

The 38th Annual IEEE/ACM International Symposium on Microarchitecture (*MICRO 2005*)

# Incremental Commit Groups for Non-Atomic Trace Processing

**Matt T. Yourst**

**Kanad Ghose**

Department of Computer Science  
State University of New York at Binghamton

{yourst, ghose}@cs.binghamton.edu

Monday, November 14, 2005

# Binary Translation (BT)

- *Source* ISA **translated** to different *native* ISA executed by hardware
- **Motivation:** simpler, faster and lower power processors versus superscalars
- In our study, native ISA uses statically scheduled **VLIW micro-ops (uops)**
- **Trace:** sequence of uops along predicted control flow path, all scheduled as a single unit
- Larger traces formed by **merging several smaller traces** (or basic blocks)
- BT software **profiles code** as it executes to **identify** and **optimize** traces
- Optimized traces stored in **translation cache**
- BT overhead **amortized** by optimizing only **frequently used** code

# Binary Translation: Examples

- **Transmeta Crusoe and Efficeon:** *all* x86 code (OS, applications) to VLIW by **code morphing software**
- **IBM Daisy and BOA:** like Transmeta, but with PowerPC to VLIW
- **Intel IA32-EL:** x86 to IA-64 VLIW (dynamic, user code only)
- **Intel Pentium 4:** x86 to RISC-like uops by hardware at trace cache fill time (minimal optimization)

Our work focuses on **full system x86-64 to VLIW translation**

# Atomic Commit/Rollback

Modern BT systems use atomic **commit/rollback**:

- Architectural state (registers and memory) **checkpointed** at start of trace
- State is **speculatively updated** by uops within trace
- If **all uops** in trace execute correctly, speculative state **commits** (overwrites last known good architectural state)
- Any exceptions cause **rollback** (restores checkpointed state)
  - **Off-trace branches**, memory ordering and speculation failures, violations of scheduling assumptions during execution
- **Required for good performance:**
  - Allows aggressive or dangerous memory reordering
  - x86 instruction semantics: must be atomic

# Motivation: Problems with Atomic Traces

## Useful computations wasted:

- Inefficient design: rollbacks occur even if only one uop fails

## Trace length limited to limit rollback penalty

- Decreases potential ILP and optimizations

## Slow Adaptation and High Overhead:

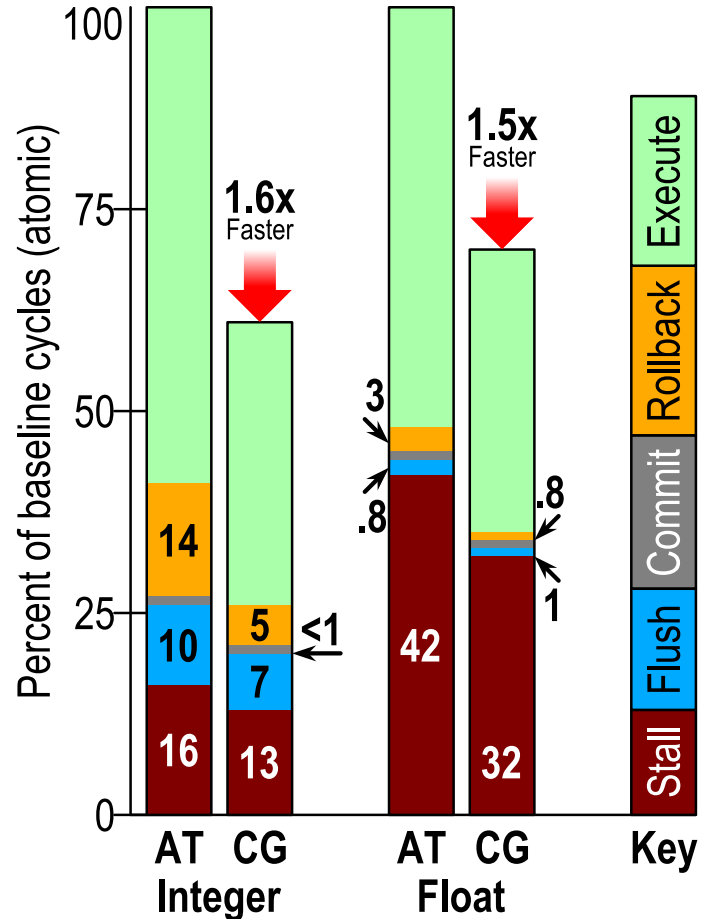
- BT system must be **very conservative** to ensure scheduling effort is not wasted
- No way to **salvage information about correctly executed uops** to guide future scheduling decisions

# Introducing Incremental Commit Groups

- Very long high ILP traces constructed by **salvaging and committing the correctly executed parts** of traces on rollbacks
- **Concept**: Divide scheduled VLIW trace into multiple ***commit groups***, separated by **commit points**
- **Rollbacks** only need to restore last commit point: **salvage all previous work**
- **Schedule uops freely across commit groups**: commit points inserted *after* scheduling, so **no ILP impact**
- **Splice working parts of traces** together into new optimized traces: **self-evolving optimizations**

# Performance Preview

- Significant Speedup vs Atomic:  $1.5\times$  faster on average
- Cuts time wasted by roll-backs by 66% (integer), 60% (FP)
- Longer traces promote better load hoisting: less stall time

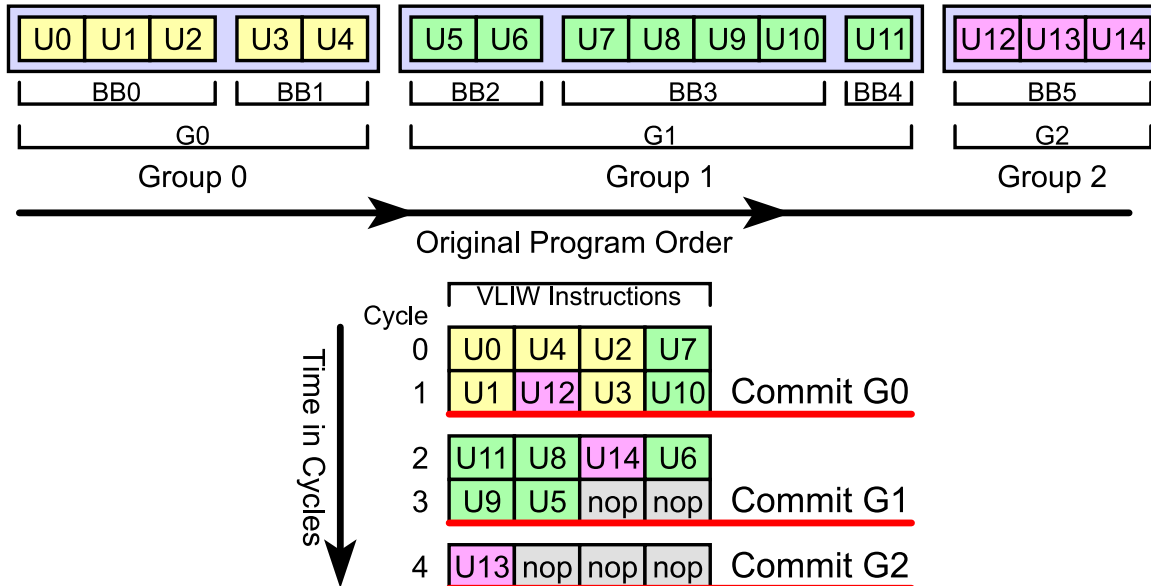


# Peptidal PT2x

- Our **VLIW** microprocessor design
- 8-way clustered core with special hardware support described later
- Translates **x86-64** code (OS + applications) into **VLIW uops** (micro-operations)
- Translation and optimization done by **PTL** (Peptidal Translation Layer), our “code morphing” software
- Specialized hardware for improved performance, as in other BT systems

