RETCON: Transactional Repair without Replay

Colin Blundell
University of Pennsylvania
blundell@cis.upenn.edu

Arun Raghavan
University of Pennsylvania
arraghav@cis.upenn.edu

Milo M. K. Martin
University of Pennsylvania
milom@cis.upenn.edu

Abstract

Over the past decade there has been a surge of academic and industrial interest in optimistic concurrency, *i.e.* the speculative parallel execution of code regions that have the semantics of isolation. This work analyzes scalability bottlenecks of workloads that use optimistic concurrency. We find that one common bottleneck is updates to auxiliary program data in otherwise non-conflicting operations, *e.g.* reference count updates and hashtable occupancy field increments.

To eliminate the performance impact of conflicts on such auxiliary data, this work proposes RETCON, a hardware mechanism that tracks the relationship between input and output values symbolically and uses this symbolic information to transparently repair the output state of a transaction at commit. RETCON is inspired by instruction replay-based mechanisms but exploits simplifying properties of the nature of computations on auxiliary data to perform repair *without* replay. Our experiments show that RETCON provides significant speedups for workloads that exhibit conflicts on auxiliary data, including transforming a transactionalized version of the Python interpreter from a workload that exhibits no scaling to one that exhibits near-linear scaling on 32 cores.

Categories and Subject Descriptors

C.1.4 Computer Systems Organization [*Processor Architectures*]: Parallel Architectures

General Terms

Design, Languages, Performance

Keywords

Transactional Memory, Parallel Programming

1. Introduction

The past decade has seen a great deal of academic and industrial interest in the area of speculative synchronization, both in the form of hardware transactional memory (HTM) [16] and in the form of speculative parallel execution of traditional lock-based critical sections [21, 29]. Speculation allows code regions with the semantics of isolation to execute in parallel by detecting conflicting accesses to the same data and rolling back when a conflict occurs. Speculation thus makes the degree of concurrency dependent on the amount of data sharing rather than the granularity of locking, potentially easing the task of writing high-performance parallel programs. In spite of much research, however, the impact of such speculation on workload performance is not clear.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISCA'10, June 19–23, 2010, Saint-Malo, France.

Copyright © 2010 ACM 978-1-4503-0053-7/10/06...$10.00

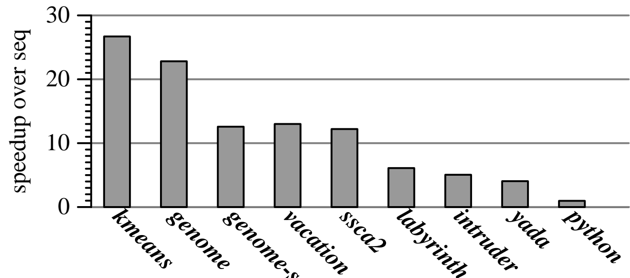
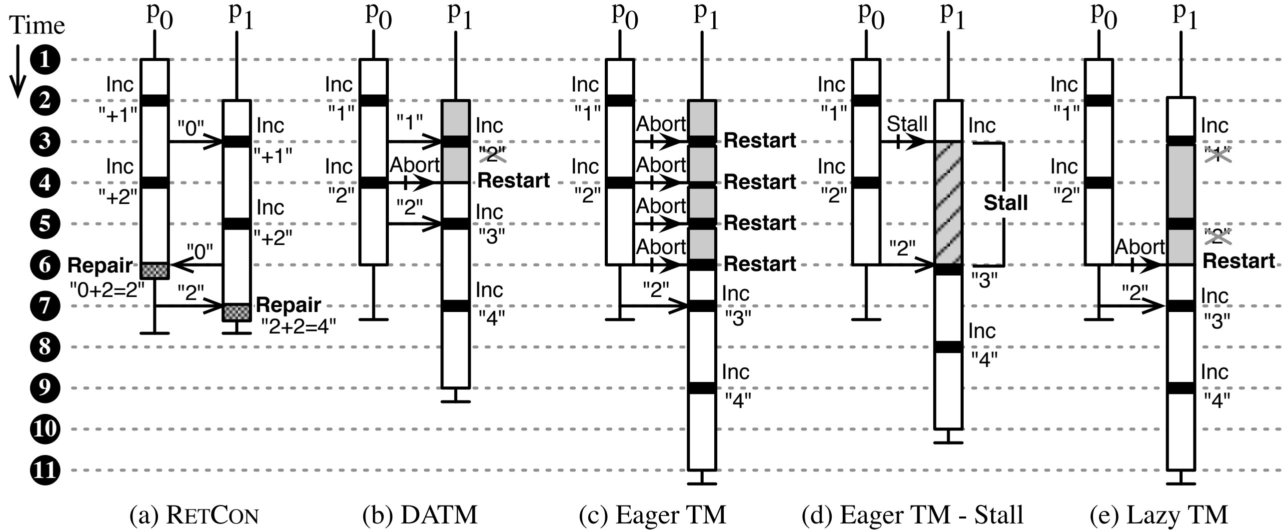


Figure 1. Scalability of aggressive HTM on 32 cores. The HTM is configured to eliminate cache overflows and has a robust timestamp-based contention management policy.

This paper undertakes a study of the potential impact of speculative concurrency on the scalability of parallel programs. To drive our study we intentionally choose benchmarks that would be challenging for speculation (described in Section 3). Our baseline system is a state-of-the-art hardware transactional memory system with reasonable contention management policies that is configured to avoid cache overflows. Nonetheless, Figure 1 shows that the performance of these workloads is mixed: although some workloads achieve near-linear speedup, nearly half of the workloads obtain a speedup of less than 10x on 32 cores.

As Section 3 describes, the dominant bottleneck to scalability is serialization arising from conflicts. We find that many sources of conflicts can be eliminated via simple software restructurings. After performing these restructurings, however, a common pattern remains of conflicts on *auxiliary data*, that is, data such as reference counters and hashtable occupancy fields that is not related to the main computation of the program itself. These conflicts cause significant performance loss on several of our workloads, for example completely serializing execution on a transactionalized version of the Python interpreter. This result is especially troubling because the operations in these workloads are conceptually non-conflicting, *e.g.*, simultaneous reads of a reference-counted shared object.

Others have reported similar patterns on a variety of workloads. Carlstrom *et al.* [7] found that a shared global identifier variable in a transactional version of SPECjbb was a bottleneck to an otherwise-scalable workload. An Azul Systems engineer analyzed the performance of Azul Systems' speculative lock elision on its clients' Java workloads and reported that programmers and library writers do not naturally write "TM-friendly code" and that counters such



(a) RetCon (b) DATM (c) Eager TM (d) Eager TM - Stall (e) Lazy TM

Figure 2. Comparison of RetCon to other approaches. P_0 and P_1 begin transactions at times ❶ and ❷ respectively. Each transaction performs two increments to a shared counter variable that is initialized to zero. *(a)* RetCon symbolically tracks the counter address and repairs its value at commit by adding the computed increment. *(b)* DATM [30] forwards the speculatively incremented counter variable at time ❸, but must abort a transaction when the second increment introduces a cyclic dependence at time ❹. *(c)* In EagerTM, P_1 suffers repeated aborts until P_0 commits at time ❻. *(d)* EagerTM-Stall stalls P_1's first increment until P_0 commits at time ❻. *(e)* In LazyTM, P_1 performs both its speculative increments but then aborts when it detects a conflict on the commit of P_0 at time ❻.

as hashtable occupancy fields lead to "a general pattern of updates to peripheral shared values" that significantly impact performance due to "presenting a true data conflict" [10].

