University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS)

Department of Computer & Information Science

11-25-2009

RETCON: Transactional Repair Without Replay

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

Arun Raghavan
University of Pennsylvania

Milo Martin
University of Pennsylvania, milom@cis.upenn.edu

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation

Colin Blundell, Arun Raghavan, and Milo Martin, "RETCON: Transactional Repair Without Replay", . November 2009.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-09-15

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/911
For more information, please contact repository@pobox.upenn.edu.

RETCON: Transactional Repair Without Replay

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 (transactions or critical sections) with the semantics of isolation from one another. This work analyzes bottlenecks to the scalability of workloads that use optimistic concurrency. We find that one common source of performance degradation is updates to auxiliary program data in otherwise non-conflicting operations, *e.g.* reference count updates on shared object reads and hashtable size field increments on inserts of different elements.

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 reference python interpreter from a workload that exhibits no scaling to one that exhibits near-linear scaling on 32 cores.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-09-15

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/911

RETCON: Transactional Repair without Replay

Colin Blundell, Arun Raghavan, Milo M. K. Martin
University of Pennsylvania

UPenn CIS Technical Report MS-CIS-09-15
November 25, 2009

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 (transactions or critical sections) with the semantics of isolation from one another. This work analyzes bottlenecks to the scalability of workloads that use optimistic concurrency. We find that one common source of performance degradation is updates to auxiliary program data in otherwise non-conflicting operations, *e.g.* reference count updates on shared object reads and hashtable size field increments on inserts of different elements.

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 reference python interpreter from a workload that exhibits no scaling to one that exhibits near-linear scaling on 32 cores.

1 Introduction

Over the past decade there has been a great deal of academic and industrial interest in the area of speculative synchronization, both in the form of hardware transactional memory [14] and in the form of speculative parallel execution of traditional lock-based critical sections [12, 19, 28, 34, 36]. Under speculative synchronization the system allows code regions with the semantics of isolation from one another to execute in parallel but detects racing accesses to the same data and rolls back one of the speculative regions involved in the race. Speculation thus makes the degree of concurrency achieved 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 a great deal of research, however, the impact of such speculation on overall workload performance is not clear.

This paper undertakes a workload-driven study of the potential of speculative concurrency to have an impact 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 design configured to avoid cache overflows, has reasonable contention

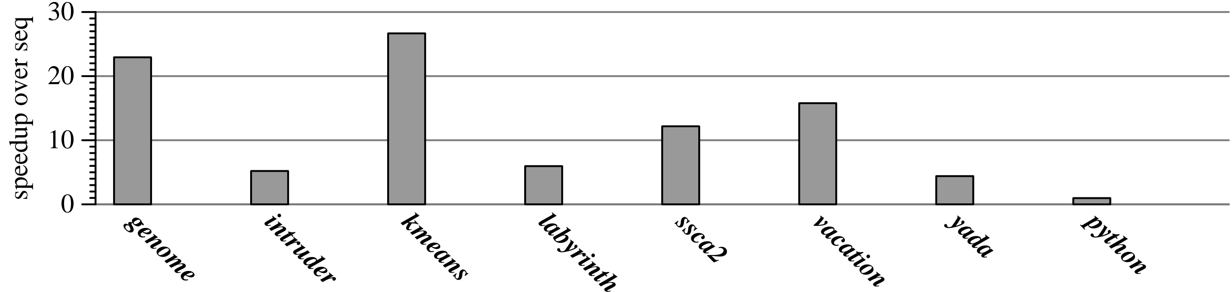

Figure 1: **Scalability of aggressive HTM on 32 processors.** The HTM is configured to eliminate cache overflow aborts, has a robust timestamp-based contention management policy, and models rollback with zero latency.

management policies, and models an efficient zero-cycle rollback latency to avoid artificially magnifying the negative impact of conflicts. Nonetheless, Figure 1 shows that the performance of these workloads is mixed: although some workloads achieve near-linear speedup, half of the workloads obtain a speedup of less than 5x on 32 cores.

We first analyze the scalability bottlenecks on these workloads. As Section 3 details, these bottlenecks come from varied sources. One workload experiences conflicts due to the nature of the algorithm itself. Other workloads experience few conflicts but still scale poorly due to other reasons, illustrating the complexity of creating high-performance parallel programs. Finally, several workloads exhibit significant conflicts that are not related to the main computation of the program itself but are rather on auxiliary data such as reference counters and hashtable size fields.

This last result is 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 as hashtable size fields commonly lead to "a general pattern of updates to peripherial shared values" that significantly impact performance due to "present[ing] 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, 15, 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. 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 (see Figure 2), with more complex computation creating a constraint that the input value remain equal at commit. To maintain symbolic information, RETCON adds a buffer to hold the original values of symbolically-tracked blocks, a buffer to hold constraints, and a buffer to hold symbolically-tracked stores. Our data shows that each of these structures can be small–*e.g.*, 16 entries.

The contributions of this work are as follows:

Analysis of bottlenecks in transactional workloads. We analyze the performance of a set of workloads with coarse-grained synchronization running on a highly-tuned hardware transactional memory system. We find first that many sources of conflicts can be eliminated via simple software restructurings. After performing these restructurings, we identify conflicts on auxiliary data updates as the largest remaining cause of performance loss. We also find that the computation performed on this auxiliary data is usually short and that remote changes to auxiliary data rarely affect the control flow of the local transaction. Inspired by proposals for employing selective re-execution in other domains, we thus propose a repair-based approach

[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.