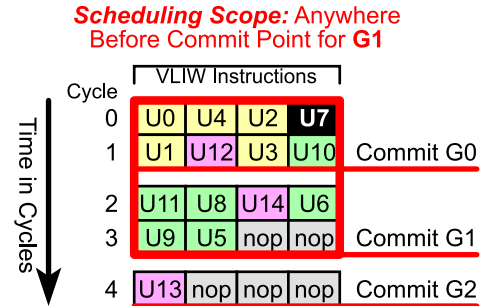
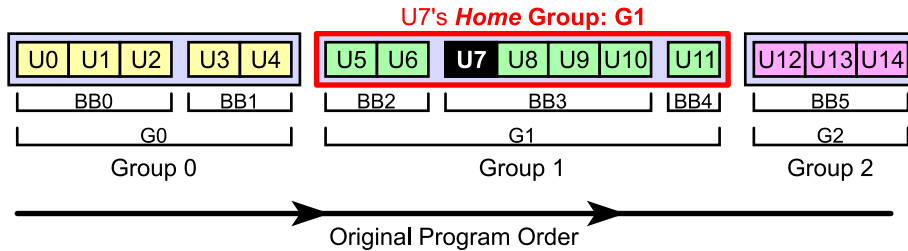
# Commit Groups: Definitions

- **Commit Group:** sequence of basic blocks in which all constituent uops atomically commit



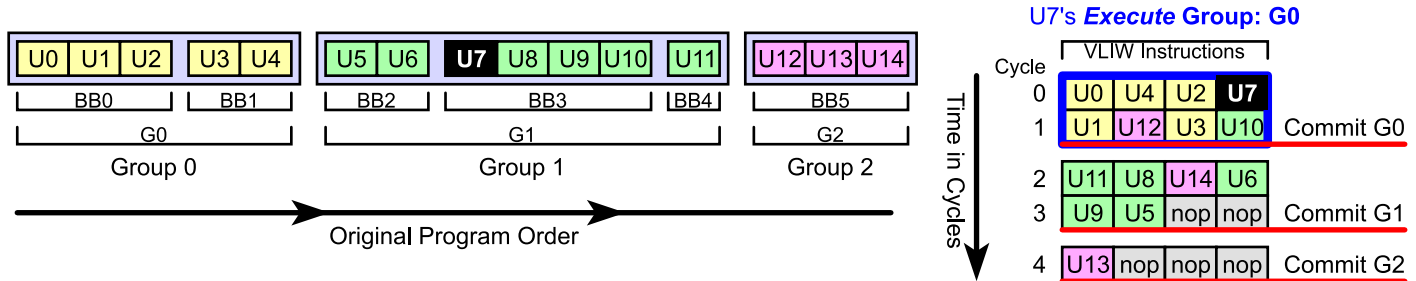
# Home Commit Group

- **Home Commit Group:** the commit group to which a given uop originally belonged in program order
- **Commit Point** for commit group  $G$ : final cycle in which any uop within  $G$  appears in the VLIW schedule
- Uops **freely scheduled anywhere before their home group's commit point** to maximize ILP



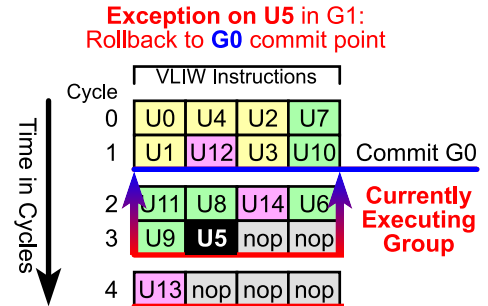
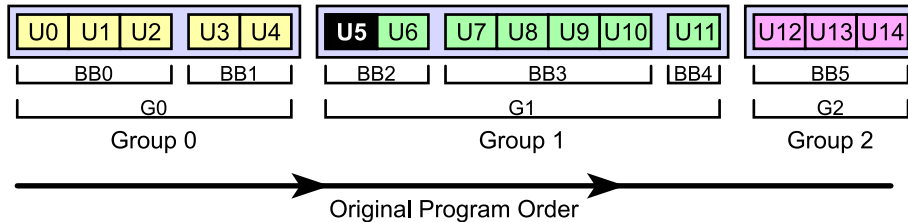
# Execute Commit Group

- **Execute Commit Group:** the commit group in which a given uop is actually scheduled
- Commit groups **always commit in original program order**, even if uops execute out of order



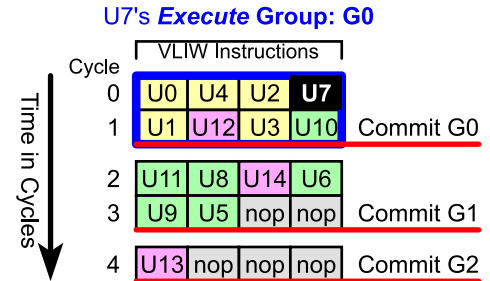
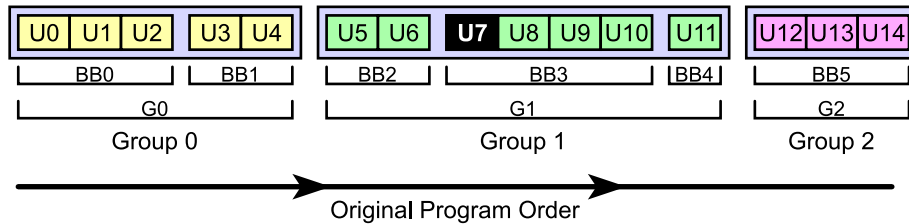
# Exception Handling and Rollbacks

- **Currently Executing Commit Group:** The execute group the processor is currently executing
- If any uop with its **home group = currently executing group** causes an exception, rollback to **previous group's commit point**
- If any uop from **future group** causes exception, **defer exception** until uop's home group is reached



# Commit Buffer

- Since uops from different groups are intermixed in schedule, results must be **separated by group**
- Commitment of results in future groups **deferred** until home group matches **currently executing group**
- *Solution?* **Commit Buffer** hardware structure



# Commit Buffer

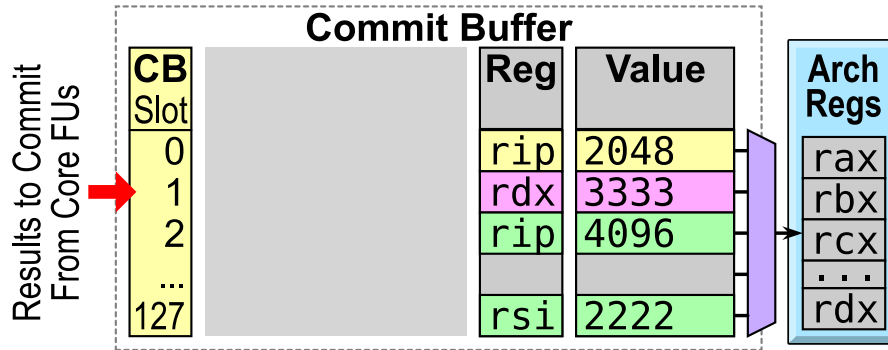
**Commit Buffer (CB)** defers updates to architectural state until appropriate commit point is reached

- Divided into **slots**; one state update (**register** or **store**) per slot
- Slot only allocated to **final uop in program order writing each unique destination** within uop's home group
- Picks **ready slots** belonging to current group and lets them **fall through** into architectural state
- **Cross commit point** only if all slots assigned to current group are **written** to architectural state and **exception free**

# Commitment Process

Every cycle, compute  $(M[G] \text{ AND } C \text{ AND } R)$  to find which slots *in the currently executing group only* are ready and still uncommitted.

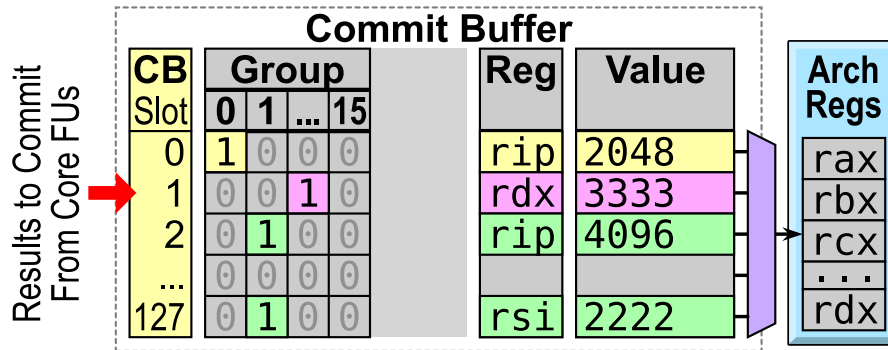
Select first match and send (register, data) pairs to speculative architectural register file. Clear slot's C bit.



