Using a Reconfigurable L1 Data Cache for Efficient Version Management in Hardware Transactional Memory

Adrià Armejach*† Azam Seyedi*† Rubén Titos-Gil‡
Ibrahim Hur* Adrián Cristal*§ Osman Unsal* Mateo Valero*†
*BSC-Microsoft Research Centre †Universitat Politècnica de Catalunya
‡Universidad de Murcia §IIIA - Artificial Intelligence Research Institute CSIC - Spanish National Research Council

***Abstract*—Transactional Memory (TM) potentially simplifies parallel programming by providing atomicity and isolation for executed transactions. One of the key mechanisms to provide such properties is version management, which defines where and how transactional updates (new values) are stored. Version management can be implemented either eagerly or lazily. In Hardware Transactional Memory (HTM) implementations, eager version management puts new values in-place and old values are kept in a software log, while lazy version management stores new values in hardware buffers keeping old values in-place. Current HTM implementations, for both eager and lazy version management schemes, suffer from performance penalties due to the inability to handle two versions of the same logical data efficiently.**

In this paper, we introduce a reconfigurable L1 data cache architecture that has two execution modes: a 64KB general purpose mode and a 32KB TM mode which is able to manage two versions of the same logical data. The latter allows to handle old and new transactional values within the cache simultaneously when executing transactional workloads. We explain in detail the architectural design and internals of this Reconfigurable Data Cache (RDC), as well as the supported operations that allow to efficiently solve existing version management problems. We describe how the RDC can support both eager and lazy HTM systems, and we present two RDC-HTM designs. Our evaluation shows that the Eager-RDC-HTM and Lazy-RDC-HTM systems achieve 1.36× and 1.18× speedup, respectively, over state-of-the-art proposals. We also evaluate the area and energy effects of our proposal, and we find that RDC designs are 1.92× and 1.38× more energy-delay efficient compared to baseline HTM systems, with less than 0.3% area impact on modern processors.

***Index Terms*—hardware transactional memory; version management; reconfigurable cache**

I. Introduction

Transactional Memory (TM) [13] aims to make shared-memory parallel programming easier by abstracting away the complexity of managing shared data. TM is an optimistic concurrency mechanism, a transaction that executes successfully commits, while one that conflicts with a concurrent transaction aborts. Hardware TM (HTM) implementations [7, 12, 20, 29] generally provide higher performance compared to Software TM (STM) or Hybrid TM (HyTM) systems. This paper focuses on HTM implementations, where three key design dimensions limit system performance: conflict detection, conflict resolution and version management [5]. Each of these dimensions could be implemented in an eager (at the time of a transactional read or write) or lazy (at transaction commit time) fashion. The first generation of HTM implementations were monolithically eager [20, 29] or lazy [7, 12].

The conflict detection policy defines *when* the system will check for conflicts by inspecting the read- and write-sets (addresses read and written by a transaction) whereas conflict resolution states *what* to do when a conflict is detected. The second generation of HTM implementations focused particularly on conflict detection and resolution policies by adopting flexible mechanisms such as detecting write-write conflicts eagerly and read-write conflicts lazily [26], detecting all conflicts eagerly and resolving them lazily [28], and providing the flexibility of detecting and resolving conflicts either eagerly or lazily, depending on the application [21]. In this paper, we focus on *version management*, the third key HTM design dimension. Version management handles the way in which the system stores both old (original) and new (transactional) versions of the same logical data.

Early TM research suggests that short and non-conflicting transactions are the common case [9], making the commit process much more critical than the abort process. However, newer studies that present larger and more representative workloads [6] show that aborts can be as common as commits and transactions can be large and execute with a high conflict rate. Thus, version management implementation is a key aspect to obtain good performance in HTM systems, in order to provide efficient abort recovery and access to two versions (old and new) of the same logical data. However, traditional version management schemes, eager or lazy, fail to efficiently handle both versions. An efficient version management scheme should be able to read and modify both versions during transactional execution using a fast hardware mechanism. Furthermore, this hardware mechanism should be flexible enough to work with both eager and lazy version management schemes, allowing it to operate with multiple HTM systems.

In Section II we introduce such a hardware mechanism: *Reconfigurable Data Cache (RDC)*. The RDC is a novel L1D cache architecture that provides two execution modes: a 64KB general purpose mode, and a 32KB TM mode that is able to manage efficiently two versions of the same logical data. The latter mode allows the RDC to keep both old and new values in the cache; these values can be accessed and modified within the cache access time using special operations supported by the RDC.

In Section III we discuss how the inclusion of the RDC affects HTM systems and how it improves both eager and lazy versioning schemes, and in Section IV we introduce two new HTM systems, *Eager-RDC-HTM* and *Lazy-RDC-*

HTM, that use our RDC design. In traditional eager versioning systems, old values are logged during transactional execution, and to restore pre-transactional state on abort, the log is accessed by a software handler. RDC eliminates the need for logging as long as the transactions do not overflow the L1 RDC cache, making the abort process much faster. In lazy versioning systems, aborting a transaction implies discarding all modified values from the fastest (lowest) level of the memory hierarchy, forcing the system to re-fetch them once the transaction restarts. Moreover, because speculative values are kept in private caches, a large amount of write-backs may be needed to publish these values to the rest of the system. With RDC, old values are quickly recovered in the L1 data cache, allowing faster re-execution of the aborted transactions. In addition, most of the write-backs can be eliminated because of the ability to keep two different versions of the same logical data.

In Section V we provide an analysis of the RDC. We introduce the methodology that we use to obtain the access time, area impact, and energy costs for all the RDC operations. We find that our proposed cache architecture meets the target cache access time requirements and its area impact is less than 0.3% on modern processors.

In Section VI we evaluate the performance and energy effects of our proposed HTM systems that use the RDC. We find that, for the STAMP benchmark suite [6], Eager-RDC-HTM and Lazy-RDC-HTM achieve average performance speedups of 1.36× and 1.18×, respectively, over the state-of-the-art HTM proposals. We also find that the power impact of RDC on modern processors is very small, and that RDC improves the energy delay product of baseline HTM systems, on average by 1.93× and 1.38×, respectively.

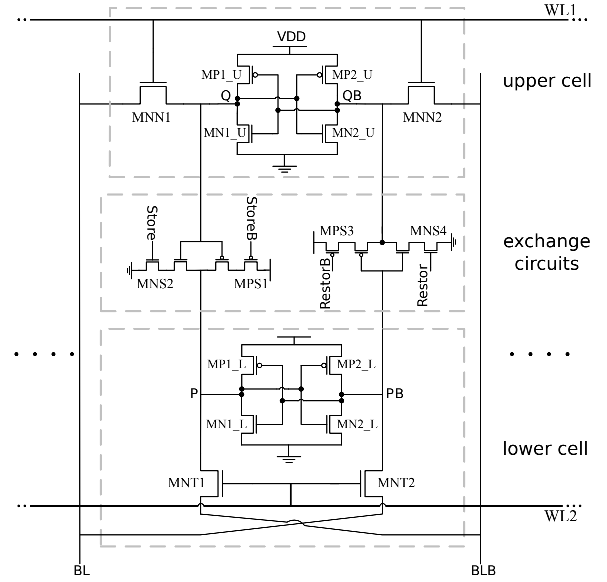


Fig. 1. Circuit schematic of the e-cell. A typical cell design is extended with an extra cell and exchange circuits.

