ADVERTIMENT. La consulta d'aquesta tesi queda condicionada a l'acceptació de les següents condicions d'ús: La difusió d'aquesta tesi per mitjà del servei TDX (www.tesisenxarxa.net) ha estat autoritzada pels titulars dels drets de propietat intel·lectual únicament per a usos privats emmarcats en activitats d'investigació i docència. No s'autoritza la seva reproducció amb finalitats de lucre ni la seva difusió i posada a disposició des d'un lloc aliè al servei TDX. No s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant al resum de presentació de la tesi com als seus continguts. En la utilització o cita de parts de la tesi és obligat indicar el nom de la persona autora.

ADVERTENCIA. La consulta de esta tesis queda condicionada a la aceptación de las siguientes condiciones de uso: La difusión de esta tesis por medio del servicio TDR (www.tesisenred.net) ha sido autorizada por los titulares de los derechos de propiedad intelectual únicamente para usos privados enmarcados en actividades de investigación y docencia. No se autoriza su reproducción con finalidades de lucro ni su difusión y puesta a disposición desde un sitio ajeno al servicio TDR. No se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al resumen de presentación de la tesis como a sus contenidos. En la utilización o cita de partes de la tesis es obligado indicar el nombre de la persona autora.

WARNING. On having consulted this thesis you're accepting the following use conditions: Spreading this thesis by the TDX (www.tesisenxarxa.net) service has been authorized by the titular of the intellectual property rights only for private uses placed in investigation and teaching activities. Reproduction with lucrative aims is not authorized neither its spreading and availability from a site foreign to the TDX service. Introducing its content in a window or frame foreign to the TDX service is not authorized (framing). This rights affect to the presentation summary of the thesis as well as to its contents. In the using or citation of parts of the thesis it's obliged to indicate the name of the author

Techniques to Improve Concurrency in Hardware Transactional Memory

Adrià Armejach Sanosa

Department of Computer Architecture

Universitat Politècnica de Catalunya

A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

April 2014

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Acta de calificación de tesis doctoral

Curso académico:

Nombre y apellidos
____

Programa de doctorado
____

Unidad estructural responsable del programa
____

Resolución del Tribunal

Reunido el Tribunal designado a tal efecto, el doctorando / la doctoranda expone el tema de la su tesis doctoral titulada ____
____.

Acabada la lectura y después de dar respuesta a las cuestiones formuladas por los miembros titulares del tribunal, éste otorga la calificación:

☐ NO APTO ☐ APROBADO ☐ NOTABLE ☐ SOBRESALIENTE

(Nombre, apellidos y firma) Presidente/a	(Nombre, apellidos y firma) Secretario/a

(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal	(Nombre, apellidos y firma) Vocal

______________________, _______ de __________________ de _______________

El resultado del escrutinio de los votos emitidos por los miembros titulares del tribunal, efectuado por la Escuela de Doctorado, a instancia de la Comisión de Doctorado de la UPC, otorga la MENCIÓN CUM LAUDE:

☐ SÍ ☐ NO

(Nombre, apellidos y firma) Presidente de la Comisión Permanente de la Escuela de Doctorado	(Nombre, apellidos y firma) Secretaria de la Comisión Permanente de la Escuela de Doctorado

Barcelona a _______ de ____________________ de __________

Acknowledgements

Pursuing a PhD is a multi-year endeavour that can turn into a tedious and never ending journey. That was not my case, and I am thankful to a lot of people and a few organisations for their guidance and support, without which I would not have been able to complete, or even start, my PhD studies. While it is not possible to make an exhaustive list of names, I would like to mention a few. Apologies if I forget to mention any name below.

My advisors Adrián Cristal and Osman Unsal gave me the opportunity to attend PhD studies. Their support, confidence, and sound technical advise have played a major role shaping my research ideas into the contributions expressed in this dissertation. I also thank Adrián and Osman for their generosity and comprehension which helped me go through the bumps of this four-year long journey. I would also like to acknowledge Ibrahim Hur for his impressive support during the initial phase of my PhD. His perseverance helped me keep up with the hard work and maintain my focus while enjoying doing research. I also thank Mateo Valero for his dedication and continuous effort in making the Barcelona Supercomputing Center such a great platform for research.

