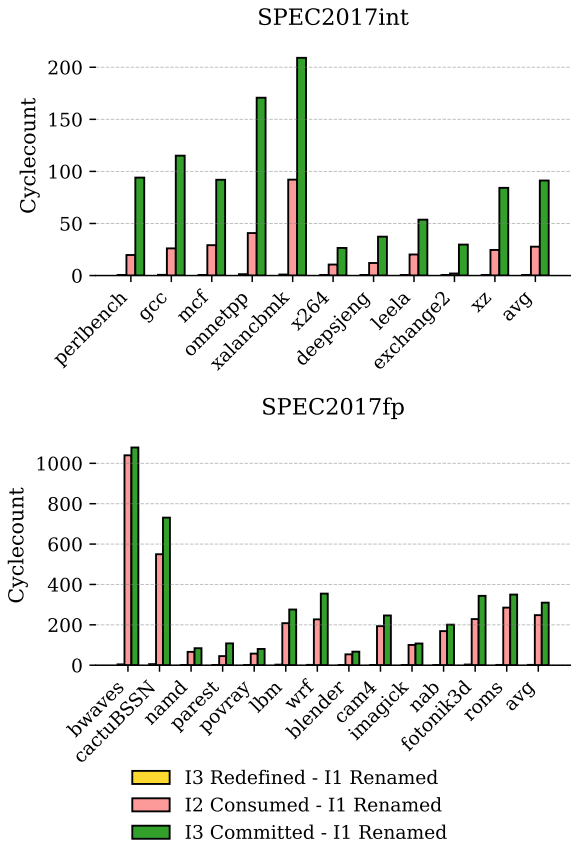


Figure 14: Average cyclecount between rename, redefine, and commit.

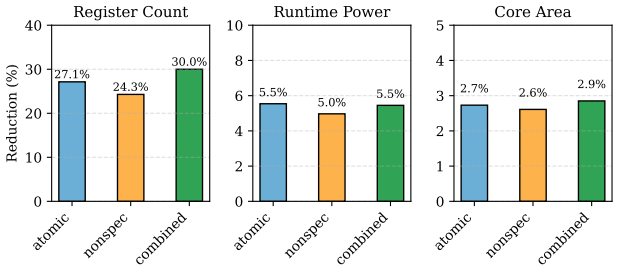


Figure 15: Overhead reduction of different schemes.

6 Related Work

Non-Speculative Early Release. As defined in subsection 2.3, non-speculative early release requires that the redefining instruction be precommitted. This precommitment condition implies that all control-flow instructions and any exception-causing instructions before the redefining instruction are resolved. Previous works relaxed this requirement by checking only conditional branches, but they cannot support precise exceptions. Note that in this paper, non-speculative early release is discussed in combination with our proposed ATR technique. Moudgill et al. [22] proposed that a physical register can be released once it is redefined and last consumed. To track the last consumer, they employed counters that record the number of pending reads for each physical register. Substantial extra hardware is required to support misprediction recovery and interrupts, including a FIFO structure storing metadata at each branch and a history buffer for each instruction. Monreal et al. [19] delayed the early-release point from I2-consumed to I2-committed, as defined in subsection 3.1. Physical registers are freed once the last instruction reading the register has committed if the redefining instruction is non-speculative. A last-use table is employed to track the last consumer, which is snapshotted at each branch for misprediction recovery. They also presented an alternative technique for tracking the confirmation of the last branch using a multi-level queue that corresponds to unconfirmed branches.

Speculative Early Release. A speculatively released register value may be reused later due to mispredictions. It is typically unsafe unless the deallocated values are backed up, which incurs high overhead. Ergin et al. [6] introduced a checkpointed register file to enable early register release, which maintains a shadow cell for each physical register bitcell. The deallocated values can be stored in the corresponding shadow cells, allowing recovery from mispredictions, interrupts, or exceptions. Building on the checkpointed register file, Jones et al. [9] utilized compiler knowledge to exactly track the last use of a register, facilitating earlier register release. Tabani et al. [31] proposed a novel register renaming technique by reusing physical registers when a value has a single consumer. This register-reusing technique can be interpreted as an early release mechanism for single-use producer physical registers. They also employ shadow banks to back up previously released values for recovery in the event of mispredictions, interrupts, or exceptions. Mehta [18] proposed a more aggressive register reclamation strategy that releases a physical register early once it is redefined. This approach leverages the observation that within loops, an architectural register is often redefined in the same or subsequent iteration, rendering the previous value unnecessary. To support this optimization, the register read stage is redesigned to occur before the dispatch stage. Additionally, a dedicated payload RAM is introduced to store these early-released values, which are then bypassed directly to the functional units.