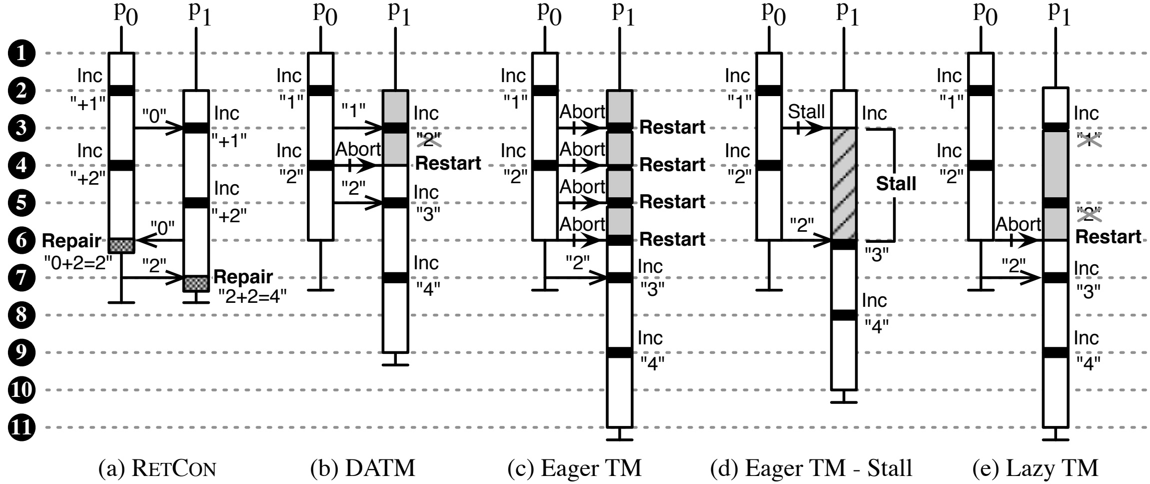

Figure 2: **Comparison of RETCON to other approaches.** P_0 and P_1 begin transactions at times ❶ and ❷ respectively. Each transaction performs two increments to a shared counter variable that is initialized to zero. *(a)* RETCON symbolically tracks the counter address and repairs its value at commit 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.

to eliminating the performance impact of these conflicts.

Proposal of RETCON, a novel approach to repair based on symbolic execution. RETCON flattens dependences during execution by symbolically tracking the relationship between inputs to a transaction and outputs produced by that transaction. Such flattening means that the amount of state required grows in terms of the number of addresses involved rather than the amount of computation performed on those addresses and admits a high degree of parallelism in the commit process.

Demonstration that RETCON can positively impact workload scalability. We quantitatively evaluate RETCON and find that it can essentially eliminate the performance impact of auxiliary data conflicts on our workloads. This results in a speedup of at least 40% on several workloads and in some cases transforms workloads that exhibited no scaling to instead exhibit near-linear scaling when combined with the simple software restructurings described above.

2 Baseline System

Supporting speculative concurrency in hardware requires several elements. First, the hardware must be able to detect racing requests (*i.e.*, conflicts) between two speculative regions. Second, it must be able to roll back to pre-speculative state to restore safety in the case of such racing requests (version management).

A contention management policy may resolve conflicts by stalling instead of rollbacks in some cases to optimize performance. 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 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 [5] as a backing mechanism for transactions that overflow the local cache hierarchy. Our baseline also includes a permissions-only cache [5] to reduce the frequency with which overflows occur. On the workloads analyzed in this work, the addition of 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. Based on prior studies in the literature [6, 29, 33] and our own experimental reconfirmation, we found 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. As described above, the processor needs to support rollback of a speculation to pre-speculative state. Various techniques have been proposed for efficient rollback support [9, 21, 23]. As our goal is to examine the impact that reducing conflicts can have on performance, the baseline is configured to use eager version management and model a zero-cycle rollback penalty. Such a configuration ensures that the baseline is not unfairly magnifying the negative performance impact of aborts.

3 Remaining Performance Challenges in Hardware Transactional Memory

We evaluate the performance of this baseline system on a variety of workloads that stress the performance of speculation (the details of these workloads are given in Table 2). The STAMP benchmark suite [20] 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`, which we could not extract useful conclusions from due to extremely high runtime variability. `labyrinth` includes a code snippet wherein a large shared grid is copied privately within a transaction and then released early

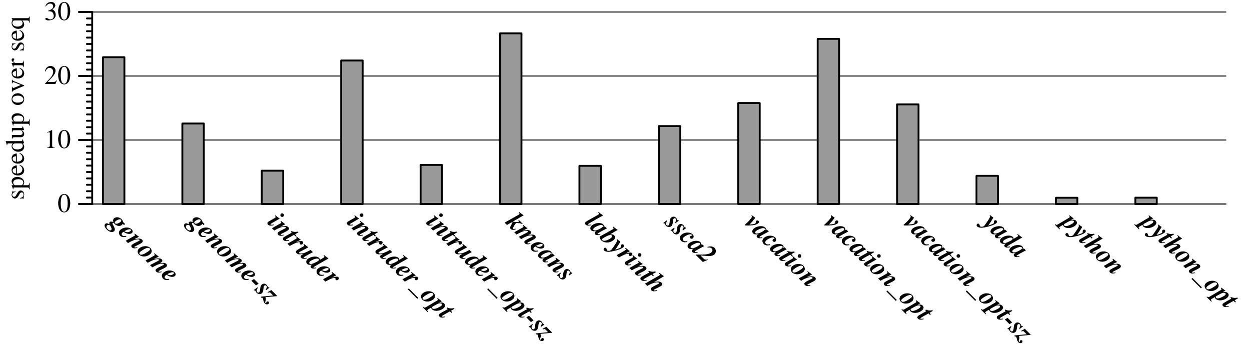

Figure 3: **Scalability of workloads on baseline system before and after applying software optimizations.** The optimizations performed to obtain the variant workloads are detailed in Table 2.