I would like to thank Professor Per Stenström, who kindly invited me to a 3 months internship at Chalmers. I had a great and productive time in Sweden thanks to Per's always positive attitude and enthusiasm. At Chalmers I also benefited from a great working environment and met a number of colleagues that made may stay even more enjoyable. Anurag, Dmitry, Bhavishya, Madhavan, Vinay, Alen and Angelos – thank you all for the fun moments and interesting conversations.

I have had the pleasure to collaborate extensively with Anurag Negi and Rubén Titos. I met Anurag while at Chalmers and I have benefited greatly from his friendship and broad technical knowledge. Rubén has been a constant source of help and I thank him for always pushing to make things better and for the long hours we shared working on the simulator. Both have been excellent research buddies, with whom I enjoyed working together.

I would also like to acknowledge all my friends and colleagues from the office that helped me throughout my PhD; for their insights and expertise in technical matters, and for their unconditional support that has been crucial to keep me sane. Many thanks go to Saša Tomić, Srđan Stipić, Azam Seyedi, Ferad Zyulkyarov, Vasilis Karakostas, Nehir Sonmez, Oriol Arcas, Vesna Smiljković, Gülay Yalçın, Vladimir Gajinov, Nikola Marković, Cristian Perfumo, Chinmay Kulkarni, Daniel Nemirovsky, Ege Akpinar, Vladimir Subotić, Paul Carpenter, and many others. I sincerely thank you all for your help and all the great moments we have had together.

I would like to thank my friends and family for supporting me during this endeavour. My deepest thanks to Marina for her love and for being there for me all the time. This dissertation would have not been possible without her.

My graduate work has been supported by the cooperation agreement between the BSC and Microsoft Research, by the Ministry of Science and Technology of Spain, by the European Union (FEDER funds) under contract TIN2008-02055-E and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

Abstract

Transactional Memory (TM) aims at making shared-memory parallel programming easier by abstracting away the complexity of managing shared data. The programmer defines sections of code, called transactions, which the TM system guarantees that will execute atomically and in isolation from the rest of the system. The programmer is not required to implement such behaviour, as happens in traditional mutual exclusion techniques like locks – that responsibility is delegated to the underlying TM system. In addition, transactions can exploit parallelism that would not be available in mutual exclusion techniques; this is achieved by allowing optimistic execution assuming no other transaction operates concurrently on the same data. If that assumption is true the transaction commits its updates to shared memory by the end of its execution, otherwise, a conflict occurs and the TM system may abort one of the conflicting transactions to guarantee correctness; the aborted transaction would roll-back its local updates and be re-executed. Even though, hardware and software implementations of TM have been studied in detail, large-scale adoption of software-only approaches have been hindered for long due to severe performance limitations.

In this thesis, we focus on identifying and solving hardware transactional memory (HTM) issues in order to improve concurrency and scalability. Two key dimensions determine the HTM design space: conflict detection and speculative version management. The first determines how conflicts are detected between concurrent transactions and how to resolve them. The latter defines where transactional updates are stored and how the system deals with two versions of the same logical data. This thesis proposes a flexible mechanism that allows efficient storage and access to two versions of the same logical data, improving overall system performance and energy efficiency.

Additionally, in this thesis we explore two solutions to reduce system contention – circumstances where transactions abort due to data dependencies – in order to improve concurrency of HTM systems. The first mechanism provides a suitable design to apply prefetching to speed-up transaction executions, lowering the window of time in which such transactions can experience contention. The second is an accurate abort prediction mechanism able to identify, before a transaction's execution, potential conflicts with running transactions.

This mechanism uses past behaviour of transactions and locality in memory references to infer predictions, adapting to variations in workload characteristics. We demonstrate that this mechanism is able to manage contention efficiently in single-application and multi-application scenarios.