As the second part of this work, therefore, we seek to provide hardware support for minimizing the performance impact of such patterns. We observe that auxiliary data is usually accessed by short, simple computations that do not affect the larger transaction.[1] This property suggests a hardware-based approach of reacquiring lost data at commit and using the current values of this data to repair local state as necessary. Such a repair-based approach was proposed by ReSlice [32] to lessen the impact of conflicts in thread-level speculation, and similar slicing has been employed in other contexts as well [8, 17, 35]. These proposals use instruction-based repair by saving the dependent instructions of a conflicting load to later re-execute these instructions with the updated value of the load (either in a special-purpose core or by re-using the resources of the processor itself).

Guided by the nature of this auxiliary data, we propose a different approach. As the processor executes a transaction, it also tracks the relationship between input values and output values symbolically. Conditionals form constraints on the acceptable range of values that an input can take when reacquired at commit. At commit time, all inputs that have been lost are reacquired, constraints are checked, and symbolic computation is reapplied to these values (see Figure 2).

Our instantiation of this approach, RetCon,[2] is tailored to fit the needs of the auxiliary data present in our workloads. RetCon tracks an input symbolically through a sequence of loads, simple arithmetic operations, branches, and stores, with more complex computation creating a constraint that the input value be the same at commit. To track symbolic information, RetCon adds a buffer to hold the initial values of symbolically-tracked blocks, a buffer to hold constraints, and a buffer to hold symbolically-tracked stores. Our data shows that these structures can be small–*e.g.*, eight entries.

Although RetCon's focus is on repairing remotely changed values, its mechanisms have the secondary benefit of reducing conflicts in other ways. RetCon's resolution of conflicts at commit implicitly provides selective lazy conflict detection [6, 33, 39]. In addition, by performing conflict detection based on values [25, 37], RetCon also avoids conflicts due to false sharing [19], silent sharing, and temporally silent sharing [20].

We quantitatively evaluate RetCon and find that it has a significant impact on the performance of our workloads. RetCon's selective laziness and value-based conflict detection contribute to this performance impact, but it is RetCon's ability to repair true conflicts on auxiliary data that provides most of the benefit. Altogether, RetCon provides a speedup of more than 40% on several workloads and in one case transforms a workload that exhibited no scaling to instead exhibit near-linear scaling.

2. Hardware Transactional Memory Baseline

Supporting speculative concurrency in hardware entails several tasks. First, the hardware must be able to detect conflicting requests between two speculative regions (conflict detection). Second, it must employ a policy to resolve conflicts by stalling or aborting one of the conflicting speculative regions (contention management). Third, it must be able to roll back to pre-speculative state on an abort (version management). Below, we describe how our baseline hardware supports these tasks.

Conflict detection. Our baseline system detects conflicts through the cache coherence protocol by adding a "speculatively-read" bit and a "speculatively-written" bit to each block in the primary data cache. As the processor accesses blocks within a speculative region, it sets the read/written bits for those blocks. External requests snoop these bits to determine conflicts, where a

[1] Throughout this paper we will generally refer to speculative regions as transactions for convenience, but they could equally well be speculative lock-based critical sections.

[2] Retcon, short for *retroactive continuity*, refers to soap operas' and comic books' practice of revising past events as necessary to match current reality.

Workload	Description and Input Parameters
genome	From STAMP, gene sequencing program, *g1024 s48 n65536*
genome-sz	Variant of genome with resizable hashtable
intruder	From STAMP, network packet intrusion detection program, *a10 l4 n2038 s1*
intruder_opt	Variant of intruder optimized with fixed-size hashtable and thread-private queues
intruder_opt-sz	Variant of intruder optimized with resizable hashtable and thread-private queues
kmeans	From STAMP, partition-based clustering program, *m15 n15 t0.05 irandom-n2048-d16-c16.txt*
labyrinth	From STAMP, shortest-distance path routing, *irandom-x32-y32-z3-n96.txt*
ssca2	From STAMP, graph kernels, *s13 i1.0 u1.0 l3 p3*
vacation	From STAMP, travel reservation system, *n4 q60 u90 r16384 t4096*
vacation_opt	Variant of vacation optimized with fixed-size hashtable
vacation_opt-sz	Variant of vacation optimized with resizable hashtable
yada	From STAMP, Delaunay mesh refinement, *a20 i633.2*
python	Python interpreter, *bm_threading.py (from Google's Unladen-Swallow [2] suite)*
python_opt	Variant of python with interpreter optimizations

Table 1. Workloads used to drive our study.

conflict is defined as an external write request to a block that has been speculatively read or any external request to a speculatively-written block. Such conflicts invoke the contention management policy, described below. This work uses OneTM [4] as a backing mechanism for transactions that overflow the local cache hierarchy. Our baseline also includes a read-only permissions-only cache [4] to reduce the frequency with which overflows occur. On the workloads analyzed in this work, the permissions-only cache essentially eliminates cache overflows entirely.

Contention management. When a conflict occurs, the processor either: (1) aborts the local speculation, (2) aborts the remote speculation, or (3) stalls the remote speculation, taking care to ensure that such stalling will not cause deadlock. Our baseline uses the "oldest transaction wins" timestamp-based conflict resolution policy. Prior studies [6, 33] and our own experimental reconfirmation have found that this policy generally performs the same or better than other policies, ensures timely forward progress, and is fairly robust across a range of workloads.

Version management. To support rollback of speculation to pre-speculative state, the baseline system uses a "clean-to-L2" approach similar to that used in prior work (*e.g.*, [5, 21]). On the first speculative write to a non-speculatively dirty block, the old value of the block is first written back ("cleaned") to the second-level (L2) cache. On an abort, the system flash-invalidates all speculatively-written blocks. The pre-speculative values of such blocks will be refetched from the lower levels of the memory hierarchy as needed via ordinary cache misses.

3. Remaining HTM Performance Challenges

We evaluate the performance of the baseline system on a variety of workloads that stress the performance of speculation (the details of these workloads are given in Table 1). The STAMP benchmark suite [22] is a set of transactional memory benchmarks that for the most part use coarse-grained transactions, with the intent of simulating the practices of naive programmers. We run all workloads in the suite except bayes, from which we could not extract useful conclusions due to high runtime variability. The benchmark labyrinth includes a code snippet wherein a large shared grid is copied privately within a transaction and then released early from conflict detection, as the algorithm itself detects conflicts on the grid at the semantic level. To achieve the same behavior without early release (a mechanism that our system does not provide), we modified the code to perform the grid copy before entering the transaction.

STAMP includes a hashtable that defaults to be non-resizable. For all workloads that use this hashtable we also run a second variant in which the hashtable is configured via STAMP's compile-time flags to automatically resize as needed. The resizable variants reflect the performance of using standard library implementations of hashtables in Java and C++, which are generally resizable by default. These variants have "-sz" appended to their names.

In addition to analyzing STAMP, we created a transactionalized version of the standard Python interpreter implementation, python. This workload is a challenging case for speculation: although the interpreter supports threading, this threading is explicitly designed for responsive graphical user interfaces and I/O events—not for supporting parallel execution on multicores. In fact, threads synchronize using a global interpreter lock (GIL). Although threads may perform selected system calls without holding the GIL, threads may interpret bytecodes only while holding the GIL. Thus, in the absence of speculation, bytecode interpretation is completely serialized by the GIL. By applying speculative lock elision to the GIL, we are attempting to extract parallelism from this complex workload that was written assuming sequential execution.

Figure 3 shows that the performance of the baseline system on the above-described workloads varies significantly. (The workload variants with "_opt" suffixes are described below.) As the execution breakdown presented in Figure 4 indicates, the primary bottleneck to scalability on many workloads is conflicts.[3] The goal of this section is to (1) quantify the impact that straightforward software restructurings can have on reducing these conflicts and (2) characterize the conflicts that remain after such restructurings.