# Commitment Process

Every cycle, compute  $(M[G] \text{ AND } C \text{ AND } R)$  to find which slots *in the currently executing group only* are ready and still uncommitted.

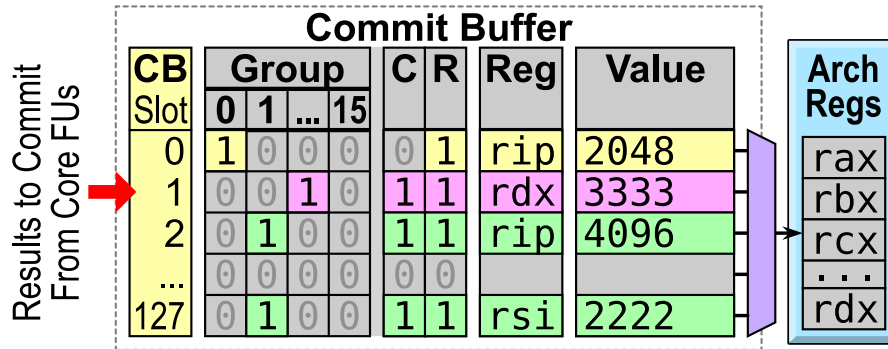
Select first match and send (register, data) pairs to speculative architectural register file. Clear slot's C bit.



# Commitment Process

Every cycle, compute  $(M[G] \text{ AND } C \text{ AND } R)$  to find which slots *in the currently executing group only* are ready and still uncommitted.

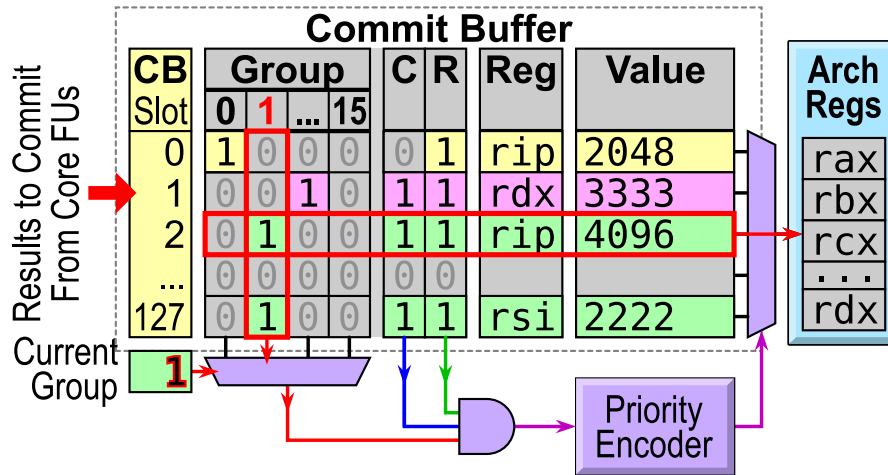
Select first match and send (register, data) pairs to speculative architectural register file. Clear slot's C bit.



# Commitment Process

Every cycle, compute  $(M[G] \text{ AND } C \text{ AND } R)$  to find which slots *in the currently executing group only* are ready and still uncommitted.

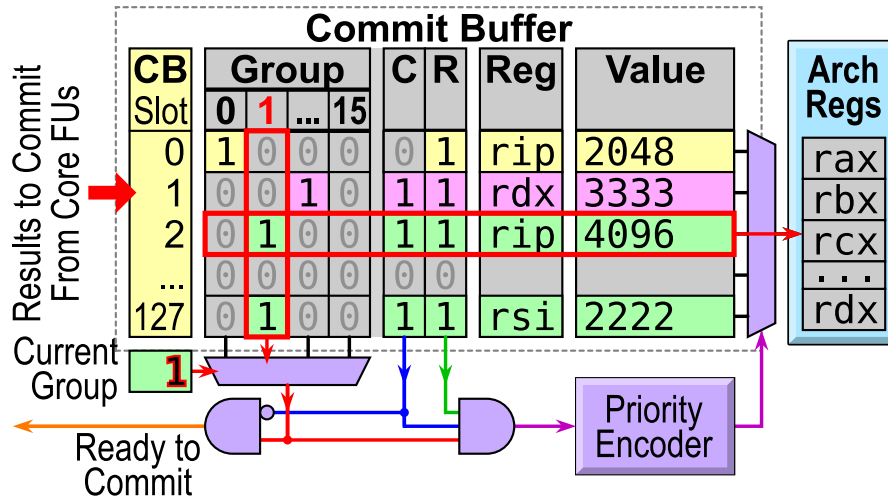
Select first match and send (register, data) pairs to speculative architectural register file. Clear slot's C bit.



# Commit Group Completion Check

At bundle specifying commit point for group  $G$ , compute  $(M[G] \text{ AND } C)$  to find which slots are still uncommitted

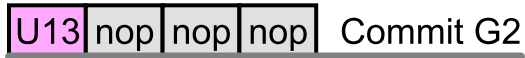
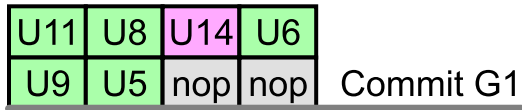
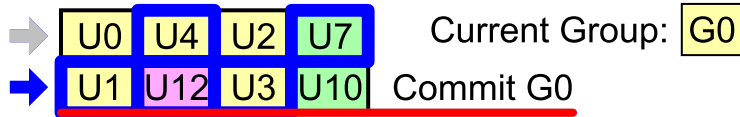
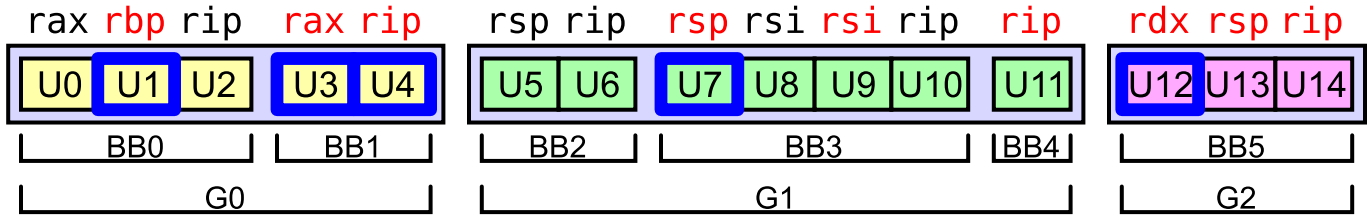
If uncommitted slots exist, **stall pipeline** (except commit unit!) until no matches remain



# Commit Buffer Hardware Complexity

- **Not on critical path:** data flow into CB is one way (processor to CB)
- **No associative search** hardware: direct access
- Easy to simplify hardware via **smart scheduling: banked design, few ports**
- Architectural RF has **zero read ports:** entire ARF latched back into physical RF for next trace

Last writer of each arch reg with each group is shown in **red**



rax	1111	U3	rsi	-	-
rbx	-	-	rdi	-	-
rcx	-	-	rbp	1234	U1
rdx	-	-	rsp	-	-
rip	2048	U4			

