Acyclic Cost-Effective Requester-Stalls Hardware Transactional Memory

Víctor Nicolás-Conesa, Rubén Titos-Gil, Ricardo Fernández-Pascual, Manuel E. Acacio, Alberto Ros
Computer Engineering Department
University of Murcia
Murcia, Spain
Email: victor.nicolasc@um.es, rtitos@um.es, rfernandez@ditec.um.es, meacacio@um.es, aros@ditec.um.es

***Abstract*—Conflict resolution remains a central challenge in best-effort Hardware Transactional Memory (HTM). Commercial implementations favor requester-wins for its simplicity but suffer from excessive squashed work due to transaction aborts and cannot guarantee progress under contention. Requester-stalls alternatives avoid many of these aborts, yet existing designs face a fundamental burden: cyclic dependencies between stalled transactions. In state-of-the-art requester-stalls policies, conservative cycle detection frequently leads to futile stalls.**

This work presents HighwayTM, a requester-stalls conflict resolution policy that can guarantee progress under contention whilst avoiding cyclic dependencies and futile stalls. Our key insight is that combining a lightweight dynamic priority (which we call a *lane*) with the static, yet simple and already available CoreID yields both cycle freedom and dependency awareness at negligible hardware cost. Lanes are assigned at conflict time and elevate transactions already part of a dependency chain over newcomers, effectively addressing the inherent problems from dynamically established dependencies. The entire mechanism fits in a 7-bit register per core with no centralized clock or global state. Evaluated in gem5 full-system simulation with a variety of benchmarks on a 16-core system, HighwayTM can reduce execution time by 22% and 15% over commercial-like requester-wins and state-of-the-art requester-stalls policies, respectively, while halving futile stalled cycles and reducing stalled transactions to under a third, with consistent results on additional concurrent data structures and in-memory applications.

***Index Terms*—Multicore; Shared Memory Systems; Parallel & distributed Architectures**

I. Introduction

Hardware Transactional Memory (HTM) was proposed over thirty years ago [1], [2]. It offered a simpler alternative to locks for programmers, attracting interest from academia and industry. Consequently, vendors like Intel [3], IBM [4], and Arm [5] started to support HTM. However, despite being available for years, it has not been universally adopted and is not yet a standard feature in most processors. Among the main reasons are some security concerns that led to companies like Intel to disable it in microcode [6], implementation challenges to ensure forward progress [5], [7]–[9], and the low performance and parallelism exploitation typically reached by existing commercial HTM implementations [10]–[12]. However, companies like Intel are picking up again with HTM given the inclusion of TSX extensions in their new products [13] and new transactional instructions [14] that are safe and mitigate Transactional Asynchronous Aborts (TAA) security holes [15].

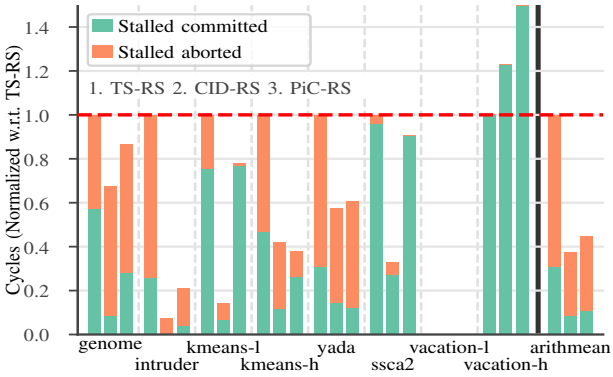


Fig. 1: Committed and aborted stalled cycles. In vacation-l no stalls are produced during the execution.

The remaining concerns have been studied over the years, with proposals covering version management [16]–[21], conflict detection [22]–[24], contention mitigation [25]–[28], and conflict resolution [29]–[34]. Commercial implementations have converged on *lazy* version management and *eager* conflict detection (LE) [4], [35]–[37]. Contention mitigation is typically handled in software or via hardware-software co-design, leaving conflict resolution as the main focus for future hardware advancements [8], [26], [27], [30].

The commercial setting is conservative in this aspect, prioritizing implementation and validation simplicity over performance and parallelism exploitation with the use of requester-wins (RW) policies [3], [36], [37]. Although this is the simplest policy, it is far from optimal, as it suffers from the *friendly-fire* pathology [29], where transactions that are able to retry their execution on abort can lead to livelocks [7], [8], [30].

A different studied approach is to stall the requester until the transaction that detected the conflict releases the address from its read or write set, i.e., until it commits. To stall

This work has been funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (ECHO project, grant agreement No 819134), the Spanish Ministry of Science, Innovation and Universities (MCIN) MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (DAMAS project, grant PID2022-136315OB-I00), and the FSRM/10.13039/100007801 (SCALE project, grant 23039/GERM/25), Spain.

a transaction, the owner of the conflicting address sends a *negative acknowledgment* (NACK) to the requester, which will then retry the access [16], [17], [38]. This policy is known as requester-stalls (RS), even though a conflict can also be resolved by aborting a transaction in some cases. However, cyclic dependencies arise, for example, when a transaction needs to wait for another transaction to release a resource, while the second transaction itself is waiting for the first transaction to release the same resource. To break such a situation, one of the transactions must abort [16], [29], [39].

It is not a trivial task to detect these cyclic dependencies [16], [17], [39] and finding the optimal transaction to abort in order to break the cycle is a current topic of study [31], [32], [34], [39]. State-of-the-art policies rely on heuristics to detect cycles. The most common heuristic is to assign every transaction a timestamp (TS) when it starts to execute. This is the case of LogTM [16], [17] with conservative cycle detection. Fig. 1 shows the execution time of stalled transactions, separating transactions that were committed or aborted. TS-RS suffers from a high number of cycles spent in futile stalls, where a stalled transaction ends up aborting, wasting both the work performed and the time spent waiting (orange bar).

A much simpler approach not yet explored by previous proposals is to use a static identifier for every core, such as the CoreID (CID), to establish a total order between transactions. If a transaction receives a conflicting request from another transaction with higher priority (higher CoreID), the lower priority transaction will simply abort, while if the requester transaction has lower priority, the receiver will send a NACK and stall the requester. Although simple, this CID-RS policy leads to problematic situations of load imbalance [29], [40] and futile stalls, as can also be observed in Fig. 1.

Other more advanced schemes use dynamic priorities to establish a partial order between transactions based on the dependencies they produce with each other, detected at conflict time [31]. A partial order ensures that if a transaction A depends on another transaction B, then B will have higher priority than A, a property that is transitive for all dependent transactions [41]. This is the approach followed in CHATS [31], which assigns a *Position in Chain* (PiC) to transactions at conflict time, creating a partial order that reflects the true dependencies between transactions. This approach can avoid aborting transactions that are not part of a cycle inside the dependency graphs they form, but it does not guarantee forward progress by itself. Thus, it cannot totally avoid cyclic situations and requires a fallback mechanism. PiC-RS suffers from the same futile stalls issue (see Fig. 1).

This work proposes HighwayTM, a novel conflict resolution policy for requester-stalls that effectively exploits the virtues of dynamic partial orders based on detected dependencies with an abstraction that we coin as *lane* while ensuring a total order between transactions. Unlike previous approaches, HighwayTM prioritizes transactions that are already part of a dependency graph. This allows transactions with established dependencies to survive to conflicts with those that have not detected any previous dependency. HighwayTM ensures a total order using the CoreID when the lane priority is equal. Because our proposal respects total order, convergence to a hierarchical, totally ordered dependency graph is guaranteed, ensuring forward progress in case of contention. HighwayTM achieves these results with minimal hardware modifications, using only a small 7-bit register to store the transaction's *lane*, which is the only additional information required to be stored.

Simulation results using gem5 show that HighwayTM reduces average execution time by 22% compared to commercial-like HTM settings and 15% compared to state-of-the-art requester-stalls policies. This is achieved by reducing futile stalled cycles to 50%, aborted transactions by 6%, and the time spent in aborted transactions by 25%, compared to previous requester-stalls policies.

The contributions of this work are the following:

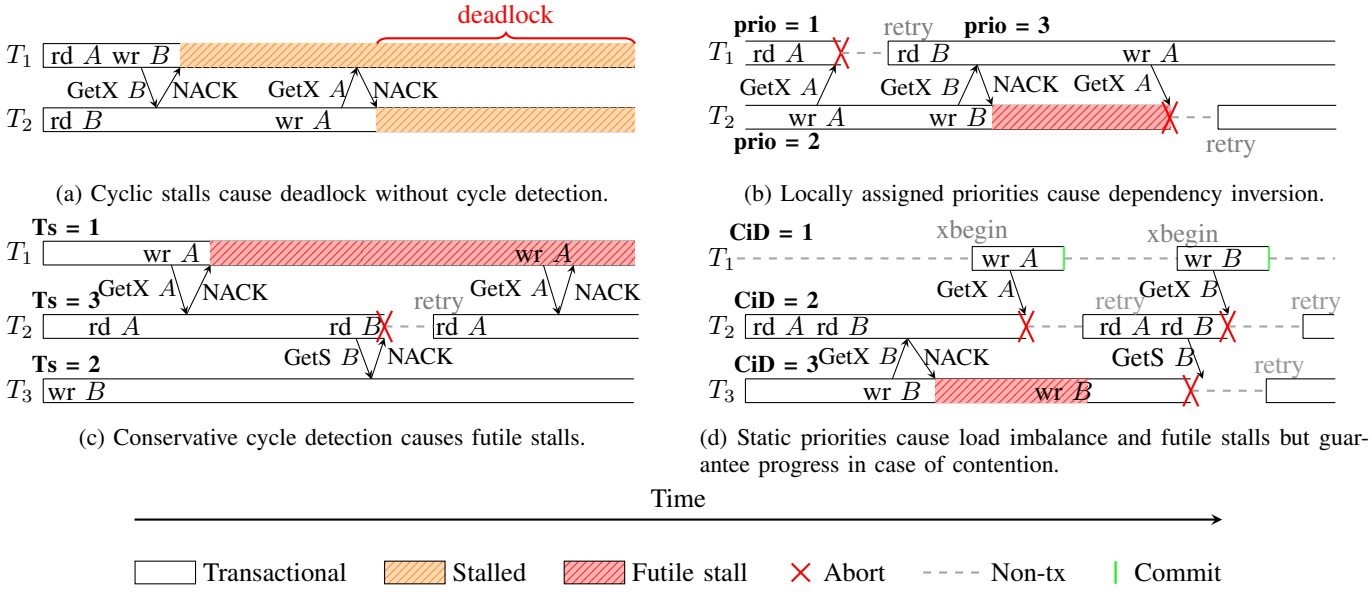
- A detailed analysis of different proposed flavors of the requester-stalls policy with simple heuristics for conflict resolution.
- HighwayTM a cycle-free requester-stalls policy that guarantees forward progress in case of contention.
- A mathematical proof of how HighwayTM policy always causes dependency graphs to converge into directed acyclic graphs.
- An analysis of how HighwayTM compares to the other proposed policies.