Operation	Description
UWrite	Write to an upper cell cache line by activating WL1
URead	Read from an upper cell cache line by activating WL1
LWrite	Write to a lower cell cache line by activating WL2
LRead	Read from a lower cell cache line by activating WL2
Store	∼Q→P: Store an upper cell to a lower cell cache line
Restore	∼PB→QB: Restore a lower cell to an upper cell cache line
ULWrite	Write to both cells simultaneously by activating WL1 and WL2
StoreAll	Store all upper cells to their respective lower cells

Fig. 2. Brief descriptions of the RDC operations.

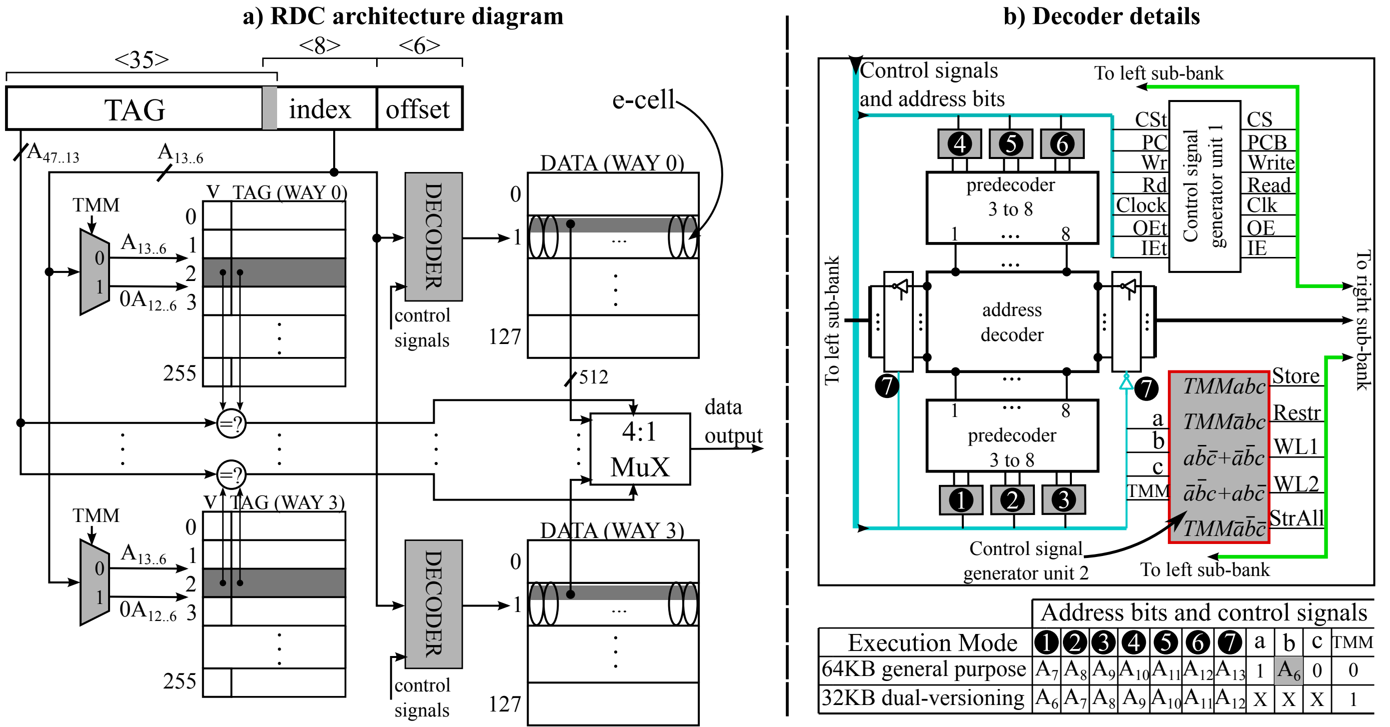
II. The Reconfigurable Data Cache

We introduce a novel L1 data cache structure: the *Reconfigurable Data Cache* (RDC). This cache, depending on the instruction stream, dynamically switches its configuration between a 64KB general purpose data cache and a 32KB TM mode data cache, which manages two versions of the same logical data. Seyedi *et al.* [25] recently proposed the low-level circuit design details of a dual-versioning cache for managing data in different optimistic concurrency scenarios. Their design requires a cache to always be split between two versions of data. We enhance that design to make it dynamically reconfigurable, and we tune it for specific TM support.

A. Basic Cell Structure and Operations

Similar to prior work [25], in RDC two bit-cells are used per data bit, instead of one as in traditional caches. Figure 1 shows the structure of the RDC cells, which we name *extended cells* (e-cells). An e-cell is formed by two typical standard 6T SRAM cells [22], which we define as the *upper cell* and the *lower cell*. These two cells are connected via two *exchange circuits*, that completely isolate the upper and lower cells from each other and reduce leakage current. To form a cache line (e.g., 64 bytes), 512 e-cells are placed side by side and connected to the same word lines (WL).

In Figure 2 we briefly explain the supported operations for the RDC. `URead` and `UWrite` are typical SRAM read and write operations performed at the upper cells; analogously, `LRead` and `LWrite` operations do the same for the lower cells. The rest of the operations cover TM version management needs, and enable the system to efficiently handle two versions of the same logical data. We use `Store` to copy the data from an upper cell to its corresponding lower cell. Basically, `Store` turns the left-side exchange circuit on, which acts as an inverter to invert Q to P; the lower cell keeps the value of P when `Store` is inactive, and it inverts P to PB, so that PB has the same value as Q. Similarly, to restore data from a lower cell to its corresponding upper cell, we activate `Restore`. Finally, `ULWrite` is used to write the same data to upper and lower cells simultaneously. All these operations work at cache line granularity; however, an operation to simultaneously copy (*Store*) all the upper cells in the cache to their corresponding lower cells is also necessary, we call this operation `StoreAll`. Note that this is an intra–e-cell operation done by activating the small exchange circuits. Therefore, the power requirements to perform this operation are acceptable, as we



	Address bits and control signals										
Execution Mode	❶	❷	❸	❹	❺	❻	❼	a	b	c	TMM
64KB general purpose	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	1	A_6	0	0
32KB dual-versioning	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	X	X	X	1

Fig. 3. (a) RDC architectural diagram. Considering a 4-way RDC, with 64B cache-lines and 48b addresses — (b) Decoder details and associated signals used for each execution mode. Depending on the TMM signal, address bits and control signals for a execution mode are generated and passed to the decoder.

show in our evaluation, because most of the components of the cache are not involved in this operation.

B. Reconfigurability: RDC Execution Modes

The reconfigurable L1 data cache provides two different execution modes. The execution mode is indicated by a signal named *Transactional Memory Mode* (TMM). If the TMM signal is not set, the cache behaves as a 64KB general purpose L1D cache; if the signal is set, it behaves as a 32KB cache with the capabilities to manage two versions of the same logical data. Figure 3 shows an architectural diagram of RDC, the decoder details and its associated signals, which change depending on the execution mode; considering 48-bit addresses and a 4-way organization with 64-byte cache lines.

