CPROB: Checkpoint Processing with Opportunistic Minimal Recovery

Andrew Hilton, Neeraj Eswaran, and Amir Roth
Computer and Information Science Department, University of Pennsylvania
{adhilton, neeraj, amir}@cis.upenn.edu

***Abstract*—CPR (Checkpoint Processing and Recovery) is a physical register management scheme that supports a larger instruction window and higher average IPC than conventional ROB-style register management. It does so by restricting mis-speculation recovery to checkpoints created at rename, and leveraging this restriction to aggressively reclaim registers that don't appear in checkpoints. The cost of CPR is checkpoint overhead, which is incurred when a mis-speculation occurs on an instruction for which a checkpoint was not created a priori. Here, CPR must recover to the immediately older checkpoint, squashing instructions older than the mis-speculation itself. In contrast, a ROB processor performs minimal recovery and only squashes instructions younger than the mis-speculation.**

CPROB is a hybrid register management scheme that preserves CPR's aggressive reclamation while opportunistically minimizing checkpoint overhead. CPROB extends CPR to track and hold the registers needed to perform minimal recovery to un-executed branches within each checkpoint. Recovery registers are held on a best-effort basis only. A checkpoint's recovery registers can be freed spontaneously when all branches in the checkpoint execute. They can also be aggressively victimized if dispatch needs registers to proceed. CPROB naturally adapts the register reclamation policy to dynamic branch behavior. When branch mis-predictions are infrequent and registers are needed to support a large window, CPROB victimizes registers and behaves like CPR. When mis-predictions are frequent and the window is small, CPROB holds on to registers and behaves like ROB. As a result, it out-performs both CPR and ROB for a given program. This performance improvement, combined with reduced checkpoint overhead, makes CPROB more energy-efficient than either ROB or CPR.

I. Introduction

Some recent work has focused on scaling the out-of-order instruction window to accommodate large numbers of instructions for the purpose of hiding last-level cache miss latencies [7], [8], [11], [20], [21], [23], [25], [30]. CPR (Checkpoint Processing and Recovery) [1] scales one of these critical window resources—the physical register file. CPR periodically checkpoints the map-table. Registers that appear in checkpoints are freed conservatively, but registers that hold inter-checkpoint temporary values are reclaimed aggressively. For a given register file size, CPR supports a larger window—and provides higher average IPC—than a processor with ROB-based register reclamation (henceforth simply "ROB"). Techniques like CFP (Continual Flow Pipelines) [30] and TCI (Transparent Control Independence) [4] build on CPR and exploit the efficient way in which it manages registers.

The cost of CPR's aggressive register reclamation is that recovery is supported only to pre-created checkpoints. A mis-speculation on an un-checkpointed instruction requires recovering to the immediately older checkpoint, squashing and redoing instructions that are older than the mis-speculation. This cost, which is termed *checkpoint overhead*, is unique to CPR. In contrast, ROB performs *minimal recovery*, squashing only instructions younger than the mis-speculation. Schemes which build on CPR inherit the checkpoint overhead problem.

Figure 1 demonstrates both aggressive register reclamation and checkpoint overhead with a short example that uses three logical registers (*r1–r3*), eight physical registers (*p1–p8*), and eight instructions (*A–H*). A map-table tracks the logical-to-physical mapping before each instruction. Checkpoints exist before instructions *B* (Checkpoint 1) and *G* (Checkpoint 2). All instructions except for branch *D* have executed.

CPR's benefits and costs can be seen by focusing on physical register *p5*. ROB can free *p5* only when *E* commits. CPR can free *p5* once *E* executes, because *p5* does not appear in the map-table or in any checkpoint and is not read by any other instruction. The fact that *p5* can be freed aggressively also means that if branch *D* mis-predicts, then recovery is only possible to Checkpoint 1—with the squash of instructions *B–D* constituting checkpoint overhead. It is not possible to recover to the state immediately before *E* because the corresponding map-table would contain *p5*.

Figure 2 quantifies checkpoint overhead for the SPEC2000 benchmarks on CPR architectures with 8 and 16 checkpoints. We show checkpoint overhead—older-than-mis-speculation instructions squashed as a percentage of instructions committed—and speedup over ROB. For many programs, high branch-prediction accuracy and, more specifically, few low-confidence branches [15] limit checkpoint overhead and expose the

	MapTable				Instruction			Held Registers		
	r1	r2	r3	PC	Raw	Rename	ROB	CPR-Insn	CPR-Chkpt	CPROB
	p1	p2	p3	**A:**	brz r1,M	brz p1,M	–	**p1**		
Chkpt1	**p1**	**p2**	**p3**	**B:**	ld [r3]->r3	ld [p3]->p4	p3	p3,p4	**p1,p2,p3**	
	p1	p2	p4	**C:**	sub r1,4->r2	sub p1,4->p5	p2	p1,p5		
	p1	p5	p4	**D:**	brz r3,Q	**brz p4,Q**	**–**	**p4**		
	p1	p5	p4	**E:**	ld [r2]->r2	ld [p5]->p6	**p5**	p5,p6		
	p1	p6	p4	**F:**	brz r2,T	brz p6,T	**–**	p6		
Chkpt2	**p1**	**p6**	**p4**	**G:**	add r1,8->r3	add p1,8->p7	p4	p1,p7	**p1,p6,p4**	**p5**
	p1	p6	p7	**H:**	ld [r3]->r2	ld [p7]->p8	p6	p7,p8		
MapTable	**p1**	**p8**	**p7**						**p1,p8,p7**	

Figure 1. An example of CPR and CPROB.

benefits of aggressive register reclamation. For others like *galgel*, *gcc* and *gzip*, checkpoint overhead is high and results in slowdowns over ROB. Using more checkpoints and allocating them more frequently reduces checkpoint overhead but also defeats aggressive register reclamation by locking down too many registers.