II. Background and motivation

Commercial HTM implementations from Intel [3], IBM [4], and Arm [5] are commonly stated to have adopted the *requester-wins* conflict resolution policy [12], [37]: a transaction aborts on receiving an invalidation for a line in its read or write set. It simplifies hardware, as most HTM builds on coherence protocols [3], [8], [37], [42], [43], but suffers well-documented pathologies [29]: *Friendly Fire* (mutual circular aborts) and livelock under contention [10], [12], [42]. Since neither hardware limits nor conflict-induced livelocks can be eliminated, these systems are *best-effort* and must fall back to a non-speculative path such as a single global lock [7], [44]: a critical bottleneck that serializes all transactions and can be triggered by one transaction failing to commit after repeated retries [7], [8], [30].

A. Requester-stalls policies

Rather than mitigating the effects of aborts, *requester-stalls* policies avoid them by issuing a *negative acknowledgement* (NACK), a message already available in state-of-the-art coherence mechanisms [45], to stall the requester rather than abort it [16], [29], [46]. On a conflicting request for a line in its read or write set, a transaction NACKs the requester, which unblocks the directory entry as for a standard data response without switching the line sharers, then retries after some optional backoff [8], [16], [30], [32], [38]. Stalling preserves speculative state [36], [42] but risks deadlock: if T_1 holds A while requesting B and T_2 holds B while requesting A, a cycle happens and both stall indefinitely (Fig. 2a) [29], [38].



(a) Cyclic stalls cause deadlock without cycle detection.

(b) Locally assigned priorities cause dependency inversion.

(c) Conservative cycle detection causes futile stalls.

(d) Static priorities cause load imbalance and futile stalls but guarantee progress in case of contention.

Fig. 2: Pathologies of requester-stalls conflict resolution

1) Cycle detection: Detecting such cycles is non-trivial: transactions usually lack information on the remote transactional state, and sharing it would add hardware complexity (tracking which transactions stall on every address), communication cost (broadcasts scale poorly), and a possible centralized bottleneck [34], [39].

LogTM [16], the canonical requester-stalls design, uses conservative cycle detection. Each transaction acquires a timestamp at its start from a shared, centralized, atomically accessed clock, unique per cycle since equal ages would be indistinguishable. On conflict it always NACKs but compares timestamps, setting a potential_cycle flag when the requester is older; a transaction that later receives a NACK from an older transaction with this flag set assumes it is in a cycle and aborts [16], [17]. Aborting whenever a cycle is possible enforces a total order (older transactions always progress over younger), but yields many false positives that abort transactions not actually in a cycle (Fig. 2c), the *Futile Stalls* [29] pathology: a transaction stalls for another to release an address yet aborts on a conflict with a third, wasting both the wait and the work. Moreover, shared atomic clocks are a bottleneck, scale poorly with core count, and are hard to implement, and carrying a timestamp on every coherence message costs bandwidth and latency [31], [39].

2) Cycle avoidance: Alternatively, cycles can be prevented altogether, an approach not explored in prior requester-stalls work to our knowledge. We identify the coreID [40], a unique per-core identifier already carried by the coherence protocol, as the best candidate for a total order without extra communication. Our implementation of conservative cycle detection with the coreID is simple: on a conflicting request, the transaction aborts if the requester has higher priority (lower CoreID), and otherwise NACKs to stall it.

This is simple yet effective: it reuses a statically assigned total order already present in the system as strapped signals and guarantees progress of at least one transaction under contention. Being static, however, causes load imbalance, since a single high-priority transaction can monopolize progress and serialize the rest. This approach is also subject to the *Futile Stalls* pathology, as a transaction can stall waiting for another transaction, but end up aborting because of a conflict with a third transaction, wasting time and work, depicted in Fig. 2d.

Dependency-aware mechanisms go beyond mere stalling by explicitly tracking and managing dependencies between transactions [31], [32], [34], [39], [47]. CHATS [31], the current state of the art in dependency-aware conflict resolution, introduces a *chaining* mechanism in which each transaction receives a dynamically assigned *Position in Chain* (PiC) upon conflict. PiC orders commits among transactions sharing speculative data, enabling some concurrent progress despite dependencies and preventing unnecessary aborts. Although initially proposed for a request-speculates policy, this priority order can be also implemented in a requester-stalls policy, where instead of forwarding a value, a NACK is sent to stall the remote transaction.

Such schemes come with little cost in hardware, but in exchange they lose the notion of total order. This is translated into two key limitations. First, the policy is susceptible to priority inversion (analogous to classic priority inversion in real-time systems [48]). Suppose that a lower-priority transaction A aborts due to a conflict with higher-priority one B. Upon retry, A may conflict with a different transaction and acquire a priority *higher* than B's, thus inverting the priority ($A \rightarrow B$ becomes $B \rightarrow A$) [41], [49]. This oscillation can repeat indefinitely, preventing either transaction from committing as can be observed in Fig. 2b. Second, dynamically assigned priorities suffer from a *stale information* problem [31]. When multiple transactions issue concurrent requests to the same

TABLE I: Requester-stalls conflict resolution policies

Policy	Fwd. Progress (Contention)	Cycle Freedom	Futile Stalls	Load Imbalance
Conserv. Cycle Det. (TS-RS) [16]	Yes*	Yes	High	No
Static Priorities (CID-RS)	Yes	Yes	High	High
Dynamic Priorities (PiC-RS) [31]	No	No	Med	No
HighwayTM	Yes	Yes	Low	Low

*Forward progress in case of contention is guaranteed under the assumption of unique and strictly increasing timestamps.

address, their messages serialize at the last-level cache. A requester updates its local PiC upon resolution, but its earlier in-flight messages still carry the old priority, causing other transactions to resolve conflicts using outdated data.

This situation can be detected when a transaction receives unexpected values from transactions it has already stalled. However, without precise information, an arbitrary decision could be made to abort the requester. Hence, the mechanism is prone to inconsistent decisions, where a transaction that aborted a remote one aborts then for the same reason, even for the same memory address. Due to the lack of total order, no core has sufficient information to determine a precedence order among inconsistent priorities [41], [49].

B. Challenges in requester-stalls policies

Requester-stalls policies ultimately face the challenge of ensuring both cycle freedom and forward progress while avoiding futile stalls and load imbalance as much as possible. As discussed in this section, there is not an optimal solution to this problem, as the approaches that guarantee forward progress and cycle freedom suffer the most from futile stalls, while the ones that try to mitigate futile stalls and load imbalance the most are the ones that cannot guarantee forward progress.

The main objective of this work is to explore a new point in the design space of requester-stalls policies that effectively exploits the virtues of partial orders based on true dependencies while ensuring a total order between transactions. In Table I we summarize the properties of the different policies we have discussed and what can be expected from our proposal. We observe that HighwayTM is capable of guaranteeing forward progress and cycle freedom leveraging some properties of static priority mechanisms while mitigating the load imbalance issue inherent from that kind of mechanism.

III. HIGHWAYTM

In this section, we present HighwayTM, a cycle-free requester-stalls conflict resolution policy that effectively exploits the virtues of partial orders based on true dependencies while ensuring a total order between transactions. Our policy is specifically designed to only use information that is available at conflict time, without relying on any global or centralized component, thus avoiding the overhead and bottlenecks associated with such approaches. HighwayTM is based on a dynamic priority abstraction that we coin as *lane*, which dynamically prioritize transactions that are already part of a dependency graph, while ensuring a total order using the CoreID when the priority is equal. This design effectively mitigates the *futile-stalls* pathology common in requester-stalls policies while guaranteeing forward progress and cycle freedom.

A. Lane-Based Priority Management

The core mechanism of HighwayTM is a dynamic priority abstraction we term as *lane*, named for its ability to organize transaction flows similar to a highway. Each transaction maintains a lane register L, which acts as the primary conflict resolution metric. Lanes are dynamically assigned at conflict time, reflecting the evolving dependency graph among transactions. A transaction with a higher lane value is considered to have higher priority and in case of a conflict, it must be resolved in its favor. We propagate lane values via NACK and request messages, so that the transaction that detects the conflict has the necessary information to resolve it.

Fig. 3 illustrates an operational example of the lane-based priority mechanism. CX refers to a core that executes a transaction whose coreID is X. The figure is split into three different groups of rows: no lane, stalled lines, and staller lines, which will be explained later in this section. Higher lanes mean higher priority, while lower coreID means higher priority. The line for lanes is wider to clarify that lane-based priority is the one that prevails, and coreID is used as a fallback if the lane mechanism does not specify a clear winner.

All transactions begin execution with an invalid lane value (using a one-bit flag) ($L = NoLane$), be it a transaction that fresh-started or a retrying one. In this initial state, conflicts are resolved using a static base total ordering. It is important to note that it is desirable to use a static total order as we do not store any memento of the previous transactional state (i.e., we never know if a transaction is retrying or the priority it had in such a case). In case a non-static order was to be used, this resolution mechanism could become inconsistent (e.g., if timestamps were to be used, a clock may reset after some time, leading to inconsistencies at the total order). To avoid such complexities and inconsistencies, we derive this priority from the physical CoreID, a unique identifier assigned to each core in the system and used to identify a node in the intercommunication mechanism. If two transactions that both lack assigned lanes conflict, the transaction that detects the conflict will compare its CoreID with the requester's. If the remote transaction has a higher priority (lower CoreID), the local transaction will abort as shown at time 0 in Fig. 3 between Core 0 (requester) and Core 1. Otherwise, it will proceed to assign itself to the initial lane (L_{init}) and send a NACK to the requester with that value as observed in time 1 with core 0 and core 2 (requester).