Opportunities for software restructurings. We find several opportunities for straightforward software restructurings. First, python contains global variables that are conceptually thread-private but were not made so due to the assumption that only one thread would be operating on them at a time. We trivially made these variables thread-private using the C "__thread" variable annotation supported by GCC and other compilers. Second, intruder dequeues work from one highly contended queue and enqueues work onto another highly contended queue; we split these queues to be thread-private. Third, both intruder and vacation have aborts due to rebalancing operations of a red-black tree used to implement a map interface. We replaced the tree implementation with STAMP's hashtable implementation. The conflicts in yada are due to irregular traversals of a shared mesh; we have not found a way to reduce these conflicts short of restructuring the algorithm, which is beyond the scope of straightforward software restructuring.

On the graphs, the _opt variants of the above workloads have the above software restructurings applied. An examination of the

[3] Exceptions are labyrinth, in which the algorithm induces load imbalance, and ssca2, which has poor caching behavior.

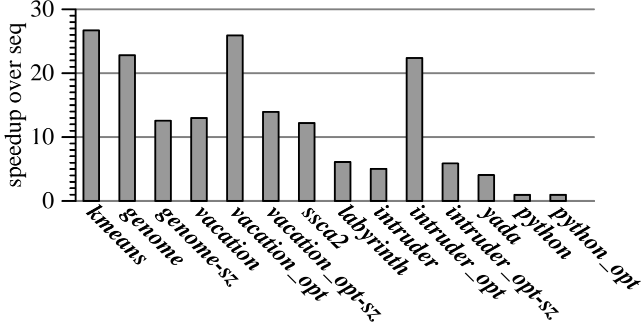


Figure 3. Scalability of workloads on baseline system before and after applying software optimizations. The optimizations performed to obtain the variant workloads are detailed in Table 1.

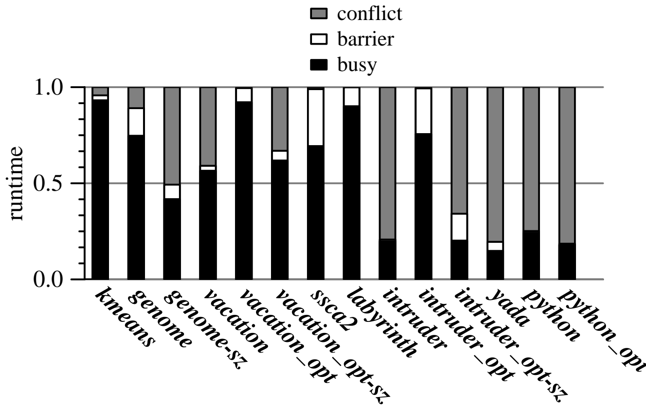


Figure 4. Time breakdown of workloads on the baseline system before and after applying software optimizations. The "busy" segment represents all time spent not stalled on synchronization. The "barrier" segment represents time stalled at a barrier, an indicator of load imbalance. The "conflict" segment represents time spent either stalled by another processor or doing work in a transaction that is ultimately aborted.

scalability of these variants (Figure 3) shows first that these simple changes have a dramatic effect on the behavior of `intruder_opt` and `vacation_opt`, increasing scalability from 5x and 13x respectively to over 20x in both cases. For the other variants, however, the picture is less rosy: the software changes do not improve scalability, and Figure 4 indicates that these workloads remain abort-bound even after elimination of the most obvious sources of conflicts.

Characterizing remaining conflicts. The `python_opt` workload conflicts on reference counts of shared objects; `vacation_opt-sz`, `intruder_opt-sz`, and `genome-sz` conflict on the occupancy field that the resizable hashtable increments on every insert to determine when to resize; and as mentioned above, `yada` conflicts on mesh operations. We note that except for `yada`, all of the remaining conflicts occur on operations that are auxiliary to the main computation of the workload. Unfortunately, these conflicts are less amenable to straightforward software restructuring (for example, distributing reference counts per-thread would result in high storage and performance overheads). Instead of exploring more sophisticated software modifications, we next explore extending hardware to transparently eliminate the performance impact of such conflicts.

4. Repair via Symbolic Tracking

Our approach to the problem of eliminating conflicts on auxiliary data exploits a few key properties of the operations on such data. First, the control flow in which this data is involved is often highly biased in one direction and relatively insensitive to the exact value of the data (for example, most hashtable inserts do not cause resizes in a well-configured hashtable). This fact implies that remote updates to auxiliary data will often leave the control flow of a transaction unaffected, changing only the output values of that transaction that are dependent on the variable in question. Second, the amount of computation performed on auxiliary data is often small relative to the computation of the transaction as a whole, *i.e.* the dependent slice of the data is much smaller than the dataflow graph of the entire transaction.

Together, these facts suggest an approach that eschews rollback in favor of a lighter-weight *selective repair*. By maintaining a record of the computation performed on auxiliary data, a transaction can allow conflicts to occur on this data without rolling back. As part of the commit process, the transaction repairs output state by atomically reacquiring all locations that have been lost and reperforming the computation dependent on those locations using current architectural values as input. As long as changes in values do not result in control flow changes, the output thus produced will be the same as if the transaction had executed using those input values in the first place.

Hardware support for such selective repair has been proposed in various contexts in the form of *instruction-based replay* [8, 11, 17, 18, 32, 35]. The proposal most relevant in our context is ReSlice [32]. Within the context of thread-level speculation, ReSlice tracks the slice of instructions dependent on a load that is likely to result in a conflict, recording this slice in an instruction buffer. At commit, ReSlice sequentially re-executes the dependent slice of all loads whose input value has changed. As long as all branch outcomes are the same (*i.e.*, control flow is unchanged) and no memory dependences have changed, such replay can successfully repair speculation, allowing it to commit. Such instruction-based replay has additionally been used for latency-tolerant checkpoint-based micro-architectures [17, 35], and some schemes are able to repair limited control-flow changes as well [3, 18].

We build upon this prior work, but we take an alternative approach to repair that is based on *symbolic tracking*. Rather than maintaining the dependent instruction slice of a conflicting load directly, our proposal tracks the symbolic relationships between inputs and outputs. As transactions execute, values are tagged with a symbolic representation of the computation that produced that value. For example, if the processor loads a value and then increments it twice, the processor tracks information indicating that it can calculate the final value by adding two to whatever value the load eventually takes. Such *symbolic values* propagate through registers and memory, while conditional operations result in constraints that the symbolic value must satisfy at the time of commit. If computation that is too complex to track symbolically occurs on a given input, the system constrains that input to remain equal to its original value when reacquired. In this case, our approach still provides performance gains due to an implicit use of lazy [6, 33, 39] and value-based [25, 37] conflict detection, as discussed in Section 5.3. Figure 2(a) gives a high-level example of symbolic tracking.

One appealing aspect of this approach is that the processor collapses the computation necessary to transform inputs to outputs during execution. The degree to which computation can be collapsed and the complexity of doing so, however, is dependent on

the nature of computation being tracked. By choosing what types of computation to symbolically track, a system designer can make an informed tradeoff between complexity and generality. The next section presents RETCON, an instantiation of symbolic tracking tailored to fit the needs of auxilary data that we have observed.

5. RETCON

This section describes RETCON, an instantiation of symbolic tracking that optimizes for design simplicity by restricting the type of computation that it can track symbolically. RETCON restricts operations to have at most one symbolic input and tracks only data (not memory addresses) symbolically. These restrictions allow RETCON to maintain symbolic information efficiently and admit a streamlined commit process. We describe the high-level RETCON algorithm below, followed by a discussion of operational details, implementation optimizations, and benefits that RETCON provides beyond the ability to repair conflicts on auxiliary data.