CPROB is a hybrid CPR/ROB register reclamation scheme which extends CPR with *best-effort* minimal recovery to un-checkpointed branches. In CPROB, a checkpoint holds not only the physical registers that appear in its map-table snapshot, but also the registers needed to perform minimal recovery to any branch within the *previous* checkpoint. Using an example from Figure 1, under CPR, Checkpoint 2 holds registers *p1*, *p6*, *p4*. Under CPROB, it also holds *p5* which allows it to support minimal recovery to instruction *E* should branch *D* mis-predict. To recover to *E*, CPROB first recovers to Checkpoint 2, CPR-style. It then works backwards serially, ROB-style. A key point is that CPROB holds recovery registers on a best-effort basis. They can be released at any time, *e.g.*, when rename needs registers to proceed. The only consequence of releasing recovery registers too early is forfeiting minimal recovery to branches that are "protected" by them, should these mis-predict.

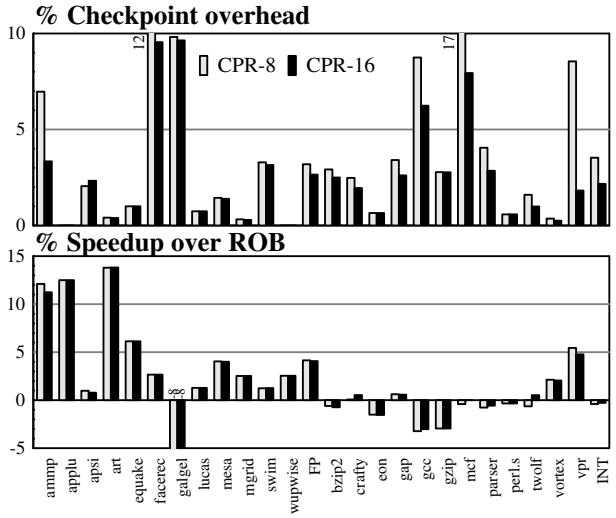


Figure 2. CPR checkpoint overhead and speedup over ROB.

Experiments show that for a range of configurations—register file sizes, ROB sizes, etc.—CPROB out-performs both ROB and CPR, eliminating CPR's performance pathologies while maintaining its advantages in the common case. CPROB does this naturally in response to dynamic register demand and branch prediction accuracy. When mis-predictions are infrequent and the window expands, CPROB victimizes recovery registers and mimics CPR. When mis-predictions are frequent, the window is small and many registers are available. Here, CPROB does not need to victimize recovery registers and mimics ROB.

CPROB improves CPR's energy-efficiency as well. Using instruction dispatch counts as rough approximations for dynamic energy consumption we show that CPR has a higher ED^2 than ROB because its small performance gains are accompanied by larger increases in instruction dispatch counts due to checkpoint overhead. CPROB improves performance *and* reduces instruction dispatch counts, resulting in a lower ED^2 than ROB.

Finally, we experiment with CPR and CPROB in the context of SMT [33]. Although SMT reduces the branch mis-prediction penalty, it actually exacerbates checkpoint overhead. CPROB mitigates this and allows SMT to benefit from CPR's efficient register management.

The main contributions of this work are:

- We introduce CPROB, a hybrid reclamation scheme that directly attacks CPR's checkpoint overhead problem, while preserving its efficient register management properties.
- We use simulation to show that CPROB outperforms both CPR and ROB for a variety of configurations and to argue that it is more energy-efficient than both designs.
- We evaluate CPR and CPROB in an SMT context.

II. BACKGROUND: CPR

Conventional ROB-based architectures reclaim registers in-order and at instruction granularity. At rename, an instruction notes the physical register which it overwrites in the map-table. This register is freed when the instruction commits. Until then, it can be used to squash the instruction and recover the map-table to the state it had before the instruction was renamed. Basic ROB-style recovery is a serial process. However, it can be accelerated by taking periodic map-table checkpoints. Recovering to a checkpointed instruction—typically a low-confidence branch [15]—can be done in one cycle. Recovery to an un-checkpointed instruction is done by recovering to the immediately *younger* checkpoint and back-tracking serially from there. With a ROB, physical registers are freed either in the order in which they were overwritten (on commit) or in the reverse order in which they were allocated (on recovery). This invariant allows the free-list—the structure that tracks unallocated registers—to be implemented as a circular queue.

CPR. CPR operates at the granularity of checkpoints, which are created at rename [1]. Instructions are committed one checkpoint at a time and recovery is permitted only to checkpoints. CPR leverages the recovery restriction to aggressively reclaim physical registers between checkpoints. At any point, CPR holds only those registers that are named in the active map-table or any map-table checkpoint, and those that are read or written by any un-executed in-flight instruction [24].

Physical register reference counting. CPR reclaims registers out-of-order—a register is typically freed when the last instruction that reads it executes, and execution proceeds out-of-order. As a result, CPR cannot track free registers using a queue. Instead, it uses reference counting. Each register is associated with a reference count—a count of zero indicates the register is free.

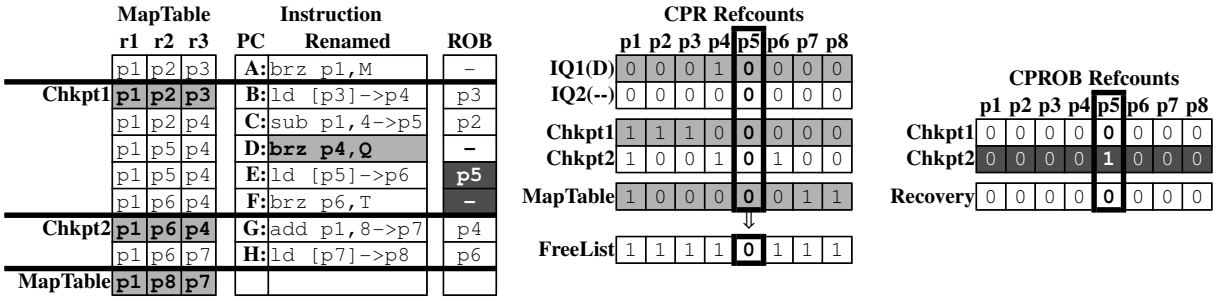
CPR's reference counting algorithm is simple. A register's reference count is incremented when it is allocated to an instruction and written to the map-table, and decremented when it is overwritten and disappears from the map-table. It is incremented when an instruction that reads or writes the register is dispatched to the issue queue and decremented when that instruction executes or is squashed. The reference count of any register that appears in a map-table checkpoint is incremented when that checkpoint is created and decremented when the checkpoint is freed.