In HighwayTM, each time the requester receives the NACK, it will check the value of the lane sent, set its own lane value to the received value minus one ($L_{remote} - 1$) and stall. This

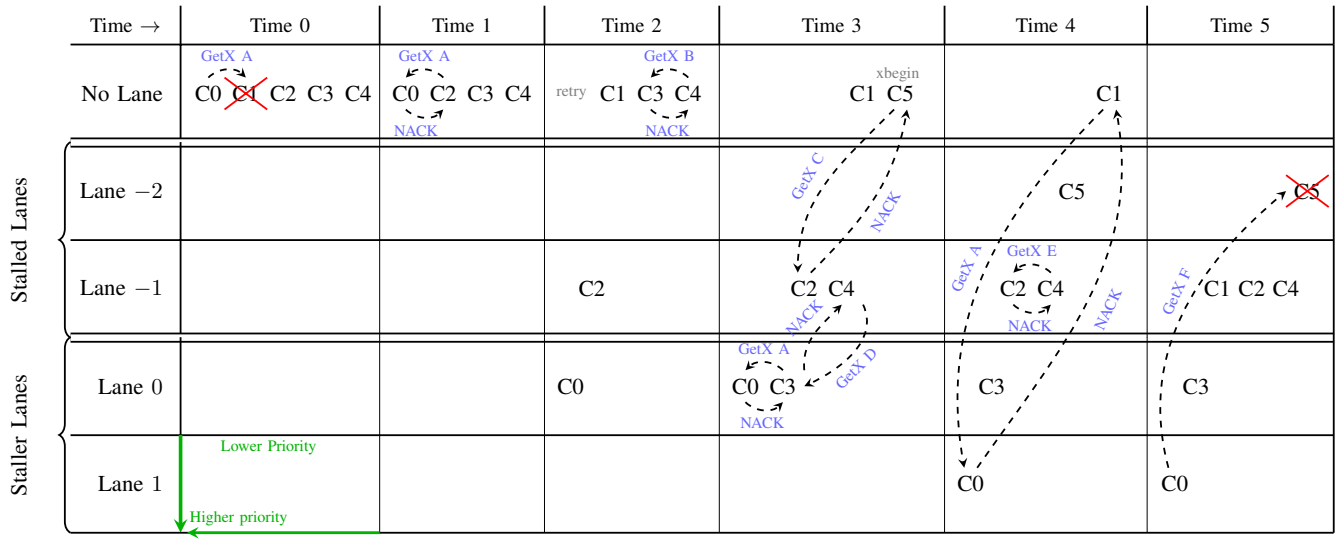


Fig. 3: Operational examples of the lane-based priority management mechanism proposed in HighwayTM

is shown at time 2 in Fig. 3, where after C0 sent a NACK with value L_{init} (lane 0), C2 sets its lane to the received minus one ($L_{init} - 1$ or lane -1). At this point, those two transactions are considered part of a dependency graph, i.e., they are stalled or have stalled another transaction. In any case, both become privileged over other transactions that are not part of any dependency graph. This is reflected by having a valid value in their lane register. With this priority elevation, we intend to avoid the head of the dependency graph (staller) to be aborted by a transaction that is not part of the dependency graph. On top of that, we also try to keep the stalled transaction alive to avoid wasting the stalled time and the work performed.

In those terms, we can consider two kinds of transaction once they enter the dependency graph: the *staller*, which is the transaction that detects the conflict and sends the NACK, and thus it had more priority than the requester; and the *stalled*, which is the transaction that receives the NACK and stalls, and thus it had less priority than the staller. It is important to keep the stalled transactions alive, but we consider more critical to keep the staller alive, since it is the one that is actually making progress and had a higher priority since the beginning. Hence, we prioritize an initial staller over any other transaction that enters the dependency graph as stalled. To do so, we split the dependency graph that can be formed in two parts: those transactions that stall a remote transaction have a priority equal or higher than the initial ($L \geq L_{init}$), those belong to a subset we called ***Staller Lanes*** (as are the ones that always enter the graph stalling other transactions). We call transactions that enter the dependency graph by being stalled as ***Stalled Lanes***, those have a priority lower than the initial lane ($L < L_{init}$). Then, we apply different rules for each of those two groups:

Staller lanes always succeed in case of conflict with a stalled lane. This means that if a transaction belonging to a staller lane requests an address to another stalled lane, the latter will abort. On the other side, if the requester is the transaction that belongs to a stalled lane, then the higher priority one sends a NACK and stalls the requester if it was not stalled before. Then, every priority tier has its own rules. Both have two invariants, if a conflict is detected with a transaction that belongs to the same line, the coreID is used to solve the tie (the transaction with lower coreID wins) and if the requester is of higher priority, the requested transaction will abort as shown at time 5 with C0 requesting an address held by C5 and aborting it. The main difference happens when the requester is of lower priority but same lane. Staller lanes are not constrained by the hard limit imposed by the initial priority assignation, that means, they can climb to higher lanes as needed. That will be required when a NACK is sent to a less-priority requester from the same lane (given by the coreID). The requested transaction will then proceed to switch to its lane plus one ($L = L + 1$) before sending a NACK as shown in Fig. 3 at times 3 and 4 between core 0 and 3. This condition is a good option only for the staller lanes as only ”original” stallers will reach this level, and in that case, the lanes will have to face the coreID limitation to grow higher, converging to a statically assigned hierarchy as will be explained later.

In case of a conflicting request with a transaction that belongs to a lower lane, the lanes will remain untouched as shown at time 3 and 4 with cores 3 and 4, where C3 does not switch its lane after sending a NACK to C4. Stalled lanes on the other side never move from their assigned lanes. They cannot switch to higher lanes to avoid priority escalation, as we do not want any ”originally stalled” transaction to grow over an ”originally staller”. This behavior is shown at time 4 where C2 and C4 conflict, as C4 has less priority than C2, C2 sends a NACK back, but it does not change its lane.

Another case to consider is conflicts with transactions that do not belong to any lane. In any case, if the requester has already a lane and the requested transaction does not, then the requested one just aborts. However, if the requested transaction is in a staller lane, the transaction will enter the lane $L_{init} - 1$,

since it cannot have a higher priority, because that would make that transaction at the same level as the original stallers. That situation is shown in Fig. 3 at times 4 and 5, where C0 sends a NACK as a response to C1 and stalls it, causing it to enter lane -1 ($L = L_{init} - 1$). Transactions that belong to stalled lanes behave differently here too, if they send a NACK as a result of conflict to a transaction that has no lane, then, the transaction enters the graph with the nacker's lane minus one ($L = L_{remote} - 1$), which reflects the dependency between the two transactions. This situation is shown in Fig. 3 at times 3 and 4 where transaction C5 asks C2 for a data and C2 answers back with a NACK situating C5 at the preceding lane -2.

B. Cycle freedom

The mechanism explained above guarantees that HighwayTM never produces a circular chain of stalled transactions. We prove this by modeling dependencies as a directed graph and showing that following any chain of stalls always leads to strictly higher priorities, which makes cycles impossible.

We represent the set of active transactions as nodes in a directed *wait-for graph* G [50], [51]. Whenever a transaction T_j sends a request that gets NACKed by T_i, T_j must wait for T_i to commit and hence release the conflicting address before it can proceed. We draw a directed edge from T_j to T_i to represent this "waits-for" relationship. A cycle in this graph, i.e., T_a waits for T_b, T_b waits for T_c, and T_c waits for T_a, would mean that every transaction in the loop is blocked, and none can ever make progress, signifying the occurrence of a deadlock.

To show that no such cycle can happen, we assign each transaction a rank $\rho(T_i) = (L_i,\ -c_i)$, where L_i is the lane value ($-\infty$ if unassigned) and c_i is the CoreID. We compare ranks lexicographically: the first component (lane) dominates, and the second component (negated CoreID) breaks ties so that a lower CoreID, which HighwayTM treats as higher static priority, maps to a higher rank. The negation is simply a notational convenience: since CoreID 0 beats CoreID 5, writing $-0 > -5$ lets us use the standard "greater means higher priority" convention for both components simultaneously.

The key observation is that *every wait-for edge points from a lower-ranked transaction to a higher-ranked one*. We verify this across all conflict resolution cases:

Case 1 ($L_i = L_j =$ No Lane; $c_i < c_j$): After resolution, T_i sets $L_i \leftarrow L_{\text{init}}$ and T_j receives $L_j \leftarrow L_{\text{init}} - 1$. Since $L_{\text{init}} > L_{\text{init}} - 1$, we have $\rho(T_i) > \rho(T_j)$.

Case 2 (T_i belongs to staller lanes vs. unassigned T_j): T_j enters at $L_{\text{init}} - 1$, while T_i holds $L_i \geq L_{\text{init}}$. The lane gap ensures $\rho(T_i) > \rho(T_j)$.

Case 3 (T_i belongs to stalled lanes vs. unassigned T_j): T_j takes $L_i - 1$, one level below T_i, so $\rho(T_i) > \rho(T_j)$.

Case 4 ($L_i = L_j \geq L_{\text{init}}$; $c_i < c_j$): T_i increments to $L_i + 1$, opening a gap above T_j's unchanged lane.

Case 5 ($L_i = L_j < L_{\text{init}}$; $c_i < c_j$): Both transactions are stalled and share the same lane. The CoreID tiebreak determines the winner, but neither transaction changes its lane– the loser aborts and leaves the graph. No wait-for edge is created, so no invariant needs to hold.

Case 6 ($L_i, L_j \neq$ No Lane; $L_i > L_j$): The higher lane wins directly; no updates needed.

Two properties ensure the invariant holds over time, not just at edge-creation time. Stalled transactions ($L < L_{\text{init}}$) freeze their lane and cannot climb closer to the transaction they wait on. Stallers ($L \geq L_{\text{init}}$) can only increase their lane, which widens the gap with anyone below. Aborted transactions reset to $L = NoLane$ and leave the graph entirely: they stop waiting and release anyone waiting on them.

A subtlety arises with retrying transactions. When a transaction T_b aborts, it drops all its edges and re-enters the system with $L =$ *No Lane*. Even if T_b subsequently climbs the lane hierarchy through new conflicts, potentially surpassing the lane of the transaction T_a that originally aborted it, no cycle can form. The old edge between T_a and T_b was destroyed at abort time, and any new edge between them reflects their *current* ranks, which satisfy the invariant. In other words, the wait-for graph only tracks live dependencies between active transactions, since it does not keep any record of past relationships that were severed by aborts.