Key to RETCON are the concepts of *symbolic locations*, *symbolic values*, and *symbolic constraints*. A symbolic location is a memory address that RETCON decides to track symbolically (determined via a predictor trained by past history of conflicts). The symbolic value of a register or memory location is a representation of the computation that was performed to calculate the concrete value of that location. As an example, the symbolic value of the register output of the first load to symbolic location *A* would be "[*A*]". Finally, symbolic constraints are a combination of a symbolic value, a boolean operator, and a constant. An *equality constraint* is a special type of constraint that simply specifies that a given symbolic location must be equal to the value first read for that location by the transaction.

Initiating symbolic tracking. During transaction execution, loads and stores not involved with symbolic repair use the conflict detection mechanism of the underlying TM system. A *symbolic load*, a load that reads from a symbolic location, initiates symbolic tracking of dependent operations by associating a symbolic value with the load's output register (recorded in the symbolic register file, described in Figure 5). The value written to the register file by a symbolic load is the processor's best-guess value for the location (*i.e.*, the architectural value at the time or a value prediction [26, 27, 37]) and execution continues based on that concrete value. The first load to a symbolic location also records the *initial concrete value* of the location (recorded in the *initial value buffer*, described in Figure 5). A load to a symbolic location does *not* set the read bit in the cache, thus allowing the block to be stolen away without triggering a conflict.

Execution with symbolic tracking. Transaction execution is determined entirely by the concrete values, but symbolic values are tracked and propagated from instruction to instruction. If an instruction's specific operation is one that can be tracked symbolically, the symbolic input is propagated to the symbolic value output. The processor performs as much computation as it can during this propagation. As an example, if a register with concrete value 5 and symbolic value "[*A*]+7" is used as input to an increment instruction, the output register's concrete value would be 6 and its symbolic value would be "[*A*]+8".

Symbolic tracking through memory. When writing a register with a symbolic value to memory, both the concrete and symbolic values are recorded and propagated to subsequent loads (using the *symbolic store buffer,* described in Figure 5). Correspondingly, all loads check the symbolic store buffer (as well as the initial value buffer, as described above) in parallel with the data cache. When a load forwards from a store that has a symbolic value, it copies that symbolic value (rather than initializing its symbolic value to the address of the store). In essence, RETCON collapses all store-load forwarding during execution. Figure 6 contains a flowchart of the operation of loads and stores in RETCON.

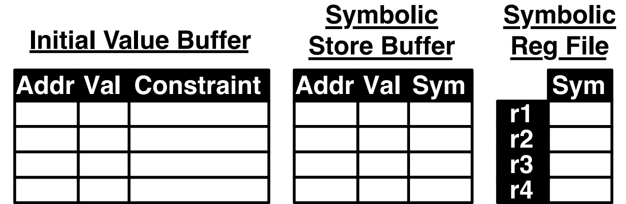


Figure 5. RETCON structures. The **initial value buffer** is a cache-like structure indexed by data address. Each entry contains the address tag bits, the initial concrete value of the symbolic memory location, and the symbolic constraints associated with that memory location (if any). The **symbolic store buffer** records symbolically-tracked stores. It is indexed by data address and accessed like a conventional cache-like unordered store buffer. Each entry contains the address tag bits, the store's concrete value, and the store's symbolic value (if any). The **symbolic register file** records the current symbolic value (if any) for each register. The value recorded in the traditional register file is the concrete value of each register, which is used to guide execution.

Symbolic control-flow constraints. If the source register of a branch has a symbolic value, RETCON adds a symbolic constraint to capture the necessary condition to ensure consistent control flow. For example, a taken branch based on a register with symbolic value "[*A*]+1" is greater than 5 would generate the constraint: "[*A*]+1>5" or, simplified, "[*A*]>4". Non-taken branches record the negation of the branch condition ("[*A*]<=4" in this case). The constraint is recorded in the initial value buffer entry corresponding to the root memory location of the symbolic value.

Equality constraints. An equality constraint is set whenever a symbolic input is used in a way that cannot be tracked symbolically, thus ensuring that any difference in the initial value and final value will result in an abort. Equality constraints are introduced when symbolic values are supplied as inputs to (1) complicated arithmetic operations the implementation has chosen not to track symbolically (*e.g.*, integer divide) or (2) the address calculation of loads or stores (but, critically, not the data input of store instructions). In addition, if an operation has multiple symbolic values as inputs, equality constraints are set on all but one to maintain the invariant that all operations have at most one symbolic input.

Pre-commit repair. As part of the commit process, the system enforces symbolic constraints by re-loading the *final concrete value* for each symbolic location. Symbolic constraints are evaluated using this final concrete value to ensure that the control flow remains valid. Next, RETCON generates the final concrete values for each symbolic register value and symbolic memory value (*i.e.*, stores with symbolic data input register values). This involves updating the concrete value in the register file and writing concrete values into the data cache. To ensure atomicity during the repair process, all loads and stores set the read/written bits in the cache and conflict detection reverts to that of the baseline system (described in Section 2). Once repair has completed, the normal baseline transaction commit commences. Figure 7 describes the repair algorithm in detail, and Figure 8 presents a step-by-step example of RETCON's operation.

5.1 Operational Details

The above text assumes word-granularity aligned memory operations and conditionals that operate directly on register values. However, RETCON must handle features from real-world architectures

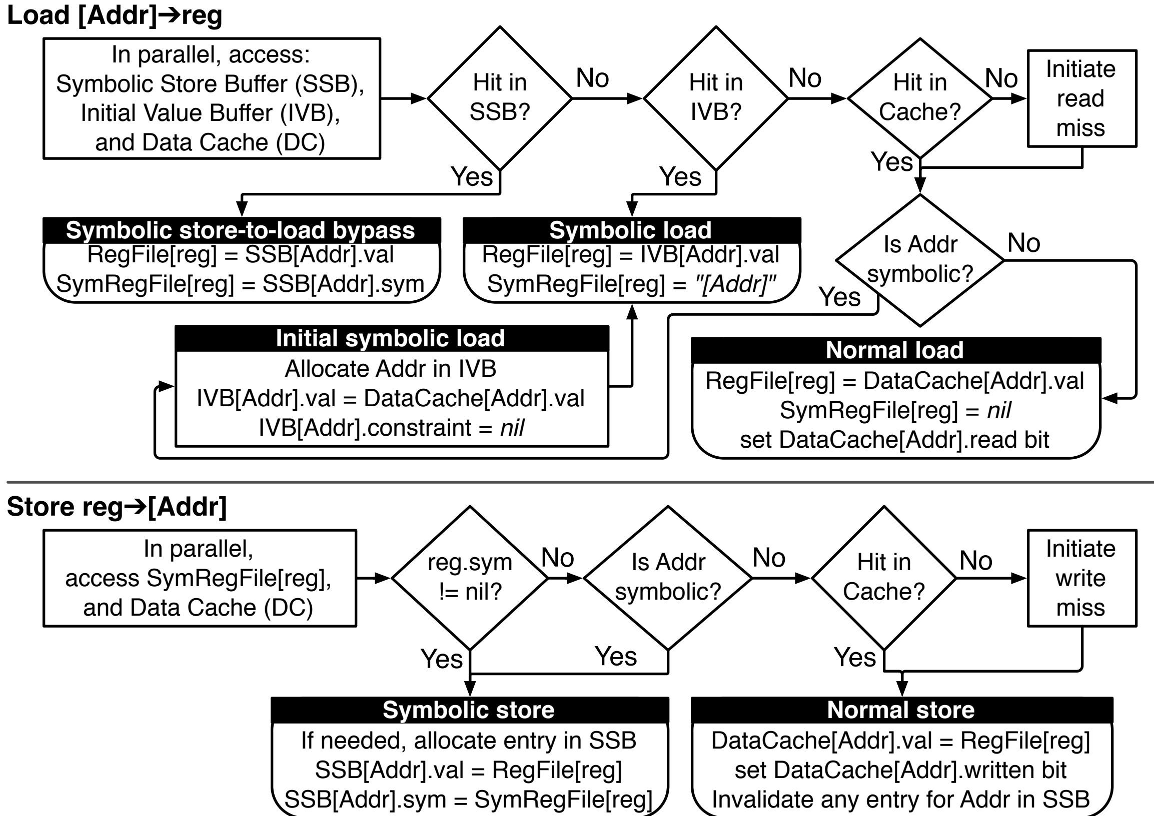


Figure 6. RETCON memory operation flowchart.