64KB general purpose mode: In this mode, the upper and lower bit-cells inside of an e-cell contain data from different cache lines. Therefore, a cache line stored in the upper cells belongs to cache set *i* in way *j*, while a cache line stored in the corresponding lower cells belongs to set *i+1* in way *j* (i.e., consecutive sets in the same way). This mode uses the first four operations described in Figure 2, to perform typical read and write operations as in any general purpose cache. Figure 3a shows an architectural diagram of the RDC. As can be seen in the figure, the most significant bit of the index is also used in the tags to support the 32KB TM mode with minimal architectural changes, so tags have fixed size for both modes (35 bits). The eight index bits ($A_{13..6}$) are used to access the tags (since TMM is not set) and also sent to the decoder. In Figure 3b it can be seen how the seven most significant bits of the index are used to address the cache entry while the least significant bit (A_6) determines if the cache line is located in the upper or the lower cells, by activating *WL1* or *WL2* respectively.

32KB TM mode: In this mode, each data bit has two versions: old and new. Old values are kept in the *lower* cells and new values are kept in the *upper* cells. These values can be accessed, modified, and moved back and forth between the upper and lower cells within the access time of the cache using the operations in Figure 2. To address 32KB of data, only half of the tag entries that are present in each way are necessary. For this reason, as can be seen in Figure 3a, the most significant bit of the index is set to '0' when the TMM signal is active. So, only the top-half tag entries are used in this mode. Regarding the decoder (Figure 3b), in this mode, the most significant bit of the index is discarded, and the rest of the bits are used to find the cache entry, while the signals *a*, *b*, and *c* select the appropriate signal(s) depending on the operation needed.

Reconfigurability considerations: Reconfiguration is only accessible in kernel mode. The binary header of a program indicates whether or not a process wants to use a general purpose cache or a TM mode cache. The OS sets the RDC to the appropriate execution mode when creating a process, and switches the mode when context switching between processes in different modes. In order to change the RDC execution mode, the OS sets or clears the TMM signal and flushes the cache in a similar way the WBINVD (write back and invalidate cache) instruction operates in the x86 ISA.

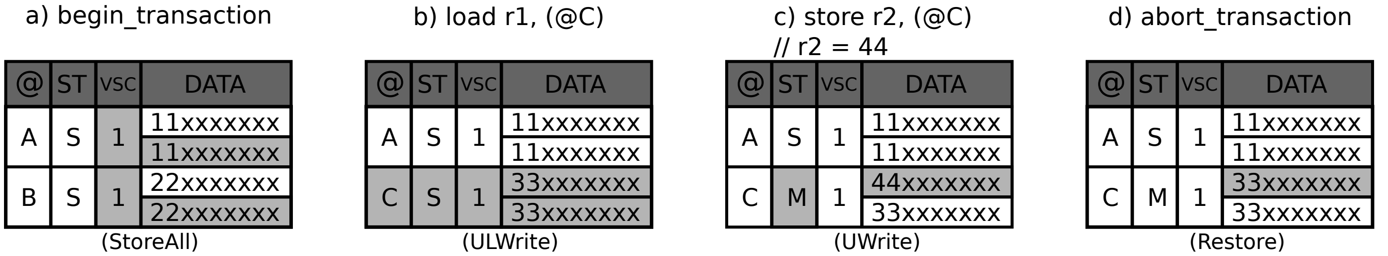


Fig. 4. A simple protocol operation example, assuming a 2-entry RDC. Shaded areas indicate state changes. (a) Creation of the shadow-copies, in the lower cells, at the beginning of a transaction — (b) A load operation that modifies both the upper and the lower cells in parallel (ULWrite) — (c) A line update, both old and new values are sharing the same cache entry — (d) Restoring old values in the RDC when the transaction is aborted.

III. Using the Reconfigurable Data Cache in Hardware Transactional Memory: RDC-HTM

In this section, we describe how our RDC structure can be used in both eager and lazy version management HTM schemes. For the rest of this section, we consider that the RDC executes in 32KB TM mode. In HTM systems, we distinguish four different execution phases when executing a transactional application: (1) non-transactional execution, (2) transactional execution, (3) commit, and (4) abort. When the RDC is used as L1 data cache during the non-transactional execution phase, the system follows the rules established by the underlying coherence protocol, but in the other three phases special considerations are required, which we detail in the following subsections.

A. Transactional Execution

One key insight of a RDC-HTM system is to maintain, during the execution of a transaction, as many valid committed values (non-transactional) as possible in the lower cells of the RDC. We name these copies of old (non-transactional) values *shadow-copies*. By providing such shadow-copies, in case of abort, the system can recover pre-transactional state with fast hardware `Restore` operations, partially or completely, performed over transactionally modified lines.

Figure 4 depicts a simple scenario of the state changes in RDC during a transactional execution that aborts. At the beginning of the transaction, the system issues `StoreAll` that creates valid shadow-copies for the entire cache in the lower cells (Figure 4a). We assume that this operation is triggered as a part of the begin_transaction primitive. In addition, during the execution of a transaction, shadow-copies need to be created for the new lines added to the L1 RDC upon a miss. This task does not take extra time, because the design of the RDC allows for concurrent writing to the upper and lower cells using the `ULWrite` operation (Figure 4b).

We add a *Valid Shadow Copy* (VSC) bit per cache-line to indicate whether the shadow-copy is valid or not for abort recovery. The system prevents creation of shadow-copies if a line comes from the L2 cache with transactionally modified state. Thus, if a shadow-copy needs to be created, an `ULWrite` operation is issued, otherwise an `UWrite` operation is issued. The VSC bit is set for a specific cache line if a `Store` or an `ULWrite` is issued; but, if an `StoreAll` is issued, the VSC bits of all lines are set. The VSC bit does not alter the state transitions in the coherence protocol.

Note that without VSC bits, in a lazy version management system, the use of more than one level of transactional caches would allow speculatively modified lines to be fetched from the L2 to the L1 cache, creating shadow-copies of non-committed data. A similar problem would occur in eager versioning systems as well, because transactional values are put in-place. Therefore, in both version management schemes, creating shadow-copies of non-committed data could lead to consistency problems if data was later used for abort recovery.

Eager version management. In traditional eager versioning systems, to recover pre-transactional state in case of abort, an entry with the old value is added in the undo log for every store performed during transactional execution [20, 29]. In a RDC-HTM implementation, on the other hand, the system keeps old values in shadow-copies, which are created either at the beginning of a transaction (Figure 4a) or during its execution (Figure 4b) with no performance penalty.

Note that in a RDC-HTM system, logging of old values is still necessary if the write-set of a transaction overflows the L1 cache. We define the new logging condition as an eviction of a transactionally modified line with the VSC bit set. When this logging condition is met, the value stored in the shadow-copy is accessed and logged. As an example, in Figure 4c, if the cache-line with address `C` was evicted, the system would log the shadow-copy value (lower cells) to be able to restore pre-transactional state in case of abort. To cover the cost of detecting the logging condition, we assume that logging process takes one extra cache operation; however, because the RDC-HTM approach significantly reduces the number of logged entries, the extra cache operation for logging does not affect performance, see Section VI-B.

Lazy version management. Lazy versioning systems, in general, do not write-back committed data to a non-transactional level of the memory hierarchy at commit time [7, 28], because that incurs significant commit overhead. Instead, only addresses are sent, and the directory maintains the ownership information and forwards potential data requests. Thus, repeated transactions that modify the same cache-line require a write-back of the cache-line each transaction. When using the RDC, however, unlike previous proposals [3, 7, 28], repeated transactions that modify the same blocks are not required

to write-back, resulting in significant performance gains, see Section VI-C, and less pressure for the memory hierarchy.