A simple and efficient implementation of reference counting uses "unary" matrices [10], [28]. In a reference count matrix, each column represents a physical register and each row represents a resource that can hold a physical register, *e.g.*, an issue queue entry or a checkpoint. A bit in the matrix is 1 if the given resource references the given physical register. A bitvector-style free-list is constructed by ORing together all the bits in a column. Registers are allocated from this free list using encoders.

Figure 3a shows the familiar instruction sequence from Figure 1. Figure 3b shows CPR's reference counting matrices and their contents for this execution snapshot. There are 8 physical registers *p1–p8* so all of the reference counting structures have 8 columns. Map-table reference counts are tracked using a bitvector. The map-table names registers *p1*, *p8*, and *p7* and those bits are set in the map-table bitvector. Checkpoint and issue queue reference counts are tracked using matrices. There is a two entry issue queue and a corresponding two-row issue queue matrix. There is only one un-executed instruction, *D*, and the row corresponding to *D* has a bit set in the column corresponding to *p4*, the only register *D* reads or writes. In the checkpoint reference count matrix, each row has set bits that correspond to the registers in that checkpoint's map-table snapshot. The free-list is formed by ORing the matrices and bitvector column-wise. In the example, only *p5* is free.

The map-table reference count bitvector is updated incrementally each cycle and is implemented using flip-flops. However, the reference count matrices do not support incremental updates. They support only three operations—writing a row, reading a row, and clearing one or more rows—and are implemented as SRAM tables. For instance, when a checkpoint is created, the map-table bitvector is copied into the table. When a checkpoint is restored, the map-table bitvector is restored from the table. When a checkpoint is freed, its row is cleared.

Checkpoint trade-offs. A large number of checkpoints and frequent checkpointing reduces checkpoint overhead. However, several effects favor a moderate number of checkpoints. Obviously, a large number of checkpoints increases the latency, area, and power consumption of the register map-table [29]. But checkpoints also lock down registers and checkpointing too frequently restricts the aggressiveness with which CPR can reclaim inter-checkpoint registers. Releasing checkpoints out-of-order [2] can perhaps mitigate this problem, but this is more difficult to do in CPR than in ROB. In CPR, many structures track instructions by their checkpoint number. Releasing checkpoints out-of-order means that checkpoint numbers may change and that structures that track checkpoint numbers have to efficiently support incremental updates.



(a) Instruction sequence (b) CPR reference counts (c) CPROB reference counts

Figure 3. CPR and CPROB implementation

III. CPROB

CPR tracks and holds physical registers that appear in every map-table checkpoint. CPROB additionally tracks and holds the registers that are over-written by instructions between every un-executed branch and the next younger checkpoint. These "recovery" registers are used to support minimal recovery.

The difference between the "map-table" registers held by CPR and the recovery registers held by CPROB is that the latter are not needed for correctness. They are held on a best-effort basis and can be freed at any time, *e.g.*, when the front end needs physical registers in order to continue dispatching instructions. However, as long as the recovery registers for a given branch are held, CPROB supports minimal recovery to that branch.

A. Mechanism

If we consider a CPR checkpoint to correspond to a map-table snapshot, then by convention, the instructions "in" a checkpoint are those younger than that checkpoint. In Figures 1 and 3, instructions *A–F* are in Checkpoint 1. By the same convention, the recovery registers for a branch in Checkpoint I are actually associated with Checkpoint I+1. In the figures, the recovery register for instruction *E* (*p5*) is associated with Checkpoint 2. The recovery registers for the "open" tail checkpoint are updated incrementally and are tracked using a (flip-flop) bitvector. The oldest checkpoint has no recovery registers associated with it.

CPROB uses a table to checkpoint and restore the recovery bitvector. The recovery reference count bitvector and table parallel the map-table reference count bitvector and table, and are shown in Figure 3c. They are incorporated into the free-list. CPROB also uses a ROB as shown in Figure 3a.

The recovery bitvector tracks the registers over-written by renamed instructions starting at the oldest un-executed branch in the "tail" checkpoint. The reason Figure 3c shows the recovery bitvector as empty is that neither *G* nor *H* are branches. Similarly, Checkpoint 2's row in the recovery reference count table contains only *p5* and not *p2* and *p3*—of these only *p5* is over-written by an instruction younger than un-checkpointed branch *D*. By the same logic, CPROB opportunistically clears the recovery bitvector whenever the number of un-executed branches in the tail checkpoint drops to zero.

Recovery bitvector and table actions parallel those of map-table bitvector and table actions. When checkpoint I is created, the map-table bitvector is copied into row I of the checkpoint reference count table. In parallel, the recovery bitvector is copied into row I of the recovery reference count table. Parallel actions on both tables also take place when checkpoints are restored or freed. Using the checkpoint number to index both the checkpoint and recovery reference count tables requires the recovery reference count table to have C rows even though at most C–1 rows have useful information at any one time—recall, the oldest checkpoint does not have recovery information because there are no older branches associated with it.

Minimal recovery. On an un-checkpointed mis-speculation, CPROB first determines if minimal recovery is possible. Due to the manner in which the recovery registers are managed—they can only be cleared for an entire checkpoint—if an instruction's own recovery register is alive, then all younger instructions in the checkpoint necessarily hold their recovery registers as well. Therefore, this decision can be made by remembering the oldest instruction in each checkpoint which holds a recovery register.