Checkpointing-based Early Release CPR [1] introduced by Akkary et al. proposes a speculative early register release mechanism based on checkpoints that aggressively releases registers as soon as they are redefined and fully consumed. CPR is speculative because it does not prevent early release for non-atomic regions. Instead, it uses re-execution from a checkpoint to ensure correctness, which renders flushes, memory nukes, and other misspeculated events very expensive. Cherry [12, 17] is an alternative implementation that combines checkpointing with a conventional ROB to minimize re-execution in some cases, still allowing for the speculatively early release of register resources.

Speculative Allocation Elimination. During renaming, registers are not allocated if they are speculatively predicted to be forwardable, shareable, or unused. These values are typically backed up in memory or shadow registers to enable recovery in case of misprediction. This technique is compatible with the proposed ATR approach. Balkan et al. [2] proposed a technique that avoids the

need for register allocation by forwarding transient values that are single-use and short-lived. To identify such instructions, a transient value prediction is applied during instruction fetch and verified before writeback. An additional buffer is still required for storing deallocated values to handle cases where the transient prediction is incorrect. Yan et al. [34] also leveraged network forwarding to allocate virtual registers instead of physical registers for short-lived values on VLIW-based embedded processors. To handle interrupts and exceptions, the values in pipeline registers are stored as part of the processor state. Butts et al. [4] presented a mechanism to predict dead instructions, eliminating the allocation of unconsumed destination registers while also optimizing scheduling and execution. The mechanism learns instructions whose destinations are never consumed and eliminates them. On a misprediction (the destination is indeed required), the mechanism flushes the pipeline and replays the previously eliminated instructions. Jourdan et al. [11] proposed register sharing to reduce physical register pressure by allowing multiple instructions to reuse the same physical register. Perais et al. [25] proposed further optimizations to improve the efficiency of register sharing.

Late Allocation. Gonzalez et al. [7] proposed virtual-physical registers to delay physical register allocation until the corresponding instruction finishes its execution. This is achieved by introducing an additional virtual table, which maps architectural registers to virtual-physical registers during renaming and virtual-physical registers to physical registers during execution. Late allocation can be combined with other renaming techniques. Monreal et al. [20] integrated late allocation with non-speculative early release to further optimize the register lifecycle.

Move Elimination. Move elimination [11, 21, 26, 27] eliminates certain instructions such as register-to-register moves by renaming multiple architectural destinations to the same physical register, reducing physical register usage. Move elimination is orthogonal to ATR and can be combined synergistically, usually implemented by adding a ptag reference count incremented on a move elimination and decremented on redefinition commit. ATR needs to be extended to decrement ref counts on early-release and the flush walk needs to be modified to decrement instead of release.

Physical Register Inlining. Lipasti et al. [14] proposed another technique to improve register file efficiency. The mechanism stores small (e.g., 8-bit) values in the SRT directly without allocating a ptag. Register packing [5] packs multiple smaller register values, e.g., two 32-bit registers in a single 64-bit register. Both techniques are orthogonal to ATR and can be combined for additional storage efficiency.

Trace processors. Rotenberg et al. [28] proposes local register files to reduce the lifetime of registers local to a trace. While traces cannot be considered atomic as they can contain flush or exception-causing instructions the work proposes another form of speculative out-of-order register reclamation.

7 Conclusion

In this paper, we observe that registers allocated within *atomic commit regions*—instruction sequences that contain neither conditional branches nor exception-causing instructions—can be safely released out of order. This is because all instructions within such regions either commit together or are flushed.

Our analysis shows that over 17% of all allocated registers in SPEC2017int and 13% in SPEC2017fp fall within *atomic commit regions*. Based on this insight, we propose a novel register renaming technique that enables out-of-order register release by exploiting atomic commit regions. This approach preserves precise exception handling while eliminating the need for misprediction recovery support in these cases.