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 support), we modified the code to perform the grid copy before entering the transaction. STAMP includes a hashtable that is by default set to be non-resizable. For all workloads that use this hashtable we also run a variant of the workload in which the hashtable is set to be resizable; this variant has "`-sz`" appended to its name.

In addition to analyzing STAMP, we created a transactionalized version of the standard Python interpreter implementation, `cpython`. 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. We linked `cpython` with Hoard [4] to provide a multicore-friendly drop-in replacement for `malloc`.

As described in Section 1 and shown in Figure 3, the performance of the baseline system on the above-described workloads varies significantly. (The workload variants with "`_opt`" suffixes are described below.) By examining a breakdown of execution time presented in Figure 4, we observe that 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.

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

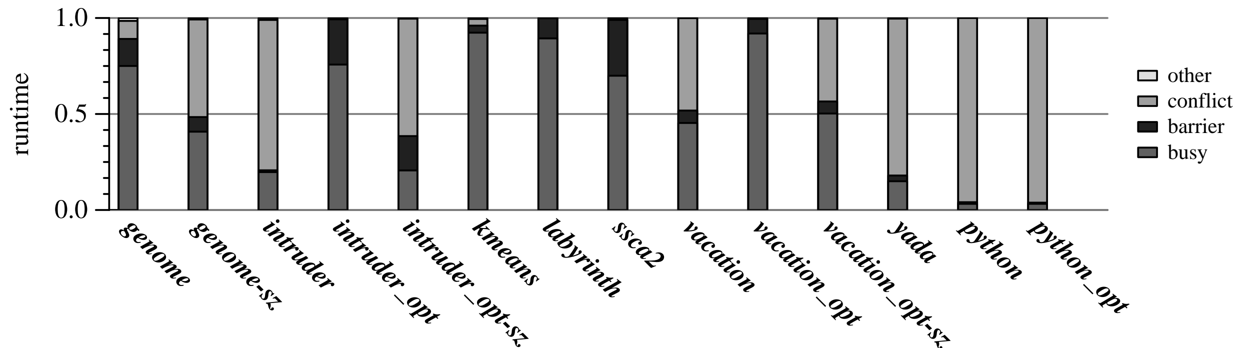

Figure 4: **Time breakdown of workloads on the baseline system before and after applying software optimizations. "busy"** represents all time spent not stalled on synchronization. **"barrier"** represents time stalled at a barrier, an indicator of load imbalance. **"conflict"** represents time spent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

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 a hashtable. 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. As with `genome`, the presence (absence) of the suffix `-sz` for `vacation_opt` and `intruder_opt` implies that the hashtable is resizable (versus fixed-size). An examination of the 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 15x respectively to well over 20x in both cases. For the other variants, however, the picture is less rosy: the software changes do not affect scalability, and Figure 4 indicates that these workloads remain abort-bound after elimination of the most obvious sources of conflicts. We characterize the remaining conflicts in these workloads below.

Characterizing remaining conflicts. After removing the above-described sources of conflicts, the remaining conflicts are as follows. The `python_opt` workload conflicts on reference counts of shared objects;

vacation_opt-sz, intruder_opt-sz, and genome-sz conflict on the internal size field of the resizable hashtable; 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, eliminating the performance impact of these auxiliary operations via software restructuring alone is a daunting task. For example, the requirements of reference counting make straightforward reorganizations such as distributing the count over each thread infeasible, as it is crucial to application performance that reference counts take up little space (they are stored with every object) and that accesses to them be efficient (they are accessed on every object access). Instead of exploring more sophisticated modifications to the software, we next explore extending hardware to transparently eliminate the performance impact of such conflicts.

4 RETCON: Resolving Conflicts without Rollbacks via Symbolic Tracking

This section describes RETCON, a hardware mechanism with the goal of transparently eliminating conflicts on auxiliary or bookkeeping data[4] such as those described at the end of the previous section. To accomplish this elimination, we exploit a few key properties of the operations on such bookkeeping 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 a bookkeeping variable 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 allows data to be stolen away from a transaction *without rolling back* by maintaining enough information about the computation done on that data to be able to *repair* the output state of the transaction at commit. The transaction can then commit atomically by 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.

[4] We will use these two terms as synonyms.

Hardware support for such transactional repair has been proposed in the form of *instruction-based replay* [8, 15, 16, 32, 35]. Within the context of thread-level speculation, ReSlice [32] tracks the slice of instructions dependent on a load that is likely to result in a conflict. These instructions are recorded in an instruction buffer. At commit, ReSlice sequentially re-executes the entire dependent slice of all loads whose memory input value has changed. As long as all branch outcomes are the same (*i.e.*, control flow is unchanged) and no memory dependencies have changed, such replay can successful repair speculation, allowing it to commit. The ReSlice paper contains a full discussion of its implementation details and repair limitations. In addition to ReSlice, such instruction-based replay has been used for load-tolerant checkpoint-based micro-architectures [15], and some schemes are able to repair limited control-flow changes as well [3].