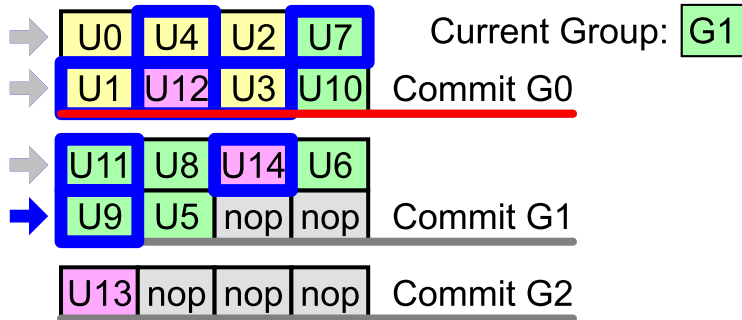
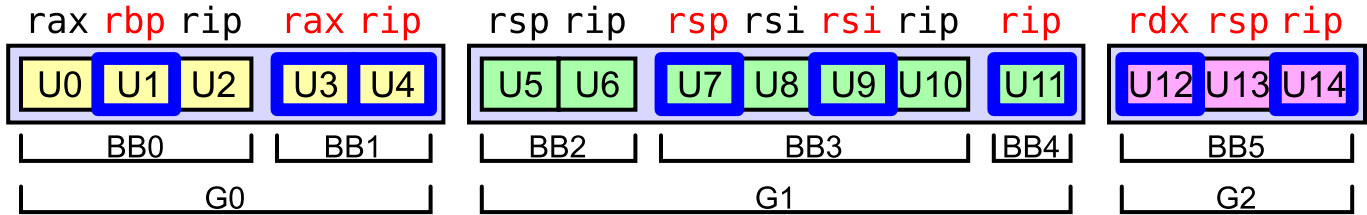
CB	Group			C	R	Reg	Value	uop
	0	1	2					
0	1	0	0	0	1	rip	2048	U4
1	0	1	0	1	1	rsp	1000	U7
2	1	0	0	0	1	rbp	1234	U1
3	0	0	1	1	1	rdx	3333	U12
4	1	0	0	0	1	rax	1111	U3
5	0	0	0	0	0			
6	0	0	0	0	0			
7	0	0	0	0	0			

**C R**                      **C R**

0 0 Unused                1 0 Wait to commit

0 1 Committed            1 1 Ready to commit

Last writer of each arch reg with each group is shown in **red**



CB	Group			C	R	Reg	Value	uop
	0	1	2					
0	0	0	0	0	0			
1	0	1	0	0	1	rsp	1000	U7
2	0	0	0	0	0			
3	0	0	1	1	1	rdx	3333	U12
4	0	0	0	0	0			
5	0	1	0	0	1	rip	4096	U11
6	0	0	1	1	1	rip	9110	U14
7	0	1	0	1	1	rsi	2222	U9

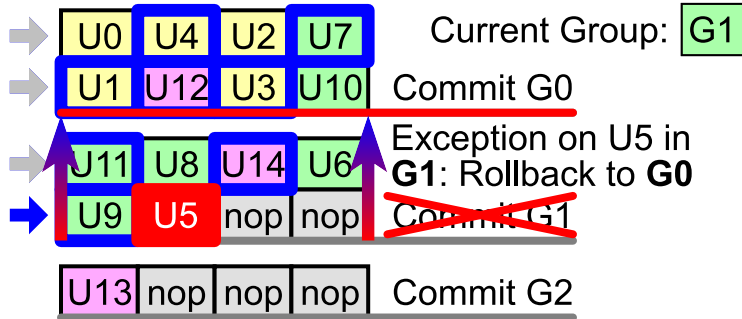
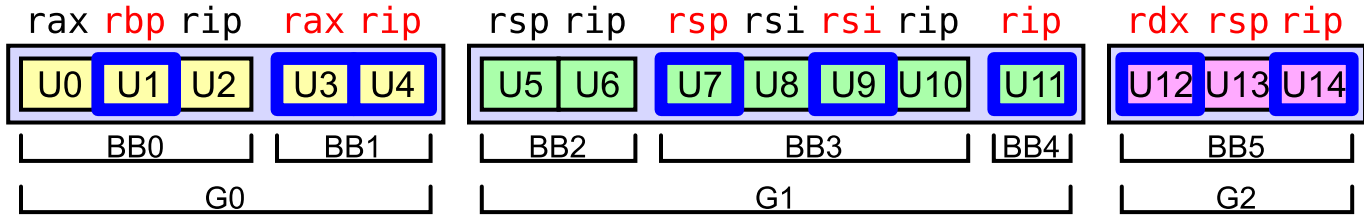
rax	1111	U3	rsi	-	-
rbx	-	-	rdi	-	-
rcx	-	-	rbp	1234	U1
rdx	-	-	rsp	1000	U7
rip	4096	U11			

**C R**                      **C R**

0 0 Unused                1 0 Wait to commit

0 1 Committed            1 1 Ready to commit

Last writer of each arch reg with each group is shown in **red**

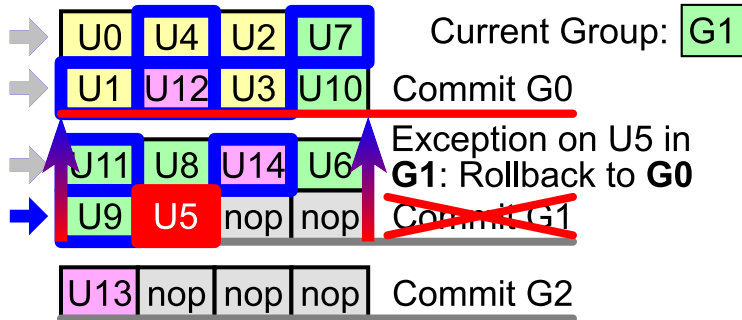
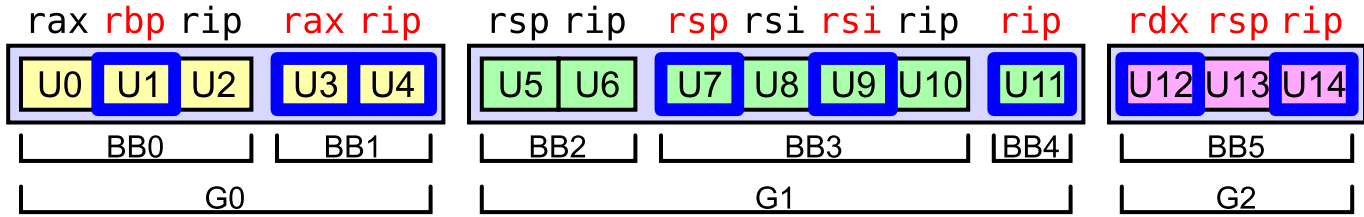


CB	Group			C	R	Reg	Value	uop
	0	1	2					
0	0	0	0	0	0			
1	0	1	0	0	1	rsp	1000	U7
2	0	0	0	0	0			
3	0	0	1	1	1	rdx	3333	U12
4	0	0	0	0	0			
5	0	1	0	0	1	rip	4096	U11
6	0	0	1	1	1	rip	9110	U14
7	0	1	0	1	1	rsi	2222	U9

rax	1111	U3	rsi	-	-
rbx	-	-	rdi	-	-
rcx	-	-	rbp	1234	U1
rdx	-	-	rsp	1000	U7
rip	4096	U11			

C	R		C	R	
0	0	Unused	1	0	Wait to commit
0	1	Committed	1	1	Ready to commit

Last writer of each arch reg with each group is shown in **red**



CB	Group			C	R	Reg	Value	uop
	0	1	2					
0	0	0	0	0	0			
1	0	1	0	0	1	rsp	1000	U7
2	0	0	0	0	0			
3	0	0	1	1	1	rdx	3333	U12
4	0	0	0	0	0			
5	0	1	0	0	1	rip	4096	U11
6	0	0	1	1	1	rip	9110	U14
7	0	1	0	1	1	rsi	2222	U9

rax	1111	U3	rsi	-	-
rbx	-	-	rdi	-	-
rcx	-	-	<b>rbp</b>	1234	U1
rdx	-	-	rsp	-	-
<b>rip</b>	2048	U4			

**C R**                      **C R**

0 0 Unused              1 0 Wait to commit

0 1 Committed          1 1 Ready to commit

# Shadow Register File

Commit buffer alone **not sufficient** to ensure we can accurately restore checkpointed state:

- As soon as group  $N$  becomes the currently executing group, any CB slots assigned to group  $N$  are transferred **asynchronously** and in **arbitrary order** into architectural register file (subject to result readiness and banking constraints)
- **Direct commits:** uops where (home group = execute group) **bypass commit buffer and go straight to register file**

Since other uops in the current group may still have exceptions, use **shadow bitcells** to checkpoint state as of last commit point

- Same operation as in other BT systems, except that commit/rollback may occur in middle of trace too