When minimal recovery is possible, recovery to the immediately younger checkpoint is performed as in CPR. From that point, serial recovery is performed as in ROB. As recovery takes place, bits in the restored map-table and recovery bitvectors are updated to reflect

Branch	48 Kbyte 3-table PPM direction predictor [22]. 2K-entry 4-way set-associative target buffer. 16K-entry confidence estimator.
Pipeline	4-way superscalar with 14 stages: 3 fetch, 2 decode, 1 rename, 1 dispatch, 1 issue, 2 regread, 1 execute, 1 complete, 1 commit. 13 cycle minimum branch misprediction penalty.
Execution	128/128 integer/FP physical registers, 32/32 integer/FP issue queue entries, 4-way issue with up to 4 integer, 2 FP, 2 loads, 1 store, and 1 branch per cycle. 3 Kbyte, distance-based memory-dependence predictor.
Window	8 checkpoints. 1024-entry fully-associative load queue. 512-entry fully associative store queue. 256-entry ROB.
Memory	32 Kbyte, 8-way set-associative, 64 byte line, 3-cycle instruction and data caches with 8-entry victim buffers. 1 Mbyte, 16-way set-associative, 64-byte line, 20-cycle L2 with 8-entry victim buffer, and 8 16-entry stream buffers. 400 cycle memory latency. 4 byte/cycle memory bus. 32 outstanding misses.

Table I
SIMULATED PROCESSOR CONFIGURATIONS

changes to the reference counts.

The ROB. The above description may make it sound as if CPROB needs a large ROB capable of holding all in-flight instructions. The ROB is a simple SRAM and is not on any critical execution paths so a large ROB may not be a concern. However, CPROB does not actually require a large ROB. CPROB only uses the ROB for best-effort minimal recovery, not for instruction commit which is handled in CPR fashion. Because instructions are only minimally-recoverable as long as their over-written registers are held, they no longer need ROB entries after these registers are released. This means that when the oldest checkpoint releases its recovery registers, its instructions can also be removed from the ROB in bulk.

B. Policy

CPROB's recovery registers are held only for performance and may be released at any time. This degree of freedom requires policies which specify: i) when checkpoints should spontaneously release their recovery registers, and ii) if registers are needed to avoid dispatch stalls, which checkpoint's recovery registers should be victimized.

Spontaneous policy: release once all branches execute. Intuitively, a checkpoint should spontaneously release its recovery registers when all branches in it execute. At that time, the chances of an un-checkpointed recovery in that checkpoint are greatly reduced. An un-checkpointed recovery can still be triggered by a load/store ordering violation, or by an exception—both of these are much rarer than branch mis-predictions.

However, the spontaneous policy alone is insufficient for adequate performance. A branch which depends on a long latency operation may not execute for many cycles. Waiting for such a branch to execute and release the checkpoint's recovery registers could stall rename for a significant period on the small chance that of a mis-prediction on a branch that was sufficiently confident to not be assigned a checkpoint.

Victimization policy: release oldest "locked" checkpoint. To avoid stalling rename, it may be necessary to victimize a checkpoint and force it to release its recovery registers. Empirically, there is little variation between "reasonable" policies such as victimize oldest checkpoint, or victimize checkpoint with fewest un-executed branches. However, victimizing the oldest checkpoint is preferred because it meshes nicely with ROB management. When the oldest checkpoint is victimized for registers, its corresponding ROB entries are freed as well. Conversely, if ROB space is needed, the instructions at the head of the ROB can be victimized, along with the recovery registers of the corresponding checkpoint. Being able to victimize old ROB entries allows CPROB to exploit a much smaller ROB than a "ROB-only" processor could.

IV. EXPERIMENTAL EVALUATION

A. Simulation Environment

We evaluate CPROB on the SPEC2000 benchmark suite using cycle-level simulation. The benchmarks are compiled for the ALPHA AXP ISA with optimization level –O4. All benchmarks run to completion using their training inputs. We use 2% periodic sampling with 2 million instructions per sample. Each sampling period is preceded by a 2 million instruction warmup period. The simulator models a 4-way issue dynamically scheduled superscalar processor with a 17-stage pipeline, 256-entry instruction window and 64-entry issue queue. Table I shows our configuration in more detail. All configurations use large fully associative load and store queues, isolating the effects of the register management schemes, which we focus on. Previous work has proposed realistic load and store queues for CPR [14].

B. Comparative Performance

Figure 4 shows checkpoint overhead for CPR and CPROB (top), and their performance relative to ROB (bottom). The number printed above each benchmark is its baseline ROB IPC. All architectures have 8 checkpoints and 256 physical registers. CPROB and ROB use 256-entry ROBs.

CPROB dramatically reduces CPR's checkpoint overhead, effectively eliminating it on SpecFP and reducing