RETCON Pre-Commit Process

Step #1. Reacquire all lost blocks to obtain final concrete values, record them in the initial value buffer, and check that all control-flow constraints are satisfied by the current values of the blocks:

```
foreach Address A in Initial Value Buffer (IVB):
  if not already in cache, obtain read permission to A
  set DataCache[A].read bit
  IVB[A].value <- DataCache[A].value
  if new value does not satisfy IVB[A].constraint:
    abort
```

Step #2. Update memory and register state based on the values in the initial value buffer (which as of step #1 above, now contains the final concrete values):

```
foreach Address A in Symbolic Store Buffer (SSB):
  if not already in cache, obtain write permission to A
  set DataCache[A].written bit
  if SSB[A].sym == nil:
    DataCache[A].value <- SSB[A].value
  else:
    DataCache[A].value <- SSB[A].sym evaluated with value from Initial Value Buffer (IVB)

foreach Register R in Symbolic Register File (SRF):
  if SRF[R].sym != nil:
      RF[R].value <- SRF[R].sym evaluated with value from Initial Value Buffer (IVB)
```

Figure 7. RETCON pre-commit repair algorithm. To ensure atomicity of the commit process, the speculatively read/written bits are set when reacquiring lost blocks and before writing values into the data cache. If a conflict occurs during this pre-commit process, the baseline conflict management logic is invoked. Once the above repair has completed, the normal baseline transactional commit commences.

Time	Operation	Initial Value Buffer: Addr	Val	Constraint	Reg	Concrete Reg File: Val	Symbolic Reg File: Sym	Symbolic Store Buffer: Addr	Val	Sym	Data Cache: Addr	Val	R/W
Time ❶	ld [A]→r1	A	5	--	r1	5	A				A	5	--
					r2						B	7	--
Time ❷	r1+1→r2	A	5	--	r1	5	A				A	5	--
					r2	6	A+1				B	7	--
Time ❸	br r2 > 1 (t) (A+1 > 1)	A	5	0<A	r1	5	A				A	5	--
					r2	6	A+1				B	7	--
Time ❹	st r2→[B]	A	5	0<A	r1	5	A	B	6	A+1	A	5	--
					r2	6	A+1				B	7	--
Time ❺	ld [B]→r1 (remote write to A)	A	5	0<A	r1	6	A+1	B	6	A+1			
					r2	6	A+1				B	7	--
Time ❻	r1→r1+2	A	5	0<A	r1	8	A+3	B	6	A+1			
					r2	6	A+1				B	7	--
Time ❼	br r1<10 (t) (A+3 < 10)	A	5	0<A<7	r1	8	A+3	B	6	A+1			
					r2	6	A+1				B	7	--
Time ❽	st r1→[A]	A	5	0<A<7	r1	8	A+3	B	6	A+1			
					r2	6	A+1	A	8	A+3	B	7	--
Time ❾	st 0→[B]	A	5	0<A<7	r1	8	A+3						
					r2			A	8	A+3	B	0	W
Time ❿	commit; load A; verify constr.	A	6	0<A<7	r1	8	A+3				A	6	R
					r2			A	9	A+3	B	0	W
Time ⓫	repair values; write to cache				r1	9	--				A	9	--
					r2						B	0	--

Figure 8. Example of RETCON operation. This diagram shows the symbolic execution and commit operations of a processor due to block A that gets stolen away mid-transaction. The load to register r1 at time ❶ initiates symbolic execution, populates the initial value buffer and the symbolic and concrete register files for r1; note that the read bit in the cache is not set. The symbolic data flows to r2 via the register file at time ❷, which in turn introduces a constraint for A at time ❸. At time ❹, r2 is stored in address B, causing it to be tracked in the symbolic store buffer. The load from B at ❺ forwards from the symbolic store buffer. Also at this time, A is removed from the cache due to a remote request; no conflict is triggered because A's read bit was not set. At time ❻, register r1 is overwritten with a new offset. The branch at time ❼ updates the initial value buffer with an additional constraint on A. At time ❽, the symbolic store to the symbolically tracked address A introduces another store buffer entry. The store to block B at time ❾ is non-symbolic, hence B's entry is invalidated in the symbolic store buffer and the store value is written speculatively into the cache. The commit process begins at time ❿ by fetching A into the cache speculatively, verifying that the new value of A (6) still satisfies the constraint and computing the new value to be stored to A in the symbolic store buffer. In the final step of the commit process, r1's concrete value is written back into the register file and store to A is drained from the store buffer to the cache, and any speculative bits in the cache are flash-cleared.

Parameter	Value
Processor	32 in-order x86 cores, 1 IPC
L1 cache	64 KB, 4-way set associative, 64B blocks, 1-cycle hit latency
L2 cache	Private, 1MB, 4-way set associative, 64B blocks, 10-cycle hit latency
Memory	100-cycle DRAM lookup latency
Permissions-only cache	4KB, 4-way set associative, read-only
Coherence	Directory-based protocol, 20-cycle hop latency
RETCON structures	8-entry initial value buffer, 8-entry constraint buffer, 32-entry symbolic store buffer
RETCON predictor	512-entry table, 8-bit saturating counters, 1:100 up/down training ratio

Table 2. Simulated machine configuration.

such as condition codes, sub-word memory operations and unaligned memory operations. Furthermore, RETCON must prevent transactions operating on potentially-inconsistent data from raising spurious exceptions or entering infinite loops.

To handle condition codes, each condition code register is extended with a symbolic constraint field. When an arithmetic operation with a symbolically-tracked input updates a condition code register, the symbolic constraint of the condition code is updated to reflect the constraint required for that condition code to retain the same value. The form of the constraint depends on the semantics of the condition code. For example, for the "equal-to-zero" condition code, the constraint operator is one of equality if the condition code is set to true and one of inequality if the condition code is set false. When a conditional operation is performed on a condition code being tracked symbolically, the condition code's constraint is added as a constraint on its root address.

To handle sub-word operations, RETCON adds a size field to symbolic values. If store-load communication becomes too complex (*e.g.*, an 8-byte load forwards from two 4-byte stores or a 4-byte store partially overwrites an 8-byte symbolic load), RETCON sets equality constraints on the relevant inputs. Similarly, RETCON treats unaligned memory operations as computation that cannot be tracked symbolically, adding equality constraints to the input word(s) of the operation.

RETCON's approach to the inconsistent data problem is similar to that taken by prior proposals [30]. When a transaction that has lost a block raises an exception, the hardware restarts the transaction with RETCON disabled. To prevent transactions from entering infinite loops due to inconsistent data, RETCON periodically reacquires stolen blocks and validates constraints.

5.2 Implementation Optimizations

The basic structures and operations described above admit several optimizations, which we describe below.

