Chaining Transactions for Effective Concurrency Management in 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*—Hardware Transactional Memory (HTM) offers the opportunity to ease parallel programming. However, driven by hardware limitations, commercial implementations eschew the complexity involved in early sophisticated proposals from academia, and, among other things, opt for simple conflict resolution policies that inevitably increase transaction aborts. To increase thread level parallelism, previous works propose conflict resolution schemes that, instead of aborting, add a second level of speculation consisting in using not-yet-committed data from another transaction. This policy, which we refer to as *requester-speculates*, has not yet been considered in the context of the kind of *best-effort* HTM support provided by commercial processors.**

This work proposes CHAining TransactionS (CHATS), a simple yet effective realization of the requester-speculates conflict resolution policy in which cyclic dependencies between transactions are avoided and the commit ordering respects the dependencies that transactions make once speculative values are communicated. The ultimate result is a best-effort HTM implementation that forces a partial order between transactions in a way that ensures effective utilization of forwarded data and that gets away from the complexity of previous proposals. Simulations using gem5 demonstrate the effectiveness of CHATS in both commercial-like setups and academic state-of-the-art best-effort systems (22% and 16% reduction in execution time, on average, respectively). These improvements are achieved by requiring less than 280 bytes of extra storage.

I. Introduction

Beginning roughly two decades ago with the advent of multicores, Hardware Transactional Memory (HTM) sparked great interest among researchers, who picked up on the seminal work by Herlihy and Moss [17] because of its potential to simplify synchronization in parallel programs. The 2000s witnessed a myriad of publications in transactional memory (TM), where researchers attempted to broaden the kinds of transactions supported in hardware with the goal of simplifying their use as a synchronization primitive, for instance, by accommodating large, coarse-grain transactions or ensuring progress despite high contention. Years later, the semiconductor industry began to commercialize HTM-equipped chips [1], [18], [20], [23], [42] that have proved significant advantages over traditional locks in specific niches [22], [43].

In spite of such success stories, the limitations exhibited by most commercial HTM implementations, commonly referred to as *best-effort* HTM, have prevented a broader shift away from locks. This is because HTM support found in real-world chips gives no assurances that a transaction will ever commit [18], [44]. Though transactions might fail for a variety of reasons, like lack of buffering resources or the need to handle hardware interrupts, the prevalent cause of aborts are conflicts. In fact, a frequent critique of HTM today is that system performance quickly deteriorates when multiple transactions are trying to access the same data at the same time and, at least one of them, is trying to update it. Behind this shortcoming lays the decision adopted by manufacturers of employing the simplest possible approach for resolving conflicts, which consists in rolling back a transaction whenever it observes a remote memory access to a block in its read or write sets that would threaten atomicity [9], [32], [38], [44]. As a result of this choice, programs using existing HTM support are liable to experience livelocks [9], [31], [32], [44]. To escape from them, the use of HTM support must be combined with an alternative fallback path employing traditional locks to ensure forward progress [10], [44]. Because mutual exclusion is required to guarantee the atomicity of transactions executed non-speculatively, transactional workloads often take a severe performance hit as soon as moderate contention arises and the fallback path is taken more frequently [7], [9], [10], [44].

An alternative approach to manage conflicts, rather than aborting at least one of the transactions involved, is to allow the requester to speculate past the conflicting access as if no conflict had happened, so that a transaction can consume not-yet-committed data from another transaction (or update data that is still in its read-write set). This conflict resolution policy, referred to in this paper as *requester-speculates*, unlocks additional concurrency and, as shown in this work, can achieve conflict serializability and ensure forward progress by means of simple mechanisms that respect the producer-consumer dependencies created by the forwardings. Several prior works have exploited this idea [2], [29], [34], [35], [39], yet they represent a significant departure in terms of hardware complexity from a typical best-effort HTM: they modify critical components such as the coherence protocol (e.g., extra states), the communication fabric (e.g., dedicated signals/messages to enforce commit order) or virtual memory.

Part of the complexity brought by prior works implementing requester-speculates stems from the desire to allow the speculative forwarding of data that has itself been received speculatively, which requires tracking *multiple* versions of *the same* conflicting cache block [34], [35]. This means that cache

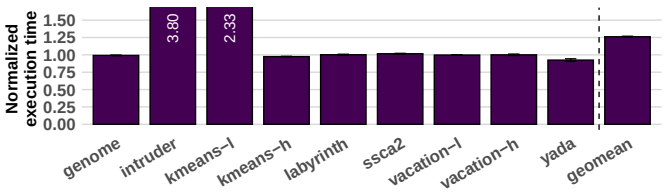


Fig. 1: Normalized execution time of a naive realization of the requester-speculates policy over a best-effort baseline

ownership over the conflicting block needs to be transferred (i.e., updated in the directory) from the forwarder to the requester transaction as if no conflict had occurred, so that subsequent conflicting requests can obtain the latest speculative version. Furthermore, it also requires to keep precise information of dependencies between transactions to meet ordering requirements, significantly increasing the complexity of the design in general and of checking the correctness of the speculation in particular. Therefore, we advocate for a slightly restricted but much simpler approach, in which coherence ownership for conflicting cache blocks is never relinquished until the owner (producer) transaction commits, and the coherence information in the directory is not updated in case of a speculative forwarding. Our design allows for the creation of long chains of transactions due to successive forwardings, as long as the involved cache blocks are different.

Although other authors [30] have explored a similar strategy to incorporate requester-speculates into an HTM design, such prior work also has several important shortcomings. First, it builds atop a *full-blown* virtualized HTM [28] so as to leverage its expensive timestamp-based deadlock avoidance scheme. Second, their design conservatively imposes severe restrictions that dramatically reduce the scenarios where requester-speculates policy can be utilized, disallowing the creation of chains of length greater than 1. Third, it does not consider the producer-consumer dependencies when resolving conflicts, leading to cascading aborts. Thus, said prior work comes short of demonstrating the full potential of requester-speculates and its applicability in the context of a best-effort HTM designs that reuse existing hardware components.

Indeed, implementing requester speculates in a best-effort system may seem easy to do in principle, since it can be treated as a value speculation problem using a validation mechanism that aborts transactions in case of misspeculation. However, a naive application of these design principles leads to performance degradation, as shown in Fig. 1 (see Section VI for details on the methodology). This degradation is mainly due to ineffective cyclic dependency managing when consumer transactions also produce data for its own producer, thus causing all transaction in the cycle to abort and hindering forward progress. To the best of our knowledge, no previous work has succeeded in putting forward a feasible implementation of requester-speculates in the context of best-effort HTMs.

This work attempts to fill such gap, by describing a practical realization that employs very simple mechanisms to create *CHAins of TransactionS* (CHATS). A fundamental observation of this work is that to effectively implement requester speculates in best-effort systems is that forwarding of speculative data creates a producer-consumer relationship between transactions that cannot be ignored. Firstly, these relationships must not be cyclic to be able to find a valid serialization of the involved transactions that allows to commit all of them. Secondly, these dependencies need to be respected upon commit, so that a transaction that has received speculative data from another can never commit before the producer.

CHATS piggybacks standard coherence traffic and relies on the usual transfer of coherence permissions when isolation is released over conflicting blocks upon commit or abort. It also performs value-based validation to ensure the correctness of input values. Our solution also allows for chains of transactions of any length and width (as long as conflicts occur on different cache blocks). Unlike prior works, CHATS does not need explicit communication between producers and consumers for commit ordering, nor deadlock avoidance.

With minimal hardware modifications and less than 280 bytes of extra storage, CHATS obtains average reductions of 22% in execution time when compared to a commercial-like best-effort HTM implementation that resembles the Intel RTM support [18]. Moreover, performance advantage increases to 28% on average when CHATS is combined with a state-of-the-art approach, PowerTM [12], that manages transactions using two different levels of priority. All of the above comes as a consequence of the significant reduction in the number of aborts that CHATS implies (34% and 49%, respectively).

The paper makes the following contributions:

- We show that blindly forwarding speculative data in a best-effort HTM system brings no performance benefits.
- We show that the *Position in Chain* (PiC), a small integer per core, is enough to encode sufficient information about producer-consumer relationships between transactions to be used to guide later forwarding decisions and to infer a good commit order.
- We show how to avoid the creation of cyclic dependencies using the information encoded in the PiC by choosing for each conflict whether to apply a requester-speculates policy or the underlying baseline policy.
- We propose CHATS, which provides support for effective use of a requester-speculates policy on top of a best-effort HTM with minimal complexity and hardware overhead.
- We demonstrate that CHATS can be effectively combined with a state-of-the-art proposal, PowerTM [12], which uses dual priority to bring even more performance gains.

II. Background and Related Work

HTM systems must guarantee that all instructions within a transaction are either completed in its entirety or not executed at all, while providing each transaction with the illusion of having exclusive access to shared memory. The simplest approach to ensure correctness that existing best-effort HTM realizations take, is to abort the transactions whose atomicity is threatened. However, such rollbacks are not strictly required as long as conflicting transactions commit in the adequate order.