# Store Commit Buffer

- Just like register commit buffer, except destination is a **physical address**, not a register
- Stores are always ready at time of issue, so **no R (Ready) bit** needed in each entry
- **Speculatively updates L2** cache using locked/dirty line scheme
- Last store to given physical address **may change at runtime**:
  - PTL uses **profiling** to speculatively identify **final store** in each group to **each physical address**
  - **Similar to** techniques for **alias prediction** and redundant store elimination

# Commit Depth Prediction

- **Commit Depth Prediction:** predict *where* trace will terminate:
  - Number of successfully completed groups
  - Final completed RIP (required next trace to recover)
- If predicted to complete N groups, after fetching bundle at commit point N, **begin fetching next predicted trace instead**
- **Depth mispredictions** require pipeline flush to refill frontend (5 cycles) with **rest the current trace** or the **next trace**
- **Avoids double penalty** of both a rollback and a pipeline flush to start a recovery trace
- Effectively transforms **rollbacks** from an event to be avoided into a **normal and predictable part of execution**

# Extremely Fast Schedule Convergence

- Commit groups also promote **significantly faster convergence** on optimized traces
- PTL can **salvage partially executed traces** and **splice the working parts together** into new optimized traces
  - Existing BT systems have to **start from scratch** after an atomic trace fully rolls back
  - Therefore, ***useful long traces are formed very quickly***
- PTL naturally **evolves** pool of active traces, surprisingly similar to a **genetic algorithm**

# Commit Group Formation Algorithm

Commit points selected *after* scheduling: fall at “natural” locations, instead of constraining schedule itself:

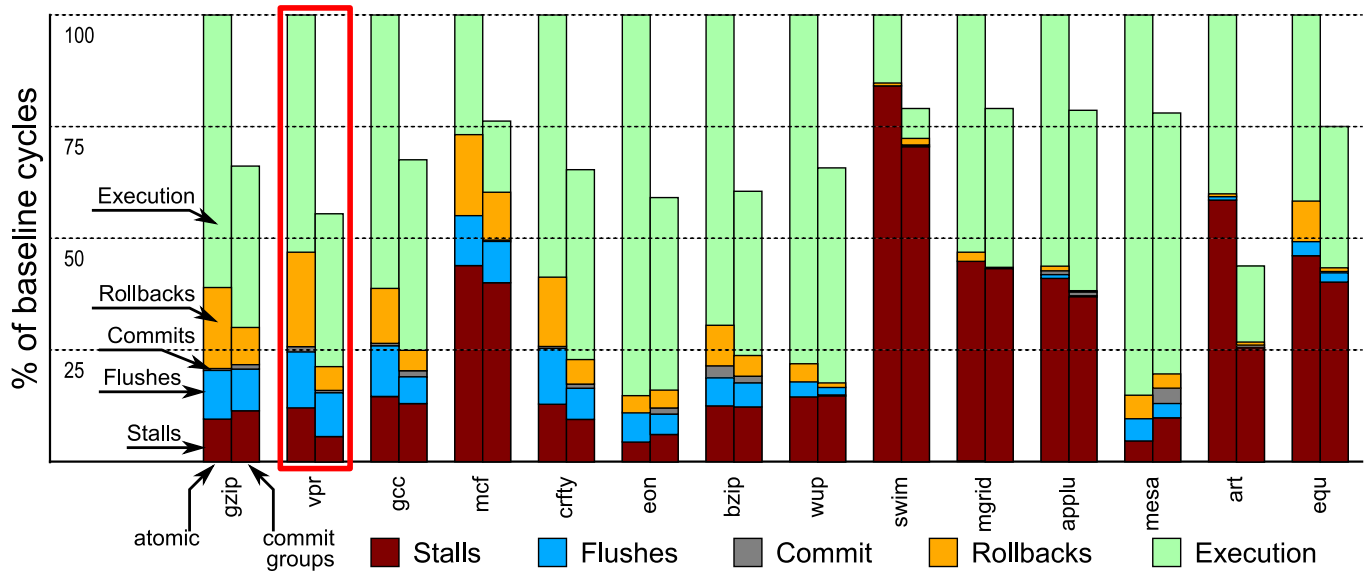
- Find cycle of **latest uop** in schedule within each original basic block
- **Merge** basic blocks with common final cycles in schedule into one **commit group**
- **Insert** commit point at cycle in which each group is finished in schedule
- Identify **final writer** of each architectural register and physical address within each group
- **Assign** commit buffer slot to each final writer
  - Assign slots according to banked CB structure to maximize throughput
  - Slots recycled after each owning group commits

# Benchmarking Methods

- **PTLsim**: our advanced cycle accurate simulator
- **SPEC 2000** benchmarks, ~2 billion x86-64 instructions each, starting in middle of main loops
- **Atomic Traces (*baseline*)** versus **Commit Group Traces**
- **Same PTL optimizations** and heuristics enabled for both cases
- **No predication or multi-path execution**
  - Shorter traces, but isolates commit group traces vs. atomic traces

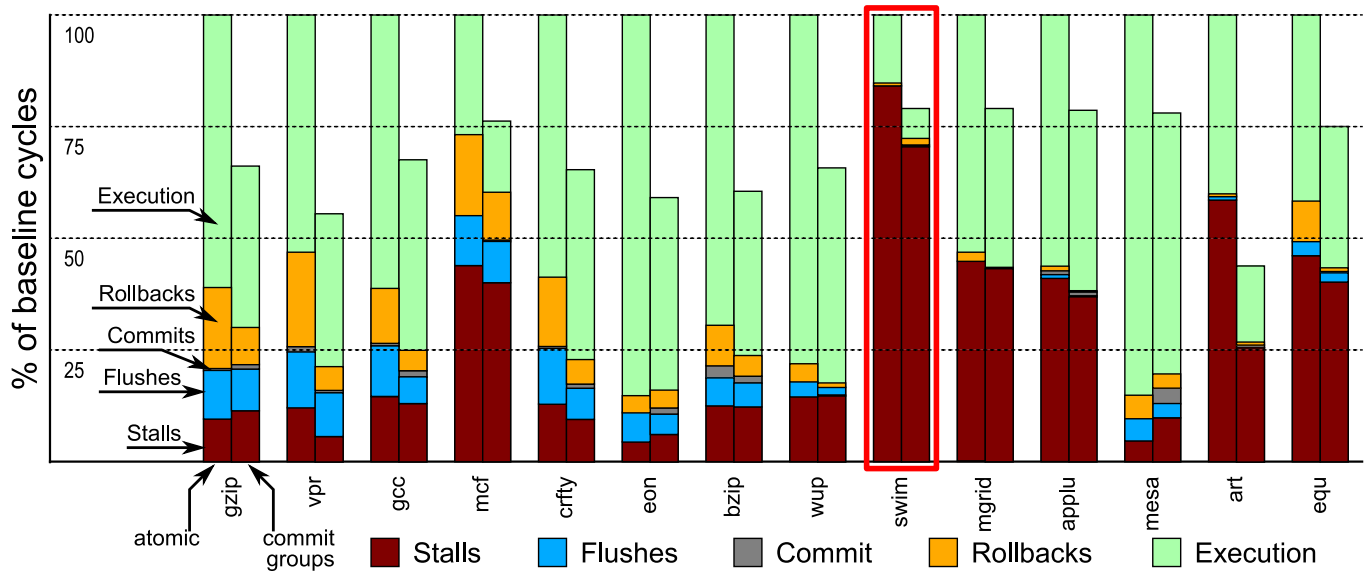
# Results by Benchmark

- Significant Speedup vs Atomic: **1.5× faster** on average
- Biggest gains on branch heavy integer: cuts rollbacks by **66%**