We propose an alternative approach that maintains the information necessary for commit-time repair as *symbolic relationships* between inputs and outputs. Rather than maintaining the dependent slice of a given location directly, we instead *tag* values with the computation that produced that value. For example, if a value is loaded and then incremented twice, the register with the incremented value is tagged with information that informs the processor that it can recreate the value for this register by adding two to whatever value the load eventually takes. Such *symbolic values* then propagate through registers and memory, while conditional operations result in constraints that the symbolic value must satisfy at the time of commit. Figure 2(a) illustrates how such symbolic tracking can reduce conflicts.

One appealing quality of such an approach is that the processor collapses the computation necessary to transform inputs to outputs during execution, as indicated by the above example. Tracking general-purpose computation symbolically in this fashion, however, would likely be complex. Fortunately, the computation performed on auxiliary data is often relatively simple, meaning that it is possible to track this computation symbolically without needing to support fully general computation. Below, we present RETCON, an instantiation of symbolic tracking that restricts the nature of computation that can be performed symbolically to admit an efficient implementation.

4.1 RETCON Overview

Symbolic tracking has two main tasks: (1) maintaining symbolic information during execution and (2) resolving conflicts and updating symbolic outputs at commit. To simplify these tasks, RETCON restricts operations to have at most one symbolic input and tracks only data (not memory addresses) symbolically. As described below, these restrictions allow RETCON to maintain symbolic information efficiently and streamline its commit process.

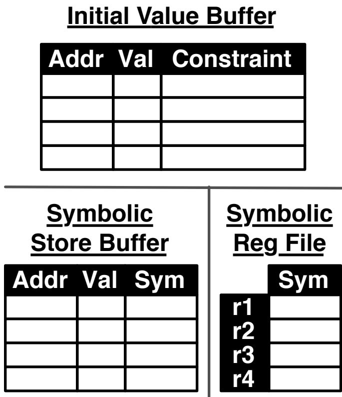

Figure 5: **RETCON structures.** The **Initial value buffer** is a cache-like structure indexed by data address. Each entry contains the address tag bits, the initial concrete value of the symbolic memory location, and the symbolic constraints associated with that memory location (if any). The **Symbolic 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.

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 (*e.g.*, 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 the 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 validation constraint that simply specifies that a given symbolic location must be equal to the value first read for that location by the transaction. Before discussing optimizations and implementation issues, we first describe a general RETCON algorithm that is agnostic to the type or amount of computation that can be tracked symbolically with the exception of the restrictions mentioned above.

4.2 Operation

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 *concrete 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) and execution continues based on that concrete value. The first symbolic load to a location also records the *initial concrete value* of the location (recorded in the initial value buffer, described in Figure 5).

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

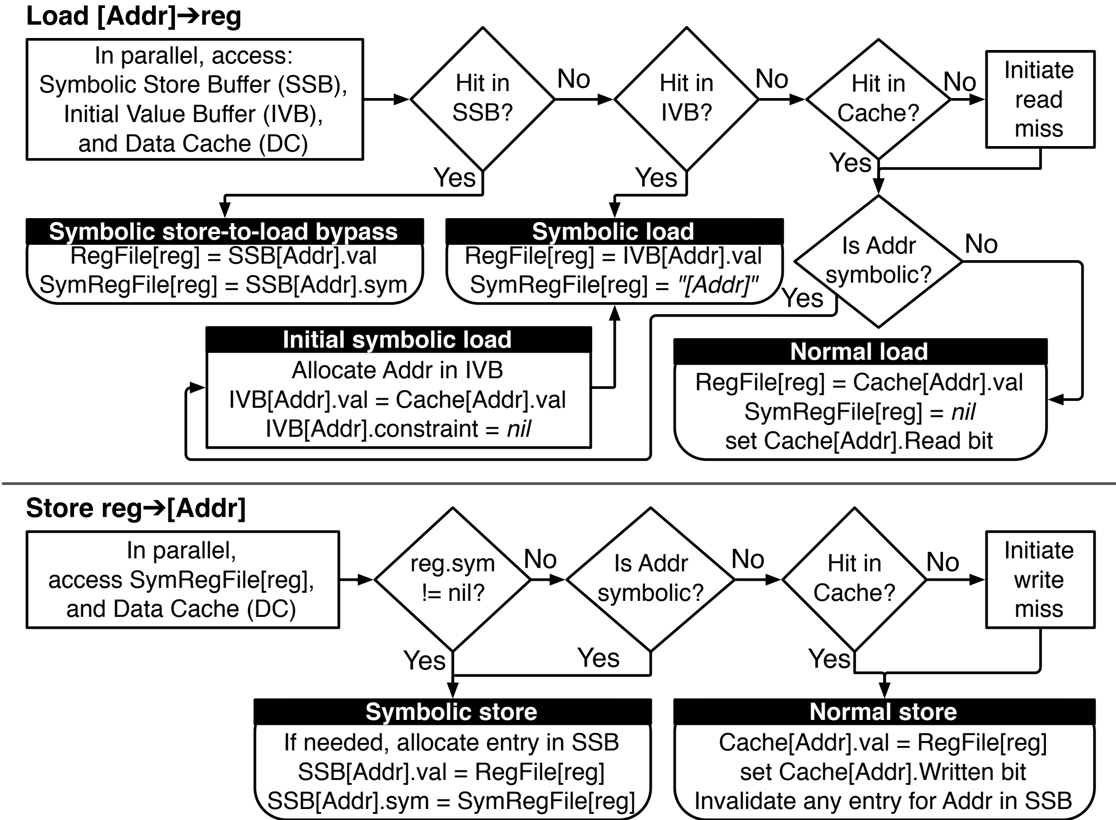

Figure 6: **RETCON memory operation flowchart**

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. Figure 6 contains a flowchart of the operation of loads and stores in RETCON.