Cache replacements and data requests from other cores need additional considerations. If a previously committed line with transactional modifications, i.e., the committed value in the shadow-copy (lower cells) and the transactional value in the upper cells, is replaced, the system first writes back the shadow-copy to the closest non-transactional level of the memory hierarchy. If a data request is forwarded by the directory from another core and if the VSC bit of the related cache line is set, the requested data will be stored in the shadow-copy (lower cells), because the shadow-copy always holds the last committed value. Note that a shadow-copy can be read with an `LRead` operation.

B. Committing Transactions

In eager versioning systems, the commit process is a fast and a per-core local operation, because transactional values are already stored in-place. Committing releases isolation by allowing other cores to load lines that are modified by the transaction. In contrast, lazy systems make transactional updates visible to the rest of the system at commit time, and conflicting transactions are aborted. A RDC-HTM system needs one additional consideration at commit time, to flush-clear the VSC bits. At the beginning of the succeeding transaction, all shadow copies are created again, setting the VSC bits, and proceeding with the transactional execution process.

C. Aborting Transactions

Eager version management. In typical eager version management HTMs, pre-transactional values are stored in an undo log that is accessed using a software handler. For each entry in the log, a store is performed with the address and data provided. This way memory is restored to pre-transactional values.

With our proposal we intend to avoid the overhead of the undo log, either completely or partially. The abort process in an eager RDC-HTM is two-folded. First, as shown in Figure 4d, transactionally modified lines in the L1 cache, if their VSC bits are set, recover pre-transactional state using a hardware mechanism, `Restore`, provided by the RDC. Second, if there is any entry in the undo log, it will be unrolled issuing a store for each entry. By reducing the abort recovery time, the number of aborts decreases and the time spent in the backoff algorithm is minimized, as we show in our evaluation.

Lazy version management. In typical lazy version management HTMs, aborting transactions need to discard transactional data in order to restore pre-transactional state. Lazy systems invalidate the lines, in transactional caches, that are marked as transactionally modified with a fast operation that modifies the state bits. Invalidating these lines on abort implies that once the transaction restarts its execution the lines have to be fetched again. Moreover, current proposals [7, 28] often use multiple levels of the memory hierarchy to track transactional state, making the re-fetch cost more significant.

Cores	16 cores, 2 GHz, single issue, single-threaded
L1D cache	64KB 4-way, 64B lines, write-back, 2-cycle hit
L2 cache	8MB 8-way, 64B lines, write-back, 12-cycle hit
Memory	4GB, 350-cycle latency
Interconnect	2D mesh, 3-cycle link latency
L2 directory	full-bit vector sharers list, 6-cycle latency
Signatures	perfect signatures

Fig. 5. Base eager systems configuration parameters.

Because memory pre-transactional state is kept, partially or completely, in the RDC shadow-copies, it can be restored within the L1 cache with a `Restore` operation, see Figure 4d. Fast-restoring of the state in the L1 cache has three advantages: (1) it allows a faster re-execution of the aborted transaction, because transactional data is already in L1, (2) it allows more parallelism by reducing pathologies like convoying [5], and (3) it alleviates pressure in the memory hierarchy.

IV. RDC-BASED HTM SYSTEMS

In this section we introduce two new HTM systems, Eager-RDC-HTM and Lazy-RDC-HTM, that incorporate our RDC design in the L1 data cache. Both of these systems are based on state-of-the-art HTM proposals.

A. Eager-RDC-HTM

Eager-RDC-HTM extends LogTM-SE [29], where conflicts are detected eagerly on coherence requests and commits are fast local operations. Eager-RDC-HTM stores transactional values in-place but saves old values in the RDC, and if necessary, a per-thread memory log is used to restore pre-transactional state.

Figure 5 summarizes the system parameters that we use. We assume a 16-core CMP with private instruction and data L1 caches, where the data cache is implemented following our RDC design and with a VSC bit per cache-line. The L2 cache is multi-banked and distributed among cores with directory information. Cores and cache banks are connected through a mesh with 64-byte links that use adaptive routing. To track transactional read- and write-sets, the system uses signatures; because signatures may lead to false positives and in consequence to unnecessary aborts, to evaluate the actual performance gains introduced by Eager-RDC-HTM, we assume a perfect implementation, i.e., not altered by aborts due to false positives, of such signatures.

Similar to LogTM-SE, Eager-RDC-HTM uses stall conflict resolution policy. When a conflict is detected on a coherence message request, the requester receives a NACK (i.e., the request cannot be serviced), it stalls and it waits until the other transaction commits. This is the most common policy in eager systems, because it causes fewer aborts, which is important when software-based abort recovery is used. By using this policy we are also being conservative about improvements obtained by Eager-RDC-HTM over LogTM-SE.

The main difference between LogTM-SE and our approach is that we keep old values in the RDC, providing faster handling of aborts. In addition, although, similar to LogTM-SE, we have a software logging mechanism that stores old

Cores	32 cores, 2 GHz, single issue, single-threaded
L1D cache	64KB 4-way, 64B lines, write-back, 2-cycle hit
L2 cache	1MB 8-way, 64B lines, write-back, 10-cycle hit
Memory	4GB, 350-cycle latency
Interconnect	2D mesh, 10 cycles per hop
Directory	full-bit vector sharers list, 10-cycle hit directory cache

Fig. 6. Base lazy systems configuration parameters.

values, unlike LogTM-SE, we use this mechanism only if transactional values are replaced because of space constrains. In our approach, in case of abort, the state is recovered by a series of fast hardware operations, and if necessary, at a later stage, by unrolling the software log; the processor checks an *overflow bit*, which is set during logging, and it invokes the log software handler if the bit is set.

Logging policy implications. Since an evicted shadow-copy may need to be stored in the log, it is kept in a buffer, which extends the existing replacement logic, from where it is read and stored in the log if the logging condition is met, or discarded otherwise. Note that deadlock conditions, regarding infinite logging, cannot occur if the system does not allow log addresses to be logged, filtering them by address; because, for every store in the log (L1), the number of candidates in L1 that can be logged decreases by one.

We use physical addresses in the log because virtual addresses are not available when logging is postponed, as in our system, to a later stage. LogTM-SE stores virtual addresses in the log in order to support paging of transactional data, which is unlikely to happen because memory is touched recently. Thus, our system aborts transactions affected by paging, which incurs only a small overhead because paging within a transaction is not common. Considering that the logging is done by hardware, the log pages are defined as not accessible by other threads, and the abort handler register cannot be redefined by the user, using physical addresses is safe, even for undoing the log because paging is not supported.

B. Lazy-RDC-HTM

Lazy-RDC-HTM is based on a Scalable-TCC-like HTM [7], which is a directory-based, distributed shared memory system tuned for continuous use of transactions. Lazy-RDC-HTM has two levels of private caches tracking transactional state, and it has write-back commit policy to communicate addresses, but not data, between nodes and directories.

Our proposal requires hardware support similar to Scalable-TCC, where two levels of private caches track transactional state, and a list of sharers is maintained at the directory level to provide consistency. We replace the L1 data cache with our RDC design, and we add the VSC bit to indicate whether shadow copies are valid or not. Figure 6 provides the system parameters that we use.

We use Scalable-TCC as the baseline for three reasons: (1) to investigate how much extra power is needed in continuous transactional executions, where the RDC is stressed by the always-in-transaction approach, (2) to explore the impact of not writing back modified lines by repeated transactions, and (3) to present the flexibility of our RDC design by showing that it can be adapted efficiently to significantly different proposals.