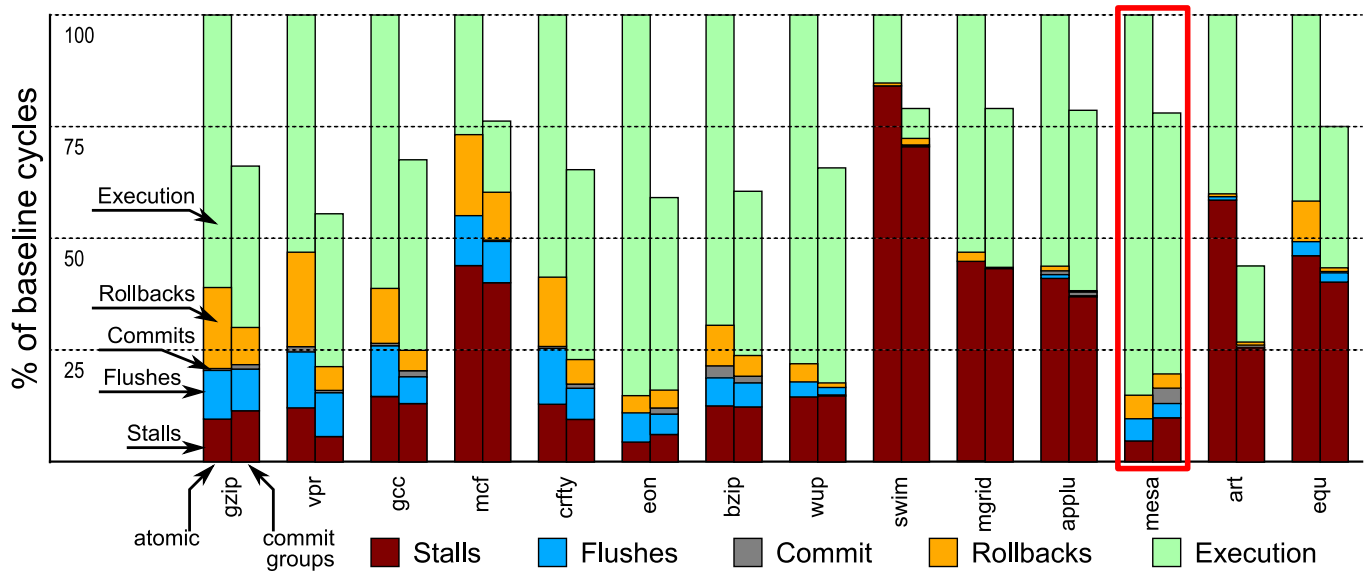
# Results by Benchmark

- Significant Speedup vs Atomic:  $1.5\times$  faster on average
- Biggest gains on **branch heavy integer**: cuts rollbacks by 66%

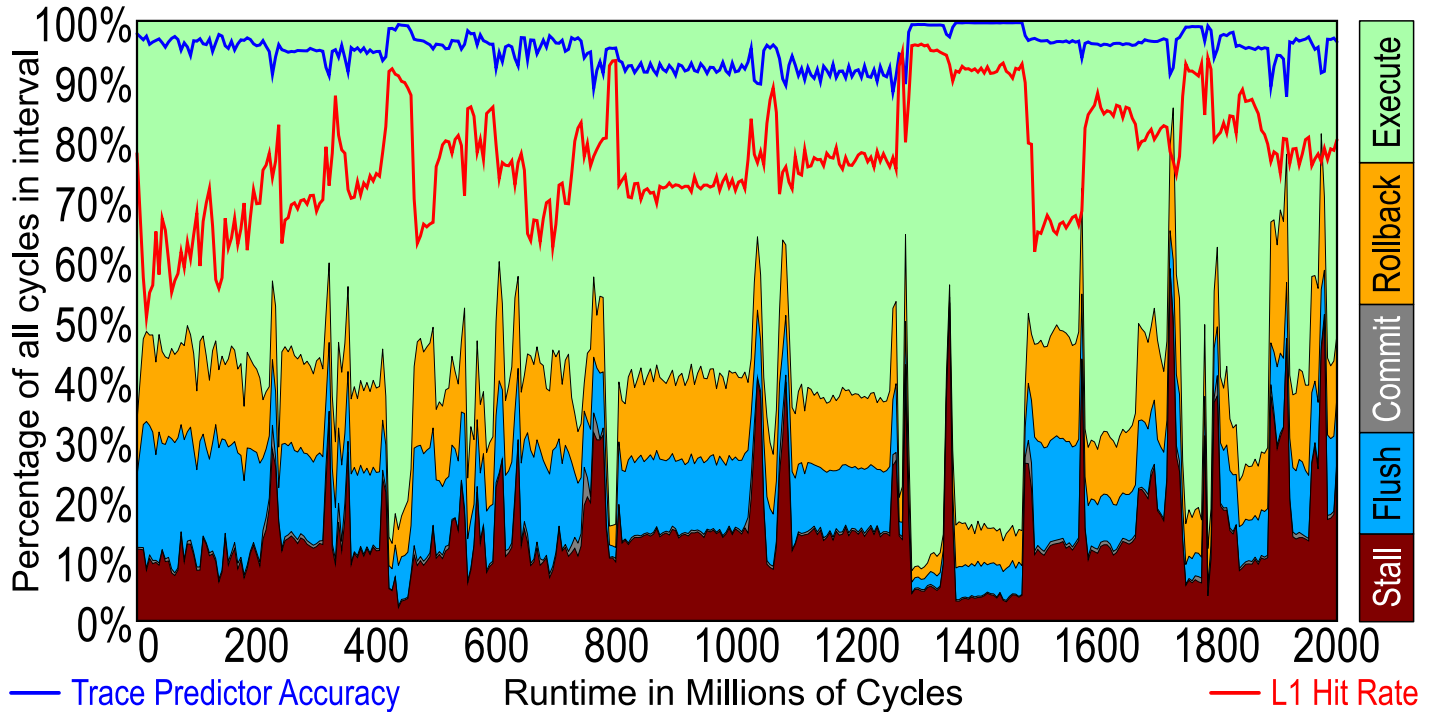


# Results by Benchmark

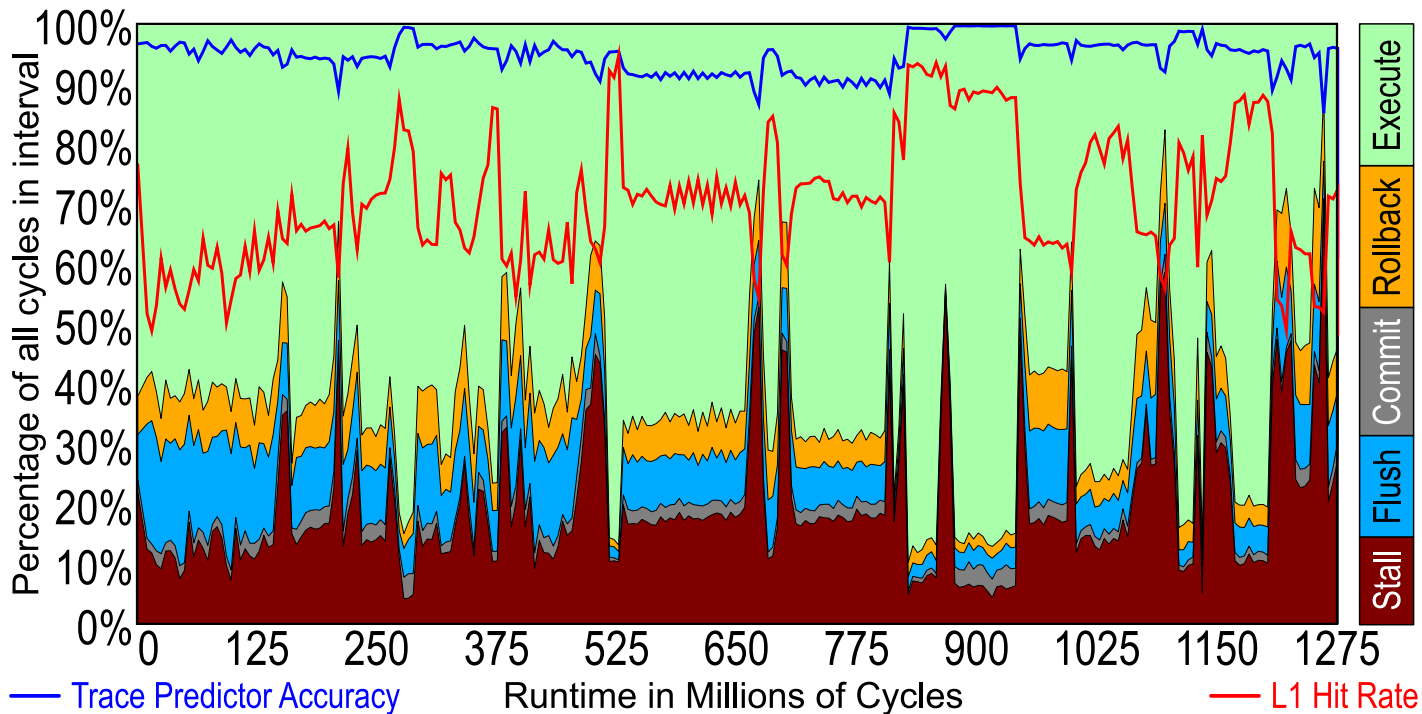
- Significant Speedup vs Atomic:  $1.5\times$  faster on average
- Biggest gains on **branch heavy integer**: cuts rollbacks by 66%