Now suppose, for contradiction, that a cycle $T_{i_1} \to T_{i_2} \to \cdots \to T_{i_k} \to T_{i_1}$ exists. Each edge requires the target to outrank the source, so following the cycle yields $\rho(T_{i_1}) < \rho(T_{i_2}) < \cdots < \rho(T_{i_k}) < \rho(T_{i_1})$. This implies $\rho(T_{i_1}) < \rho(T_{i_1})$, which is impossible since no value is strictly less than itself. The wait-for graph is therefore acyclic at all times.

The direct consequence is *guaranteed forward progress on contention*: because the graph has no cycles, at least one transaction sits at the top of every chain (it waits on nobody). That transaction holds the highest rank among any other conflicting transaction, so no incoming conflict can force it to abort or stall. It runs to completion, and once it commits, the transactions waiting on it resume. In this way, contention is guaranteed to be eventually resolved.

It is important to note that although we guarantee forward progress, we do not guarantee that work will be perfectly balanced between cores. A transaction with a low CoreID could be outranked by others with higher CoreIDs in cases of high contention. However, HighwayTM's design tries to mitigate this risk by elevating transactions into lanes once a dependency is detected. In the evaluation we will show that we effectively mitigate the imbalance inherent to the static priorities we used leaving this as a corner case more than a common occurrence.

C. Implementation

The implementation of HighwayTM requires minimal hardware modifications. Each transaction needs to maintain a lane register, which can be implemented as a small 7-bit register (1 bit for validity and 6 bits for lane value) added to the existing transactional metadata. The number of lanes bounded to the width of the register being used for its local tracking. If a transaction receives a NACK with a lane value that underflows the counter, then, the transaction will directly abort, as else it

TABLE II: System parameters

CPU Settings	
Cores	16, out-of-order
Core width	Fetch: 6, Decode: 6, Rename: 6, Issue: 12, Commit: 8
Structure size	ROB: 512, LQ: 192, SQ: 114, IQ: 140
Branch predictor (BP)	L-TAGE [58]
Memory Settings	
L1 D cache	Private, 48KiB, 12-way, 1-cycle hit latency
L1 I cache	Private, 32KiB, 8-way, 1-cycle hit latency
L2 cache	Private, 1.25MiB, unified, 10-way, 4-cycle minimum roundtrip
L3 cache	Shared, 32MiB, unified, 16-way 30-cycle minimum roundtrip
Memory	8GiB DDR4-2400
Protocol	MESI, directory-based
Network Settings	
Topology	Crossbar
Flit size / Message size	16 bytes / 5 flits (data), 1 flit (control)
Link latency / bandwidth	1 cycle / 1 flit per cycle

could break the ordering constrain of the global coreID. An overflow from the upper side can only happen if a transaction won a conflict over other using the global ordering mechanism. In that case, the transaction is not aborted and keeps its maxed out lane value, which has already provided an ordering constrain regarding the aggressor using the coreID, which is static. Our system needs support for negative acknowledgments on the coherence mechanism, something already present in state-of-the-art coherence protocols like AMBA CHI [45]. We need to extend NACK, GetX and GetS messages to carry the lane information (7 bits), which must be forwarded to the L1 cache controller that will detect the conflict and apply the conflict resolution policy. The rest of the system remains unchanged, as all the necessary information for conflict resolution is already present in the system (CoreID and NACKs). The lane values are only used for conflict resolution and do not need to be stored in the cache lines or sent with every coherence message.

IV. Methodology

A. Simulation environment

We conduct our evaluation using the gem5 simulator [52] operating in full-system mode. The guest operating system is an unmodified Ubuntu Linux 16.04 distribution running kernel 4.8.13. We model a 16-core x86-64 processor featuring out-of-order execution and a three-level cache hierarchy. Table II summarizes the system parameters, which have been tuned to approximate the characteristics of an Intel Golden Cove microarchitecture [53]–[55]. To ensure high-fidelity modeling for the on-chip communication, we integrate HeteroGarnet [56] for the interconnection network. Our transactional runtime implements fallback synchronization via eager lock subscription, following Intel RTM guidelines [7], [44], and implements software pre-faulting [57] to prevent page faults from triggering spurious aborts.

B. Policies evaluated

In our study, we perform several evaluations comparing HighwayTM with different conflict resolution policies. All configurations are modeled as best-effort systems that resort to a fallback path that consist on a global lock [7] after a fixed number of retries, ensuring forward progress. To achieve a fair comparison between policies, we use the best configuration in terms of number of retries for every system (note since progress is guaranteed in case of contention, HighwayTM is unbounded, which coincide to be the better configuration). These are the configurations that have been modeled for our study:

Baseline. We utilize a best-effort, requester-wins, HTM model based on Intel Restricted Transactional Memory (RTM), with lazy versioning and eager conflict detection. Speculative state is buffered within the L1 data cache using a *speculatively modified* (SM) bit for each cache line to identify write-set members. Before a block enters the speculative state, its clean version is evicted to the L2 cache, which serves as the preservation level for non-speculative data. Transactional rollback is achieved through gang-invalidating SM-marked lines in the L1, with the coherence protocol supporting silent invalidation of exclusively owned data. Finally, allowing read-sets that exceed private cache capacity, mirroring real-world RTM behavior [3], [11], we employ a perfect signature [24], [59][1] for read-set tracking. The rest of the systems evaluated are built on top of this baseline, modifying only the conflict resolution policy and number of retries before fallback.

Static Core ID-based Requester Stalls (CID-RS). This policy assigns a static priority based on the unique CoreID embedded in coherence messages (lower ID denotes higher priority). Conflicts are resolved by strictly favoring higher priority transactions: the lower priority requester will stall, while higher priority requests force lower priority nodes to abort. In order to cause a stall, a NACK is sent to the lower priority requester. This strict static ordering inherently precludes any dependency cycle and deadlock.

Timestamp-based Requester-Stalls (TS-RS). This configuration employs a timestamp-based conflict resolution policy akin to that used in LogTM [16] but using lazy version management, leveraging the cache for versioning in the same way as the baseline. For this model, we assume a globally synchronized, atomically accessed clock to dispense timestamps, implicitly disregarding hardware complexities of distributed timekeeping and counter overflow. Conflicts are resolved by stalling the requester; the holder sends a negative acknowledgement (*NACK*) forcing the requester to retry the access until successful. To detect and resolve deadlocks without a full resource dependence graph, we employ a heuristic based on these timestamps. Each transaction keeps a *potential_cycle* flag. When a transaction sends a *NACK* to an older transaction (one with a smaller timestamp), it sets this flag, indicating that it has impeded a higher priority transaction. Subsequently, if this transaction receives a *NACK* from an older transaction while the *potential_cycle* flag is active, it aborts. This logic safely breaks potential dependencies: a transaction effectively

[1]We intentionally decided to use a perfect signature so the comparison between the different conflict resolutions was not biased by any other factor, like capacity aborts.

"steps aside" if it is both blocking an older transaction and being blocked by one.

Position in Chain (PiC-RS). This system applies the priority scheme of CHATS [31] to a requester-stalls policy, replacing early value forwarding with priority-based requester-stalls behavior. It utilizes a local *PiC* register to dynamically assign priorities upon conflict detection. When a conflict occurs, the transaction with the higher PiC wins: if the requester wins the conflict, the requested aborts; on the opposite side, it issues a *NACK*, stalling the requester. Priority assignment follows specific rules related to chain formation: Initially, all transactions are Not Connected (NC). When a transaction is requested, if it is NC, it sets its PiC to $init$ (or $Remote + 1$ if the requester has a higher priority) and sends a *NACK*. Conversely, a remote transaction receiving a *NACK* updates its PiC to $Received_PiC - 1$ if it lacks any priority, thereby placing itself logically behind the provider in the dependency chain.

C. Benchmarks

We evaluate the proposed designs using the STAMP benchmark suite [60] configured with medium inputs. Experiments run with 16 threads, aligning with the core count of our simulated system. We exclude *bayes* from the set due to the non-deterministic nature of its search algorithm [61], which introduces significant variability in committed instruction counts. Similarly, *labyrinth* is omitted as it effectively serializes execution in the absence of read set early-release [62] techniques. We further extend the evaluation with two additional sets of benchmarks. *CDSB* (concurrent data-structure benchmarks) [63], [64] comprises six transactional data structures (a binary search tree, deque, hash map, queue, sorted list, and stack) where each transaction performs a single operation on the shared structure.

The second set, which we group under the name *TBench*, consists of four transactional workloads that we implemented ourselves for this study. These are not a pre-existing suite but microbenchmarks we built specifically to place a controlled, tunable form of contention on the conflict-resolution mechanism, complementing the application-level behavior of STAMP and the single-operation accesses of CDSB. Table III summarizes the configuration of each workload for the medium input used in the evaluation.

Bank is a classic account-transfer stress test: each transaction reads and debits one source account and credits several destination accounts in a single fan-out transfer, so its write set spans several balances at once. A global money-conservation invariant, namely that the sum of all balances stays constant, forces the whole transfer to be atomic. The account array is small and the participating accounts are chosen at random, so a few hot balances are written by many concurrent transactions; the fan-out widens each write set and, with it, the conflict footprint, yielding frequent short write-write conflicts. The number of accounts and the fan-out degree are tunable, which lets us dial contention from mild to severe.

TABLE III: Configuration of the *TBench* workloads (*medium* input, 16 threads)

Workload	Configuration
Bank	512 accounts, fan-out 32, 8,192 transactions per thread, 5×10^7 initial balance
Gcon	4,096 vertices, density 32, 8,192 transactions per thread
Tpcc	1 warehouse, 75% new-order transactions, 512 customers per district, 1,024 items per warehouse, 64 order lines, 32,384 transactions
Ycsb	2,048 records, 32 keys per transaction, 60% writes, Zipfian $\theta = 0.75$, 65,536 transactions

Gcon performs parallel graph contraction: it repeatedly merges the two endpoints of an edge into a single supernode, splicing their adjacency lists and rewriting every neighbor to reference the new supernode, all within a single transaction. Because each merge changes which vertices are adjacent, and therefore which later merges collide, successive contractions build data-dependent chains over the evolving graph. This irregular, pointer-chasing pattern mirrors the graph-analytics and machine-learning kernels that are still run predominantly on CPUs, and its contention is structural, intensifying as the graph shrinks into fewer, higher-degree supernodes.