Symbolic control-flow constraints. If the source register of a branch has a symbolic value, RETCON adds a symbolic constraint to captures the necessary condition to ensure consistent control flow. For example, a taken branch based on if 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

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.

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. Whenever the implementation uses a symbolic input in a way that cannot be tracked symbolically, an equality constraint is set on the symbolic location from which that input was produced, thus ensuring that any change in the value of that input 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 the symbolic constraints by re-loading the *final concrete value* for each symbolic location that was stolen away during execution. The

Time	Operation	Initial Value Buffer Addr	Val	Constraint	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	5	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.** Figure 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 original value buffer and the symbolic and concrete register files for r1. 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. At time ❻, register r1 is overwritten with a new offset and the branch at time ❼ further constrains r1. 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.

symbolic constraints are evaluated using the 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. Once the above repair has completed, the normal baseline transactional commit commences. Figure 7 describes the repair algorithm in detail.

Example. Figure 8 contains a step-by-step example of RETCON's operation, symbolic tracking through registers and memory, and pre-commit repair process.

4.3 Discussion

RETCON has two properties that enable an efficient implementation. First, the fact that each operation is restricted to at most one symbolic input serves to eliminate an explosion that might otherwise occur in the amount of state required to hold symbolic values as well as the amount of computation required to evaluate these symbolic values to concrete values at commit. Second, RETCON collapses all store-load forwarding during execution: 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. As a result, there is now no dependence between the symbolic values of the load and the store. This fact admits a high degree of parallelism in the commit process.

The above text assumes word-granularity aligned memory operations and conditionals that operate directly on register values. However, real-world architectures feature condition codes, sub-word memory operations and unaligned memory operations. Hence, RETCON must handle these technicalities as well.

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's code 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 or words of the operation.

4.4 Optimizations

The basic structures and operations described above admit several optimizations.

Maintenance of initial value buffer entries at cache-block granularity. To prevent multiple cache misses on symbolic loads to different words of an invalidated cache block, the initial value buffer can maintain entries at cache-block granularity. A symbolic load would now start symbolic tracking of the entire block. Constraints can be maintained by a separate address-indexed and word-granularity buffer.

Compressed representation of equality constraints. Equality constraints can be represented by per-word "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. The baseline RETCON will take two misses on blocks that it has symbolically read and written during the commit process, as it will first acquire the block via a read to check constraints and then cause an upgrade miss when it performs the write to the block. To avoid this second miss, per-block written bits can be added to initial value buffer entries, with a block being reacquired with write permissions in the initial precommit phase if its written bit is set.

Efficient representation of symbolic computation. Limiting the type of computation that can be symbolically tracked to be only additions and subtractions admits several optimizations in representation, including: (1) tracking symbolic values succintly as "`(input_address, increment)`" pairs, (2) collapsing all arithmetic computation on symbolic values to cumulative increments, and (3) representing constraints by succinct intervals. Any number of constraints with $(\leq, <, =, >, \geq)$ can be represented precisely by the *most restrictive interval* bounding the symbolic value. Any number of not-equal-to constraints can be represented similarly by an interval that the symbolic value must remain *without* with some loss of precision.

5 Experimental Evaluation

Our experimental evaluation demonstrates RETCON's ability to eliminate the performance impact of conflicts due to bookkeeping operations and 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.

Parameter	Value
Processor	32 in-order x86 cores, 1 IPC
L1 cache	64 KB, 4-way set associative, 64B blocks
L2 cache	Private, 1MB, 4-way set associative, 64B blocks, 10-cycle hit latency
Memory	100 cycles DRAM lookup latency
Permissions-only cache	4KB, 4-way set associative
Coherence	Directory-based protocol, 20 cycle hop latency
RETCON structures	16-entry original value buffer, 16-entry constraint buffer, 32-entry symbolic store buffer

Table 1: Simulated machine configuration

Workload	Description and Input Parameters
`genome`	From STAMP, gene sequencing program, *g256 s16 n16384*
`genome-sz`	Variant of `genome` with resizeable 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 resizeable 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` with fixed-size hashtable
`vacation_opt-sz`	Variant of `vacation` with resizeable hashtable
`yada`	From STAMP, Delaunay mesh refinement, *-a20 -i 633.2" -a 633.2.ele 633.2.node 633.2.poly*
`python`	Python interpreter, *bm_threading.py (from Google's Unladen-Swallow [2] suite)*
`python_opt`	Variant of `python` with interpreter optimizations

Table 2: Workloads used in RETCON evaluation

5.1 Methodology

RETCON variants. RETCON differs in multiple dimensions from the baseline HTM system described in Section 2: in addition to admitting transaction commits wherein a value read has been changed remotely (the focus of RETCON), RETCON can reduce conflicts versus the baseline system because RETCON's behavior implicitly provides selective lazy conflict detection [6, 27, 33]. By performing conflict detection based on values, RETCON also avoids conflicts due to false sharing [17], silent sharing, and temporally silent sharing [18]. To decouple these aspects of RETCON from our focus on eliminating aborts due to true non-silent sharing, 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. By comparing the performance of these two variants of RETCON, we can isolate the impact of RETCON's ability to commit in the presence of true sharing from its support of lazy and value-based conflict detection.