# Results In Detail: gcc (Atomic Traces)

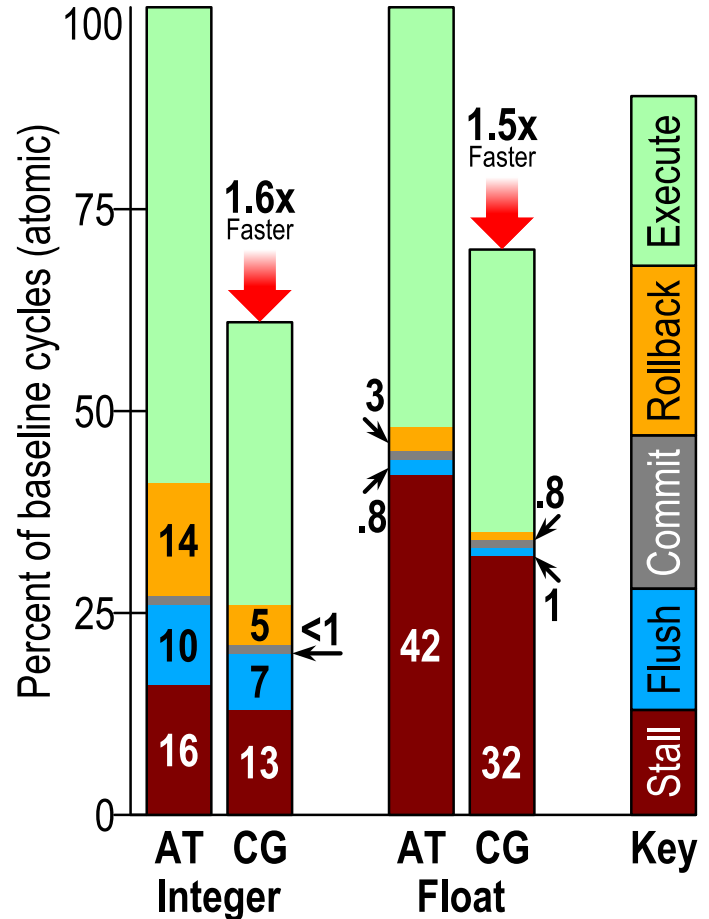


# Results In Detail: gcc (Commit Groups)



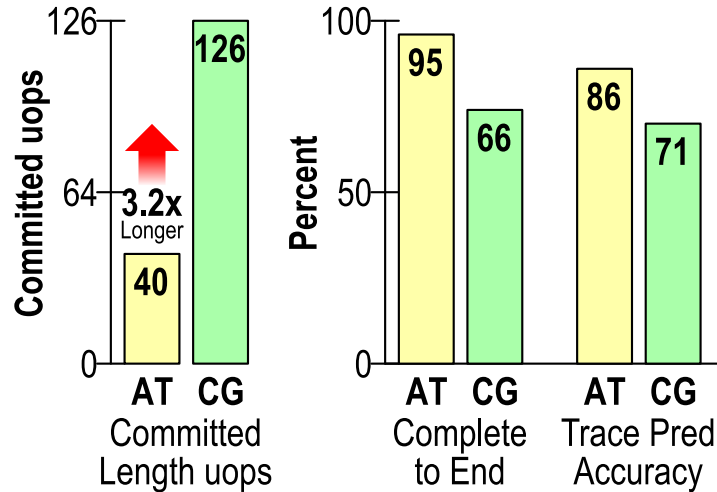
# Results Summary

- Significant Speedup vs Atomic:  $1.5\times$  faster on average
- Cuts time wasted by roll-backs by 66% (integer), 60% (FP)
- Longer traces promote better load hoisting: less stall time



## Results: More Statistics

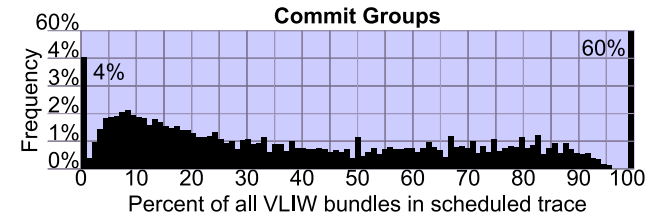
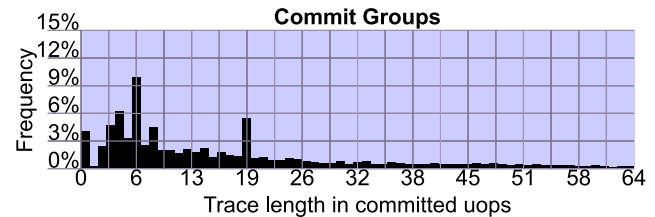
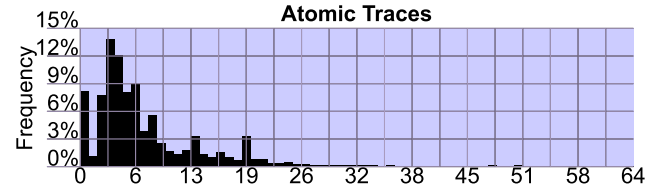
- Significantly longer traces possible with CG
- Many side exits from trace with CG (but no performance loss!)
- Trace predictor accuracy lower with CG, but still acceptable



# Commit Depth Statistics

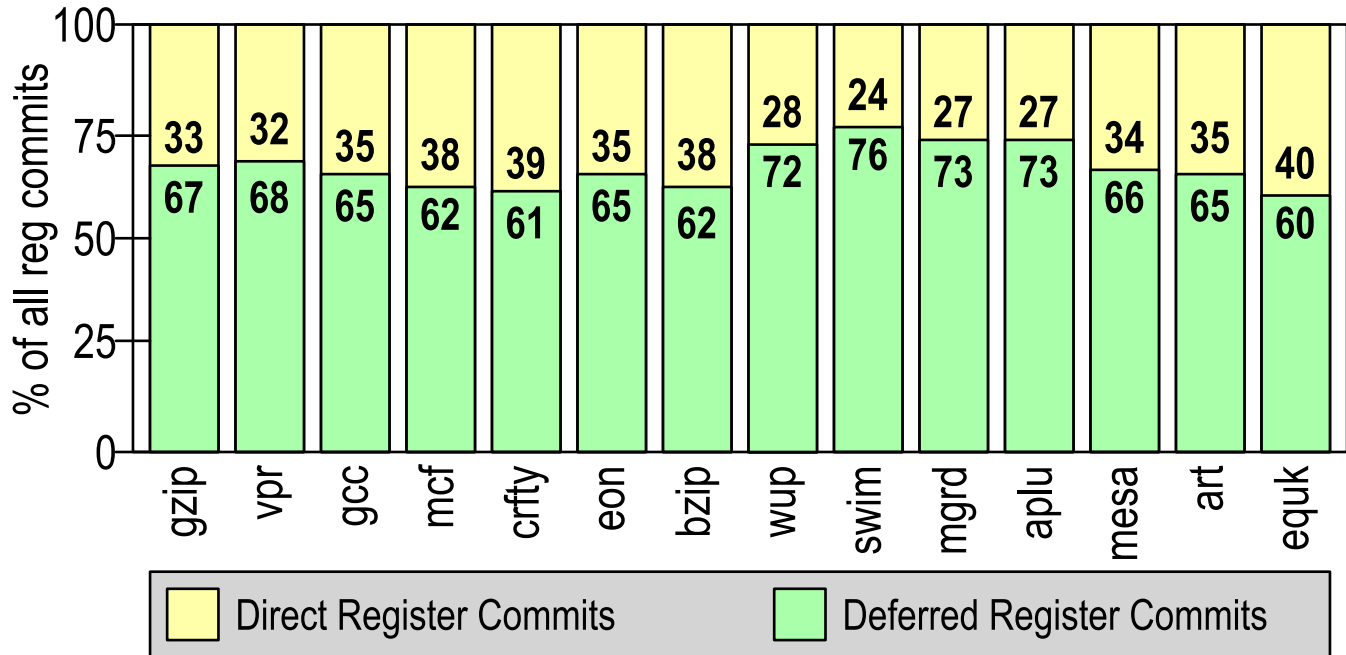
- **Atomic traces:** most traces are **short** (average: 15-20 uops)
- **Commit groups:** longer traces (average: 40 uops)
- Remember, **no predication or multipath execution** was used in this study (so traces are shorter)
- Wide range of **trace exit points** (only 60% of CG traces fully complete)