Tpcc is an in-memory implementation of the TPC-C [65] order-entry workload, the canonical OLTP benchmark. Its dominant new-order transaction reads a warehouse and district, increments a hot per-district order counter, and inserts an order together with its line items across the warehouse, district, and order tables. The per-district counter is a sharp single hotspot on which every new-order for that district serializes, while the multi-table insert that follows forms a short cross-table dependency chain, so a single workload exercises both of the contention regimes we study. The warehouse and district counts are tunable, controlling how many transactions collide on each counter.

Ycsb is a transactional port of the Yahoo! Cloud Serving Benchmark [66], a standard key-value-store workload. Each transaction issues a small, configurable mix of read and update operations over keys drawn from a skewed Zipfian distribution, so that a few hot keys absorb most of the accesses while the long tail remains uncontended. Varying the Zipfian skew and the read-to-update ratio sweeps contention continuously from near-uniform to a tight hotspot, and lets us probe how each policy handles read-write versus write-write conflicts.

V. Evaluation

We divide the evaluation into two sections. We begin with low- and medium-contention benchmarks before addressing high-contention scenarios independently. We classify benchmarks considering the average number of aborts per committed transaction, which allows us to better understand the behavior of each policy under different contention levels. The STAMP, CDSB and TBench benchmarks are discussed together throughout, grouped by the contention they generate rather than by suite.

Figure 4 shows the execution time breakdown for all evaluated policies, normalized to the RW baseline. Each bar decomposes execution into non-transactional cycles, committed transactional cycles, cycles holding or waiting for the fallback lock, aborted cycles, and stalled cycles, further split into those belonging to transactions that ultimately committed versus those that were aborted (i.e., futile stalls).

Figure 5 classifies all executed transactions in different groups; first, committed transactions; then, aborted transactions by cause: conflict-induced aborts, capacity-related aborts (abort size), fallback-lock-related aborts (abort FL), and other abort sources. The secondary axis tracks the average time a transaction spends for a given configuration in a given benchmark to abort. This last item is represented with a dot line plot per benchmark.

Figure 6 isolates cycles spent in stalled transactions, normalized to TS-RS, and distinguishes *committed stalled cycles* (productive stalls that led to successful commits) from *aborted stalled cycles* (futile stalls that ended in aborts). This figure measures the efficiency of the stalling mechanism: committed stalled cycles represent productive waiting, while aborted stalled cycles represent pure waste, the *futile stalls* pathology identified by Bobba *et al.* [29].

Figures 7, 8, and 9 present heatmaps of aborted transaction counts per CoreID, with totals annotated above each column. These figures assess how fairly each policy distributes the cost of conflict resolution across cores in general, and for intruder and yada. These two cases are special since are the ones where HighwayTM shows the most significant improvement over the rest of policies, thus, these figures help us to comprehend the reasons.

Finally, Figure 10 reports a strong-scaling study of the average execution time across the STAMP benchmarks as the core count grows from 1 to 16, each point normalized to single-core RW. It assesses how each policy relative behavior evolves as contention intensifies with parallelism.

A. Low and medium contention benchmarks

Several benchmarks barely exercise the conflict-resolution mechanism: STAMP's *ssca2* and *vacation*, together with the CDSB *hash map*, whose operations scatter across buckets. All policies finish within a few percent of RW, confirming that HighwayTM adds no overhead when contention is absent. The CDSB *bst* sits just above this regime: brief contention on a few hot nodes gives the requester-stalls policies a small ($\sim$12%) edge over RW that only CID-RS forfeits to load imbalance (Fig. 4).

Between the negligible cases and genuine moderate contention sits *kmeans-l*: its short transactions update shared cluster centers frequently enough to generate real conflicts, yet each is small, so aborts are cheap and every requester-stalls policy gains only about 6% over RW. HighwayTM is marginally the slowest here ($\sim$4%), as its lane machinery adds a small overhead that such cheap, frequent conflicts do not repay.

Under moderate contention, conflicts occur often enough to matter but are not pervasive: stalling clearly beats aborting and every requester-stalls policy improves over RW, yet a wrong decision does not cascade as it does under high contention. The access pattern still decides how much the policy itself matters. *Genome* runs long, read-heavy transactions whose conflicts are pairwise and one-shot, so no dependency chains form for the dynamic policies to exploit. TS-RS, PiC-RS, and HighwayTM therefore cut execution time by about a quarter and perform near-identically: the gain comes entirely from not discarding a long, hard-won read set on abort, while the static-priority policies spend marginally more time stalling as threads reconverge at the per-round barriers (Figs. 4, 6).

B. High contention benchmarks

Under high contention the conflict-resolution policy becomes decisive: a single wrong decision cascades into wasted stall time, discarded speculative work, or fallback-lock serialization. These workloads fall into two regimes. The first, single-hotspot serialization, funnels every thread through one shared location, so conflicts reduce to a single question of admission order, which a fixed, pre-assigned order settles directly. The second, long dependency chains, links transactions into chains along which the transaction that should be prioritized shifts as execution proceeds, so no static ordering stays correct and resolution must instead track the dependencies as they form. The two regimes reward opposite strategies, and a policy tuned for one tends to misfire on the other.

The first regime comprises *intruder*'s packet dequeue, the CDSB *deque*, *queue*, and *stack*, and the shared accounts of the *bank* application. A static, pre-assigned order is ideal here, it lets one transaction win each contended access and serializes the rest cleanly, as requester-wins does by aborting but without discarding work. CID-RS and HighwayTM therefore lead this regime, cutting execution time by up to 45% over RW, whereas the dynamic PiC-RS and the timestamp TS-RS end up slower than RW: PiC-RS oscillates as aborted transactions re-enter with inverted priorities, and TS-RS turns most of its stalls futile, spending up to 3.7$\times$ RW's time on *stack*. HighwayTM closely tracks CID-RS across these benchmarks, but because it elevates active stallers into lanes rather than trusting the CoreID alone, it acquires the fallback lock far less often and wastes far fewer stalled cycles (Figs. 5, 6).

Intruder is the most demanding of these: under RW it produces the highest absolute abort count in the suite (Fig. 8), yet RW is not the worst policy. TS-RS and PiC-RS are, because their heuristics make transactions stall on the contended red-black tree and then abort anyway, more expensive than aborting outright. CID-RS is fastest, and HighwayTM follows closely while never needing the fallback lock that CID-RS's load imbalance eventually forces; here the spread of aborts across cores (Fig. 8) reflects the benchmark's inherent serialization rather than imbalance, which both static policies handle correctly.

The second regime comprises *yada*'s mesh refinements, the CDSB *sorted-list*, *gcon*'s graph contraction, and the ordered

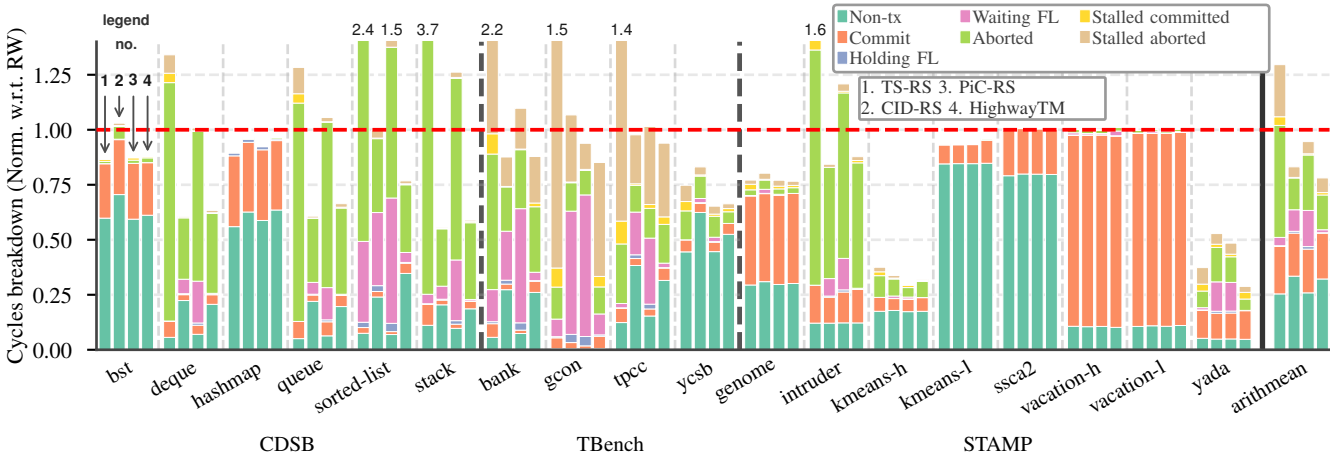


Fig. 4: Execution time breakdown, normalized to RW

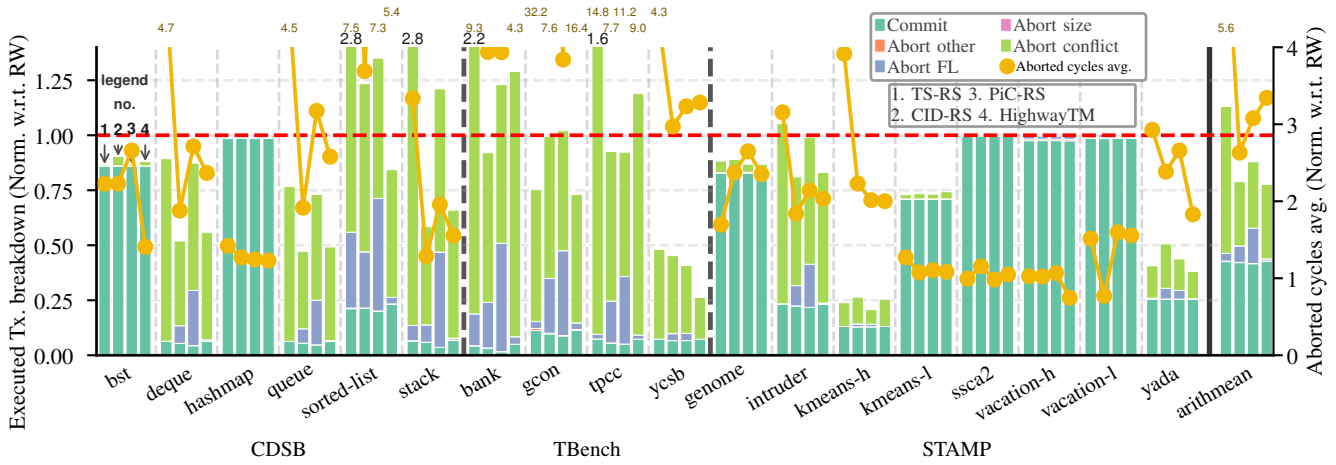


Fig. 5: Abort breakdown by cause, normalized to RW

or key-skewed accesses of *kmeans-h* and *ycsb*. Priorities must now follow the dependencies that form at conflict time, which a purely static order cannot capture, CID-RS even falls below RW on *gcon*, whose vertex merges continually reshape the dependency graph. HighwayTM is fastest on *yada*, *sorted-list*, and *gcon*, and ties the dynamic PiC-RS where access is merely ordered or skewed (*kmeans-h*, *ycsb*): its lanes carry the same dependency information PiC-RS exploits, but anchored to a total order, so they never oscillate the way PiC-RS does on *sorted-list*. TS-RS instead collapses on the traversal-heavy *sorted-list* and *gcon*, aborting long transactions on the mere suspicion of a cycle.