Experimental setup. We model RETCON and our baseline transactional memory implementation (al-

ready described in 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 1). The version of RETCON that we evaluate employs the optimizations discussed in Section 4.4. We limit RETCON to support tracking 16 blocks symbolically and maintain constraints to 16 addresses. Vagaries of our simulator prevented us from easily bounding the size of the symbolic store buffer; we analyze this number below, finding that a 32-entry buffer would be sufficient. We conservatively assume that RETCON's commit-time reacquire requests are made serially and that commit-time stores are reperformed serially after all blocks have been reacquired. RETCON uses a predictor to determine which data blocks invoke value-based and symbolic tracking. 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.

5.2 Impact of RETCON on Performance

Figure 9 presents the workload scalability over sequential execution when run under the three configurations. In several cases, the ability of RETCON to repair conflicts without rollbacks 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 (30x on 32 cores) by eliminating the performance impact of reference counter updates. Similarly, RETCON's ability to resolve conflicts on hashtable size field updates without rollbacks changes the characteristics of `genome-sz` (66% speedup over *lazy-vb* by converting a 14x speedup over the baseline into a 24x speedup), `intruder_opt-sz` (211% speedup over *lazy-vb* by converting a 6x speedup into a 21x speedup), and `vacation_opt-sz` (26% speedup over *lazy-vb* by converting a 19x speedup into a 24x 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 where the performance improvements of RETCON come from, 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. Most of these savings are due to RETCON's ability to repair local state after remote changes in values, as the *lazy-vb* variant of RETCON experiences a significant speedup over the baseline system only on `vacation` and `vacation_opt-sz`.

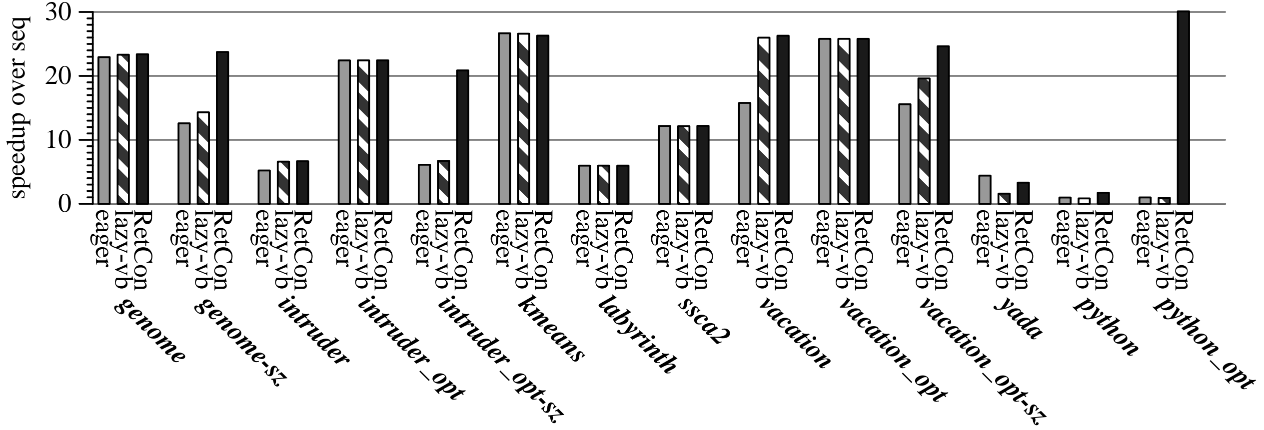

Figure 9: **Scalability over sequential execution.**

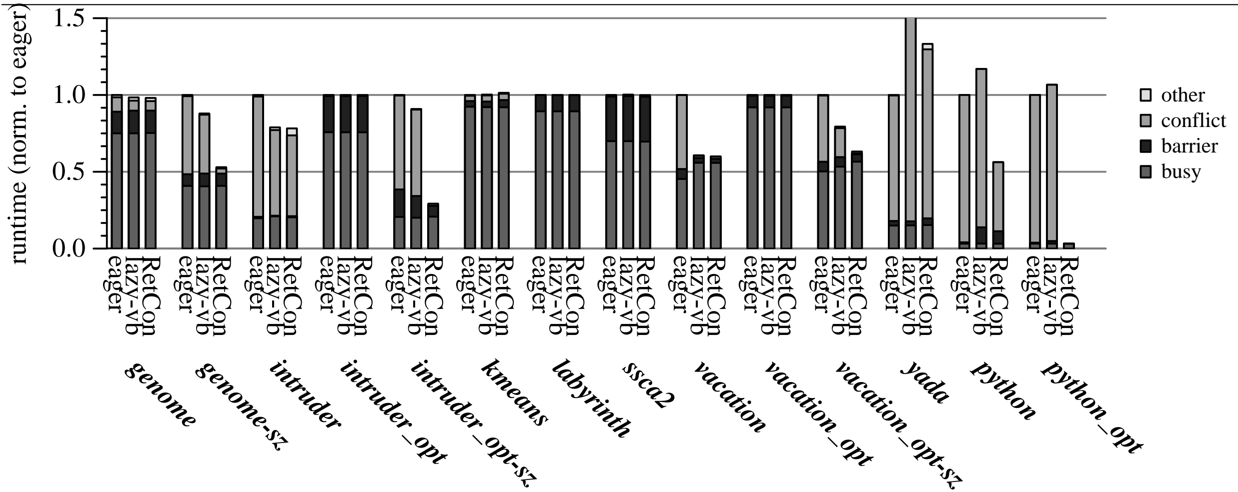

Figure 10: **Breakdown of execution time**

On `intruder_opt-sz` and `python_opt`, reduction in time spent in conflicts has the secondary effect of reducing load imbalance and thus time spent waiting at barriers.