Implementing conflict serialization in HTM can enhance performance at the cost of higher complexity [2], [34], [35]. Dependencies among transactions running in different cores must be tracked, as this information is essential for enforcing commit order, avoiding deadlocks that stem from circular dependencies, propagating abort to dependent transactions, etc. Additional challenges that must be addressed include ensuring that a transaction observes the correct version for a speculated cache block (depending on its position in the dependency chain), and dealing with the fact that dependencies among transactions may not be visible to all cores.

Ramadan *et al.* [35] were pioneers in proposing a requester-speculates HTM design, DATM. It augments a coherence protocol with additional states to track forwarded and received data, while taking advantage of a bus-based environment to greatly simplify deadlock avoidance (as information about dependencies is easily kept consistent across all cores).

Titos *et al.* [39] adopted a simpler yet more restricted approach to requester-speculates HTM that extends LogTM [28] to enable reader-writer sharing in a distributed directory-based protocol. The authors propose extensions to allow writer transactions to obtain exclusive ownership over blocks in the read-set of other transactions, while ensuring that readers commit first and observe the correct data version.

Within the scope of simultaneous multithreading (SMT) processor design, Qian *et al.* proposed a design [33] that takes advantage of the fast communication between local contexts to enable concurrency between dependent transactions.

A subsequent proposal by the same authors [34] proposes a scheme for serialization of conflicting transactions in a distributed directory environment, without employing any centralized hardware structure to track dependencies. A downside of OmniOrder is that it requires broad modifications across the memory hierarchy and dedicated commit/abort messages. Apart from replicating history buffers and successor registers across the directory, OmniOrder keeps a history of speculative updates for each word that makes it unable to reuse the L1 cache for speculative versioning. Instead, an L0 cache solely dedicated to keep speculatively modified blocks is used, potentially affecting the critical path of all memory accesses.

The works by Pant *et al.* [29], [30] have also investigated how to improve concurrency of conflicting transactions atop LogTM [28]. In VP-TM [29], the authors explore the potential of resolving conflicts by means of a centralized, memory-level value predictor, while in LEVC [30] they propose a constrained design based on their prior observations. These proposals share a key similarity with our work, as neither of them update the coherence state for a conflicting block when its value is forwarded to another transaction, unlike other prior works. A key shortcoming is that LEVC is liable to livelocks, because the underlying deadlock avoidance scheme it relies upon is unaware of the dependencies created due to forwarding of speculative values: The highest-priority transaction may be aborted after having consumed speculative data from a lower-priority transaction that the underlying timestamp-based scheme selects as victim. Moreover, it also imposes draconian restrictions in both the length of dependency chains and the number of speculative consumers of a data block.

In SONTM [2], the authors propose a hardware implementation of a conflict serializability technique previously shown to improve performance for software transactions, based on assigning each access a serial number. To manage such numbers, extensive changes, including operating system and virtual memory, are required. A fundamental drawback is the complexity and lack of scalability of commit: a centralized lock must be held to ensure atomicity, while serial numbers are broadcast to other cores and written to a data structure in virtual memory. This bottleneck, coupled with intrinsic instrumentation overheads and the complexity of managing serial numbers, make it an unappealing solution for adopting requester-speculates into a best-effort HTM system.

Finally, Jafri *et al.* investigated how to combine a token-coherence-based HTM system [8] with software support to achieve conflict serializability of transactions [21]. In Wait-n-GoTM, a great deal of the complexity involved by dependence tracking found in prior works is shifted to software: exception handlers are invoked on commit and abort are used to identify dependent transactions, enforce ordering for conflicting transactions, etc. However, because it builds atop token coherence [26], the mechanisms proposed do not fit into today's multicores that employ directory-based coherence.

III. CHAined TRANSACTIONS (CHATS)

We propose CHATS (alluding to the chatty behavior of transactions) as an innovative extension to typical best-effort HTM implementations, aimed at providing better concurrency management by supporting a requester-speculates conflict resolution. The main goal of CHATS is to reduce the number of aborts due to conflicts through the forwarding of not-yet-committed (speculative) values. To maximize chances of profitable value forwarding, our proposal ensures a commit ordering that respects the dependencies created by forwardings.

By keeping imprecise yet sufficient information about producer-consumer dependencies, CHATS avoids forwarding in cases of potential cycles. This way, CHATS can effectively serialize the commit of the transactions involved when the requester-speculates conflict resolution policy kicks in.

Let us consider as example two concurrent transactions, one of them attempting to read data modified by the other. The only privately cached copy of the block resides in the writer's cache, and so the transactional load will miss in caches, a request will be sent to the coherence directory, which will redirect the request to the current block owner. When the cache observes that the block is part of the write set, a conflict is detected. In HTM systems that implement requester-wins conflict resolution [42], [43] the transaction gets immediately aborted, the speculative values in cache are discarded, and the request gets serviced by lower levels with the non-speculative data version. The operation of CHATS starts to diverge from a typical HTM at this point: instead of aborting, the writer transaction detects the conflict and sends a copy of the data block (the current speculatively modified

version) to the requester, with the hope that no more stores from that transaction will target the block. The forwarding of the data initiates the *chat* between transactions, builds the chain of transactions, and dictates commit order. The requester obtains a speculative copy of the block, stores it in its cache, adds it to the write set and serves the load miss. Meanwhile, another copy of the speculatively received data is saved in a buffer away from cache and later used for validation. The directory is informed that the request was denied, so that the writer stays as block owner.

A. Validation of forwarded data

Validation of the speculatively received data involves requesting exclusive ownership for each data block, until non-speculative data is received with coherence permissions. Transactions are not permitted to commit until all blocks that have been received speculatively are successfully validated.

Upon each validation attempt, the data received in response is compared against the speculatively consumed value that was saved when the originating cache miss was serviced:

Value mismatch: The transaction must abort. Such validation failures can happen in three scenarios: i) the producer transaction has overwritten the data (the consumed data was an intermediate version); ii) the producer has aborted (the response contains an *earlier* non-speculative version, or a speculative version forwarded by a different producer); iii) the producer has committed but the block was subsequently modified by other core before this consumer validates it.

Value match: No abort is signaled, but the block cannot be considered validated until a response with non-speculative data is received and the requester becomes actually the owner for the block. Only when that happens, the validation of the block is successful and the original copy buffered is discarded. The copy allocated in cache as part of the write set becomes the current version. Note that data are only discarded from the aforementioned buffer, not from the cache.

This validation mechanism involves minor modifications to the coherence protocol: avoiding ownership transitions during forwarding. In fact, the directory is oblivious to the forwarding and it receives a cancellation (unblock) of its request. Instead, the burden of checking the correctness of the speculation lies entirely on the consumer.

Executing transactions using non-committed data may lead to accesses to out-of-bounds memory addresses or to endless loops. The first case is already covered by the HTM support as exceptions result in aborting transactions. To cover endless loops, the validation process must be performed periodically to ensure the detection of incorrectly speculated data.

Our approximation of value-based validation eliminates the need for tracking of producer-consumer dependencies and allows for graceful handling of common situations that prior works resolve through dedicated hardware support for communication of commit/abort signals:

Cascading aborts: When a producer transaction aborts, the abort is propagated to all dependent transactions through validation, without requiring any explicit message sent by the producer. If a value received by a validation request differs from the original speculative data, it triggers an abort for the consuming transaction due to the detected inconsistency. This abort will propagate in the same way through all the levels of the chain.

Multiple consumers: When several transactions receive speculative copies of the same data block from a producer, they serialize their commit one after another. For instance, consider transactions T1 and T2, both consumers of data from T0; then T0 commits, and eventually both T1 and T2 attempt validation. If T1's validation request arrives first at the directory, it becomes the next owner of the block. Then, T2 will receive speculative data from T1 in its next validation attempt and its commit will be serialized after T1. This happens through the usual coherence protocol, without having to adjust any state specific to forwarding. If T1 modified the speculative data it received from T0, T2 will abort when it observes the value mismatch; otherwise, if T1 did not modify the data, T2 will be allowed to commit after T1 commits (e.g., when it obtains a non-speculative copy of the data).

As it can be noted from the previous example, the best-effort approximation we adopt in CHATS trades off stricter serialization orders for simplicity, reusing existing mechanisms found in typical best-effort HTMs whenever possible. In the aforementioned example, while T2 may not have true dependencies with T1, our design serializes their commit. In particular, we opt for inserting a speculatively received block in cache and mark it as being part of the write set to simplify its management, since that ensures that it never leaves the cache unless the transaction commits: if the transaction aborts, the block will be discarded by the conditional gang-invalidation of write-set blocks that L1 caches in best-effort HTM typically support. Similarly, the eviction of said block from cache will result in a capacity-induced abort, as it would with the eviction of any other block from the write-set.

B. Avoidance of cyclic forwarding

The forwarding of data creates a producer-consumer relationship between transactions. This relationship needs to be acyclic to be able to serialize the committing of the transactions and thus should drive the commit order. The validation mechanism described in the previous section ensures the commit order of acyclic chains, because transactions cannot commit until they have validated all their speculatively received blocks, and that validation implies that the producers have already committed. But transactions would not be able to commit if they are involved in a cyclic chain. Hence, it is necessary to avoid the formation of cycles.