Yada is the extreme of this regime: its transactions exceed 10,000 instructions and run fully transactionally [60], so every abort discards enormous work, under RW it alone aborts 43,060 transactions (Fig. 9). HighwayTM achieves its best result here, finishing in just 28.9% of RW's cycles, with the lowest abort count of any policy (6,931) and a flat distribution across cores. Lane elevation shields a refinement that has already made others wait from freshly dequeued, lane-less transactions; the acyclic guarantee removes the false cycle-breaking that still costs the otherwise-fair TS-RS many futile stalls; and the CoreID tiebreaker governs only fresh transactions, so neither the systematic aborts that cripple CID-RS nor the priority inversion that drives PiC-RS to the fallback lock arise. The stalled-cycle breakdown (Fig. 6) confirms that HighwayTM's stalls here are overwhelmingly productive.

tpcc combines both regimes in a single workload. Its new-order transactions serialize on a handful of hot per-district counters, a pattern that rewards a pre-assigned static order; but each transaction also threads a short dependency chain across the warehouse, district, and order tables, a pattern that rewards dynamic information. HighwayTM is the only policy that captures both: its static entry serializes the counter hotspots much as CID-RS does, while its lane elevation additionally follows the per-order chains, and it almost never falls back to the global lock, so it is the fastest policy on *tpcc*. CID-RS, lacking the dynamic component, trails slightly and is driven

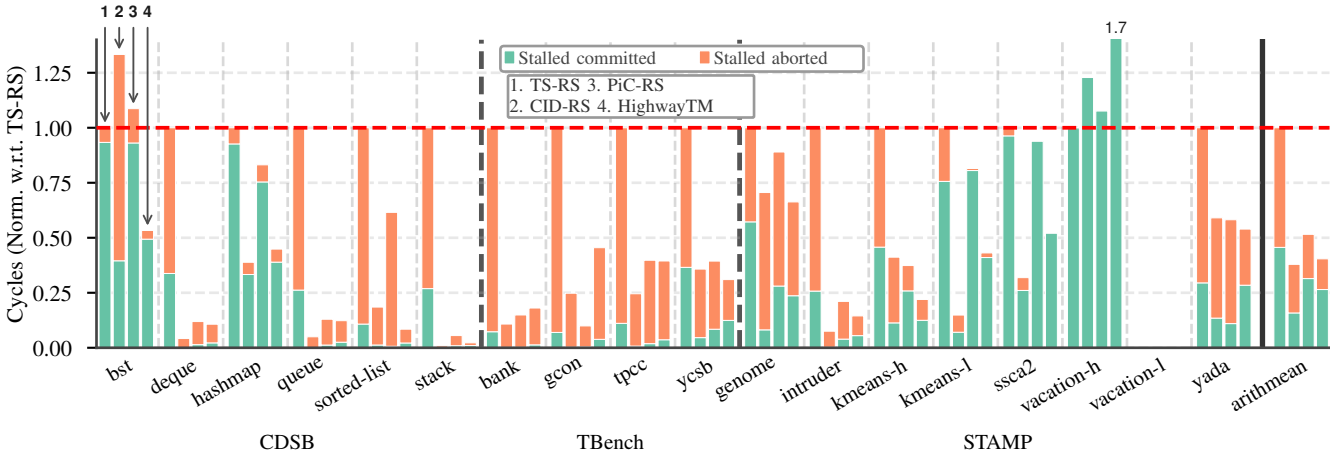


Fig. 6: Committed and aborted stalled cycles, normalized to TS RS

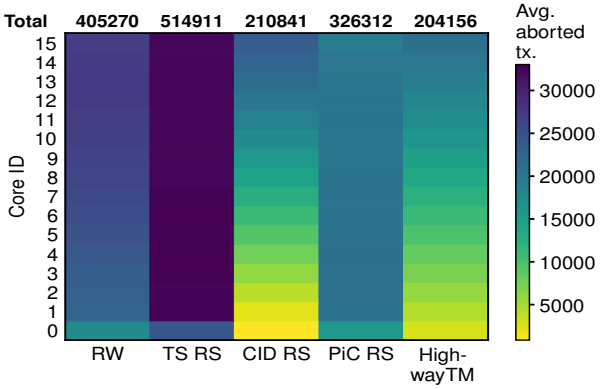


Fig. 7: Average aborted transactions per core across all benchmarks

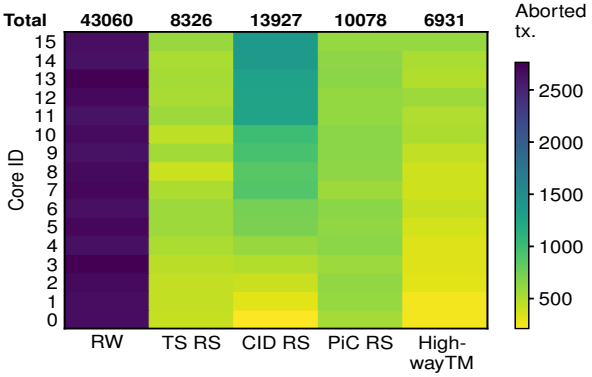


Fig. 9: Aborted transactions per core for *yada*

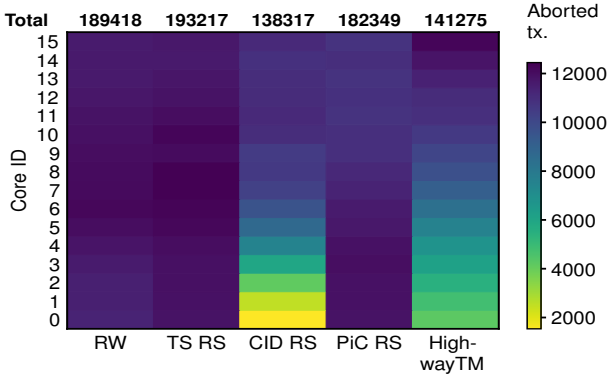


Fig. 8: Aborted transactions per core for *intruder*

onto the fallback lock far more often by its load imbalance; the purely dynamic PiC-RS oscillates on the hot counters and only matches RW; and TS-RS, whose conservative cycle detection turns most of its stalls futile, is the slowest by a wide margin.

Fig. 7 explains why the policy with the fewest aborts is not the fastest. CID-RS reaches the lowest aggregate abort count, but only by imposing a rigid CoreID order whose cost is a steep load imbalance: per-core aborts climb monotonically with the core index, from about fifteen thousand on the highest-priority core to over four hundred thousand on the lowest, a roughly 27× spread. Those persistently deprioritized cores are the ones repeatedly driven to the fallback lock (Fig. 5), whose serialization erases the advantage of the low abort count. HighwayTM accepts more raw aborts than CID-RS, yet because lane position rather than the CoreID settles contended conflicts it never starves a block of cores into the fallback lock, and its stalls stay the most productive of any policy, which is why its aborts yield a lower execution time than CID-RS's despite their greater number.

Across the full suite, HighwayTM is the only policy that is never the worst and always lands at or near the best: it wins outright on the dependency-heavy workloads, matches the strongest static policy on the serializing ones, and adds no overhead when contention is absent, always with the lowest fallback-lock reliance and the fewest futile stalls (Figs. 7, 5). Anchoring dynamic lane priorities to a static total order thus unifies the strengths of both policy families. That the CDSB and TBench workloads, drawn from concurrent data structures

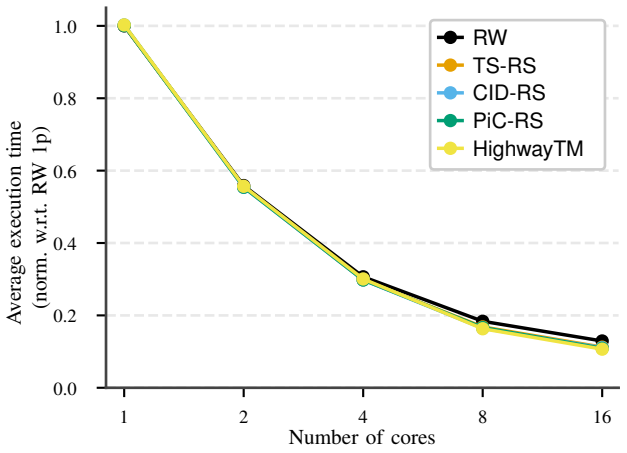


Fig. 10: Strong scaling of average execution time over STAMP, normalized to single-core RW.

and modern in-memory applications, reproduce the very contention patterns already present in STAMP also confirms that STAMP remains representative of these potentially state-of-the-art workloads.

C. Scalability analysis

Figure 10 reports a strong-scaling study: the average execution time across the STAMP benchmarks as the core count grows from 1 to 16, with every point normalized to single-core RW. At one and two cores the five policies are essentially identical: contention is so rare that the conflict-resolution mechanism almost never fires, confirming that none of them, HighwayTM included, adds overhead in the uncontended case. They separate only as cores are added and conflicts become frequent: RW, which discards work on every conflict, falls progressively behind, trailing HighwayTM by 2% at four cores, 13% at eight, and 21% at sixteen. HighwayTM is moreover the fastest policy at both eight and sixteen cores, so its advantage over requester-wins does not merely persist but *widens* with the core count. The fan-out is super-linear: the policies stay within 3% of one another through four cores, where conflicts are still rare enough that the resolution choice is second-order, and they separate sharply only once contention becomes the bottleneck; by sixteen cores HighwayTM achieves a $1.21\times$ speedup over requester-wins on the fixed STAMP workload. Since contention grows with parallelism, this trend indicates that the benefit of HighwayTM will only become more pronounced on the larger core counts of future systems.