## Histograms for *gcc*



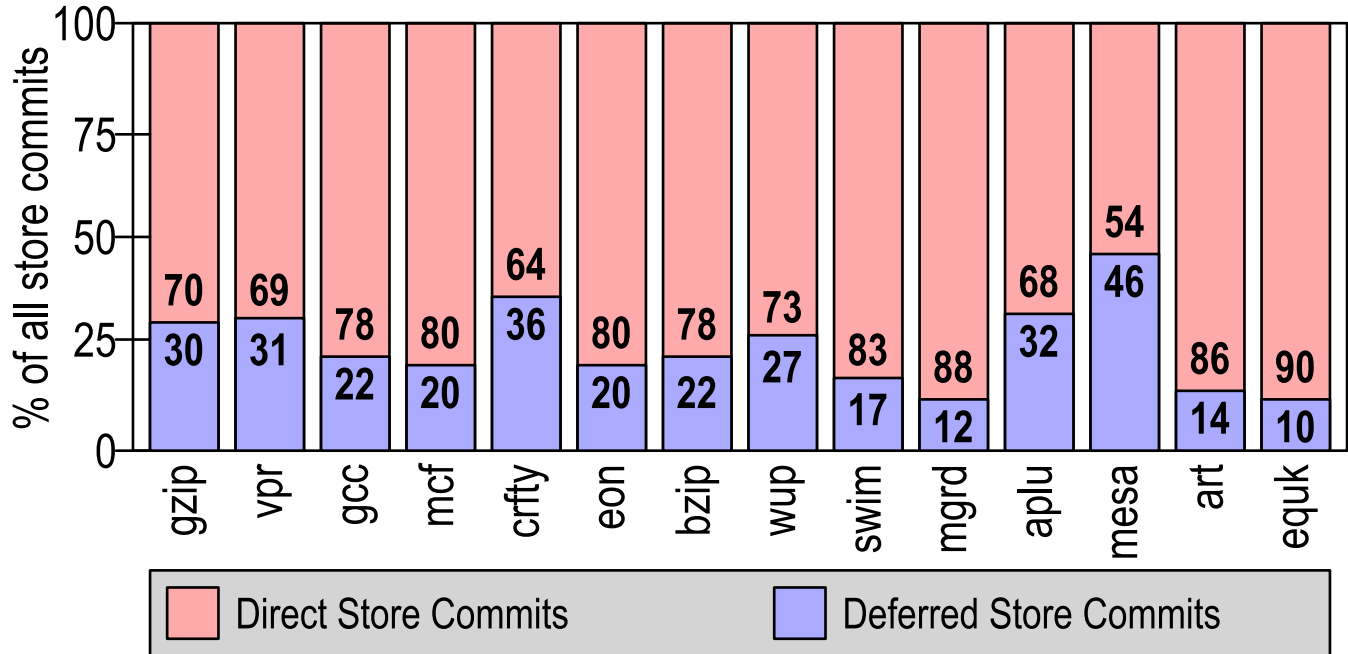
# Register Commits: Direct vs Deferred (via CB)

- 2/3 deferred in integer, more in FP

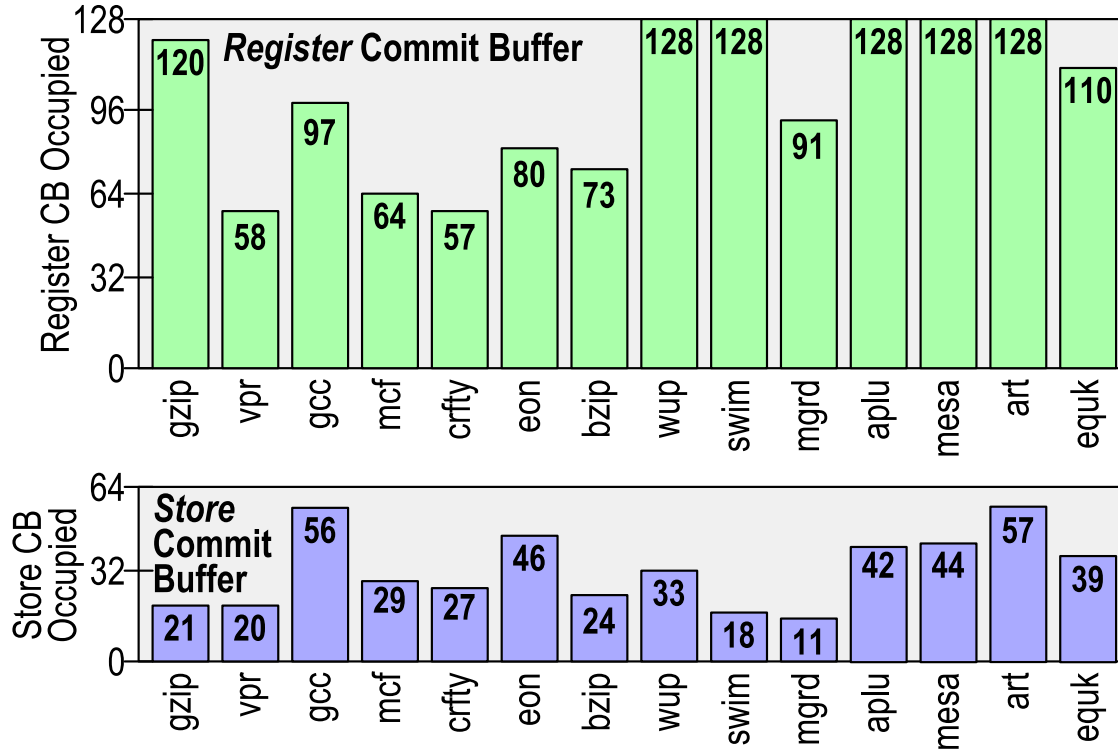


## Store Commits: Direct vs Deferred (via CB)

- **Varies widely**, but many more direct commits (within home group) of stores



# Commit Buffer: Occupancy



# Future Work

- Improvement of **trace predictor accuracy**:
  - **Commit depth prediction** significantly complicates trace predictor
- **Tapering Trace Problem**. IPC greatly decreases at end of trace as execution reduces to single dependency chain
  - Serious issue in **floating point** codes, since FP latencies are so long
  - **Overlap** first group of incoming trace with last group of outgoing trace
- Addition of **predication** and **multi-path commit** to complement commit groups

# Summary

- *Incremental commit groups* allow **long high ILP traces**, while **minimizing wasted computations** by rollbacks
- *Two sources of performance*: **reduced rollback penalties** and **long high ILP traces**
- **Commit Buffer** defers updates to architectural state until appropriate commit group reached
- **Commit depth predictor** eliminates pipeline flush stalls when rollbacks can be predicted
- **Significantly faster convergence** on **optimized traces** by splicing together partially executed traces
- **NET RESULT: 1.5X performance gain** vs atomic, **70% reduction in wasted rollback cycles**

## *Questions?*

Check out <http://www.ptlsim.org> for our x86-64 out of order simulator, which features many of the simulation components used in this study