Keeping full track of producer-consumer relationships is expensive and involves excessive complexity. As an alternative, CHATS implements a straightforward yet powerful method based on storing imprecise yet sufficient information of the dependencies among transactions. Our method keeps a local value in each core, known as *Position in Chain* (PiC) that represents the place of the transaction in a chain of transactions

that have forwarded values among them. If a transaction is not a producer nor a consumer then its PiC is invalid.

The PiC is sent along with data requests and speculative responses. On a conflicting request, the local and the remote PiCs of the transactions are compared, and the conflict is resolved in the following way:

- Requester-speculates: speculative data can be sent to the requester when either (i) any of the transactions has an invalid PiC (i.e., was not part of a chain), (ii) the received value (of the requester) is lower than the local PiC, or (iii) the local transaction has not received any speculative data yet or has validated all the speculatively received data (so it can increase its PiC value). In all those cases, the local transaction (producer) ends up with a PiC higher than the remote transaction (see Section IV for details), thus preventing cyclic forwarding scenarios.
- Requester-wins: Local transaction must abort when it has received speculative data which has not yet been validated (so it cannot increase its PiC without potentially matching the PiC of its producer) and either the received value is higher than the local PiC or both PiCs are identical. Cycles are hence resolved in those scenarios by aborting transactions.

Section IV offers further details about how and why CHATS use PiC to choose between those conflict resolution policies.

C. Correctness guarantees

CHATS relies on the fact that the order of memory operations inside a transaction is irrelevant from the point of view of the rest of the threads, since all the accesses will be made effective atomically at the serialization point, i.e., when the transaction commits [32], [41].

When a conflict occurs in CHATS, the thread that responds (T_P –producer) continues undisturbed once it has answered with the current data of the conflicting location (*L*). When the requesting transaction (T_C –consumer) receives that response, from the point of view of the rest of the system it continues executing as if no conflict had happened, and as if the memory access had not been attempted. In fact, neither the coherence state nor ownership information change in the directory. The execution continues assuming some value (*X*) for that access which may be correct or incorrect in the end. A correct value (*Y*) would be one that T_C would have received if it stopped its execution until T_P committed, and then reattempted the memory access. Note that *Y* would not necessarily be the data written at *L* by T_P when it commits, because other transactions or non-transactional accesses can write to that location between the commit and the reattempted access.

To check whether speculation succeeded, T_C must perform the access again at some point before commit and obtain requested permissions and non-speculative data (*Y*). If *X* and *Y* are equal, speculation succeeded. Otherwise, *X* was wrong and T_C must abort. The important detail is that, from the point of view of the consistency model, the access has occurred only once atomically at commit time along the rest of accesses performed by the transaction. One could say that no real data forwarding occurs, only value speculation. The trick is that the speculation is done after another thread informs of the value of *L* at some point in time, and we bet that the value will be the same if we perform the access later. Speculation will often succeed as long as T_C commits after T_P (which provides the speculative data as a hint), but it can fail not only if T_P decides later to modify the data before it commits, but also if another transaction that commits between them (or a non-transactional access) writes a *different* value to *L*.

Note that the speculation can succeed even if the location is modified between the commits of T_P and T_C as long as the value Y read by T_C is equal to the *X* that it speculated. The atomicity of committing transactions is ensured by the underlying HTM mechanisms, which we do not modify. The commit ordering enforced by the PiC mechanism increases the probability of successful speculation (but does not guarantee it) and ensures that no cycles can appear, which could lead to livelock situations due to repeated aborts. Livelocks due to wrongly speculated values are handled by performing periodic validation, as explained in III-A.

The above description may raise concerns about the case where the same location is written repeatedly by more than one transaction, but ends up with the same value as originally (ABA problem). Say a first transaction, T_1, speculates a value A for an access to memory location L, because it receives a message from a second transaction, T_2, which previously had written A. Later, a third transaction, T_3, writes a B value to L, and then, a fourth one, T_4, writes again at L the value A. The speculation done by T_1 will be correct if it tries to commit just after T_2 or T_4, but not if it does so after T_3 (note that, in this case, CHATS ensures that T_1 would be aborted). For correctness, it does not matter that T_1 guessed the value A because it was provided by T_2 or T_4, or whether it just obtained the value randomly: it got the correct value A for address L anyway. Note that, in each case, all the non-speculative values read by T_1, including the one used to validate the speculation, are those written by all the previously committed transactions (serialized) and non-speculative accesses, and the observed behavior is the same as if no speculation had occurred.

IV. Implementation of CHATS

An advantage of CHATS is its simplicity, as it requires no CPU modifications and no complex logic. CHATS requires minor modifications to the cache coherence protocol), a 4-entry *Validation State Buffer* (VSB) used to keep the original copy of each speculatively received block, a 5-bit register per core to keep track of the PiC, and a one-bit register named *Cons* to record if the transaction is currently consuming any speculative data pending of validation (Section IV-C). Fig. 2, depicts the hardware structures employed by CHATS. Utilizing the standard 64-byte cache line granularity and assuming 48 bits for physical addresses, the overall cost in terms of memory overhead of CHATS is less than 280 bytes (279.5 bytes) per core.

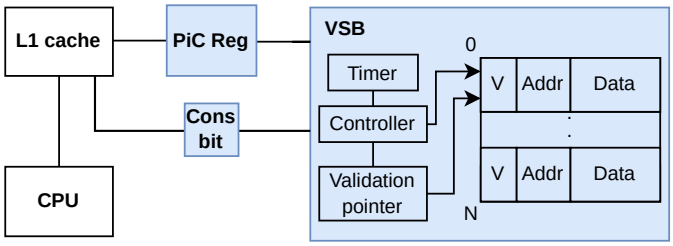


Fig. 2: CHATS overview (extra structures in blue)

A. Modifications to the coherence protocol

The coherence protocol requires minimal changes. It needs to be extended to allow private caches to answer with speculative data responses (SpecResp) to requests forwarded to them by the directory from other cores. In a typical best-effort system using the requester-wins policy, such a conflicting forwarded request would cause an abort of the transaction prior to sending a response with data and possibly a transference of ownership (if the request was exclusive). However, when using CHATS the cache receiving the request can choose to act exactly the same as just described or, instead, to neither abort the transaction nor relinquish the ownership of the data block, but still answer to the requester with a SpecResp message including speculative data and inform the directory that the request has been canceled. Note that whenever a conflicting non-transactional request is received, the system always resorts to requester-wins and aborts the transaction.

When a SpecResp is received, the requester cache will update its state as if it was a standard response with data and exclusive access to the block (even if the original request was only a read); however, it will also insert a copy for later validation in the VSB and, crucially, it will inform the directory that the coherence request has been canceled and the coherence state and ownership for the block must remain unchanged. Thus, the forwarder remains as the exclusive owner of the cache block from the point of view of the coherence protocol. The core will work with the copy of the data stored in the cache as if it had effectively acquired ownership of the data in all respects. The validation mechanism described in Section IV-B will ensure that such fiction becomes actually real before the transactions commits. The only behavioral difference between such speculatively received blocks and those genuinely owned by the transaction is that the former cannot be forwarded speculatively to another transaction, simply because the core does not observe coherence traffic for them.

B. The VSB and validation of speculative data

As mentioned in Section III, a requester that receives a speculative response must keep an unmodified copy of the data for later validation of the speculation. The unmodified copy of the block is saved in the *Validation State Buffer* (VSB). The VSB keeps copies of speculative data received from other caches until they have been validated, and must be empty before a transaction can commit. The contents of the VSB are immediately discarded upon abort.

Each VSB entry contains a valid bit, the block address, and a copy of the speculatively received data block, as shown in Fig. 2. The VSB has two associated pointers: one points to the next available entry and another to the entry that must be validated next. The VSB includes a simple controller that takes care of the validation process.

An associated timer periodically triggers the validation process if there is any block present in the VSB. Each time this happens, the controller makes an exclusive coherence request for the address of the block located at the entry pointed by the validation pointer, and increases that pointer. The timer will be activated again once the response is received.

When a response is received, it could be again a SpecResp or a standard coherence response granting ownership of the block. In either case, the received data is compared against the copy in the VSB. If it does not match, it means that the data has been modified by the original forwarder (or someone else) after we received it and hence the transaction must be aborted immediately. It the data matches, the transaction can continue; additionally, if the response was not a SpecResp but a standard coherence message, then the speculative data of the block has been validated, the core is now the real owner of the data, and the block can be removed from the VSB.

During the validation process, the PiC of each response is compared with the local PiC (that of the consumer). If the local PiC is higher or equal than the remote PiC, the local transaction is aborted. This is necessary to handle cycles created due to race conditions as mentioned at the end of Section IV-C.

When the transaction has validated all the speculated data, the VSB becomes empty and the Cons bit can be reset, but the PiC still remains valid until the transaction commits, as it may be the producer of another data.

C. Using the PiC to avoid cycles

As explained in Section III, CHATS encodes the producer-consumer relationships created by value forwardings using the *Position in Chain* (PiC), which represents the position of the transaction in a chain of forwardings. If set for a transaction, its PiC will always be higher than that of all transactions that have received speculative data from it. If unset, it means that the transaction is not part of any forwarding chain.