D. Interconnect cost analysis

A requester-stalls policy trades discarded work for additional NACK traffic, raising the question of whether it overloads the interconnect. Table IV reports, across the full suite, the NACKs each policy injects together with four network-contention metrics normalized to RW. The two effects pull in opposite directions: every stall sends a NACK, but every

TABLE IV: Interconnect cost: injected NACK traffic and network-contention metrics normalized to RW

Policy	NACKs injected		Network (norm. RW)			
	/run ($\times 10^6$)	/Tx	Flits	Link util.	Pkt. lat.	Q. lat.
TS-RS	7.45	160.7	1.808	1.808	0.982	1.074
CID-RS	1.15	39.3	1.041	1.041	0.954	0.924
PiC-RS	1.74	57.0	1.284	1.284	0.978	1.019
HighwayTM	1.95	58.2	1.089	1.089	0.960	0.953

avoided abort removes the re-execution traffic a discarded transaction would otherwise have generated. TS-RS marks the pessimistic extreme 160.7 NACKs per committed transaction and $1.81\times$ the flits and link utilization of RW because its conservative cycle detection turns many stalls futile and re-issues them. The remaining requester-stalls policies inject an order of magnitude fewer NACKs (39–58 per transaction) and keep flits and link utilization within 4–28% of RW.

HighwayTM injects marginally more NACKs than CID-RS (58.2 vs. 39.3 per transaction) yet settles at only $1.089\times$ the flits of RW, since the aborts it avoids (Fig. 5) more than offset the added stall messages. Packet latency stays at or below the RW baseline for every requester-stalls policy (0.95–$0.98\times$), and HighwayTM's flit-queue latency drops to $0.953\times$: the interconnect is not merely absorbing the extra NACKs but is genuinely less congested, as fewer aborts leave fewer re-executions competing for link bandwidth.

VI. Related Work

Several requester-wins proposals mitigate livelock and unfair aborts without changing the coherence protocol's conflict detection. PowerTM [8] and WoperTM [30] grant a software-assigned *power token* to a single transaction; any conflict involving the token holder is resolved by aborting the opposing transaction, ensuring forward progress while preserving parallelism for the rest. PleaseTM [43] embeds *plea bits* in the coherence responses of transactions, signaling the requester that aborting the responder would discard significant work. LosaTM [67] dynamically adapts the resolution strategy, switching between aggressive and conservative modes based on observed contention. DeTraS [42] delays stores to reduce the vulnerability window of write-set lines. The common thread across these works is to use a more informed policy to decide which transaction to preserve. However, they all rely on heuristics, software hints, or centralized mechanisms to make that decision.

Requester-stalls policies, discussed in Section II, avoid aborts by NACKing the requester. LogTM [16] introduced the paradigm with conservative timestamp-based cycle detection, and LogTM-SE [17] later decoupled transactional state from the L1 cache so that transactions could survive evictions, an orthogonal improvement that benefits any requester-stalls policy built on top of it. Wait-n-GoTM [38] observes that most cyclic dependencies originate from a few heavily contended

cache blocks; rather than resolving all conflicts uniformly, it allows transactions to *go past* most conflicts but serializes predicted cycle-inducing accesses, achieving a hybrid between lazy detection and deadlock-free stalling at the cost of per-core dependency tracking state and a cycle-predictor. Nagabhiru and Byrd [40] use the CoreID already present in coherence messages to establish a total order for very short transactions, showing that even minimal priority information suffices when transactions are small enough to absorb load imbalance. These policies share the core principle of stalling requesters instead of aborting them, but they differ in how they determine which transaction to stall and how they manage the resulting dependencies.

The closest work to our proposal comprises dependency-aware mechanisms that explicitly manage inter-transaction ordering to allow conflicting transactions to coexist rather than aborting one. DATM [34] was the first to observe that many conflicts are conflict-serializable and do not require an abort: it builds a dependency graph and forwards speculative data between ordered transactions. OmniOrder [39] removes the centralized component by embedding conflict serialization into the directory protocol itself: coherence messages carry the history of speculative updates so that receivers can independently determine ordering constraints. Pant explore two complementary paths: VPTM [47] predicts the values a stalled transaction will eventually read, allowing it to continue speculatively, while LEVC [32] forwards tentative values early to shorten stall durations. CHATS [31], the current state of the art in this kind of policies, introduces chained transaction ordering through dynamically assigned Position-in-Chain (PiC) values upon conflict, enabling a partial ordering of dependent transactions without forwarding speculative data and without a centralized graph. As we discussed in Section II, the lack of a total order makes CHATS susceptible to priority inversion and stale-information pathologies that prevent it from guaranteeing forward progress by itself.

VII. Conclusion

We presented HighwayTM, a simple requester-stalls conflict resolution policy for hardware transactional memory. HighwayTM reconciles the benefits of dependency-aware prioritization with the guarantees of a total order between transactions. Our key insight is that transactions already involved in a dependency chain have accumulated speculative work worth preserving. HighwayTM realizes this principle through a lane abstraction that dynamically elevates transactions upon conflict, while restricting others to enter in the mechanism, and falling back to a static CoreID-based ordering when no dependency information exists. We formally proved that the resulting wait-for graph is acyclic at all times, which guarantees forward progress under contention. In addition, this technique should also be considered for other general parallel programming techniques that rely on wait-based synchronization mechanisms. Our evaluation shows that HighwayTM reduces average execution time by approximately 15% over state-of-the-art requester-stalls policies. The gains are most pronounced under high contention, where HighwayTM reduces execution time by up to roughly 70% over requester-wins while almost never resorting to the fallback lock. HighwayTM cuts futile stalled cycles and reduces the average time spent in aborted transactions by roughly 20% on average, confirming that lane elevation steers the stalling mechanism toward productive waits. Importantly, under low contention HighwayTM introduces no measurable overhead, and its entire hardware cost amounts to a single 7-bit register per core and 7 additional bits on three coherence message types.

References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," *SIGARCH Computer Architecture News*, vol. 21, no. 2, pp. 289–300, 1993.

[2] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, "Multiple reservations and the oklahoma update," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 4, pp. 58–71, 1993.

[3] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013, pp. 1–11.

[4] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike, "Transactional memory support in the ibm power8 processor," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 8:1–8:14, 2015.

[5] Arm Ltd. (2020) Arm transactional memory extensions documentation. [Online]. Available: https://developer.arm.com/documentation/101028/0012/16--Transactional-Memory-Extension--TME--intrinsics

[6] Intel Corporation. (2023) Intel® transactional synchronization extensions (intel® tsx) memory and performance monitoring update for intel® processors. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000059422/processors.html

[7] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy, "Improved single global lock fallback for best-effort hardware transactional memory," in *Transact 2014 Workshop*, 2014, p. 54.

[8] D. Dice, M. Herlihy, and A. Kogan, "Improving parallelism in hardware transactional memory," *Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 1, pp. 1–24, 2018.

[9] Intel Corporation. (2013) Intel® transactional synchronization extensions (intel® tsx) programming considerations. WaybackMachine/Intel. [Online]. Available: https://web.archive.org/web/20131031020527/http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/GUID-54A84479-DEC6-4D2F-9895-46D278EDA820.htm

[10] V. Nicolás-Conesa, R. Titos-Gil, R. Fernández-Pascual, M. E. Acacio, and A. Ros, "On the interactions between ILP and TLP with hardware transactional memory," *Microprocessors and Microsystems (MICPRO)*, vol. 104, p. 104987, 2024.

[11] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenstrom, "Performance and energy analysis of the restricted transactional memory implementation on haswell," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 615–624.

[12] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 127–136.

[13] Intel Corporation. (2026) Intel® xeon® 698x processor specification. Intel. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/243857/intel-xeon-698x-processor-336m-cache-2-00-ghz/specifications.html

[14] ——. (2023) Intel® architecture instruction set extensions and future features. [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf

[15] ——. (2026) Intel® transactional synchronization extensions (intel® tsx) asynchronous abort. WaybackMachine/Intel. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-tsx-asynchronous-abort.html

[16] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: Log-based transactional memory," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2006, pp. 254–265.

[17] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Logtm-se: Decoupling hardware transactional memory from caches," in *Symposium on High Performance Computer Architecture (HPCA)*, 2007, pp. 261–272.

[18] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2005, pp. 316–327.

[19] M. Lupon, G. Magklis, and A. González, "Fastm: A log-based hardware transactional memory with fast abort recovery," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2009, pp. 293–302.

[20] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2, p. 102, 2004.

[21] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas, "Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors," *Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 3, pp. 247–279, 2005.

[22] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson, "Si-tm: Reducing transactional memory abort rates through snapshot isolation," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS)*, 2014, pp. 383–398.

[23] S. Park, C. J. Hughes, and M. Prvulovic, "Forgive-tm: Supporting lazy conflict detection in eager hardware transactional memory," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 192–204.

[24] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[25] W. N. Scherer III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Symposium on Principles of Distributed Computing (PODC)*, 2005, pp. 240–248.

[26] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon, "Robust contention management in software transactional memory," in *OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, ser. OOPSLA '05, no. CONF, 2005.

[27] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *International Symposium on Microarchitecture (MICRO)*, 2009, pp. 156–167.

[28] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic scheduling for hardware transactional memory," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 3, pp. 1–40, 2017.

[29] 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," *IEEE Micro*, vol. 35, no. 2, pp. 81–91, 2007.

[30] V. Nicolás-Conesa, R. Titos-Gil, R. Fernández-Pascual, M. E. Acacio, and A. Ros, " WoperTM: Got Nacks? Use Them! ," *IEEE Computer Architecture Letters (CAL)*, vol. 24, no. 01, pp. 157–160, Jan. 2025. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/LCA.2025.3565199

[31] ——, "Chaining transactions for effective concurrency management in hardware transactional memory," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 840–855.

[32] S. M. Pant and G. T. Byrd, "Limited early value communication to improve performance of transactional memory," in *International Conference on Supercomputing (ICS)*, 2009, pp. 421–429.

[33] R. Titos, M. E. Acacio, and J. M. Garcia, "Speculation-based conflict resolution in hardware transactional memory," in *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009, pp. 1–12.

[34] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *International Symposium on Microarchitecture (MICRO)*, 2008, pp. 246–257.

[35] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "Eazyhtm: Eager-lazy hardware transactional memory," in *Symposium on Microarchitecture (MICRO)*, 2009, pp. 145–155.

