ASeD: Availability, Security, and Debugging Support using Transactional Memory

JaeWoong Chung, Woongki Baek, Nathan G. Bronson, Jiwon Seo, Christos Kozyrakis, Kunle Olukotun

Stanford University

Motivation

•Transactional Memory (TM)
- •Simplifies parallel programming using atomic blocks
- •Easy to use & high performance

•TM systems should provide ACI features
- •Atomicity: rollback to a safe system state
- •Consistency: guarantee system-level invariants
- •Isolation: limit the propagation of side-effects

•Key insights: ACI of TM can also be used for ASeD
- •Availability
- •Security
- •Debugging

Contributions

•Investigation of the feasibility of ASeD on HTMs
- •Demonstrate the great synergy between ASeD and ACI

•Implementation of the ASeD on HTM by addressing:
- •Tightly coupled ACI components in HTM
- •Enhancing performance-oriented ACI in HTM for ASeD
- •Simultaneous use of ACI for concurrency and ASeD

•Quantitative evaluation of the proposed HTM with ASeD
- •Overall, achieving ASeD with small runtime overhead

ASeD Design Philosophy

•Tool vs. Solution
- •Providing a tool with key primitives for flexibility
- •Providing HW acceleration for the common case

•Integration vs. Versatility
- •Proposing an integrated design instead of a collection of separate HW extensions

•Cost-efficiency vs. Performance
- •Avoiding additional HW just to accelerate a single feature
- •Maximizing HW resources between ASeD & HTM

Availability Features

•ASeD addresses both permanent & transient faults
- •Permanent: loss in cores or caches, etc.
- •Transient: packet loss, logic errors, etc.

•Availability primitives
- •Global ckeckpoint: system-wide state
- •AI (atomic/isolated) regions: thread-specific state

•Use case
- •Permanent faults
  - •Global checkpoints are periodically taken
  - •Upon faults, roll-back to the latest global checkpoint
- •Transient faults
  - •Code fragments are enclosed by AI regions
  - •Upon faults, just roll-back the faulty thread
  - •Significantly reduced MTTR

Security Features

•Security primitives
- •Fine grained read/write barriers
  - •Accessing a specific address is detected & notified
- •Isolated execution
  - •Similar to the AI regions

•Use case
- •Buffer overflow detection
  - •Mark the address of canaries with write barriers
  - •Overwrite to a canary is detected by the ASeD
  - •Significantly lower overhead than SW canaries

Debugging Features

•Debugging primitives
- •Global checkpoint
- •Fine grained read/write barriers

•Use case
- •Scalable watchpoints
  - •Provides arbitrary # of watchpoints using R/W barriers
  - •Negligible performance impact
  - •Supports coarse-grain watchpoints with the runtime
- •Parallel bookmark & step-back are also supported

Feature Decomposition

Primitive	Function	Used for	SW Interface	Components
AI region	atomic & isolated region (single thread)	local recovery, isolated-execution	AI_begin, AI_end, AI_abort	register checkpoint/rollback/release, memory versioning/rollback/commit, access conflict detection
Global checkpoint	system-wide checkpoint	global recovery, parallel bookmark, step-back	CP_take, CP_release, CP_restore	cache line writeback, register checkpoint/rollback/release, system-wide memory logging/rollback/release
Address protect	fine-grain barrier	R/W barrier, canary, watchpoint	AP_set, AP_reset	access conflict detection
Software handler	callback function from HW	R/W barrier, canary, watchpoint	register_hd, remove_hd	SW handler triggered on conflict, runtime support

ASeD on Transactional Memory

•Implementing the ASeD on HTM using similarity of ACI
- •Reduces HW costs of ASeD support
- •An integrated system of TM with the advantages of ASeD

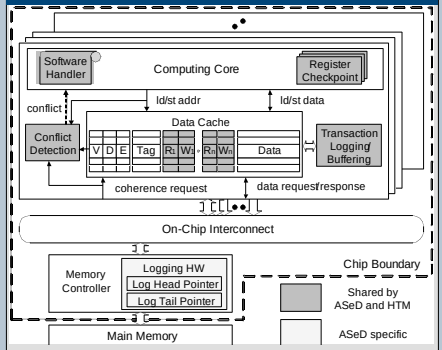
ASeD ∩ HTM ≈ ASeD

ACI	ASED Component	HTM Availability
A	Register checkpoint/ rollback/release	√
	Memory versioning/ rollback/commit	√ (per-thread)
	System-wide memory logging/rollback/release	×
C & I	Conflict detection	√ (per cache line metadata)
	Software handler	√
Inter-face	AI_begin/end/abort	√ (TM_begin/end/abort)
	AP_set/reset	~ (only within transactions)

•Except global checkpoint, all primitives are available in HTM
- •AI regions ➔ atomic transactions
- •Address protection ➔ conflict detection
- •SW handlers ➔ SW contention manager

•Small changes in conflict detection
- •Word granularity conflict detection ➔ hybrid (HW + SW)
- •Coexisting Tx's and AIs ➔ dedicated nesting level for AI

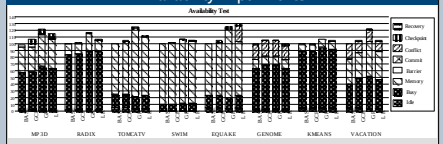
ASeD – HTM ≠ Φ



•Major difference is global checkpoint (system-wide)
- •Periodic interrupts stop/complete all cores/comm's
- •Flush all dirty cache lines to memory
- •HTM mechanisms are used to checkpoint reg. state

•When an error is detected
- •All cores/caches are reset
- •All logs are applied in reverse order
- •Register checkpoint is restored and system resumes

Availability Experiments



•Normalized execution of 8 apps with 4 versions
- •Base: without checkpoints & faults
- •GCP: global checkpoint every 50K cycles, no faults
- •GR: same as GCP with fault injection/1M cycles
- •LR: same as GR with local recovery using AIs

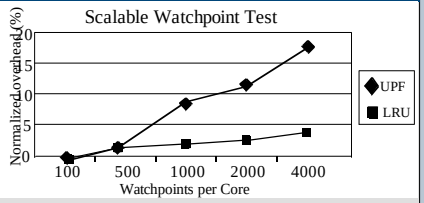
•Summary of results
- •GCP overhead : < 3.5%, GR overhead: 20-30%
- •LR reduces the overhead of GR (Equake: false conflicts)

Security Experiments

	Gzip	Polymorph	Ghttpd	Nullhttpd
ASeD	0.2%	0.4%	0.3%	0.3%
StackGuard	3.1%	3.6%	3.3%	2.0%

•Overhead of ASeD is significantly lower than StackGuard
- •StackGuard: a number of inst's to set/check canaries
- •ASeD: only two inst's at prologue & epilogue of functions

Debugging Experiments



•A u-bench that randomly sets watchpoints
•Two cache replacement policies
- •LRU: least recently used
- •UPF: unprotected first

•UPF performs worse (L1 is filled watchpoints data)
•LRU overhead: overflow mechanism for L1 metadata bits

Conclusions

•We proposed & evaluated the ASeD on top of HTM
- •Availability using global & local checkpoints
- •Security using R/W barriers & isolated executions
- •Debugging: using global checkpoints

•Overall, enhanced HTM with ASeD with minimal HW costs