Having an always-in-transaction approach can considerably increase the power consumption of the RDC, because at the beginning of every transaction an `StoreAll` operation is performed. We modify this policy by taking advantage of the fact that the cache contents remain unchanged from the end of a transaction until the beginning of the following transaction. Thus, in our policy, at commit time, the system updates, using an `Store` operation, the shadow-copies of the cache lines that are transactionally modified, i.e., the write-set.

Because, at commit time, the system writes back only addresses, committed values are kept in private caches and they can survive, thanks to the RDC, transactional modifications. Our approach can save significant amount of write-backs that occur due to modifications of committed values, evictions, and data requests from other cores.

In lazy systems, the use of multiple levels of private caches for tracking transactional state is common [7, 28] to minimize the overhead of virtualization techniques [8, 23]. Although our proposal is compatible with virtualization mechanisms, we do not implement them, because we find that using two levels of caches with moderate sizes is sufficient to hold transactional data for the workloads that we evaluate.

V. Reconfigurable Data Cache Analysis

We use CACTI 5 [27] to determine the optimal number and size of the components present in a way for the L1 data cache configuration that we use in our evaluation (see Figure 5). We construct, for one way of the RDC and one way of a typical 64KB SRAM, Hspice transistor level netlists that include all the components, such as the complete decoder, control signal units, drivers, and data cells. We simulate and optimize both structures with Hspice 2003.03 using HP 45nm Predictive Technology Model [1] for V_{DD}=1V, 2GHz processor clock frequency, and T= 25°C. We calculate the access time, dynamic energy, and static energy per access for all operations in RDC and SRAM. Our analysis indicates that our RDC design meets, as the typical SRAM, the target access time requirement of two clock cycles. Figure 7 shows the energy costs for typical SRAM and RDC operations.

In Figure 8 we show the layouts [2] of both the typical 64KB SRAM and RDC ways. Both layouts use an appropriate allocation of the stage drivers, and we calculate the area increase of the RDC over the typical SRAM as 15.2%. We believe that this area increase is acceptable considering the relative areas of L1D caches in modern processors. To support our claim, in Figure 9 we show the expected area impact of our RDC design on two commercial chips: IBM Power7 [14, 15], which uses the same technology node as our baseline systems and has large out-of-order cores, and Sun Niagara [16, 24], which includes simple in-order cores. We find that, for both chips, the sum of all the L1D areas represents a small percentage of the die, and our RDC proposal increases the overall die area by less than 0.3%.

Operation	Energy (pJ)	
	SRAM 64KB	RDC 64KB
Read/URead	170.7	188.2
Write/UWrite	127.3	159.1
LRead	-	190.0
LWrite	-	159.9
Store	-	175.3
Restore	-	180.4
ULWrite	-	168.5
StoreAll	-	767.8
Static	65.1	90.8

Fig. 7. Typical SRAM and RDC energy consumption per operation.

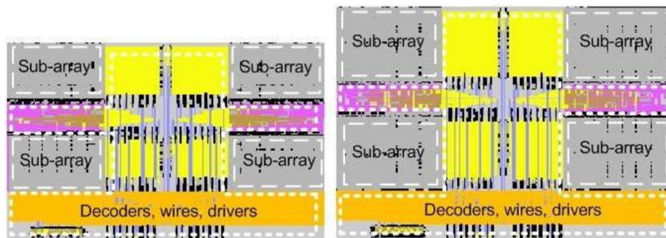


Fig. 8. Typical 64KB SRAM (left) and RDC (right) layouts. Showing one sub-bank, address decoders, wires, drivers, and control signals. The second symmetric sub-banks are omitted for clarity.

	IBM Power7	Sun Niagara
Technology node	$45nm$	$90nm$
Die size	$567mm^2$	$379mm^2$
Core size (sum of all cores)	$163mm^2$	$104mm^2$
L1 area (I/D) (sum of all cores)	$7.04/9.68mm^2$	$8.96/5.12mm^2$
L1 area (I/D) % of die	1.24/1.71%	2.36/1.35%
Die size increase with RDC	0.26%	0.21%

Fig. 9. Expected area impact of our RDC design on two commercial chips: the RDC increases die size by less than 0.3%.

VI. Evaluation

In this section we evaluate the performance, power, and energy consumption of Eager-RDC-HTM and Lazy-RDC-HTM using the STAMP benchmark suite [6]. We first describe the simulation environments that we use, then we present our results. In our evaluation we try to make a fair comparison with other state-of-the-art HTM systems; however, we do not intend to compare our systems against each other.

A. Simulation Environment

For Eager-RDC-HTM and LogTM-SE we use a full-system execution-driven simulator, GEMS [19], in conjunction with Simics [18]. The former models the processor pipeline and memory system, while the latter provides functional correctness in a SPARC ISA environment. GEMS introduces indeterminism to our experiments, so we repeat each experiment four times with different seeds and we average the results. For the evaluation of Lazy-RDC-HTM and Scalable-TCC we use M5 [4], an Alpha 21264 full-system simulator. We modify M5 to model a directory-based distributed shared memory system and an interconnection network between the nodes.

We use the STAMP benchmark suite with nine different benchmark configurations: Genome, Intruder, KMeans-high, KMeans-low, Labyrinth, SSCA2, Vacation-high, Vacation-low, and Yada. “high” and “low” workloads provide different conflict rates. We use the input parameters suggested by the developers of STAMP. Note that we exclude Bayes from our evaluation, because this application spends excessive amount of time in barriers due to load imbalance between threads, and this causes the results being not representative of the characteristics of the application.

B. Performance Results for Eager-RDC-HTM

Figure 10 shows the execution time breakdown for LogTM-SE with 64KB L1D, which serves as our baseline, for Eager-RDC-HTM, and for an Idealized eager versioning HTM. The Idealized HTM that we simulate is the same as Eager-RDC-HTM except that it has zero cycle abort and commit costs, it has an infinite RDC L1D cache, and the cache operations that involve storing and restoring values have no cost. In the figure, execution time is normalized to a 16-threaded LogTM-SE execution, and it is divided into non-transactional time (*non-tx*), barriers time (*barrier*), useful transactional time (*useful tx*), wasted work from aborted transactions (*wasted tx*), time spent in abort recovery (*aborting*), time spent by a transaction waiting for conflicts to be resolved (*stalled*), and time spent in the backoff algorithm executed right after an abort (*backoff*).

We find that, Figure 10, the overall performance of Eager-RDC-HTM is 1.36× better than LogTM-SE, and it is very close (within 1%, on average) to the Idealized HTM for all the workloads that we evaluate. We obtain significant speedups, e.g., 6.3× with Intruder, in applications which have contention and which are not constrained by large non-transactional or barrier execution times.

We identify three main reasons for the better performance of Eager-RDC-HTM over LogTM-SE. First, by providing a mechanism to successfully handle two different versions of the same logical data, we reduce the time spent in abort recovery process, on average, from 4.0% to 0.0%. The statistics in Figure 11 reveal that almost all aborts are resolved by hardware using the RDC capabilities, and for the majority of the workloads not even a single entry is unrolled from the log. Second, reducing the aborting time of a transaction prevents other transactions from aborting, because data owned by an aborting transaction cannot be accessed by other transactions. Figure 11 shows that, in Eager-RDC-HTM, the percentage of transactions that are aborted by another transaction, which executes the abort recovery phase, %AB Conf., decreases significantly, along with the total percentage of aborts. Finally, as a consequence of reduced abort rates, the time spent in stall and backoff phases is also reduced. The backoff and stall execution time in applications with high abort rates or with large transactions can represent a big percentage of the total execution time in LogTM-SE, e.g., up to 60% and 50% in Intruder and Yada, respectively. Thus, for this type of applications Eager-RDC-HTM performs significantly better than LogTM-SE.