We finally note that there are several abort-bound workloads that RETCON is unable to help (*e.g.*, `yada`). We discuss the reasons for the lack of performance improvement in these workloads in Section 5.4.

5.3 RETCON Implementation Considerations

In this subsection, we analyze the amount of state that RETCON requires and study the characteristics of the RETCON precommit repair process. Table 3 presents various characteristics about transactional behavior under RETCON; we discuss this data below.

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 fills (16 blocks tracked) on only four workloads, while the constraint buffer fills (16 constraints) on only five. In addition, the "private stores" column indicates that a

Application	**blocks lost**	**blocks tracked**	**symbolic registers**	**private stores**	**constr. addrs.**	**commit cycles**	**commit stall %**
bayes	0.3 (6.0)	1.8 (15.0)	0.2 (2.0)	1.5 (34.0)	1.4 (16.0)	26.0	0.0
genome	0.0 (10.0)	1.7 (16.0)	0.1 (1.0)	0.1 (19.0)	1.5 (16.0)	2.0	0.1
genome-sz	0.0 (8.0)	2.0 (16.0)	0.2 (1.0)	0.2 (15.0)	1.5 (16.0)	3.0	0.1
intruder	0.4 (8.0)	2.1 (14.0)	0.0 (1.0)	0.5 (11.0)	1.7 (16.0)	14.0	0.2
intruder_opt	0.0 (1.0)	0.2 (2.0)	0.0 (1.0)	0.0 (3.0)	0.0 (4.0)	1.0	0.2
intruder_opt-sz	0.2 (2.0)	0.4 (3.0)	0.2 (2.0)	0.2 (7.0)	0.2 (5.0)	22.0	3.2
kmeans	0.0 (2.0)	0.4 (2.0)	0.0 (0.0)	0.2 (17.0)	0.0 (0.0)	2.0	0.6
labyrinth	0.0 (1.0)	0.2 (2.0)	0.0 (0.0)	0.1 (2.0)	0.3 (4.0)	1.0	0.2
ssca2	0.0 (1.0)	0.2 (2.0)	0.0 (2.0)	0.2 (1.0)	0.2 (1.0)	1.0	0.8
vacation	1.0 (4.0)	3.8 (12.0)	0.0 (1.0)	0.1 (4.0)	3.9 (12.0)	23.0	0.3
vacation_opt	0.0 (0.0)	0.0 (1.0)	0.0 (1.0)	0.0 (0.0)	0.0 (2.0)	1.0	0.1
vacation_opt-sz	0.6 (3.0)	1.7 (3.0)	0.9 (1.0)	0.1 (3.0)	0.1 (3.0)	48.0	2.5
yada	0.4 (14.0)	2.2 (16.0)	0.3 (1.0)	1.2 (34.0)	1.7 (16.0)	27.0	0.1
python	10.4 (16.0)	13.0 (16.0)	0.0 (0.0)	17.9 (30.0)	15.1 (16.0)	553.0	0.6
python_opt	5.2 (9.0)	5.2 (9.0)	0.0 (0.0)	6.1 (10.0)	7.4 (13.0)	258.0	0.7

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 and *(g)* percentage of time spent during pre-commit repair as compared to the lifetime of a transaction.

32-entry symbolic store buffer would be sufficient to hold virtually all the private stores of these workloads.

RETCON pre-commit process. As the "blocks lost" column of Table 3 shows, 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), even though we have conservatively configured RETCON to reacquire blocks serially at commit and reperform stores serially after acquiring all blocks. In fact, the time spent in the pre-commit process is under 1% of total transaction execution time on all but two of our workloads and under 4% on all workloads (the "commit stall %" column).

Comparison to idealized system. The above data suggests that neither the limits on amount of state that RETCON can track nor the conservative assumptions about the pre-commit repair process greatly impact performance. To validate this conclusion, we ran a variant of RETCON that could track unlimited state, reacquired blocks in parallel at commit, and assumed no latency to reperform stores into the cache at commit. These changes did not significantly impact results on any of our workloads.

5.4 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 the values on which there is contention are used to index into memory.

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 will 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 values (*e.g.*, the pointer to the head of the queue in `intruder`) such as dependence-aware transactional memory (DATM) [30] may be more useful. We discuss tradeoffs between RETCON and DATM as well as the potential for integrating the two techniques in the next section.

6 Related Work

Several proposals have focused on mitigating the performance limitations caused by conflicts in optimistic concurrency mechanisms like transactional memory [14], speculative locking [19, 28], or thread-level speculation [12, 34, 36]. Bobba *et al.* [6] examine the performance pathologies present in abort-based conflict resolution schemes, including eager or lazy conflict detection [11, 21, 33]. Figure 2 illustrates an example situation in which RETCON's repair has an advantage over traditional eager or lazy conflict detection. 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 [24]. Value prediction has been explored in the context of thread level speculation [36]. Recent works on value prediction and transactional memory [25, 26, 37] observe that conflicts may be prevented by predicting false sharing or repeated sharing patterns. However, such schemes are not designed to capture unpredictable shared variables such as reference counts.