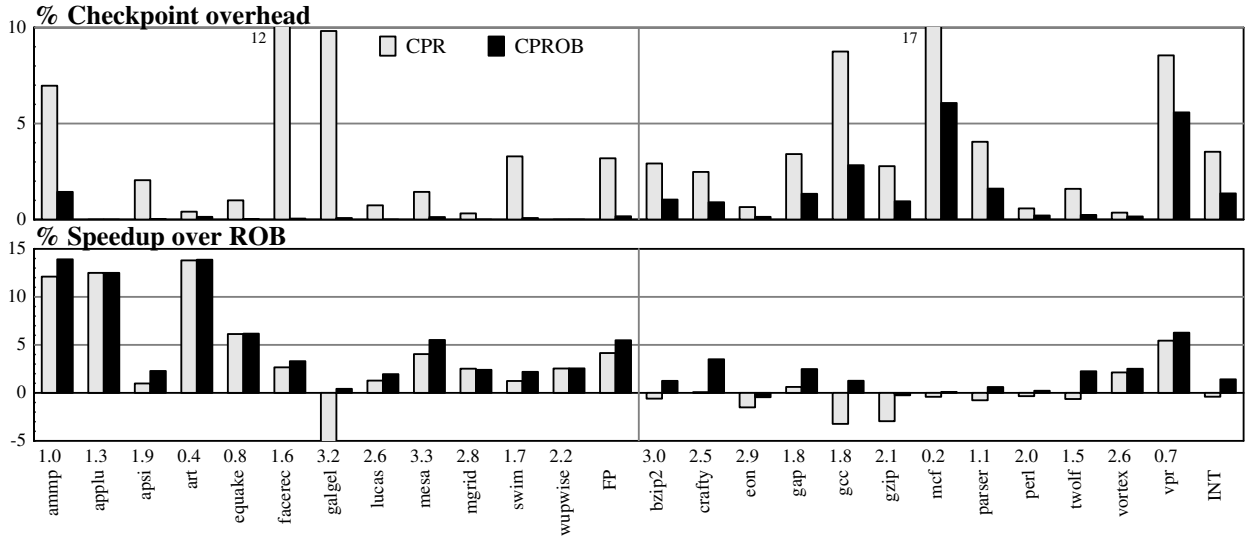


Figure 4. CPR and CPROB: Checkpoint overhead and speedup over ROB

it by 60% on SpecINT. The few cases of non-trivial checkpoint overhead in CPROB stem from load miss-dependent mis-predicted branches. Because of their long resolution latencies, their enclosing checkpoints are often victimized before the mis-prediction is detected.

In most cases, reduction in checkpoint overhead results in substantial performance improvement. However, this is not always the case. In *facerec*—where checkpoint overhead decreases from 12% to almost 0% but performance hardly changes—CPROB unmasks store queue capacity stalls. In *mcf*, L2 misses are prevalent and issue queue stalls dominate with or without checkpoint overhead.

The most important observation from Figure 4 is that—with a few exceptions of less than 0.5%—**CPROB always outperforms both CPR and ROB.** More specifically, **CPROB eliminates CPR's instances of pathological slowdown relative to ROB.** CPROB achieves this by mimicking the architecture best suited to the situation the program is experiencing at the moment. It does this naturally in response to correlated changes in register demand and recent branch behavior. High register pressure implies a full window which in turn implies that no mis-predictions have been recently uncovered. CPROB responds to this situation by victimizing old checkpoints and behaving like CPR. When squashes due to branch mis-predictions clear the window and free registers, CPROB does not need to victimize old checkpoints and acts like ROB for recovery purposes. But even here CPROB outperforms ROB because it does not need to hold *all* over-written registers, only registers over-written by instructions immediately younger than un-checkpointed branches.

C. Area and Energy

Although we do not engage in detailed energy modeling, we argue that CPROB not only outperforms CPR, but also reduces its energy consuption and ED^2. In fact, unlike CPR, CPROB has lower ED^2 than ROB. A lower ED^2 implies that CPROB is more energy-efficient than ROB in a voltage-independent way, *i.e.*, for the same energy it would yield higher performance than applying DVFS (dynamic voltage frequency scaling) to ROB [19]. Our argument has two parts: one considers the overhead of new structures, the other considers dynamic instruction counts.

We use a modified version of CACTI-4 [31] to model the areas of the register management structures for ROB, CPR, and CPROB. The ROB structures occupy a total of 0.100 mm^2 divided into a 64-entry map-table with 12 read ports, 4 write ports and 4 checkpoints (0.094), a 256-entry, 4-bank ROB with 1 read port and 1 write port (0.003) and a 196-entry, 4-bank free list with 1 read port and 1 write port (0.003). CPR's structures occupy 0.159 mm^2—a 64-entry map table with 8 read ports, 4 write ports and 8 checkpoints (0.070), two 128-column, 32-row issue queue reference count matrices with 4 write ports (0.070), and a 256-column, 8-row checkpoint reference count matrix with 1 read-write port (0.019). Notice, CPR has fewer map-table read ports than ROB because it does not need to track over-written registers. However, it uses more checkpoints than ROB (see Section IV-D). CPROB *does* need to track over-written registers, but it requires fewer checkpoints and

so its structures occupy 0.186 mm^2 divided into ROB-style map table and ROB (0.097 combined) and CPR-style reference count tables (0.089 combined). CPROB's area overhead over CPR is 0.033 mm^2; its overhead over ROB is 0.086 mm^2. These are small relative to the area of a typical out-of-order core in 45nm technology—*e.g.*, Intel's Core2 is approximately 25 mm^2. Even if we assume that these new structures consume energy at a rate that is 10 times higher than that of the average structure, their energy consumption would constitute approximately 1% of total core energy.

The second part of our argument uses IPCs and dynamic instruction counts. For SPECfp, CPR outperforms ROB by an average of 4% whereas CPROB outperforms it by an average of 6%. At the same time, our simulations show that CPR dispatches 3.2% more instructions than ROB—presumably due to checkpoint overhead—while CPROB dispatches only 0.4% instructions more than ROB. If we roughly equate dynamic instructions dispatched with dynamic energy consumption and add a 1% sur-charge for CPR and CPROB to account for the new structures, we find that CPR reduces ED^2 by 3.8% over ROB while CPROB reduces it by 10.5%. We repeat the calculation for SPECint. Here, CPR under-performs ROB by 0.5% *and* dispatches 6% more instructions resulting in an ED^2 *increase* of 8.1%. CPROB out-performs ROB by 1.5% while dispatching only 1.6% more instructions. For SPECint, CPROB's ED^2 is 0.4% lower than ROB. In other words, CPR is on average somewhat faster than ROB but it is also less energy efficient. CPROB not only improves CPR's performance and corrects its performance pathologies, it also improves its energy efficiency to the point where it is more energy efficient than ROB.

D. Sensitivity Analysis

To further analyze the performance characteristics of CPROB, Figure 5 shows average relative performance across both benchmark suites for ROB, CPR, and CPROB when varying three micro-architectural parameters: number of checkpoints, ROB size, and number of physical registers. In each graph, one parameter is varied, holding the other two constant. The exception is the physical register experiments which scale the ROB size to match the register file. All results are normalized to the main ROB configuration with 256 registers, 256 ROB entries, and 8 checkpoints.