Figure 11 shows the percentage of transactions of which write-sets overflow the L1D cache. Note that in Yada, 9.3% of transactions overflow L1D with their write-sets, but the

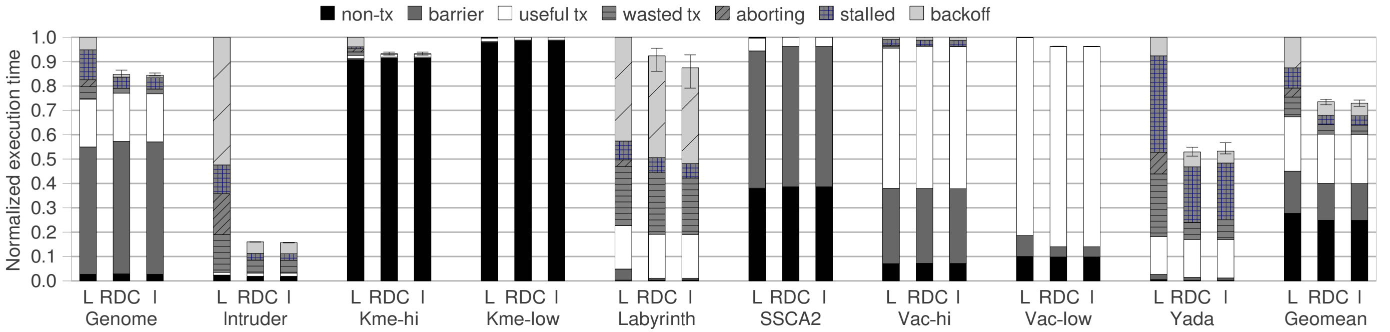


Fig. 10. Normalized execution time breakdown for 16 threads, in eager systems.
L — LogTM-SE; **RDC** — Eager-RDC-HTM; **I** — Idealized eager HTM

Application	Commits	LogTM-SE			Eager-RDC-HTM				
		%AB	%AB Conf.	Unrolled Entries	%AB	%AB Conf.	Unrolled Entries	%HW AB	%TX OVF
Genome	5922	34.8	19.7	6996	9.0	0.5	0	100.0	0.0
Intruder	11275	96.0	31.7	329891	86.2	2.1	0	100.0	0.0
KMeans-high	8238	50.7	30.2	6	3.0	0.0	0	100.0	0.0
KMeans-low	10984	4.5	33.8	0	0.6	0.0	0	100.0	0.0
Labyrinth	224	98.9	6.3	37602	98.4	0.1	35	99.8	13.2
SSCA2	47302	0.7	19.6	0	0.3	0.0	0	100.0	0.0
Vacation-high	4096	5.0	0.1	853	0.6	0.0	0	100.0	0.3
Vacation-low	4096	0.1	0.0	14	0.0	0.0	0	100.0	0.2
Yada	5330	69.6	7.7	164594	47.5	0.9	83	98.5	9.3

Fig. 11. Benchmark statistics for LogTM-SE and Eager-RDC-HTM.

Legend: **%AB** — Percentage of aborts, calculated as aborts/(aborts+commits); **%AB Conf.** — Percentage of aborts caused by aborting transactions; **Unrolled Entries** — Total number of log entries restored due to software aborts; **%HW AB** — Percentage of aborts resolved entirely by hardware; **%TX OVF** — Percentage of transactions of which write-set overflows L1.

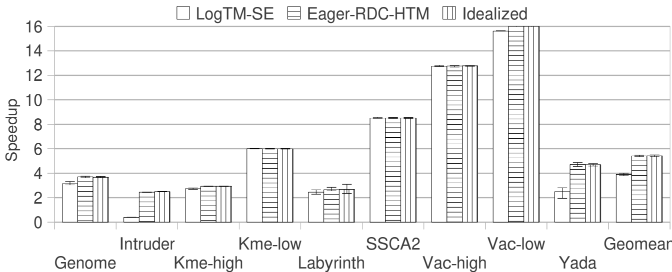


Fig. 12. Relative performance of 16-threaded applications compared to single-threaded LogTM-SE.

number of unrolled entries of the log in Eager-RDC-HTM is still much lower than it is in LogTM-SE, and its performance is close to Ideal. We believe that, even for coarser transactions, Eager-RDC-HTM can perform similar to the Idealized eager versioning HTM, because the two-folded abort process is a hybrid solution that minimizes the use of the log.

In Figure 12, we present the scalability results for LogTM-SE, Eager-RDC-HTM, and Ideal, each running 16-threaded applications. Applications with low abort rates, such as SSCA2, KMeans-low, and Vacation, have good scalability for all the evaluated HTM systems, and consequently Eager-RDC-HTM performs similar to LogTM-SE. In contrast, applications with coarser transactions and/or with high conflict rates, such as Genome, Intruder, and Yada, have worse scalability, and in general they fail to scale in LogTM-SE. However, Eager-RDC-HTM improves the performance of such applications significantly, being closer to ideal. Labyrinth does not improve substantially, because (1) it has large transactions with large data-sets, and (2) it has a notable abort rate that is not influenced by additional aborts due to other aborting transactions, putting pressure in the conflict resolution policy.

C. Performance Results for Lazy-RDC-HTM

Figure 13 shows the execution time breakdown for the lazy HTM systems that we evaluate, namely Scalable-TCC (with 64KB L1D), Lazy-RDC-HTM, and Idealized lazy HTM. The Idealized lazy HTM extends Lazy-RDC-HTM with instantaneous validation and data write-back at commit time, keeping a copy in the shared state; it serves as a good upper bound because it emulates a perfect, with limited hardware resources, lazy version management policy.

The results in Figure 13 are normalized to Scalable-TCC 32-threaded execution, and they are split into seven parts, namely *Barrier*, *Commit*, *Useful*, *StallCache*, *Wasted*, *WastedCache*, and *Aborting*. For committed transactions, we define “Useful” time as one cycle per instruction plus the number of memory accesses per instruction multiplied by the L1D hit latency, and we define “StallCache” as the time spent waiting for an L1D cache miss to be served. Analogously, for aborted transactions we define “Wasted” and “WastedCache”. The “Aborting” time