Also, every request message or forwarded probe and every SpecResp message must include the current PiC of the core. PiCs are expected to be very small numbers (5 bits), but their range limit the length of the chains supported by CHATS. Note also that to ease extending chains from either end, the initial value of PiCs (PiC_{init}) should be in the middle of the range. Another value should be reserved to denote an unconnected transaction with unset PiC ($\text{PiC}_{\varnothing}$).

Fig. 3 explains how the PiC is updated. Upon reception of a conflicting request, if current PiC is invalid ($\text{PiC}_{\varnothing}$), indicating that the transaction has not received or forwarded any speculative data, the PiC is set to the its initial value (PiC_{init}). This PiC is then transmitted to other transactions along with coherence messages (a response in this case).

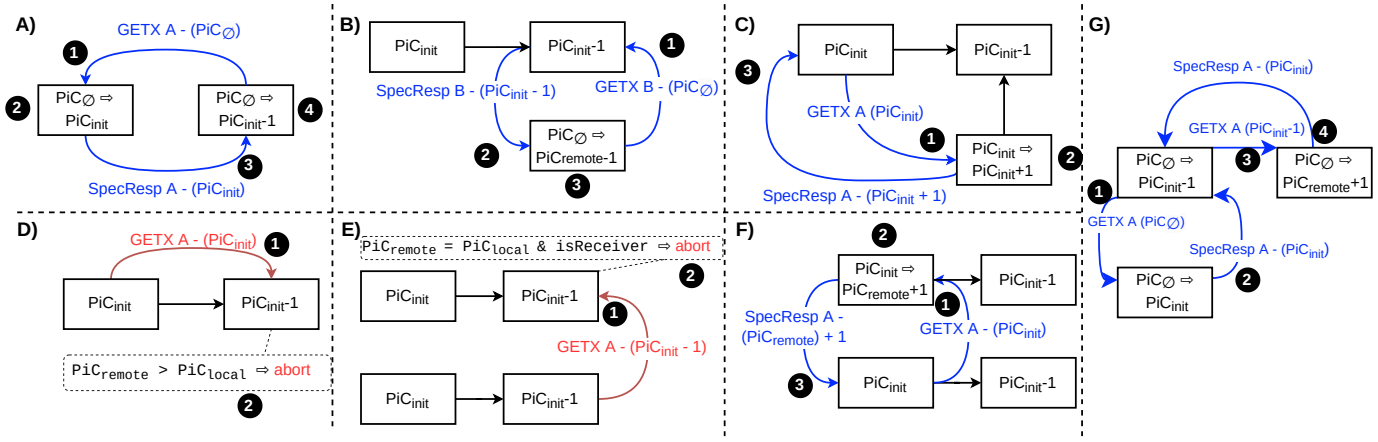


Fig. 3: A) Starting point for CHATS, two unconnected transactions $PiC = PiC_{\varnothing}$ and one requester. B) PiC usage allow already receiving transactions to become forwarders to other unconnected or with lower PiC transactions. C) A transaction can forward data to other transactions if it was already doing it. Checkings over PiC will be made in this case, ensuring that it cannot be modified if the transaction has already received an SpecResp. D) A transaction that already received speculative data cannot forward anything to other with same or higher PiC. E) A transaction that already received speculative data cannot forward anything to other with same or higher PiC. F) A transaction that already forwarded data becomes forwarder to another producer E) and D) will raise an abort signal. G) Two transactions produce to a consumer

The requester will set its own PiC to $PiC_{fwd} - 1$ if it was not already set. Note that the PiC of a transaction cannot be changed once the transaction uses any speculative value received from another transaction. In case of transactional abort, PiC is reset to $\text{PiC}_{\varnothing}$.

In the event of a conflict between two transactions, the exact course of action depends on the PiC of the requester (PiC_{remote}), the PiC of the transaction that receives the request PiC_{local}, and whether it has already received speculative data not yet validated:

- If both are $PiC_{\varnothing}$ (as depicted in Fig. 3A), this means that these transactions are not part of any chain, and a requester-speculates policy can be used to handle this request. The forwarder sets its PiC to PiC_{init}, and transmits it to the requester along a SpecResp message. On the requester side, the PiC is set to one less than the received value ($PiC_{init} - 1$), marking these transactions as chained, and the Cons bit is set.
- If an unchained transaction ($PiC_{local} = PiC_{\varnothing}$) receives a request from a chained transaction ($PiC_{remote} \neq PiC_{\varnothing}$), it will use a requester-speculates policy too, but in this case it will set PiC_{local} to $PiC_{remote} + 1$ (Fig. 3C).
- If a chained transaction ($PiC_{local} \neq PiC_{\varnothing}$) receives a request from an unchained transaction ($PiC_{remote} = PiC_{\varnothing}$), the requester-speculates policy will be used, and its PiC_{local} will be unchanged. A SpecResp message will be sent, which will cause the remote transaction to update its PiC to $PiC_{local} - 1$ and set its Cons bit (Fig. 3B).
- If both are different than $PiC_{\varnothing}$ and $PiC_{remote} > PiC_{local}$, that means that the local transaction has not sent speculative data to the requester nor any of its producers, but may have received speculative data from them. In this case, the local transaction reads its Cons bit to check if it has indeed received any speculative value. If it has, then its PiC_{local} cannot be updated because doing so could increase this PiC past that of a producer of this transaction; therefore, the requester-wins conflict resolution policy must be used, and the local transaction is aborted (Fig. 3D and Fig. 3E). If, on the contrary, the Cons bit is unset, this transaction has not received any speculative value or it has already validated all those values, which implies that all its producers have already committed. In this case, the requester-speculates policy can be used, and a SpecResp message is sent to the requester after updating PiC_{local} to $PiC_{remote} + 1$ (Fig. 3F).

In any of the above cases, if the updates required to the PiC in either the local or the remote transaction would cause an overflow or underflow, then the requester-wins policy needs to be used (aborting the local transaction).

Note that the PiC received by the local transaction may be outdated once it is used to choose the response policy. Hence, in some cases, cycles can actually be created by our forwarding mechanism. These cycles will be detected during the validation process by looking at the local PiC and the PiC of the received message, aborting the validating transaction if necessary, which will cause cascading aborts to the rest of transactions in the cycle.

V. Interactions with Coherence and Consistency and Progress Guarantees

Our proposal aims to minimize the necessary hardware modifications, in line with existing best-effort HTM designs. This section elaborates on the (minimal) changes to the cache coherence protocol required by CHATS, and discusses its

interactions with the memory consistency model, and liveness guarantees.

A. Interactions with cache coherence

As stated in Section IV-A, CHATS needs a mechanism that allows a transaction that receives a conflicting request to ignore it with respect to coherence (i.e., not perform the request) and instead send a *SpecResp* message in response. The meaning of those messages from the coherence protocol point of view is to inform the requester that the memory access has not been completed and may be re-tried later. This kind of *NACK* message has also been introduced in other previous HTM proposals to implement requester-loses or requester-stalls policies [9], [12], [25].

Additionally, in CHATS these messages carry some data that is communicated to the requester in the response. It is important to highlight that the mentioned data is just a hint that may be used or not by the requester to speculate values, but the content is irrelevant from the point of view of the coherence protocol. Once this message is sent, the responder notifies the directory that it did not give away the permissions. Thus, if the requester continues to execute, it must do so speculating the conflicting value. A requester speculating on a value needs to later reissue the request for such value, which will be done by the validation mechanism described in Section IV-B. A transaction will acquire permissions for the block only after a successful validation.

In addition, whenever a requester receives a *SpecResp* in response, it buffers the speculative data, stores it in the L1 cache, and adds it to its write-set. This addition to write-set is critical as it allows to reuse the usual HTM mechanisms to detect further conflicts and to discard the data in case of misspeculation. Thus, if a transaction aborts, any speculated block stored in cache is invalidated along with the blocks modified inside the transactional region. Additionally, if a validation is successful, there is no need to take any extra step as the data speculation proves to be correct, and therefore any later modifications locally performed to the involved block are valid as well. However, this can lead to false capacity-induced aborts if speculative read blocks are evicted from the cache (although this situation is unlikely since the replacement algorithm favors write-set blocks).

B. Interactions with consistency model

As previously stated in Section III-C, any memory access that receives a *SpecResp* message as a response remains speculative until validation. From the point of view of the transaction it can be seen as delaying the memory operation, but it is guaranteed that it will be validated (i.e. actually performed) before committing. Since memory accesses within transactions are made visible to the memory system atomically, delaying loads or stores inside a transaction does not affect the consistency model. Therefore, delaying any subsequent operation over any speculative block does not affect or interact in any way with the memory consistency model. This has been shown in previous works where transactional memory accesses are delayed but performed before commit, like [41]. Any races are therefore avoided, including the ABA problem as explained in Section III-C.