We show that, across the SPEC benchmarks, the proposed scheme achieves an average speedup of 5.13% for a 64-entry, and 1.48% for a 224-entry register file. Alternatively, it enables an average reduction of over 27.1% in register file size while maintaining IPC degradation below 3% relative to the baseline. Finally, we demonstrate that our technique can be integrated with other register renaming approaches to further enhance performance.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and helpful feedback. This work was generously supported by Intel's Center for Transformative Server Architectures (TSA), the Center for Research in Storage Systems (CRSS), and NSF grants #2111688 and #1942754.

References

[1] Haitham Akkary, Ravi Rajwar, and Srikanth T Srinivasan. 2003. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 423–434.

[2] Deniz Balkan, Joseph Sharkey, Dmitry Ponomarev, and Kanad Ghose. 2006. SPARTAN: speculative avoidance of register allocations to transient values for performance and energy efficiency. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques.* 265–274.

[3] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE, 265–275.

[4] J Adam Butts and Guri Sohi. 2002. Dynamic dead-instruction detection and elimination. *ACM SIGPLAN Notices* 37, 10 (2002), 199–210.

[5] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. 2004. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *37th International Symposium on Microarchitecture (MICRO-37'04).* IEEE, 304–315.

[6] Oguz Ergin, Deniz Balkan, Dmitry Ponomarev, and Kanad Ghose. 2004. Increasing processor performance through early register release. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.* IEEE, 480–487.

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

[8] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[9] Timothy M Jones, MFR O'Boyle, Jaume Abella, Antonio González, and Oguz Ergin. 2005. Compiler directed early register release. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05).* IEEE, 110–119.

[10] Timothy M Jones, Michael FP O'boyle, Jaume Abella, Antonio González, and Oğuz Ergin. 2009. Exploring the limits of early register release: Exploiting compiler analysis. *ACM Transactions on Architecture and Code Optimization (TACO)* 6, 3 (2009), 1–30.

[11] Stephan Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. 1998. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture.* IEEE, 216–225.

[12] Meyrem Kirman, Nevin Kirman, and Jose F Martinez. 2005. Cherry-MP: Correctly integrating checkpointed early resource recycling in chip multiprocessors. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05).* IEEE, 12–pp.

[13] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: An integrated power, area, and timing modeling

framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*. 469–480.
[14] Mikko H Lipasti, Brian R Mestan, and Erika Gunadi. 2004. Physical register inlining. *ACM SIGARCH Computer Architecture News* 32, 2 (2004), 325.
[15] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.
[16] Salvador Petit Marti, Julio Sahuquillo Borras, Pedro Lopez Rodriguez, Rafael Ubal Tena, and Jose Duato Marin. 2009. A complexity-effective out-of-order retirement microarchitecture. *IEEE Transactions on computers* 58, 12 (2009), 1626–1639.
[17] José F Martínez, Jose Renau, Michael C Huang, and Milos Prvulovic. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.* IEEE, 3–14.
[18] Sanyam Mehta. 2023. Speculative register reclamation. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1182–1194.
[19] Teresa Monreal, Víctor Viñals, Antonio González, and Mateo Valero. 2002. Hardware schemes for early register release. In *Proceedings International Conference on Parallel Processing*. IEEE, 5–13.
[20] Teresa Monreal, Victor Vinals, José González, Antonio González, and Mateo Valero. 2004. Late allocation and early release of physical registers. *IEEE Trans. Comput.* 53, 10 (2004), 1244–1259.
[21] Andreas Moshovos and Gurindar S Sohi. 1997. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 235–245.
[22] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. 1993. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of the 26th annual international symposium on Microarchitecture*. IEEE, 202–213.
[23] Stefan Nikolić, Francky Catthoor, Zsolt Tőkei, and Paolo Ienne. 2021. Global is the new local: FPGA architecture at 5nm and beyond. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 34–44.
[24] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. 2024. Udp: Utility-driven fetch directed instruction prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1188–1201.
[25] Arthur Perais and André Seznec. 2016. Cost effective physical register sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 694–706.
[26] Vlad Petric, Anne Bracy, and Amir Roth. 2002. Three extensions to register integration. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002.(MICRO-35). Proceedings.* IEEE, 37–47.
[27] Vlad Petric, Tingting Sha, and Amir Roth. 2005. Reno: a rename-based instruction optimizer. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 98–109.
[28] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. 1997. Trace processors. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 138–148.
[29] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017 Benchmark Suite. https://www.spec.org/cpu2017/.
[30] Ivan Sutherland, Robert F Sproull, and David Harris. 1999. *Logical effort: designing fast CMOS circuits*. Morgan Kaufmann.
[31] Hamid Tabani, Jose-Maria Arnau, Jordi Tubella, and Antonio Gonzalez. 2018. A novel register renaming technique for out-of-order processors. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 259–270.
[32] Jessica H Tseng and Krste Asanović. 2003. Banked multiported register files for high-frequency superscalar microprocessors. In *Proceedings of the 30th annual international symposium on Computer architecture*. 62–71.
[33] Kaifan Wang, Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Xi Chen, Lingrui Gou, Xuan Hu, Yue Jin, Qianruo Li, et al. 2023. XiangShan open-source high performance RISC-V processor design and implementation. *Journal of Computer Research and Development* 60, 3 (2023), 476–493.
[34] Jun Yan and Wei Zhang. 2007. Virtual registers: Reducing register pressure without enlarging the register file. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 57–70.
[35] Kenneth C Yeager. 1996. The MIPS R10000 superscalar microprocessor. *IEEE micro* 16, 2 (1996), 28–41.