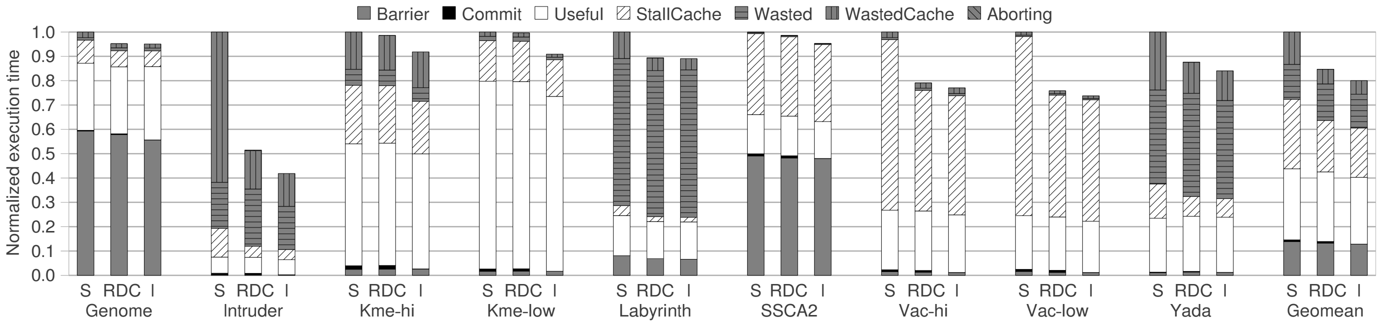


Fig. 13. Normalized execution time breakdown for 32 threads, in lazy systems.
S — Scalable-TCC; **RDC** — Lazy-RDC-HTM; **I** — Idealized lazy HTM

Application	Scalable-TCC				Lazy-RDC-HTM					
	Commits	Cycles CTX	WB per CTX	%AB	Cycles CTX	WB per CTX	%AB	%WB saved	Restore ATX	%AB time
Genome	11823	9896	5.2	7.7	8925	4.6	8.0	28.7	6.6	0.0
Intruder	20115	2686	21.0	75.6	1652	8.9	73.4	21.3	6.3	0.3
KMeans-high	26708	1717	0.9	33.6	1709	0.8	32.7	0.6	0.1	0.0
KMeans-low	68331	1811	0.4	5.3	1806	0.4	5.3	1.0	0.1	0.0
Labyrinth	1126	103056	277.5	37.5	91121	252.6	26.9	5.2	149.5	0.0
SSCA2	113122	3073	5.0	0.7	3026	4.9	0.7	2.0	1.2	0.0
Vacation-high	9332	10640	20.4	2.4	8364	18.8	3.1	23.1	4.8	0.0
Vacation-low	9261	9206	17.3	1.4	6965	14.7	1.9	27.4	5.1	0.0
Yada	5907	24921	67.9	41.5	20608	48.1	34.6	7.8	53.0	0.1

Fig. 14. Benchmark statistics for the Lazy-RDC-HTM system.

Legend: **Cycles CTX** — Average number of execution cycles for committed transactions; **WB per CTX** — Number of write-backs per committed transaction; **%WB saved** — Percentage of write-backs saved during execution; **Restore ATX** — Number of restores per aborted transaction; **%AB time** — Percentage of "Aborting" execution time.

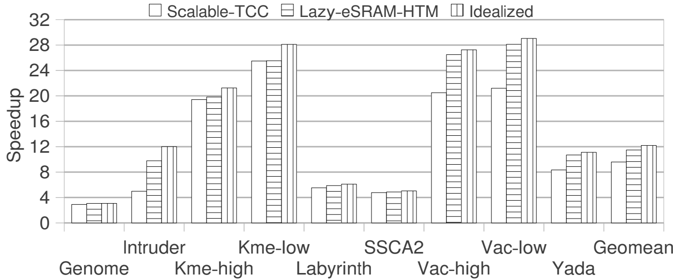


Fig. 15. Relative performance of 32-threaded applications compared to single-threaded Scalable-TCC.

is the overhead to restore the old values for recovering pre-transactional state, and it is defined as the number of L1D lines that are in the write-set and have a valid old value multiplied by the L1D hit latency. Note that because the "Aborting" time is a very small fraction of the total time, it is not noticeable in the figure.

Lazy-RDC-HTM reduces the average time spent in "StallCache" from 28.6% to 21.2% and in "WastedCache" from 13.3% to 6.0%. Because, on an abort we can recover pre-transactional state in the L1D cache by restoring the old values (shadow-copies) present in the lower cells, which makes re-execution of aborted transactions faster, as shown in Figure 14 (Cycles CTX). Moreover, since the L1D can operate with both old and new values, it is not necessary to write-back transactionally modified data for consecutive transactions that write the same set of lines, unless those lines need to be evicted or are requested by another core.

With Lazy-RDC-HTM, we achieve significant speedups over Scalable-TCC for Intruder, Labyrinth, Vacation, and Yada. For these benchmarks, the average execution time of committed transactions is reduced considerably, due to a lower number of write-back operations and the possibility to recover pre-transactional state at the L1D level. Genome and SSCA2 are constrained by extensive use of barriers, and KMeans with its small write-set and few aborts does not have much margin for improvement.

Figure 15 shows the scalability results for the 32-threaded executions. We find that Scalable-TCC achieves 9.6× speedup over single-threaded execution, while Lazy-RDC-HTM presents about 11.7× speedup. We also observe that the performance advantage of our approach is much higher for Intruder, Vacation and Yada compared to other applications.

Finally, in Figure 16, we present the effects of memory latency for four applications that achieve high performance improvements with Lazy-RDC-HTM. Notice that even for a low latency of 150 cycles, Intruder and Vacation workloads maintain good performance. Also, for latencies higher than 350 cycles, all applications obtain better results.

Applications	Eager HTM Systems					Lazy HTM Systems				
	L1D Power (mW)		Speedup	Power	EDP	L1D Power (mW)		Speedup	Power	EDP
	LogTM-SE	RDC HTM	(×)	Inc. (×)	Ratio (×)	STCC	RDC HTM	(×)	Inc. (×)	Ratio (×)
Genome	85.5	112.0	1.15	1.006	1.32	66.5	92.3	1.05	1.008	1.09
Intruder	73.1	98.7	6.31	1.007	39.48	73.6	110.3	1.93	1.010	3.70
KMe-high	92.2	117.7	1.09	1.006	1.18	78.5	103.6	1.01	1.006	1.02
KMe-low	94.6	118.2	1.00	1.005	1.00	83.7	108.2	1.00	1.006	1.00
Labyrinth	75.2	99.8	1.06	1.007	1.11	92.9	120.4	1.12	1.006	1.24
SSCA2	87.8	112.5	1.00	1.006	0.99	66.5	92.3	1.01	1.008	1.02
Vac-high	84.3	108.8	1.00	1.006	1.00	71.9	99.4	1.26	1.008	1.59
Vac-low	78.2	102.6	1.05	1.006	1.09	71.1	98.8	1.32	1.008	1.72
Yada	80.9	104.1	2.24	1.006	4.99	86.2	115.7	1.14	1.007	1.29

Fig. 17. Power consumption comparison of L1D caches and Energy Delay Product (EDP) comparison of entire systems.

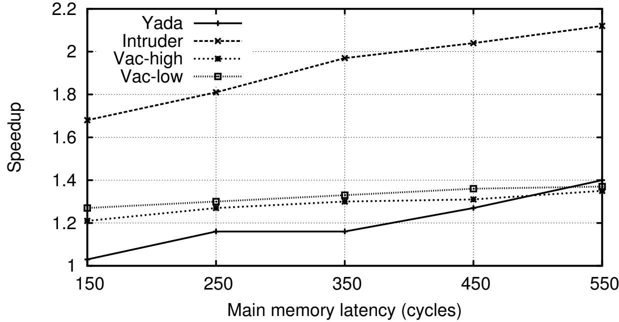


Fig. 16. Speedup variation with respect to main memory latency in lazy systems.