TABLE I: 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 [37]
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

C. Forward progress guarantees

CHATS is a best-effort HTM implementation, and as such, no guarantees over forward progress are offered at hardware level, just as in all other HTM proposals evaluated in this work. CHATS, as any other best-effort HTM, resorts to a software fallback lock mechanism [10], [12] to guarantee forward progress. On the other hand, our transaction ordering mechanism based in PiC guarantees that all transactions either commit respecting any dependency induced by speculation or abort if a potential cyclic situation is detected. This guarantee covers those cases where races may result in the transmission of out-of-date PiC values, as stated in Section IV-C. Finally, the validation mechanism is ensured to always finish, checking that forwarded data is correct and continuing, or detecting modifications to speculated blocks and aborting, thus avoiding any livelock situation due to data periodic validation.

VI. Methodology

A. Simulation environment

We use the gem5 simulator [5] to evaluate the benefits of CHATS. We use the full-system simulation mode to boot an unmodified Ubuntu Linux 16.04, kernel 4.8.13. The simulated system is a 16-core, out-of-order x86-64 processor with a three-level cache hierarchy. Table I shows the relevant configuration parameters of the simulated system, which have been set to resemble a modern microarchitecture such as Golden Cove [4], [11], [36]. We use DRAMsim3 [24] to accurately model main memory and HeteroGarnet [3] for the interconnection network.

B. HTM systems evaluated

Baseline. We choose as baseline an Intel RTM-like (Restricted Transactional Memory) best-effort system with lazy versioning and eager conflict detection. In our chosen baseline, the L1 data cache is used for speculative versioning, by introducing a *speculatively modified* (SM) bit per cache line to track write-set blocks. Non-speculative values are written back

to L2 before a block in L1 is speculatively modified thus, L2 is used to store non-speculative values. On abort, conditional gang-invalidation of SM blocks in L1 suffices to discard speculatively written values, while the coherence protocol tolerates such silent invalidations of exclusively owned blocks. Following the features seen in commercial RTM implementations, which can accommodate transactions whose read-sets easily exceed the size of the private cache level [14], [43], we use a perfect signature [6], [15] to track read sets.

Naive requester-speculates (Naive R-S) This configuration extends the baseline by allowing implementing the naive requester-speculates conflict resolution policy, whose performance results were reported in Section I. This naive nature allows transactions to always forward speculative data without restrictions. To break cyclic dependencies, consumer transactions use a counter that gets decremented upon each unsuccessful validation attempt for a speculatively received block; the counter is reset on a successful validation. If the counter drops to zero, the transaction aborts to escape from potential deadlocks. We use a 4-bit global counter on each core that allows up to 16 unsuccessful validations before aborting.

CHAined TransactionS (CHATS) It extends the baseline with all the elements described in Section III and Section IV to achieve transaction chaining. The concept of PiC eliminates the need to limit validation attempts since cyclic dependencies are prevented. For simplicity, we refrain from imposing restrictions on the number of sharers a transaction can have for speculative data or the number of transactions included in a chain (which are limited by the 5-bit PiC register).

Power transactions (Power). We implement a system similar to PowerTM [12], where software triggers an elevated priority status after the second conflict-induced abort. Our runtime environment ensures the existence of at most one Power transaction, and conflicts involving both elevated and non-elevated priority transactions consistently favor the former. Following [12], we implemented a mechanism to allow negative coherence responses without invalidating the data in the cache. These *nacks* can only be sent by transactions with elevated priority.

Powered CHAined TransactionS (PCHATS). This configuration combines CHATS with PowerTM [12] as they can synergize and yield additional performance benefits together. Here, Power transactions are exclusively producers of speculative data, never consumers, and are always at the top of the priority chain (they do not need a PiC). Conflicts are systematically resolved in favor of Power transactions, and transactions receiving data from a Power transaction do not modify their PiC. The validation mechanism alone is sufficient to ensure commit serialization against Power transactions.

Limited Early Value Communication over Best-efffort idealized (LEVC-BE-Idealized). This is a best-effort adaptation of the ideas presented in [30]. LEVC is built over a non-best-effort mechanism [28] which would not be appropriate to evaluate in the context of this work. We have designed LEVC-BE-idealized by applying techniques used by LEVC for value forwarding and cycle avoidance but applying them on top of a best-effort requester-stall design. Our LEVC model uses ideal timestamps that never roll over, are instantly acquired and propagated in all coherence messages at no extra overhead. Whenever a conflict happens, the requested transaction sends a speculative version of the value along with a $nack$ message. The amount of speculative data blocks that can be consumed is the same than for CHATS (VSB size) for better comparison. Additionally, as this implementation is best-effort we have determined the optimal number of retries, specified in Table II.

TABLE II: HTM system configurations.

System	Block state	Retries	VSB size	Cycles valid.
Baseline	NA	6	NA	NA
Naive R-S	Rrestrict/W	2	4	50
CHATS	Rrestrict/W	32	4	50
Power	NA	2	NA	NA
PCHATS	Rrestrict/W	1	4	50
LEVC-BE-Idealized	Rrestrict/W	64	4	0

C. Benchmarks

To conduct our experiments, we utilized the STAMP benchmark suite [27] with the recommended medium inputs. We run 16 threads as the scalability of most of the STAMP benchmarks is very limited beyond this number. Due to the inherent randomness exhibited by *bayes*, whose search algorithm may result in varying amounts of work for the same input [13], we opted to exclude it from our evaluation.

We also implemented two synthetic microbenchmarks, *llb* and *cadd*, to measure the effectiveness of CHATS. These are implemented having in mind patterns typically employed in software. *llb* emulates several threads traversing a linked list where elements are searched, then modified. We run it using two inputs for low/high contention flavours using 16/64 elements per thread, in both cases with lists lengths of 512 and 256 iterations. As for *cadd*, it uses a vector populated with queues of integers, called clusters. Every thread modifies a shared variable and iterates over all the elements in the cluster calculating the sum of every element plus the modified version of the variable. We run *cadd* with 512 clusters of length 64. The results obtained for the microbenchmarks are shown in Figs. 4 to 7, yet they have not been accounted for in the calculation of the arithmetic and geometric means (to avoid overstating the benefits that could be seen in practice).

We make use of our TM library with functions for transactional begin and end that follow RTM recommendations [19] in regards of eager lock subscription. To isolate our evaluation from side effects such as page-fault-induced aborts, we employ a software scheme of pre-faulting [40].

D. Configuration parameters defined in CHATS

CHATS defines several configuration parameters that dictate its concrete behavior. They must be adjusted at design time considering the particular characteristics of the underlying hardware. We include in Table II the optimal values for our hardware setup. Section VII-A includes a sensitivity analysis to quantify their impact on abort rate and performance.

Blocks that can be forwarded. We consider three configurations that differ in which blocks are eligible for forwarding:

- *Forward all* (or R/W): blocks belonging to both read and write sets are allowed to be forwarded.
- *Forward written* (or W): Blocks belonging only to the write set can be forwarded.
- *Forward restricted read and written* (or Rrestrict/W): blocks belonging to the read and write sets are allowed to be forwarded. However, a heuristic check for currently in-flight writes from the local core is performed to avoid forwarding blocks known to be invalidated shortly afterward.

Retries before resorting to the fallback path. All configurations are best-effort, and thus, all of them need to specify a fallback path. It will be executed either with power token, in configurations based on PowerTM [12], or by holding a global lock to ensure execution serialization [10], in the rest.

Maximum number of forwarded blocks and validation periodicity. Validation relies on the *Validation State Buffer* (VSB) to track received speculative blocks and conduct periodic checks. The frequency of validation can impact performance (besides network traffic), as modifications or commits to speculated blocks may be detected sooner or later. The size of the VSB dictates the maximum number of cache lines that each transaction can receive speculatively.

VII. Results

We start our evaluation with the execution time achieved by CHATS compared to the other configurations discussed in Section VI-B, shown in Fig. 4 (the lower, the better). Execution time is always normalized to the baseline.

Ssca2 and vacation exhibit very low contention between transactions (the total number of aborts ranges between 0 and 10 for the entire execution) and hence all configurations achieve virtually the same performance. In this case, there are no opportunities to forward values between transactions. On the other hand, labyrinth shows no improvements given its scarce parallelism when its shared data structure cannot be *early released* [16] from the read set of its main transaction.

For the other four STAMP benchmarks (genome, intruder, kmeans, and yada), we observe large improvements in execution time with CHATS and PCHATS. In kmeans, there is a shared structure with points that need to be classified using clustering. Most of the transactional code is dedicated to updating the centers of each cluster for each point. There are three transactions, the first is the most contended one, where the centers are updated, while the rest are just to update two global variables. Contention occurs when more than one transaction acts on the same center. In any case, this update is a minimal task and every thread memory access pattern is the same when accessing the centers. As a result, kmeans is a benchmark that hugely benefits from correct data forwarding as contending threads have the same data access patterns. Once a transaction modifies one of the dimensions for the center, there is no further update, so this data can be safely forwarded to other threads. This migration pattern is successfully exploited using CHATS. The use of our mechanism results in a noticeable reduction in the amount of produced conflicts as can be observed in Fig. 5 reducing the occurrence of conflicts in roughly 75% in both, CHATS and PCHATS, mostly due to fewer forwarder transactions being aborted as shown in Fig. 6. In high contention situations as in kmeans-h, it significantly reflects in performance. In genome, the same behavior is expected since genome sequencing follows an analogous behavior of producer-consumer dependencies. Nevertheless, the contention is lower and although less reflected in performance, CHATS is still able to give a solid improvement of 25% in execution time and 75% reduction in conflicts.