Proposals such as open nested transactions [7, 11, 22], early release, and transactional boosting [13] seek to increase concurrency by providing programming abstractions. To be effective, these schemes require significant programmer effort in exposing concurrency and deducing correctness.

Dependence-aware transactional memory (DATM) [30] forwards speculatively written data between transactions. A globally enforced commit order guarantees atomicity and forward progress. DATM pre-

vents 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. Figure 2 illustrates an example situation handled by RETCON that causes an abort due to a cyclic dependence in DATM. As discussed above, however, DATM's speculative value forwarding can in some cases avoid conflicts that RETCON cannot repair. As RETCON already uses value-based conflict detection, a future direction is to incorporate DATM-like speculative value forwarding into the RETCON framework.

ReSlice [32] proposed selective tracking and repair of architected state in the context of speculative execution in checkpointed processors. RETCON is also based on this concept, but tracks dataflow differently and is tailored to track the common idioms described in this paper. RETCON also differs from ReSlice in that the commit process can repair architected state without re-executing dependent chains of instructions. Whereas RETCON avoids treating memory addresses symbolically, however, ReSlice allows memory addresses to change in re-execution as long as the new address does not change the dataflow of the slice.

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 `cpython` is that `PyPy` uses a conservative garbage collector and thus does not raise the problem of conflicts on reference counts.

7 Conclusions

This work proposed RETCON, a hardware mechanism for hardware transactional memory that allows transactions to lose data during execution and transparently repairs the effects of remote modifications at commit. RETCON is inspired by prior work on instruction-based replay but uses symbolic tracking of the relationships between inputs and outputs to achieve repair without replay. Our quantitative results show that RETCON is able to eliminate the performance impact of conflicts on auxiliary data. When combined with simple software restructurings, RETCON's elimination of auxiliary data as a source of performance loss results in speedups of 40% or more on several workloads including a 30x speedup on `cpython`, 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 can be avoided.

Acknowledgements

The authors thank the members of Blundell's dissertation committee, Rajeev Alur, André DeHon, Maurice Herlihy and Amir Roth, as well as Andrew Hilton and Santosh Nagarakatte for comments on this work. This work is supported in part by NSF awards CCF-0644197 and CCF-0905464, donations from Intel Corporation and Sun Microsystems, and an IBM PhD Fellowship for Blundell.

References

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

[2] The Unladen-Swallow Benchmark Suite. 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 *ISCA*, June 2007.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS*, Nov. 2000.

[5] 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 *ISCA*, June 2007.

[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 *ISCA*, June 2007.

[7] B. D. Carlstrom, J. Chung, A. McDonald, H. Chafi, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *PLDI*, June 2006.

[8] S. Chaudhry, R. Cypher, M. Ekmans, 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 *ISCA*, June 2009.

[9] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded Page-Based Transactional Memory. In *ASPLOS*, Oct. 2006.

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

[11] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *ASPLOS*, Oct. 2004.

[12] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *ASPLOS*, Oct. 1998.

[13] M. Herlihy and E. Koskinen. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *PPoPP*, Mar. 2008.

[14] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, May 1993.

[15] A. D. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating all-level cache misses in in-order processors. In *HPCA*, Feb. 2009.

[16] A. D. Hilton and A. Roth. Ginger: control independence using tag rewriting. In *ISCA*, June 2007.

[17] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: making use of incoherence. In *ASPLOS*, Oct. 2004.

[18] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *ASPLOS*, Oct. 2002.

[19] J. F. Martinez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *ASPLOS*, Oct. 2002.

[20] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*, 2008.

[21] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, Feb. 2006.

[22] 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 *ASPLOS*, Oct. 2006.

[23] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *ISCA*, June 2007.

[24] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: a Dynamic Binary Rewriting Approach to Software Transactional Memory. In *PACT*, 2007.

[25] S. M. Pant and G. T. Byrd. Extending concurrency of transactional memory programs by using value prediction. In *CF 2009*, 2009.

[26] S. M. Pant and G. T. Byrd. Limited early value communication to improve performance of transactional memory. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, 2009.

[27] C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-Lazy Hardware Transactional Memory. In *MICRO*, Dec. 2009.

[28] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*, Dec. 2001.

[29] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: transactional memory for an operating system. In *ISCA*, June 2007.

[30] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-Aware Transactions for Increased Concurrency. In *MICRO*, Nov. 2008.

[31] N. Riley and C. Zilles. Hardware tansactional memory support for lightweight dynamic language evolution. In *OOPSLA*, Oct. 2006.

[32] S. R. Sarangi, J. T. Wei Liu, and Y. Zhou. ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing. In *MICRO*, Nov. 2005.

[33] A. Shriraman and S. Dwarkadas. Refereeing Conflicts in Hardware Transactional Memory Systems. In *ICS*, June 2003.

[34] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *ISCA*, June 1995.

[35] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS*, Oct. 2004.

[36] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving Value Communication for Thread-Level Speculation. In *HPCA*, Feb. 2002.

[37] F. Tabba, A. W. Hay, and J. R. Goodman. Transactional Value Prediction. In *TRANSACT*, Feb. 2009.