Checkpoints. The left graph varies the number of map-table checkpoints from 2 to 16. While ROB is almost completely insensitive to checkpoint count, CPR requires at least 4 to match ROB performance, and 8 to show noticeable benefits. CPROB inherits checkpoint-insensitivity from ROB—it is capable of minimal recovery to most mis-speculations and suffers minimal checkpoint overhead even with only two checkpoints.

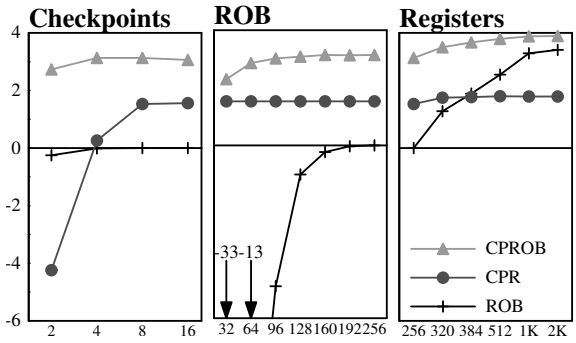


Figure 5. CPROB Sensitivity: Checkpoints, ROB, and registers.

ROB size. The middle graph varies ROB size from 32 to 256 entries. CPR does not use a ROB; its performance is shown strictly for comparison. The important observation here is that CPROB is much less sensitive to ROB size than a traditional ROB-only architecture. In a ROB-only architecture, ROB size limits window size. In CPROB, it limits the ability to perform minimal recovery.

Number of registers. The right graph varies the number of physical registers. This graph shows three important trends. First, CPR outperforms ROB when registers are relatively scarce and aggressive reclamation is important. In this scenario, CPROB victimizes frequently and its performance tracks CPR. Second, when registers are plentiful, ROB outperforms CPR—aggressive reclamation is less important and checkpoint overhead is exposed. Here, CPROB does not victimize and tracks ROB. Third, CPROB outperforms both CPR and ROB at points in between—each benchmark has phases in which it is constrained by registers and phases in which branch mis-predictions dominate. CPROB naturally adapts to the current behavior.

E. CFPROB: Continual Flow Pipelines on CPROB

The final graph of Figure 5 shows that CPR is insensitive to the number of physical registers. While CPR allows the register file to scale to accommodate large window sizes, it becomes constrained by the issue queue when the window needs to scale most—in the presence of last-level cache misses. One technique for allowing the window to scale under long-latency load misses is CFP (Continual Flow Pipelines) [30]. CFP builds on CPR and exploits its execution-based register-reclamation. As such, it could be built on CPROB just as easily. We call this configuration CFPROB.

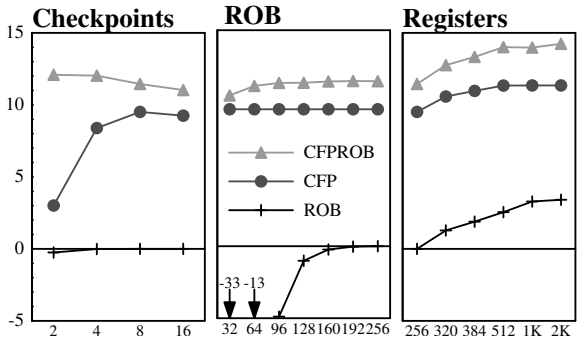


Figure 6. CFPROB sensitivity: checkpoints, ROB, and registers.

In the interest of space, we do not show per-benchmark CFP and CFPROB results. However, our experiments show that—as with CPR and CPROB—CFPROB always matches, and often outperforms, CFP. Figure 6 repeats the CPROB sensitivity analysis in the CFP context. When we vary the number of checkpoints, CFPROB exhibits a slightly unexpected pattern—its performance decreases with more checkpoints. Checkpointing too frequently can be detrimental to performance as checkpoints hold registers until they commit. Due to long latency last-level cache misses and in-order checkpoint commit, superfluous checkpoints can hold registers for hundreds of cycles. While some benchmarks (*gzip*, *vpr*) benefit from the additional protection from mis-speculations, others exhibit increased physical register stalls (most notably, *equake*). This trend begins to appear in CFP (built on CPR) at 16 checkpoints. It appears sooner in CFPROB.

Although CFP achieves very large window sizes, CFPROB is insensitive to ROB size. The key to this behavior is that miss-dependent branch mis-predictions are quite rare [30], and most branches resolve quickly. By supporting minimal recovery for a moderate number of the youngest branches, CFPROB reduces checkpoint overhead greatly even with a ROB much smaller than the effective window size.

The right graph in Figure 6 shows the effects of increasing register file size. CFP significantly outperforms ROB at all points on this graph due to its issue queue scaling benefits and last-level cache miss tolerance. However, with the penalty of last-level misses reduced, checkpoint overhead is exposed, and so CFPROB provides an additional 2–3% performance benefit.

V. CPR AND CPROB ON SMT

In addition to single-thread performance, contemporary processors target throughput via SMT (Simultaneous Multi-Threading) [33]. One of the critical resources in SMT is the physical register file, which must accommodate the architectural register state of multiple threads and still have enough registers to support a reasonable renaming window [17]. The criticality of the register file suggests that a register-efficient substrate like CPR may be a good match for SMT.