D. Power and Energy Results

To evaluate the power and energy effects of the RDC-HTM systems, we use the energy costs for the SRAM and RDC operations that we calculate in our analysis, Figure 7. We do not have at our disposal an accurate tool to measure the power consumption of our simulated HTM systems; therefore, considering the L1D areas shown in Figure 9, we assume that the L1D occupies 2% of the processor area, and we make an area-based power estimation.

We consider both dynamic and static power [10], and we present, in Figure 17, total power consumption, performance speedup, power increase, and energy delay product (EDP) effects of the RDC-HTM systems over systems with typical SRAM L1D caches. We find that the total processor power consumption increase in Eager-RDC-HTM and Lazy-RDC-HTM systems are 0.59% and 0.73%, respectively. We also find that RDC-based systems have significantly better EDP results compared to the systems with typical SRAMs: 1.93× and 1.38× better, for Eager-RDC-HTM and Lazy-RDC-HTM systems, respectively. Note that even with a much more conservative assumption of 5% for the total L1D area in the processor, total power increase due to the RDC L1D cache is about 1.5%, and the impact on our EDP results is negligible.

VII. Related Work

Ergin *et al.* [11] proposed a shadow-cell SRAM design for checkpointed register files to exploit instruction level parallelism. In that novel technique, each bit-cell has a shadow-copy cell to store a temporal value which can be recovered later. However, their design was not suitable for larger circuits, such as caches. Seyedi *et al.* [25] proposed a low-level circuit design of a dual-versioning L1D cache for different optimistic concurrency scenarios. In this design, the authors use exchange circuits between the cells, thus isolating both cells from each other, reducing leakage power. The authors give a complete description of the internal structure of the cache, with details of all its components, such as buffers, drivers, address and data interconnect, and additional circuitry. In addition, a brief discussion of the dual-versioning cache advantages in three optimistic concurrency techniques is also given. However, the authors do not show how the dual-versioning cache would be actually used in such scenarios, nor present a complete evaluation. Moreover, their design does not allow to dynamically reconfigure the cache, which implies a much larger area and power overhead for non transactional codes.

Herlihy and Moss introduced the first HTM design [13], which uses a separate small transactional cache to buffer both old and new values. In that novel design, commit and abort operations are local to the given processor and cache, and after abort, it allows transactions to be re-executed without needing to fetch lines back into the cache. However, the fully-associative transactional cache is orders of magnitude smaller than a L1 cache, limiting the size of transactions. Moreover, such design does not allow to spill lines to higher (private) levels of the memory hierarchy in lazy systems, or to use eager version management.

Transactional Coherence and Consistency (TCC) [12] implements lazy conflict detection and lazy version management. TCC guarantees forward progress and livelock-free execution without user-level intervention; however, it uses a common bus between cores, and transactions have to acquire a global token at commit time, limiting its scalability. Scalable-TCC [7] enhances TCC proposal by using a directory-based coherence protocol that supports parallel commits that send addresses but not data. Transactional updates are stored in private caches until commit time. On abort, updates are discarded from private caches, forcing re-executed transactions to fetch again these values. Moreover, because committed data is kept in private caches, it is necessary to write-back this data when a subsequent transaction wants to modify it.

Eager version management was first used by Ananian *et al.*'s UTM proposal [3] to support unbounded transactions.

UTM stores new values in-place and old values, for both loads and stores, in a log. In contrast, LogTM [20] implementation only logs cache-lines targeted by stores and detects conflicts on coherence requests. LogTM-SE [29] extends LogTM by tracking transactional information using signatures that can be easily recovered after an OS intervention. All these HTM systems need to access the log in case of abort, and the log size is at least as large as the write-set of the aborted transaction.

Lupon *et al.* proposed FASTM [17] to minimize abort overhead of LogTM-SE by leaving old values in higher levels of the memory hierarchy and discarding new values that are stored in-place (pinned in L1 caches) in case of abort, behaving similar to a lazy version management scheme. However, FASTM has several differences with respect to our proposal: (1) it modifies the cache coherence protocol with the inclusion of an additional state, (2) an entry in the log must be added for every transactional store, even if the log is not used, (3) after abort recovery, data is not present in L1, making re-executed transactions slower, and (4) in case of transactional overflow of L1, the entire log must be restored.

VIII. Conclusions

We introduce a novel hardware solution, reconfigurable data cache (RDC), for version management in HTM systems. The RDC provides two execution modes: a 64KB general purpose, and a 32KB TM mode capable of managing two versions of the same logical data efficiently. We present the architectural details and operation of the RDC, and we introduce two new HTM systems, one eager and one lazy, that utilize this cache design. We demonstrate that the new HTM systems that we propose solve existing version management problems and achieve, with an acceptable area cost, significant performance and energy delay product improvements over state-of-the-art HTM proposals.

Acknowledgments

We would like to thank all reviewers for their comments and valuable feedback. This work is supported by the agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union under contracts TIN2007-60625 and TIN2008-02055-E, by the European HiPEAC Network of Excellence and by the Velox FP7 project (216852).

References

[1] Predictive technology model. http://ptm.asu.edu/.

[2] The Electric VLSI Design System. http://www.staticfreesoft.com.

[3] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proc. 11th International Symposium on High-Performance Computer Architecture*, Feb. 2005.

[4] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 2006.

[5] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization*, Sep 2008.

[7] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[8] J. Chung, C. Cao Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, Oct. 2006.

[9] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb. 2006.

[10] S. Cosemans et al. A 3.6 pJ/Access 480 MHz, 128 kb On-Chip SRAM With 850 MHz Boost Mode in 90 nm CMOS With Tunable Sense Amplifiers. *IEEE Journal of Solid-State Circuits*, 2009.

[11] O. Ergin and D. Balkan et al. Early Register Deallocation Mechanisms Using Checkpointed Register Files. *IEEE Trans. Computers*, 2006.

[12] 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. In *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[14] R. Kalla and B. Sinharoy et al. Power7: IBM's Next-Generation Server Processor. *IEEE Micro*, 30:7–15, 2010.

[15] R. Kalla et al. Power7: IBM's Next Generation Power Microprocessor. *IEEE Symp. High-Performance Chips (Hot Chips 21)*, 2009.

[16] P. Kongetira and K. Aingaran et al. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25:21–29, 2005.

[17] M. Lupon, G. Magklis, and A. Gonzalez. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2009.

[18] P. S. Magnusson and M. Christensson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35:50–58, 2002.

[19] M. M. K. Martin and D. J. Sorin et al. Multifacets General Execution-driven Multiprocessor Simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33:2005, 2005.

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

[21] A. Negi, M. M. Waliullah, and P. Stenström. LV^*: A low complexity lazy versioning HTM infrastructure. In *ICSAMOS '10*.

[22] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital integrated circuits - A design perspective*. Prentice Hall, 2nd edition, 2004.

[23] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[24] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[25] A. Seyedi, A. Armejach, A. Cristal, O. S. Unsal, I. Hur, and M. Valero. Circuit design of a dual-versioning L1 data cache for optimistic concurrency. In *Proceedings of the 21st Great Lakes Symposium on Very Large Scale Integration*, May 2011.

[26] A. Shriraman and S. Dwarkadas. Refereeing conflicts in hardware transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing*, June 2009.

[27] S. Thoziyoor et al. CACTI 5.1. Technical Report HPL-2008-20, HP Laboratories, Palo Alto, 2008.

[28] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009.

[29] L. Yen, J. Bobba, M. M. 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 *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, Feb. 2007.