Maintenance of initial value buffer entries at cache-block granularity. To avoid reacquiring a stolen block each time a byte in the block is accessed for the first time, the initial value buffer maintains entries at cache-block granularity. A symbolic load starts symbolic tracking of the entire block. Constraints are maintained by a separate address-indexed and word-granularity buffer.

Compressed representation of equality constraints. RETCON represents equality constraints using per-byte "equality bits" added to entries in the initial value buffer. This optimization works synergistically with the previous one as it reduces pressure on the constraint buffer.

Avoidance of upgrade misses during pre-commit. As described thus far, during the commit process RETCON issues two misses for blocks that it has symbolically read and written (first to acquire the block via a read to check constraints followed by an upgrade miss when writing the block into the cache). To avoid this second miss, RETCON includes per-block written bits on initial value buffer entries. If the written bit is set, the block is acquired with write permission during the initial precommit phase.

Efficient representation of symbolic computation. Limiting the type of computation that can be symbolically tracked to be only additions and subtractions allows optimization of symbolic representations [28]. RETCON (1) tracks symbolic values succintly as "`(input_address, increment)`" pairs, (2) collapses all arithmetic computation on symbolic values to cumulative increments, and (3) represents constraints by succinct intervals. RETCON precisely represents any number of constraints $(\leq, <, =, >, \geq)$ by the *most restrictive interval* bounding the symbolic value. Similarly, RETCON compactly represents any number of not-equal-to constraints—at the cost of some loss of precision—by tracking the interval in which the value cannot reside.

5.3 Other Benefits of RETCON

Although the purpose of RETCON is to enable recovery from remotely-changed inputs, its potential benefits extend further. First, RETCON's property of selectively delaying conflict resolution until transaction commit provides a form of lazy conflict detection [12, 23]; prior work [33, 38, 39] has shown that this strategy can enhance concurrency by allowing readers to commit before a conflicting writer. Second, RETCON's value-based detection of conflicts at a sub-block granularity eliminates conflicts due to false sharing [19] and silent sharing [20] in a similar manner to prior proposals on value-based conflict detection [25, 37]. We analyze the impact of these benefits in Section 6.2.

Researchers have also proposed the usage of speculative values in order to avoid conflicts on true sharing, for example by forwarding values from in-progress transanctions [30] or by predicting values [9, 26, 27, 36]. RETCON does not currently incorporate speculative value forwarding, but it is well-suited to do so. By generating constraints throughout execution and checking that the current architectural value satisfies all generated constraints at commit, RETCON ensures correctness regardless of the source of the value used during execution (*e.g.*, the current architectural value, a forwarded value from an in-progress transaction, or a predicted value).

6. Experimental Evaluation

Our experimental evaluation demonstrates RETCON's ability to eliminate the performance impact of conflicts due to auxiliary data and analyzes the resulting workload scalability improvements. We further examine the amount of state required by RETCON to achieve these results and explore the nature of conflicts that RETCON is unable to repair to give insight on future directions.

6.1 Methodology

As discussed in Section 5.3, RETCON provides multiple benefits over the baseline HTM system described in Section 2 (denoted as *eager* on the graphs): in addition to admitting transaction commits wherein a value read has been changed remotely, RETCON can

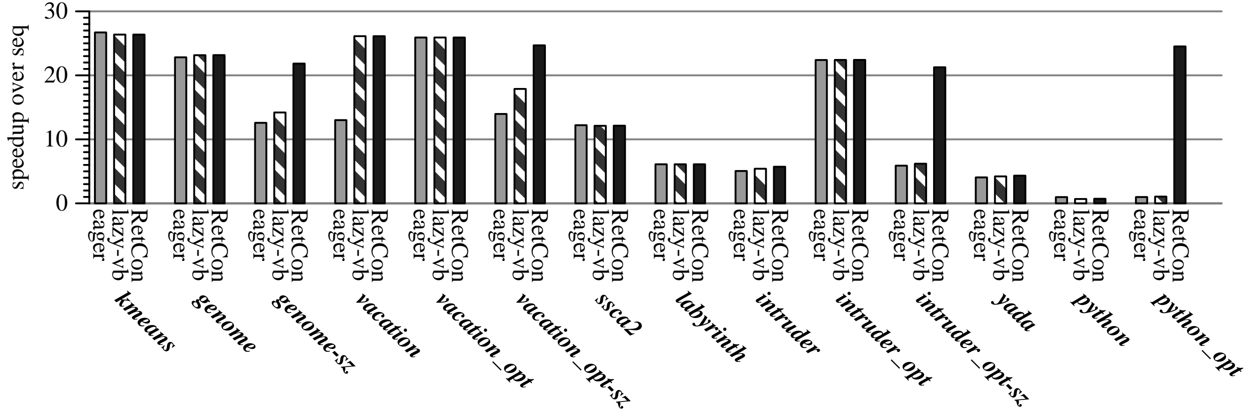


Figure 9. Scalability over sequential execution.

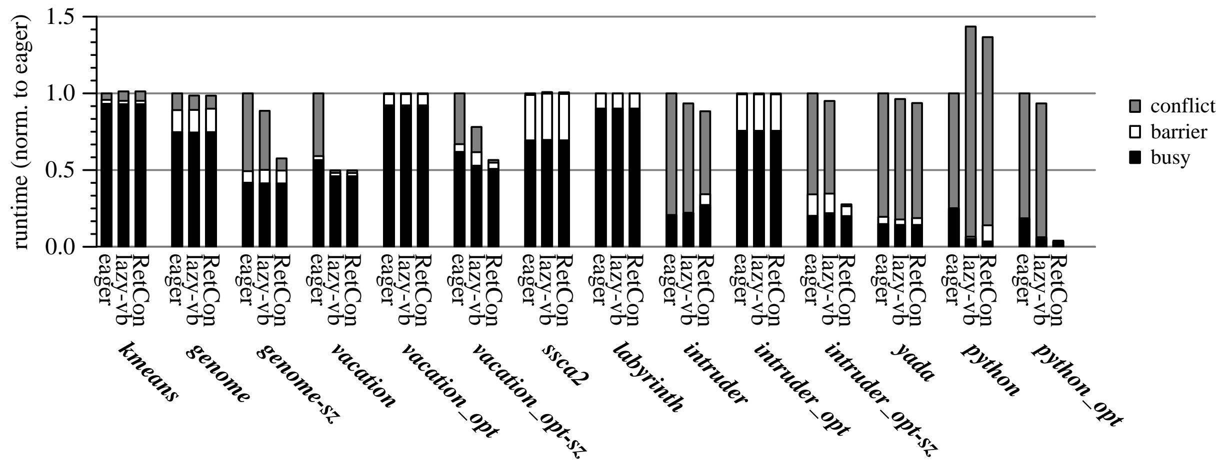


Figure 10. Breakdown of execution time.

reduce conflicts versus the baseline system due to its selective use of laziness and value-based conflict detection. To provide insight into the sources of RETCON's performance gains, we also evaluate a limited variant of RETCON in which values read are not allowed to change: instead, all reads are checked to have the same value at commit (at a precise byte granularity). This RETCON variant, which we refer to as *lazy-vb*, captures commits due to laziness and false/silent sharing but does not allow commits where a value read has been changed remotely.