[36] A. Armejach, R. Titos-Gil, A. Negi, O. S. Unsal, and A. Cristal, "Techniques to improve performance in requester-wins hardware transactional memory," *Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, pp. 1–25, 2013.

[37] T. Zhang, "Designing practical software bug detectors using commodity hardware and common programming patterns," Ph.D. dissertation, Virginia Tech, 2020.

[38] S. A. R. Jafri, G. Voskuilen, and T. Vijaykumar, "Wait-n-gotm: improving htm performance by serializing cyclic dependencies," *SIGPLAN Notices*, vol. 48, no. 4, pp. 521–534, 2013.

[39] X. Qian, B. Sahelices, and J. Torrellas, "Omniorder: Directory-based conflict serialization of transactions," *SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 421–432, 2014.

[40] M. Nagabhiru and G. T. Byrd, "Achieving forward progress guarantee in small hardware transactions," *IEEE Computer Architecture Letters (CAL)*, vol. 23, no. 1, pp. 53–56, 2024.

[41] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[42] R. Titos-Gil, R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Detras: Delaying stores for friendly-fire mitigation in hardware transactional memory," *Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 1, pp. 1–13, 2021.

[43] S. Park, M. Prvulovic, and C. J. Hughes, "Pleasetm: Enabling transaction conflict management in requester-wins hardware transactional memory," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2016, pp. 285–296.

[44] Intel Corporation. (2023) Intel® 64 and ia-32 architectures optimization reference manual: Volume 1. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/671488?explicitVersion=true&fileName=248966-Software-Optimization-Manual-048.pdf

[45] Arm Ltd., *AMBA® 5 CHI Architecture Specification*, Arm Limited, Aug. 2021, version E.b, document number: IHI 0050E.b. [Online]. Available: https://developer.arm.com

[46] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 37, no. 10, pp. 5–17, oct 2002.

[47] S. M. Pant and G. T. Byrd, "Extending concurrency of transactional memory programs by using value prediction," in *International Conference on Computing Frontiers (CF)*, 2009, pp. 11–20.

[48] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers (TC)*, vol. 39, no. 9, pp. 1175–1185, 1990.

[49] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*, 1989, pp. 215–226.

[50] R. C. Holt, "Some deadlock properties of computer systems," *ACM Computing Surveys*, vol. 4, no. 3, pp. 179–196, 1972.

[51] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–78, 1971.

[52] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[53] K. Bhati. (2021) In-depth analysis of intel 12th generation core alder lake, thread director, and other tech. SparrowsNews. [Online]. Available: https://sparrowsnews.com/2021/08/20/intel-12th-generation-core-alder-lake/

[54] Chips and cheese. (2021) Popping the hood on golden cove. [Online]. Available: https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/

[55] A. Scherer and G. Sohi, "Special issue on hot chips 33," *IEEE Micro*, vol. 42, no. 03, pp. 6–6, may 2022.

[56] S. Bharadwaj, J. Yin, B. Beckmann, and T. Krishna, "Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling," in *Design Automation Conference (DAC)*, 2020, pp. 1–6.

[57] R. Titos-Gil, R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Pftouch: Concurrent page-fault handling for intel restricted transactional memory," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 145, pp. 111–123, 2020.

[58] A. Seznec, "A 256 kbits l-tage branch predictor," *Journal of Instruction-Level Parallelism (JILP)*, vol. 9, pp. 1–6, 2007.

[59] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, Second Edition*, 2nd ed. Springer Cham, 2022.

[60] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *International Symposium on Workload Characterization (IISWC)*, 2008, pp. 35–46.

[61] A. Dragojevic and R. Guerraoui, "Predicting the scalability of an stm: A pragmatic approach," in *SIGPLAN Workshop on Transactional Computing*, 2010.

[62] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, 2003, pp. 92–101.

[63] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[64] S. Patel, R. Kalayappan, I. Mahajan, and S. R. Sarangi, "A hardware implementation of the MCAS synchronization primitive," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2017, pp. 918–921.

[65] Transaction Processing Performance Council, "TPC Benchmark C, standard specification, revision 5.11," https://www.tpc.org/tpcc/, 2010.

[66] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010, pp. 143–154.

[67] C. Fu, L. Wan, and J. Han, "LosaTM: A hardware transactional memory integrated with a low-overhead scenario-awareness conflict manager," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 12, pp. 4849–4862, 2022.

[68] V. Nicolas, "RING-5: Reproducible instrumentation for numerical graphics for gem5," RING-5 Development Team, 2026. [Online]. Available: https://github.com/nikiitin/RING-5

APPENDIX

A. Abstract

The artifact provides the complete evaluation workflow for HighwayTM: the modified gem5 full-system simulator, DRAMsim3 and HeteroGarnet memory models, benchmarks and experiment definitions, Docker/Slurm automation, frozen paper data, and RING5 analysis scripts [68]. Reviewers can quickly regenerate every simulation-derived paper figure from the frozen data and reproduce the full study. Each figure script also accepts newly generated raw `stats.txt` trees.

B. Artifact check-list

leftmargin=1.35em,itemsep=0pt,parsep=0pt,topsep=2pt

- **Program:** gem5, STAMP and concurrent data-structure benchmarks, RING5 figure scripts.
- **Compilation:** C/C++ (SCons/Make) inside Docker; Python analysis environment installed by Make.
- **Model:** Full-system x86-64; 1-16 out-of-order cores, three-level cache, directory MESI, crossbar network, DDR4-2400.
- **Data:** Raw gem5 statistics and bundled frozen per-seed/reduced CSVs containing the paper results.
- **Environment:** 64-bit Linux (Ubuntu 24.04), GCC 9.2.0, Python $\geq$3.12, Docker Engine $\geq$27.1.1 and Compose $\geq$2.29.1; KVM recommended.
- **Hardware:** At least one host CPU and 3.5 GiB RAM per parallel simulation node; 6.7 GiB for the extracted artifact.
- **Execution:** Make, Docker Compose, and Slurm; experiment matrices with seeds 0-3.
- **Metrics:** Execution cycles, transactional regions, commits and abort causes, stalled/futile cycles, per-core aborts, and network traffic.
- **Output:** Figures (PNG/PDF/PGF).
- **Experiments:** main (280 runs), scalability (800 runs), or frozen-data regeneration.
- **Time:** Frozen-data setup and plotting: typically under 15 min; build/checkpoints: around 1 hour; complete simulation: several days, depending on host capacity.
- **Availability:** Zenodo: https://doi.org/10.5281/zenodo.22303584.
- **Licenses:** RING5: GPLv3; bundled third-party software and benchmarks retain their respective licenses.
- **Archived:** Permanent DOI: `10.5281/zenodo.22303584`.

1) How to access: To download the artifact, fetch it from https://doi.org/10.5281/zenodo.22303584. It is a multipart zip file with a SHA-256 checksum that can be used to verify the integrity of the downloaded file. To verify the integrity of the parts and reconstruct the archive execute the following commands:

```
sha256sum --check SHA256SUMS
cat MICRO26-HighwayTM-Artifact.zip.part?? \
  > MICRO26-HighwayTM-Artifact.zip
sha256sum MICRO26-HighwayTM-Artifact.zip
unzip MICRO26-HighwayTM-Artifact.zip
```

The fast frozen-data evaluation needs only compatible Python and a LaTeX engine if PGF output is desired. Full simulation needs a Linux Docker host; access to `/dev/kvm` substantially reduces boot and fast-forward time. Atomic CPU mode is supplied for hosts without KVM as fallback, but is much slower. No download is required in addition to the repository fetch. Dependencies and tools used for experimentation are included and bundled in the artifact files.

The top-level `README.md` is a command-by-command manual in Markdown format, in addition to this appendix. That manual includes troubleshooting hints that were not included here. In this annex we focus on the experiment reproduction and how to extract and validate the simulation results.

2) Data sets and models: The main campaign comprises five conflict-resolution policies, 18 STAMP/CDSB/in-house TBench benchmarks, four seeds, and 16 cores (360 runs). The strong-scaling campaign uses eight STAMP benchmarks, five policies, 1/2/4/8/16 cores, and four seeds. We decided to also keep the CSVs to preserve the paper measurements and to offer replication verification; raw executions write `stats.txt` below `results/highwaytm-main` and `results/highwaytm-scalability`. The modeled machine and policy retry limits are defined in `gem5/gem5_path/scripts/MICRO2026/config.py`. Each system configuration is reflected in the methodology section of this work.

C. Installation

For figure reproduction only using the frozen data, run:

```
make build_RING-5
./figures/generate-figures --reference
```

To build the full simulation environment, if your host supports KVM, run:

```
make build
make run
docker exec slurmctld \
  /gem5/tasks-gem5 create-boot-checkpoints \
  --overwrite
```

In the opposite case, if your host does not support KVM, run:

```
make build_atomic
make run
docker exec slurmctld \
  /gem5/tasks-gem5 create-boot-checkpoints \
  --disable-kvm --overwrite
```

A successful build command should end with the message `Everything installed and verified from the offline bundle!`

D. Experiment workflow

Once the artifact is built and the boot checkpoints are created, you can connect to the virtualized Slurm cluster and enqueue the experiments with these commands:

```
make connect
cd /gem5/gem5_path/scripts/run-scripts
CAMPAIGN=all ./gem5_gen_scripts.py \
  --config-file ../MICRO2026/config.py --enqueue
```

Keep the cluster running until all jobs complete, exit with `exit`, and then, execute `make plot`. This command should automatically select raw results. To generate the results from the provided CSVs with the results, run the following command: `./figures/generate-figures --reference`.

E. Evaluation and expected results

The plot command creates 24 figure files (eight figures in three formats) and `plots/table4_network_cost.csv`. Raw-data values may vary slightly with host/tool versions, but the qualitative ordering and paper conclusions should remain. Some benchmarks on the MCAS suite may not reach completion and may abort execution due to an assertion failure. The occurrence of such failures depends on the host GCC version. We recommend using GCC 9.2.0, which is the version used to generate the data, and Ubuntu 24.04.

F. Experiment customization and notes

Individual figure scripts accept `--results PATH` for another compatible result tree and `--reparse` to invalidate their RING5 cache. Do not enqueue the same campaign twice into one output directory. Partial result trees should be completed or moved prior to live plotting. PNG and PDF are always produced; PGF requires a working LaTeX installation.