Another benchmark with huge potential is yada, which implements long-running transactions to perform re-triangulation. During transactional execution, several random memory locations are accessed in a read-modify-write fashion which CHATS can easily exploit. In particular, whenever a transaction modifies a memory location, it would not modify it again, following a migration pattern. This pattern is exploited by CHATS allowing sharers to read and use the modified data speculatively to perform other calculations on other triangles. This is reflected in Fig. 5 where conflict-induced aborts are consistently reduced to roughly 50% using CHATS with respect to the baseline. Even for Power, where aborts are expected to rise due to its ability to provide progress despite contention without resorting to the fallback lock, PCHATS manages to reduce aborts by far more than 50% compared to Power baseline and only increases them by roughly 25% compared to the baseline.

Intruder attempts to emulate a network intrusion system and only two steps from this detection are enclosed by transactions *capture* and *reassembly*. In *capture* an intrusion data structure is popped from a FIFO queue. In this transaction, however, there is a time gap between reading and modifying the structure pointer, which can be read by multiple transactions simultaneously. For policies where the requester is not necessarily the transaction that wins the conflict, this could be a source of problems. For example, if the winner of the conflict is the requested transaction, then the transaction trying to execute the last modification will try to acquire write permissions but will be systematically denied by other readers. This will cause a livelock because no transaction will be able to complete, showing a clear pathology of starving writers [9]. A similar phenomenon occurs in CHATS, but in this case, the problem is not caused by starving writers, but by false positives at cycle detection. Whenever two transactions try to write to the same address at the same time, cycles are erroneously detected due to receiving an outdated PiC as discussed in sections V-C and IV-C. This will cause all involved transactions to abort, creating a livelock situation that must be resolved by resorting to fallback-lock. The other source of contention comes from *reassembly* where a red-black tree is traversed to add nodes and occasionally re-balanced. Transactions in this case again exhibit a producer-consumer behavior with the inclusion of events that can cause generalized aborts to all transactions traversing the tree due to re-balance. Additionally, another transaction covers up recovering data from the queue where results are deposited in *reassembly* which can often cause

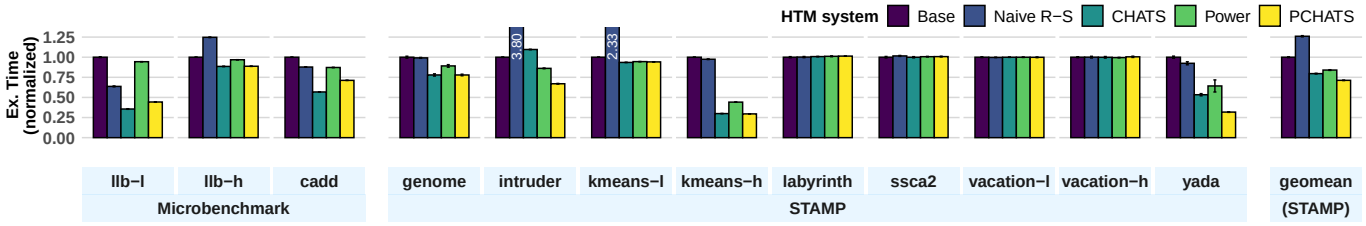


Fig. 4: Performance results in terms of execution time

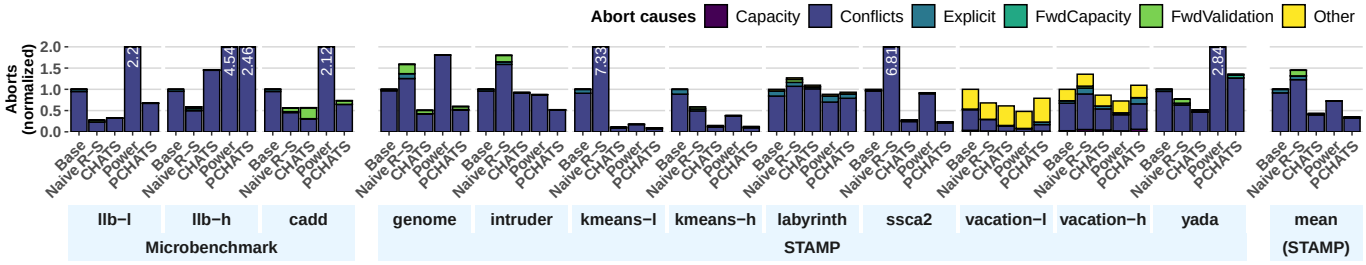


Fig. 5: Aborted transactions split by the reasons that caused the abort

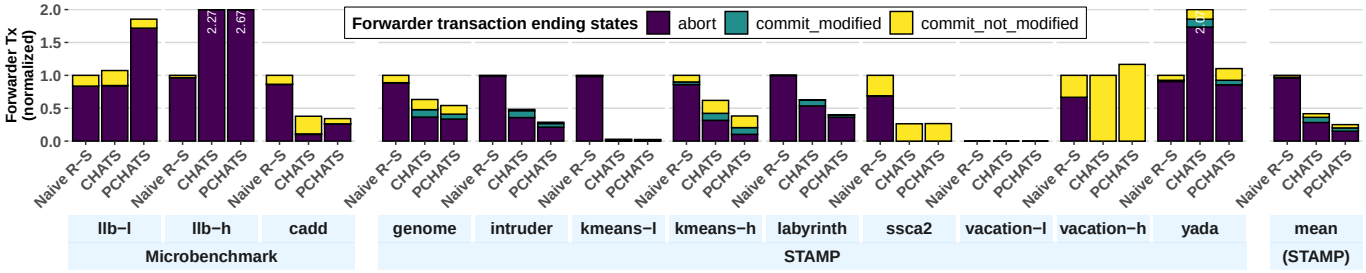


Fig. 6: Amount of executed transactions that conflicted and forwarded data. Bars are split marking how the transaction finished its execution.

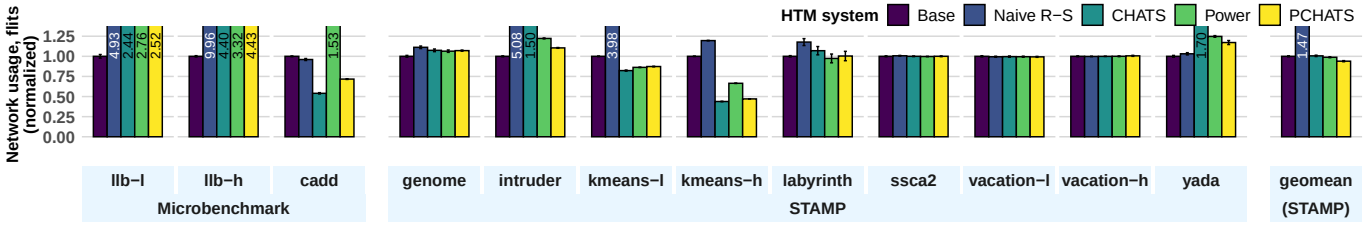


Fig. 7: Normalized network usage in flits sent through the interconnection network

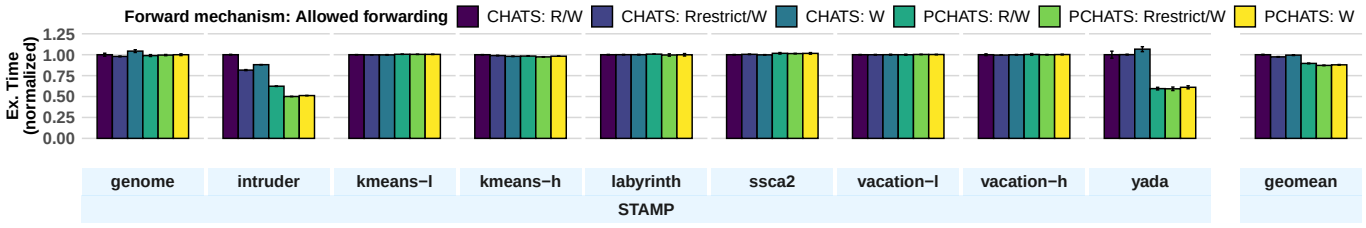


Fig. 8: Effect of enabling different types of blocks to be forwarded. R/W, W and Rrestrict/W types are considered according to what explained in Section VI-D. All results are normalized to CHATS with R/W.

conflicts in transactions that are finishing the execution of this phase. CHATS and more notably PCHATS can circumvent all those obstacles for concurrency exploitation. CHATS easily handles traversing the shared structure, but tree balancing and external dependencies with structures can pose difficulties in avoiding cyclic situations. PCHATS can observe not only locally generated context information such as PiC but also which transaction is experiencing more contention via the power token. In addition, even with such contended transactions, CHATS can forward data without risking the survivability of the producer as observed in Fig. 6, ensuring that data forwarded by it will be potentially correct. CHATS shows performance degradation over the baseline, but PCHATS manages to improve its performance by more than 30%.