Intuition suggests that SMT and CPR may be synergistic in another way as well. Specifically, SMT reduces the impact of branch mis-predictions by making each thread speculate more slowly [18]. The temptation is to think that because CPR is more vulnerable to branch mis-predictions than ROB—specifically to unchecked branch mis-predictions—that SMT would reduce checkpoint overhead as well. However, intuition is mis-leading in this case. SMT only reduces the cost of squashing instructions that are *younger* than the mis-prediction—fewer younger-than-mis-prediction wrong-path instructions are fetched before the mis-prediction is uncovered as fetch bandwidth is divided between threads. It has no direct effect on the number of instructions *older* than a mis-prediction that are squashed, *i.e.*, checkpoint overhead. Quite the opposite, by reducing the mis-prediction penalty, SMT amplifies the checkpoint overhead penalty by comparison. Not only does the thread hurt its own performance by re-fetching and re-executing older-than-mis-prediction instructions, it hurts the other thread as well. SMT further exacerbates CPR's checkpoint overhead problem by forcing threads to share a limited number of checkpoints.

Of course, SMT can use CPROB as a substrate as easily as it can use CPR. Extending CPR and CPROB's reference counting mechanisms to SMT is trivial. All that is needed is to replicate the map-table and recovery reference count bitvectors on a per-thread basis.

Experiments. Our SMT experiments use the basic single-threaded configuration from Table I: 8 checkpoints, 256 ROB entries, 256 total physical registers. The fetch policy is ICOUNT [32] augmented with the constraint that no thread may occupy more than three-quarters of the issue queue [26]. We similarly "cap" the checkpoints, disallowing any thread from occupying more than 6 out of the 8. The ROB and load/store queues are statically partitioned [16]. CPROB attempts to victimize the currently renaming thread first. If that thread is fully victimized, it victimizes the other thread.

Figure 7 shows the speedups of a CPR-based and CPROB-based SMT processor over a ROB-based SMT processor for various two-thread workloads. The plot shows "speedup" of one SMT machine (CPR or CPROB) over another (ROB), not "SMT-speedup" of concurrent execution over single-program execution. We

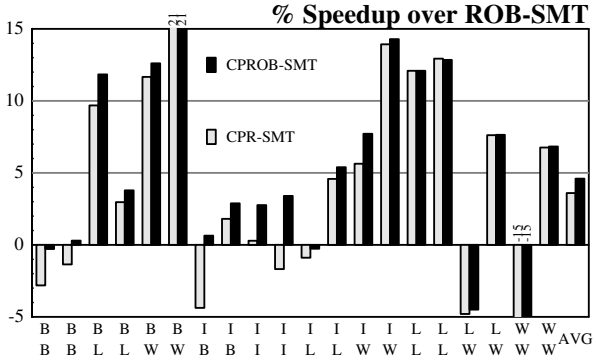


Figure 7. Two-thread experiments: speedup over ROB-SMT

construct workloads by first categorizing benchmarks as either memory-bandwidth bound (W), memory-latency bound (L), branch-misprediction bound (B), or high-ILP (I). From each category, we select 4 benchmarks and then create random pairings such that each combination of two categories (*i.e.* BB, BW, IL, etc.) has four pairings. Our multi-program workload methodology is FIESTA [12]. From each program we choose 50 samples, each of which runs for 5 million cycles when executed standalone. A multi-program run executes the samples from the different programs pair-wise. The same samples are used in all experiments. The figure shows one pairing from each category, as well as the average over all pairings.

The difference between CPR and CPROB is most pronounced in the BB and IB pairings. In some of these, CPR underperforms ROB. However, CPROB corrects this. For BL and BW pairings, CPR's register reclamation benefits dominate, causing CPR and CPROB to perform similarly. In the LW and WW pairings—both CPR and CPROB can underperform ROB by a significant amount. These slowdowns are due to increased thrashing in the data cache which is enabled by CPR's larger window—they are unrelated to checkpoint overhead.

VI. Related Work

To our knowledge, CPROB is the first work that directly attacks CPR's checkpoint overhead problem. CPROB is applicable to microarchitectures based on CPR like CFP (Continual Flow Pipelines) [30] and TCI (Transparent Control Independence) [4].

CPROB is a hybrid best-effort register reclamation scheme that targets CPR processors that use physical register reference counting [10], [28]. CPROB is unrelated to mechanisms that aggressively reclaim registers in ROB-based microarchitectures, including Early Register Release [8] and Cherry [21]. CPROB does not target ROB-based scalable microarchitectures like KILO processor and its successors [7], [25].

CPROB targets checkpoint overhead. It is orthogonal to microarchitectures that exploit control independence to reduce the branch mis-prediction penalty [4], [5], [6], [9], [13], [27]. It is potentially synergistic with TCI [4] as both use a CPR substrate.

CPROB uses serial recovery to recover to un-checkpointed instructions. As such, it is potentially synergistic with TurboROB [3], a mechanism that accelerates serial branch recovery. TurboROB also reduces the number of registers that must be held to support minimal recovery. "TurboCPROB" would victimize fewer checkpoints with un-executed branches.

VII. Conclusions

CPR trades efficient register management in the common case for checkpoint overhead—squashing of instructions older than un-checkpointed mis-speculations.

CPROB adaptively combines CPR's aggressive register reclamation with best-effort ROB-style minimal recovery. CPROB outperforms both ROB and CPR in a variety of micro-architectural configurations—typically outperforming both architectures on any given benchmark. It does so by adapting to dynamic register requirements and recent branch behavior. When registers are scarce, CPROB gracefully degrades to CPR. When registers are plentiful, CPROB holds on to registers to support minimal recovery.

Relative to a contemporary high-performance out-of-order ROB-style core, CPROB's hardware overhead—reference counting bitvectors and matrices—is negligible, likely increasing area by less than 0.1% and power consumption by less than 1%. In exchange for these overheads, CPROB provides improved performance with minimal superfluous instruction dispatch due to checkpoint overhead, yielding a design with lower ED^2.

Finally, CPROB's combination of register efficiency, checkpoint efficiency, and low checkpoint overhead, makes it an interesting and promising substrate for SMT.

VIII. Acknowledgments

We thank the reviewers for their comments. This work was supported by NSF award CCF-0541292 and by a grant from the Intel Research Council. Andrew Hilton was partially supported by a fellowship from the University of Pennsylvania Center for Teaching and Learning.