We model RETCON and our baseline transactional memory implementation (Section 2) using a full-system simulator based on the publicly-available FeS_2 simulator [1], but with further modifications. Our simulator models a 32-processor x86-64 multiprocessor with memory system parameters set to approximate a modern multicore (Table 2). All of our simulated configurations use the "clean-to-L2" version management scheme described in Section 2. The version of RETCON that we evaluate employs the optimizations discussed in Section 5.2. We limit RETCON to support tracking at most eight cache blocks symbolically and maintaining constraints on up to eight word-granularity addresses. Vagaries of our simulator prevented us from easily bounding the size of the word-granularity symbolic store buffer; we analyze this number below, finding that a 32-entry buffer would be sufficient. We configure RETCON's commit-time repair process to reacquire all lost blocks in parallel and reperform stores serially after all blocks have been reacquired; Section 6.3 analyzes the impact of an alternative configuration in which blocks are reacquired serially.

RETCON uses a predictor to determine which data blocks invoke value-based and symbolic tracking. This predictor is an untagged table of 512 eight-bit saturating counters indexed by nine low-order bits of the data block address. The predictor learns based on observed conflicts. To avoid elongating the amount of time that is spent in transactions that will eventually abort, a violated constraint causes the predictor to train down aggressively, requiring the observation of 100 conflicts on that block before attempting symbolic tracking on that block again.

6.2 Impact of RETCON on Performance

Figure 9 presents workload scalability over sequential execution when run under the three configurations. In several cases, the ability of RETCON to repair conflicts changes the qualitative behavior of the workload. Most significantly, RETCON tranforms `python_opt` from a workload that has no scaling for the *lazy-vb* configuration to one that has near-linear scaling (25x on 32 cores) by eliminating the performance impact of reference counter updates. Similarly, RETCON's ability to resolve conflicts on hashtable occupancy field updates without rollbacks changes the characteristics of `genome-sz` (50% speedup over *lazy-vb* by converting a 14x speedup over the baseline into a 21x speedup), `intruder_opt-sz` (243% speedup over *lazy-vb* by converting a 6x speedup into a 21x speedup), and `vacation_opt-sz` (38% speedup over *lazy-vb* by converting a 18x

Application	blocks lost	blocks tracked	symbolic registers	private stores	constr. addrs.	commit cycles	commit stall %	input changed %	slice size
kmeans	0.1 (2.0)	0.6 (3.0)	0.0 (1.0)	1.0 (17.0)	0.0 (0.0)	10.6	2.5	0.0	0.0 (0.0)
genome	0.0 (5.0)	2.9 (8.0)	0.2 (3.0)	0.2 (19.0)	1.2 (8.0)	2.6	0.1	0.9	3.0 (3.0)
genome-sz	0.0 (7.0)	3.9 (8.0)	0.4 (2.0)	0.3 (22.0)	1.8 (8.0)	28.4	0.9	1.2	8.8 (60.0)
vacation	0.7 (3.0)	3.3 (8.0)	0.0 (2.0)	0.2 (7.0)	3.6 (8.0)	52.9	0.7	0.0	6.0 (9.0)
vacation_opt	0.0 (0.0)	0.0 (1.0)	0.0 (1.0)	0.0 (0.0)	0.0 (1.0)	1.0	0.1	0.0	0.0 (0.0)
vacation_opt-sz	0.6 (3.0)	1.7 (5.0)	0.9 (1.0)	0.1 (3.0)	0.1 (6.0)	40.8	2.1	2.2	6.2 (15.0)
labyrinth	0.0 (1.0)	0.1 (1.0)	0.0 (0.0)	0.0 (1.0)	0.0 (3.0)	1.2	0.1	0.0	0.0 (0.0)
intruder	0.3 (6.0)	2.9 (8.0)	0.2 (2.0)	0.6 (15.0)	1.6 (8.0)	29.0	1.5	3.6	3.3 (5.0)
intruder_opt	0.0 (1.0)	0.2 (3.0)	0.0 (1.0)	0.0 (6.0)	0.0 (5.0)	1.1	0.2	0.0	0.0 (0.0)
intruder_opt-sz	0.2 (1.0)	0.4 (3.0)	0.2 (2.0)	0.2 (6.0)	0.2 (6.0)	22.1	3.2	16.6	5.0 (5.0)
yada	0.2 (6.0)	2.2 (8.0)	0.3 (2.0)	1.3 (35.0)	1.1 (8.0)	53.3	0.3	0.1	11.5 (30.0)
python	5.6 (8.0)	8.0 (8.0)	0.0 (1.0)	7.6 (12.0)	7.9 (8.0)	99.2	0.2	42.9	127.5 (230.0)
python_opt	5.1 (8.0)	5.4 (8.0)	0.0 (0.0)	5.9 (10.0)	7.2 (8.0)	319.5	0.8	89.8	199.3 (230.0)

Table 3. RETCON structure utilization and pre-commit runtime overhead. The columns, in order, show the average and maximum (in parentheses) number of *(a)* 64B blocks stolen away during a transaction, *(b)* initial value buffer entries, *(c)* symbolic registers repaired at commit, *(d)* symbolic stores performed at commit, *(e)* symbolic constraints to be checked at commit, *(f)* stall cycles during the pre-commit repair process, *(g)* percentage of transaction execution that is spent in pre-commit repair, *(h)* percentage of committed transactions that had a remotely-changed input, and *(i)* instructions in a dependent slice (*i.e.*, the number of instructions that would need to be buffered and replayed in an instruction replay-based scheme).

speedup into a 25x speedup). Whereas without RETCON the performance of these workloads is significantly worse than the corresponding variants with a fixed-size hashtable, the addition of RETCON makes them insensitive to whether the hashtable is fixed-size or resizable.

To provide insight into the performance improvements of RETCON, Figure 10 presents a breakdown of execution time. We observe that RETCON is able to completely eliminate time spent in conflicts on several workloads that are abort-bound on the baseline system. RETCON can also have secondary effects of reducing load imbalance (*e.g.*, `intruder_opt-sz`) and/or reducing contention in the memory system (the source of the reduction in busy time in `vacation_opt-sz` and `python_opt`).

Analysis of the relative performance of RETCON and *lazy-vb* shows that in most cases RETCON's performance gains come from its ability to repair remotely-changed inputs rather than its usage of lazy value-based conflict detection and false/silent sharing. One exception is `vacation`; further investigation revealed that the increase in performance on this workload is due to laziness. To test RETCON's ability to eliminate conflicts due to false sharing we ran an experiment on `raytrace`, a benchmark from the SPLASH2 suite [40] with a well-known instance of false sharing [23]; RETCON increased speedup over sequential execution from 14x to 21x on 32 cores. These results point to RETCON's ability to provide performance benefits beyond eliminating the negative impact of auxiliary data conflicts.

6.3 RETCON Implementation Considerations

State required by RETCON. Table 3 shows that the amount of state that RETCON tracks symbolically is relatively small. In fact, the initial value buffer ("blocks tracked" column) and constraint buffers ("const. addrs." column) fill on only half the workloads. In addition, the "private stores" column indicates that a 32-entry symbolic store buffer would be sufficient to hold virtually all the private stores of these workloads. We performed experiments with larger buffers (not shown), and all but one of our workloads (`python_opt`) were insensitive to increased buffering. With 16-entry initial value buffer and constraint buffers, the speedup obtained on `python_opt` increased marginally from 25x to 26x, and remained at this value even with 1024-entry buffers. Even with such large buffers the average number of blocks lost remained below 8 in all workloads except for the unmodified `python`, which averaged 11 blocks lost.

RETCON pre-commit process. The "blocks lost" column of Table 3 shows that the number of blocks lost during execution of a transaction is in general quite small on our workloads. In fact, only `python` and `python_opt` lose more than one block per transaction on average. The characteristics of the number of stores that must be reperformed prior to commit (the "private stores" column) are qualitatively similar. As a result, the average number of cycles that a transaction spends in the pre-commit process is quite small (the "commit cycles" column). In fact, the time spent in the pre-commit process is under 1% of total transaction execution time on all but four of our workloads and under 4% on all workloads (the "commit stall %" column). To study the sensitivity of this result to bandwidth with which lost blocks are reacquired, we ran a configuration that conservatively reacquired lost blocks serially (not shown). The only change was a decrease in the performance of `python_opt` from 25x to 20x.