We stated that CHATS correctly manages traversing data structures that are being modified. We confirm this with the llb benchmark where contention is consistently reduced when an affordable amount of shared memory locations are modified by all transactions. Nevertheless, we state the limits on CHATS with its high contention version, where all threads are modifying all memory locations randomly and causing extra aborts due to the need for transactional serialization. Even in this case, with an extra quantity of aborts as observed in Fig. 6, CHATS and PCHATS benefit from those producers that did not abort and committed unmodified values, thus performing better than the baseline and the naive requester-speculates versions. The behavior exposed by the intruder's *capture* phase is resembled in cadd. We observe how, even if transactions hold a shared modified memory address for a long time, CHATS manages to exploit parallelism by allowing several transactions to have local copies of those locations, leading to more transactions eventually committed.

In terms of efficiency, one might anticipate CHATS to exacerbate the utilization of the interconnection network due to the periodic validations that it requires. However, as observed in Fig. 7, the amount of flits (information units) sent through the interconnect is lower for CHATS and PCHATS configurations. This perhaps unexpected outcome is reasonable due to the reduction in the number of aborts, resulting in less wasted work and traffic. On the other hand, requester-speculates policies that do not incorporate any cycle avoidance mechanism, experience a significant increase in intercommunication network usage. This is attributed to a considerable rise in the number of aborts, as previously observed in Fig. 5.

A. Implementation decisions and sensitivity analysis

We performed several sensitivity analyses for the parameters mentioned in Section VI-D for each HTM system under consideration. The evaluation in the previous section uses the best cost-effective configuration for each system to guarantee a fair comparison between them.

1) Blocks to forward: In CHATS and, by extension, PCHATS, one configurable aspect is the selection of the type of data blocks eligible to be forwarded. There are two main options: allow forwarding of modified blocks only, or allow forwarding of both shared and modified blocks. Speculative

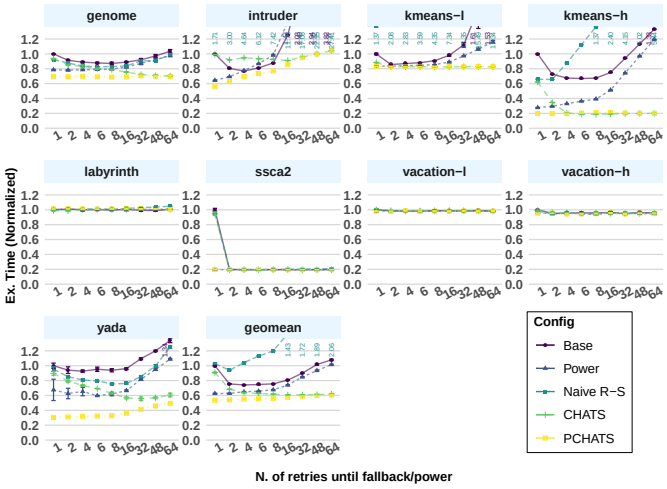
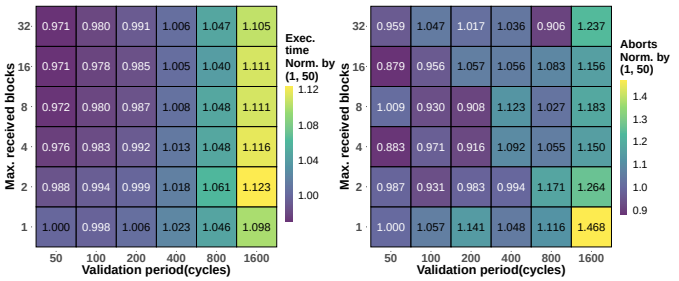


Fig. 9: Sensitivity analysis of execution time for a given number of transactional retries before resorting to fallback path

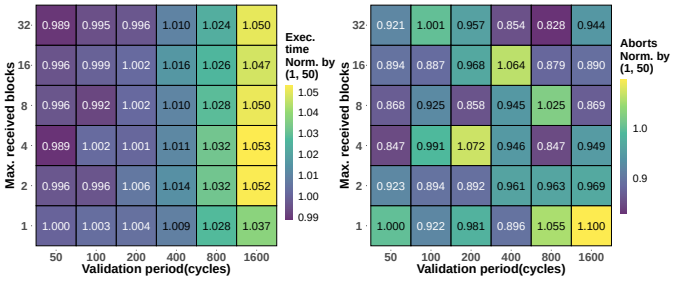
data blocks are forwarded via SpecResp messages, like modified values. This allows CHATS to forward, not only locally modified versions of data but blocks that have been only read too. Additionally, we introduced a heuristic that assesses whether an address slated for forwarding is already associated with an in-flight GETX, as explained in Section VI-D. Its effectiveness is demonstrated in our findings, as shown in Fig. 8. The results indicate a slight advantage in employing our heuristic across all implementations of CHATS. This strategy reduces the likelihood of unnecessary aborts by ensuring that forwarding decisions are informed by the current state of data requests.

2) Retry threshold until fallback path: Another critical configuration parameter to consider is the number of retries allowed before resorting to fallback mechanisms to guarantee forward progress. In Fig. 9 we observe that for the best-effort configuration, the best option is to use 6 retries in most benchmarks, while for Power-based systems it is better to reduce this number to 2. In PCHATS, using only 1 retry gives the best results. This makes sense because acquiring the Power token allows other transactions to be executed simultaneously without applying serialization. However, CHATS performs better with 32 retries for the same reason that PCHATS does with only 1. With a higher threshold, more transactions are allowed to re-execute and forward their data, giving more chances to exploit parallelism by acquiring potentially correct data and values for PiC. In PCHATS, the same happens, but power transactions themselves allow it while the token is in use.

3) Validation periodicity and VSB size: As a final measurement point, we conducted a study on key configurations for value forwarding: the periodicity of validations and how many blocks a transaction can speculate with simultaneously. Fig. 10 shows the execution time normalized to the left/down square (50 cycles / 1 block). We found a sweet spot for CHATS and PCHATS in terms of performance, efficiency, and cost-effectiveness with 4 VSB entries. With this small number of



(a) Execution time: CHATS (b) Aborts: CHATS



(c) Execution time: PCHATS (d) Aborts: PCHATS

Fig. 10: Sensitivity analysis showing how execution time (left) and aborts (right) change as the VSB size (Y-axis) and the validation interval (X-axis) vary

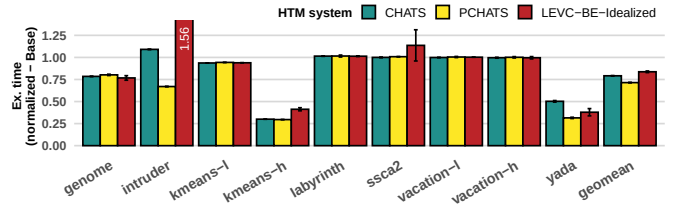


Fig. 11: Execution time over the best-effort baseline

entries, we manage to introduce a minor storage overhead of just 280 bytes. In addition, we lose only 0.005% of the average performance compared to the version that uses 32 entries for CHATS while having the best result in terms of aborts. For PCHATS, 4 entries remain the best option.

B. Comparison against similar proposals

We compare the relative performance of CHATS with an idealized version of LEVC [30] since both are requester-speculates approaches. Fig.11 summarizes the results regarding execution time, with all values normalized to the requester-wins best-effort baseline.

For intruder, CHATS and LEVC-BE-Idealized both struggle to avoid cyclic dependencies. However, PiC's local context information yields superior results in avoiding cycles than the static timestamp-based transactional ordering mechanism of LEVC-BE-Idealized. In LEVC-BE-Idealized, a producer can forward speculative data only to a single consumer, while CHATS does not have this limitation. In kmeans-h, CHATS performs optimally because of its ability to exploit producer-consumer dependencies, in contrast with the timestamp-based policy on which LEVC-BE-Idealized is based. In yada, CHATS performs worse than LEVC-BE-Idealized, mainly because its large transactions benefit from mechanisms that allow stalling. However, PCHATS outperforms LEVC-BE-Idealized in yada. Overall, we find that CHATS and PCHATS achieve average performance improvements of 4.6% and 12.3%, respectively, over LEVC-BE-Idealized, whose mechanism is considerably more complex.

VIII. Conclusion

Requester-speculates constitutes an effective conflict resolution policy for HTM, particularly useful in high contention scenarios. However, its implementation can significantly increase the complexity of HTM systems, requiring a careful management of speculative data and cyclic dependencies avoidance, and had not yet been explored in commercial-like systems, which are best-effort and abort-based.

We have proposed CHAined TransactionS (CHATS), a best-effort mechanism that dynamically selects between requester-speculates and requester-wins conflict resolution policies, based on dependencies created by previous speculatively forwarded data blocks, to avoid cycles. CHATS uses a simple mechanism to validate speculation, ensures proper commit order, avoids the transfer of ownership of speculated data, and encodes dependencies with negligible overhead.