REFERENCES

[1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. 36th Intl. Symp. on Microarchitecture*, pages 423–434, Dec. 2003.

[2] P. Akl and A. Moshovos. Branch Tap: Improving Performance with Very Few Checkpoints Through Adaptive Speculation Control. In *Proc. 20th Intl. Conf. on Supercomputing*, pages 36–45, Jun. 2006.

[3] P. Akl and A. Moshovos. Turbo-ROB: A Low-Cost Simple Checkpoint/Restore Accelerator. In *Proc. 2008 Intl. Conf. on High Performance Embedded Architectures and Compilers*, Jan. 2008.

[4] A. Al-Zawawi, V. Reddy, E. Rotenberg, and H. Akkary. Transparent Control Independence (TCI). In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 448–459, Jun. 2007.

[5] C. Cher and T. Vijaykumar. Skipper: A Microarchitecture for Exploiting Control-Flow Independence. In *Proc. 34th Intl. Symp. on Microarchitecture*, pages 4–15, Dec. 2001.

[6] Y. Chou, J. Fung, and J. Shen. Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection. In *Proc. 1999 Intl. Conf. on Supercomputing*, pages 109–118, Jun. 1999.

[7] A. Cristal, O. Santana, M. Valero, and J. Martinez. Toward KILO-Instruction Processors. *ACM Trans. on Architecture and Code Optimization*, 1(4):389–417, Dec. 2004.

[8] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing Processor Performance Through Early Register Release. In *Proc. 22nd IEEE Intl. Conf. on Computer Design*, pages 480–487, Oct. 2004.

[9] A. Gandhi, H. Akkary, and S. Srinivasan. Reducing Branch Misprediction Penalty via Selective Branch Recovery. In *Proc. 10th Intl. Symp. on High Performance Computer Architecture*, pages 254–265, Feb. 2004.

[10] A. Golander and S. Weiss. Hiding the Misprediction Penalty of a Resource-Efficient High-Performance Processor. *ACM Transactions on Architecture and Code Optimization*, 4(4), Jan. 2008.

[11] I. Gonzalez, M. Galluzzi, A. Veindenbaum, M. Ramirez, A. Cristal, and M. Valero. A Distributed Processor State Management Architecture for Large-Window Processors. In *Proc. 41st Intl. Symp on Microarchitecture*, Nov. 2008.

[12] A. Hilton, N. Eswaran, and A. Roth. FIESTA: A Sampled-Balanced Multi-Program Workload Methodology. In *Proc. 5th Workshop on Modeling, Benchmarking and Simulation*, pages 43–52, Jun. 2009.

[13] A. Hilton and A. Roth. Ginger: Control Independence Using Tag Rewriting. In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 436–447, Jun. 2007.

[14] A. Hilton and A. Roth. Decoupled Store Completion and Silent Deterministic Replay: Enabling Scalable Memory Systems for CPR/CFP Processors. In *Proc. 36th Intl. Symp. on Computer Architecture*, pages 245–254, Jun. 2009.

[15] E. Jacobson, E. Rotenberg, and J. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proc. 29th Intl. Symp. on Microarchitecture*, pages 142–152, Dec. 1996.

[16] D. Koufaty and D. Marr. HyperThreading Technology in the NetBurst Microarchitecture. *IEEE MICRO*, 23(2), Mar. 2003.

[17] J. Lo, S. Parekh, S. Eggers, H. Levy, and D. Tullsen. Software Directed Register Deallocation for Simultaneous Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):922–933, 1999.

[18] K. Luo, M. Franklin, S. Mukherjee, and A. Seznec. Boosting SMT Performance by Speculation Control. In *Proc. 15th Intl. Parallel and Distributed Processing Symp.*, Apr. 2001.

[19] A. Martin, M. Nystroem, and P. Penzes. ET2: A Metric for Time and Energy Efficiency of Computation. Technical Report CSTR:2001.007, CalTech, 2001.

[20] J. Martinez, A. Cristal, M. Valero, and J. Llosa. Ephemeral Registers. Technical Report CSL-TR-2003-1035, Computer Systems Lab, Cornell University, Nov. 2003.

[21] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *Proc. 35th Intl. Symp. on Microarchitecture*, Nov. 2002.

[22] P. Michaud. A PPM-like, Tag-Based Branch Predictor. *Journal of Instruction Level Parallelism*, 7(1):1–10, Apr. 2005.

[23] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals. Delaying Physical Register Allocation Through Virtual-Physical Registers. In *Proc. 32nd Intl. Symp. on Microarchitecture*, pages 186–192, Nov. 1999.

[24] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: An Alternative Approach. In *Proc. 26th Intl. Symp. on Microarchitecture*, pages 202–213, Dec. 1993.

[25] M. Pericas, R. Gonzalez, D. Jimenez, and M. Valero. A Decoupled KILO-Instruction Processor. In *Proc. 12th Intl. Symp. on High Performance Computer Architecture*, pages 53–64, Feb. 2006.

[26] S. Raasch and S. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proc. 2003 Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 15–26, Sep. 2003.

[27] E. Rotenberg, Q. Jacobsen, and J. Smith. A Study of Control Independence in Superscalar Processors. In *Proc. 5th Intl. Symp. on High Performance Computer Architecture*, pages 115–124, Jan. 1999.

[28] A. Roth. Physical Register Reference Counting. *Computer Architecture Letters*, 7(1), Jan. 2008.

[29] E. Safi, P. Akl, A. Moshovos, A. Veneris, and A. Arapoyianni. On The Latency, Energy, and Area of Checkpointed, Superscalar Register Alias Tables. In *Proc. 2007 Intl. Symp. on Low-Power Electronics and Design*, Aug. 2007.

[30] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Oct. 2004.

[31] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett-Packard Labs Technical Report, Jun. 2006.

[32] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. 23rd Intl. Symp. on Computer Architecture*, pages 191–202, May 1996.

[33] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 392–403, Jun. 1995.