6.4 Analyzing the Limitations of RETCON

As Figure 9 and Figure 10 show, there are three workloads with significant conflicts that RETCON does not greatly affect: the unmodified variants of `intruder`, `yada`, and `python`. The nominal reason that RETCON cannot help on these workloads is that each of these workloads experiences conflicts due to multiple threads simultaneously enqueueing and dequeueing from shared lists. As the variables on which there is contention are used to index into memory (*e.g.*, the pointer to the list head), RETCON is not able to repair these conflicts.

However, these workloads also serve as examples that a repair-based approach is not always the right one to take. In each of these cases, the data elements being operated on are central to the dataflow of the entire transaction. Therefore, a repair that involves selecting a different list element at commit than one previously selected during execution would likely involve redoing most of the work of the transaction, resulting in little savings over a full abort. In such cases, an approach based on forwarding speculative val-

ues (*e.g.*, the pointer to the head of the queue in `intruder`) such as dependence-aware transactional memory (DATM) [30] may be more useful. As discussed in Section 5.3, integrating techniques based on communication of speculative values into RETCON is a potential area of future work.

7. Related Work

Several proposals have focused on mitigating the performance limitations caused by conflicts in optimistic concurrency mechanisms such as transactional memory [16], speculative locking [21, 29], or thread-level speculation [9, 13, 34, 36]. Bobba *et al.* [6] examine the performance pathologies present in conflict resolution schemes, including eager or lazy conflict detection [12, 23, 33], and recent work proposes mixed eager/lazy strategies [33, 38, 39]. Value-based conflict detection has been used in the context of transactional memory for avoiding conflicts due to false sharing [37] as well as for compatibility with library code [25].

Researchers have proposed the use of speculative values to avoid conflicts due to true sharing. Dependence-aware transactional memory (DATM) [30] forwards speculatively written data between transactions. A globally enforced commit order guarantees atomicity and forward progress. DATM prevents aborts due to conflicts when transactions access shared data acyclically (such as incrementing a shared counter once), but aborts when there are repeated accesses to shared data between transactions (see Figure 2). Other proposals have explored value prediction in the context of thread-level speculation and transactional memory [9, 26, 27, 36].

Proposals such as open nested transactions [7, 12, 24], abstract nested transactions [14], and transactional boosting [15] seek to increase concurrency by providing programming abstractions. RETCON differs from these proposals by trading off generality for programmer transparency.

RETCON is inspired by proposals for repair via selective instruction replay [8, 11, 17, 18, 32, 35]. The tradeoffs in symbolic tracking versus instruction slicing are efficiency of representation vs. generality of recomputation. On the side of efficiency, a long transaction might, for example, perform many hashtable inserts or reference count updates; RetCon would perform a constant amount of recomputation vs. the linear amount of state tracked and computation reperformed by an instruction replay-based scheme. In contrast, an instruction replay-based scheme can support replay of more complex types of computation (*e.g.*, ReSlice allows memory addresses to change in re-execution as long as the new address does not change the dataflow of the slice).

Finally, Riley and Zilles [31] explore software restructuring and observe aborts due to false conflicts when studying the behavior of the `PyPy` python interpreter with transactional memory. One key difference in the tasks of parallelizing `PyPy` and parallelizing standard `python` is that `PyPy` uses a conservative garbage collector and thus does not raise the problem of conflicts on reference counts.

8. Conclusions

This work proposes RETCON, a mechanism for hardware transactional memory that allows transactions to lose data during execution and transparently repairs the effects of remote modifications at commit. RETCON uses symbolic tracking of the relationships between inputs and outputs to achieve repair without replay. Our quantitative results indicate that RETCON is able to eliminate the performance impact of conflicts on auxiliary data. When combined with simple software restructurings, this elimination of auxiliary data as a source of performance loss results in speedups of 40% or more on several workloads including a 25x speedup on `python`, a complex program that was never intended to be run in parallel. In the future we plan to investigate the integration of RETCON with mechanisms that use speculative value forwarding such as transactional value prediction and dependence-aware transactional memory to broaden the scope of conflicts that RETCON eliminates.

References

[1] The FeS2 simulator. URL `http://fes2.cs.uiuc.edu/acknowledgements.html`.

[2] The Unladen-Swallow Benchmark Suite. URL `http://code.google.com/p/unladen-swallow/wiki/Benchmarks`.

[3] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary. Transparent Control Independence (TCI). In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[4] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[5] C. Blundell, M. M. K. Martin, and T. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[6] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[7] B. D. Carlstrom, A. MacDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[8] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, June 2009.

[9] M. Cintra and J. Torellas. Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors. In *Proceedings of the Eighth Symposium on High-Performance Computer Architecture*, Feb. 2002.

[10] C. Click, Feb. 2009. URL `http://blogs.azulsystems.com/cliff/2009/02/and-now-some-hardware-transactional-memory-comments.html`.

[11] R. Desikan, S. Sethumadhavan, D. Burger, and S. W. Keckler. Scalable Selective Re-execution for EDGE Architectures. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[12] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[13] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[14] T. Harris and S. Stipic. Abstract Nested Transactions. In *Proceedings of the Second ACM SIGPLAN Workshop on Transactional Computing*, Aug. 2007.

[15] M. Herlihy and E. Koskinen. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2008.

[16] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[17] A. D. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating All-Level Cache Misses in In-Order Processors. In *Proceedings of the 14th Symposium on High-Performance Computer Architecture*, Feb. 2008.

[18] A. D. Hilton and A. Roth. Ginger: Control Independence Using Tag Rewriting. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[19] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence Decoupling: Making Use of Incoherence. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[20] K. M. Lepak and M. H. Lipasti. Temporally Silent Stores. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[21] J. F. Martinez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[22] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, 2008.

[23] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, Feb. 2006.

[24] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting Nested Transactional Memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[25] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary Rewriting Approach to Software Transactional Memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[26] S. M. Pant and G. T. Byrd. Limited Early Value Communication to Improve Performance of Transactional Memory. In *Proceedings of the 23rd International Conference on Supercomputing*, June 2009.

[27] S. M. Pant and G. T. Byrd. A Study of Conflicting Data in TM Programs and Methods to Increase Concurrency Using Value Prediction. In *Proceedings of the Sixth ACM Conference on Computing Frontiers*, May 2009.

[28] V. Petric, T. Sha, and A. Roth. RENO: A Rename-Based Instruction Optimizer. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[29] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[30] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-Aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2008.

[31] N. Riley and C. Zilles. Hardware Transactional Memory Support for Lightweight Dynamic Language Evolution. In *Proceedings of the 21st SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2006.

[32] S. R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.

[33] A. Shriraman and S. Dwarkadas. Refereeing Conflicts in Hardware Transactional Memory Systems. In *Proceedings of the 23rd International Conference on Supercomputing*, June 2009.

[34] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[35] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[36] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proceedings of the Eighth Symposium on High-Performance Computer Architecture*, Feb. 2002.

[37] F. Tabba, A. W. Hay, and J. R. Goodman. Transactional Value Prediction. In *Proceedings of the Fourth ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2009.

[38] R. Titos, M. E. Acacio, and J. M. Garcia. Speculation-Based Conflict Resolution in Hardware Transactional Memory. In *Proceedings of the International Parallel and Distributed Processing Symposium Symposium*, May 2009.

[39] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-Lazy Hardware Transactional Memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2009.

[40] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.