With very little hardware overhead (280 bytes per core), CHATS can effectively increase concurrency between transactions, though impeding cyclic chains that would need to be broken and would cause cascaded aborts. As a result of this increased overlap in transactional execution, CHATS brings remarkable reductions in execution time (22% on average). Additionally, we show that our mechanism is capable of improving the performance of a state-of-the-art proposal (PowerTM), also conceived for best-effort HTM implementations (average reductions of 16% in execution time). Finally, we also demonstrate that the performance advantage obtained from the combination of CHATS and PowerTM (what we call PCHATS) increases to 28% on average.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (ECHO, grant agreement No. 819134), from the Spanish Ministry of Science, Innovation and Universities (MCIN) MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (DAMAS, grant PID2022-136315OB-I00), and from the MCIN grant MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR (HEEDA, grant TED2021-130233B- C33).

References

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

[2] U. Aydonat and T. S. Abdelrahman, "Hardware support for relaxed concurrency control in transactional memory," in *International Symposium on Microarchitecture (MICRO)*, 2010, pp. 15–26.

[3] 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.

[4] 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/

[5] 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, p. 1–7, 2011.

[6] 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.

[7] C. Blundell, A. Raghavan, and M. M. Martin, "Retcon: Transactional repair without replay," in *International Symposium on Computer Architecture (ISCA)*, 2010, p. 258–269.

[8] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood, "Tokentm: Efficient execution of large transactions with hardware transactional memory," in *International Symposium on Computer Architecture (ISCA)*, 2008, p. 127–138.

[9] 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," *SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 81–91, 2007.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] T. Harris, J. Larus, and R. Rajwar, *Transactional memory*, 2022.

[16] 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.

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

[18] 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

[19] 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

[20] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for ibm system z," in *International Symposium on Microarchitecture (MICRO)*, 2012, pp. 25–36.

[21] 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.

[22] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner, "Improving in-memory database index performance with intel® transactional synchronization extensions," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 476–487.

[23] 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.

[24] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *Computer Architecture Letters (CAL)*, vol. 19, no. 2, pp. 106–109, 2020.

[25] 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.

[26] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: decoupling performance and correctness," in *International Symposium on Computer Architecture (ISCA)*, 2003, p. 182–193.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] X. Qian, B. Sahelices, and J. Torrellas, "Bulksmt: Designing smt processors for atomic-block execution," in *International Symposium on High-Performance Computer Architecture (ISCA)*, 2012, pp. 1–12.

[34] 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.

[35] 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.

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

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

[38] A. Shriraman and S. Dwarkadas, "Refereeing conflicts in hardware transactional memory," in *International conference on Supercomputing (ICS)*, 2009, pp. 136–146.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

APPENDIX

A. Abstract

This artifact consist on a whole cloud environment that gives support for full-system execution and fast-forwarding with virtualization over container for gem5 simulator. Additionally, it holds the code and simulator used to perform all the experiments in CHATS paper. Following the detailed instructions given in this appendix and in the *readme.md* document in the repository, the user will be able to consistently reproduce exactly the same experiments to obtain the results that appear in this manuscript.

B. Artifact check-list (meta-information)

- **Algorithm:** Container orchestration and simulation
- **Program:** gem5, Docker, Docker-compose, KVM, Slurm, Containerd
- **Compilation:** Docker: 27.1.1, Docker-compose: 2.29.1, Containerd: 1.7.19, Slurm: 24.05
- **Data set:** STAMP benchmarks and microbenchmarks from section VI (included in the full-system image)
- **Run-time environment:** Ubuntu 20.04. Nevertheless, modules from this version are locally binded to the container. It is required to have KVM support.
- **Hardware:** Memory: min. 4GiB
- **Output:** All the statistic files generated from the experiments.
- **Experiments:** Base RW, Naive R-S, CHATS, Power, PCHATS, LEVC-BE-idealized. Sensibility analysis and final comparison with the best-case configuration for each system.
- **How much disk space required (approximately)?:** 60GiB
- **How much time is needed to prepare workflow (approximately)?:** 30min.
- **How much time is needed to complete experiments (approximately)?:** Depends on your system, but around 3 days.
- **Publicly available?:** Yes
- **Workflow automation framework used?:** Docker-compose, Make
- **Archived (provide DOI)?:** https://zenodo.org/doi/10.5281/zenodo.13741948

C. Description

This artifact is contains all the instructions and specification files to build, run and orchestrate a set of containers that will contain the built ready-to-use gem5 simulator. Additionally, some scripts are provided to exactly replicate all the experiments that were performed for this paper.

The application itself is built up as a Slurm-based system (modified from https://github.com/giovtorres/slurm-docker-cluster) that can be used to submit gem5 jobs that will be executed in every node that is isolated for this purpose. All nodes are automatically configured and prepared to execute when the application is built so the user do not have to mess with any system detail. Once the simulations finish, all results will be placed in a volume (folder) shared between the host machine and all the containers. All this can be performed by the user with simple commands like make build.

1) How to access: The application is fully available (source and orquestrator instructions) at https://github.com/nikiitin/MICRO24-521.

2) Hardware dependencies: Allow virtualization in kernel at your BIOS/UEFI configuration.

3) Software dependencies: This application makes use of Docker (27.1.1), Docker compose (2.29.1), Make, KVM, wget and Shell-based command interpreter.

4) Data sets: CHATS evaluation is performed using STAMP benchmark suite and the microbenchmarks mentioned in section VI. All the benchmarks are already included in the disk image loaded to the container.

D. Installation

To install the application, first clone the repository from git. Once cloned, enter in the directory and build the application with the following commands:

```
cd MICRO24_521
# NOTE: here root permissions are required as docker
     must be executed as sudo
sudo make build
```

You should have, either root privileges or be able to execute with root permissions as docker and docker compose require it to build. During the application build, the make script will try to detect its missing dependencies. Whenever a dependency is missing, the user will be asked if he wants to install it or not, if yes (y), then it will require sudo permissions to install the software in the kernel.

In case your machine does not have virtualization enabled, you can avoid using KVM by building a version that uses AtomicSimpleCPU model from gem5 to fast-forward until region of interest begins. Note that using a simulated model will take by far more time than using virtualization, but it should be compatible with any system. However, sudo will still be required as docker needs to build the containers for the application.

```
sudo make build_atomic
```

The application additionally, will prompt the user with the information of the current build step and what is being performed. Along all this information, it will print the configuration being used. This configuration is calculated taking into account the system specifications to automatically tune the container in an optimal way. Nevertheless, the user is able to change these values from the Makefile. Finally, whenever all the software is installed correctly and containers correctly built, it will print a message with “Everything built correctly!”. This is the signal to go into the next objective and do make run, again with sudo privileges.

```
    sudo make run
```

This will load and start all the containers with all the cluster nodes and the built and ready to use gem5 application. The next step is to connect with the container executing the slurm controller daemon. This will be the ”user” to access the rest of the system.

```
    sudo make connect
```

This command will connect to the controller node with a /bin/bash process directed to your shell. Here you can find the gem5 directory, along all the system files.

E. Experiment workflow

Once inside the node, dive into the folder that contains all the scripts to submit your experiment jobs.

```
cd /gem5/gem5_path/scripts/CHATS
```

Here you will find already configured python scripts with the simulation parameters that were used for this manuscript. The relation with figures is:

- config.chats.blocks.sensibility.py: Fig.10
- config.chats.fwdrvsfwdw.py: Fig.8
- config.chats.main.py: Fig.1, Fig.4, Fig.5, Fig.6, Fig.7
- config.chats.retry.sensibility.py: Fig.9

All them contains the definition for the experiments to perform and will generate a file hirerarchy at */gem5/result* folder, which is binned to the host system. All these results will be available to use by the host in the binned-volume at path-to-application/results.

F. Evaluation and expected results

Those mentioned configuration files have each one matching bash script file. Those files are the ones that must be executed in order to submit works to slurm. Each one will be used to submit the experiments configured with its python configuration file. To execute it, use the following command in CHATS script directory:

```
./generate-simulations-(Experiment) [--enqueue]
```

If executing without the *–enqueue* option, it will only generate the file hierarchy aforementioned and write all the simulations prepared to execute but without submitting a work for it. With *–enqueue*, all those simulation scripts will be submitted to slurm and will be executed by the nodes. You can check the status of each simulation with following commands:

```
squeue
/gem5/gem5_path/scripts/check_simulations.sh /
gem5/results
```

The second listing is a script that prompt the user the current status of the simulation and the folder it is stored in.

Finally, inside every simulation folder, you will find each *stats.txt* file with all the results from the simulation. All those files are available from outside the container to the host machine due to the binded volume used in the folder *path-to-application/results*. Once checked that all simulations have finished for all experiments, there is another make rule that produces all the plot images with the statistic files:

```
# From the repository root file
make plot
```

This will execute RING-5 (https://github.com/nikiitin/RING-5) to generate all the plots for every configuration. This tool should had been installed at build time and will produce the plots (along with csv files with all the results) in the plots folder.

G. Experiment customization

You can customize the parameters for each experiment at */gem5/gem5_paths/scripts/CHATS*. In each *config.chats.*.py* you can find all the explicit parameters that each job will use. You can modify already-existing ones or make new configuration files that are submitted by its bash script matching file. You can define new parameters from those defined at */gem5/gem5_path/scripts/run-scripts/options.py*. To define several simulations varying only one parameter, write an entry like the following one:

```
parameter_to_specify: Vary(different, values),
```

H. Notes

For more information refer to the *Readme.md* file from the repository.