8 Appendix

8.1 Abstract

This artifact contains the complete source code for the ATR release scheme implemented in the Scarab simulator, targeting the x86 goldencove-like architecture. It also includes the necessary tooling to execute the SPEC2017int and SPEC2017fp benchmark suites within Scarab. To facilitate reproducibility, the artifact provides Docker-based workflows for setting up simulation environments and replicating the key experimental results. Additionally, accompanying plotting scripts are included to regenerate Figures 4, 6, 10, and 12.

8.2 Artifact check-list (meta-information)

- **Program:** Scarab simulator and Docker tool
- **Compilation:** Automated by tooling
- **Data set:** SPEC2017int and SPEC2017fp
- **Hardware:** Intel x86 64 processor
- **Execution:** Automated by tooling
- **Output:** Graphs of IPC speedups
- **Experiments:** Automated by tooling
- **How much disk space required (approximately)?:** 50GB
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes to install Docker and build a docker image
- **How much time is needed to complete experiments (approximately)?:** 24 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT license
- **Data licenses (if publicly available)?:** MIT license
- **Workflow automation framework used?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.16734823

8.3 Description

8.3.1 How to access. All source code is open-source and available on GitHub. The Scarab simulator, including the proposed ATR feature, is provided at: https://github.com/litz-lab/scarab/tree/MICRO2025-ATR. The simulation workflow requires cloning the repository at https://github.com/litz-lab/scarab-infra, which automates the installation of Scarab, all necessary dependencies, and the generation of plots corresponding to the figures presented in the paper.

8.3.2 Hardware dependencies. Intel x86-64 processors are required, as Scarab relies on Intel's PIN, a dynamic binary instrumentation framework that supports the IA-32, x86-64, and MIC instruction set architectures.

8.3.3 Software dependencies. The `conda` environment is required to run the simulation infrastructure. The corresponding environment specification file is available at https://github.com/litz-lab/scarab-infra/blob/main/quickstart_env.yaml.

In addition, the `slurm` environment is required. The installation steps are introduced in https://github.com/litz-lab/scarab-infra/blob/main/docs/slurm_install_guide.md.

8.3.4 Data sets. SPEC2017int and SPEC2017fp traces are available by following the instructions provided in the `README` file of the `MICRO2025-ATR` branch.

8.4 Installation

(1) Install the Docker Engine, `conda` and `slurm`.
(2) Clone the Scarab infrastructure repository and checkout the `MICRO2025-ATR` branch.

8.5 Experiment workflow

(1) Download the SPEC2017int and SPEC2017fp traces using `gdown`.
(2) Prepare the JSON configuration file and place it in the `json/` directory.
(3) Launch the simulation with `./run.sh -simulation atr`.

8.6 Evaluation and expected results

Figures 4, 6, 10, and 12 can be reproduced using the provided scripts in the `MICRO2025-ATR` branch.