Finally, this thesis also analyses initial real-world HTM protocols that recently appeared in market products. These protocols have been designed to be simple and easy to incorporate in existing chip-multiprocessors. However, this simplicity comes at the cost of severe performance degradation due to transient and persistent livelock conditions, potentially preventing forward progress. We show that existing techniques are unable to mitigate this degradation effectively. To deal with this issue we propose a set of techniques that retain the simplicity of the protocol while providing improved performance and forward progress guarantees in a wide variety of transactional workloads.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

During the last decades the number of transistors in a single chip has increased exponentially, from the first home computers that had a few thousands of transistors to today's designs that involve hundreds of millions; with desktop-oriented chips being close to 1 billion transistors, and server-oriented chips surpassing the 2 billion transistors mark. These ever-increasing transistor densities led to substantial performance improvements of sequential processors [64]. However, computer architects ended up hitting the power wall, i.e., undesired levels of power consumption associated to the increase of operation frequency [15]. In order to continue delivering performance improvements, manufacturers have shifted towards designs that integrate several processing units or *cores* on a single chip. Unfortunately, software developers can no longer rely on the next generation of processors to improve performance of their sequential programs, making thread-level parallelism the new challenge to achieve high performance.

The advent of such *multi-core* chips has moved parallel programming from the domain of high performance computing to the mainstream. Now, software developers have the difficult task to write parallel programs to take advantage of multi-core hardware architectures. However, in spite of years of research, writing parallel programs using existing parallel programming methodologies is extremely hard, error prone, and difficult to debug.

1.1 Parallel Programming Problems

Multi-cores usually operate under a shared memory model, allowing parallel tasks of an application to cooperate by concurrently accessing shared resources using a common address space. Each task can be seen as a sequential thread of execution that performs useful computation. Thus, a parallel programming model has to create and manage several tasks that need to synchronise and communicate to each other. However, having concurrent parallel tasks may introduce several new classes of potential software bugs, of which data races (e.g., data dependencies) are the most common [63]. Today's programming models commonly target this problem via lock-based approaches. In this parallel programming technique, locks are used to provide mutual exclusion for shared memory accesses that are used for communication among parallel tasks.

Unfortunately, when using locks, programmers must pick between two undesirable choices:

- Use coarse-grain locks, where large regions of code are indicated as critical regions. This makes the task of adding coarse-grain locks to a program quite straightforward, but introduces unnecessary serialisation that degrades system performance.
- On the other side, fine-grain locking aims at critical sections of minimum size. Smaller critical sections permit greater concurrency, and thus scalability. However, this scheme leads to higher complexity, and it is usually difficult to prove the correctness of the resulting algorithm.

This two choices establish a programming effort versus performance trade-off. The complexity associated with fine-grain locking can lead to incorrect synchronisation, i.e., data races, which could manifest in the form of non-deterministic bugs, producing incorrect results for certain executions of an application. This fact makes lock-based programs difficult to debug, because bugs are hard to reproduce. Synchronisation errors may also result in deadlock or livelock conditions. Using multiple locks requires strict programmer discipline to avoid cyclic dependencies where two or more threads create circular requests to acquire locks, leading to a deadlock scenario where threads are blocked and no forward progress is made. On the other hand, livelocks occur when two or more threads cease to make forward progress while performing the same piece of work repeatedly.

Even correctly parallelised applications may behave poorly due to coherence or unnecessary contention in critical sections. Parallel applications have to modify a certain amount of shared data. Modifying the same data in different cores causes cache-lines to move between private caches, penalising system throughput. Mutual exclusion enforced by locks restricts parallelism even if two critical sections would not access the same shared data, in such cases an opportunity for greater performance is lost due to the restrictive nature of lock based concurrency.

1.2 Transactional Memory

To address the need for a simpler parallel programming model, Transactional Memory (TM) [39, 40] has emerged as a promising paradigm to provide good parallel performance and easy-to-write parallel code.

Unlike in lock-based approaches, with TM programmers do not need to explicitly specify and manage the synchronisation among threads; however, programmers simply mark code segments as transactions that should execute *atomically* and *in isolation* with respect to other code, and the TM system manages the concurrency control for them. It is easier for programmers to reason about the execution of a transactional program since transactions are executed in a logical sequential order according to a serialisable schedule model.

To provide atomicity, the TM system ensures that transactions are executed under all-or-nothing semantics, i.e., either the entire region of code in the critical section is executed, or none of it is executed. Isolation is provided by ensuring that no partial results are visible to the rest of the system, results are made visible only when a transaction completes its execution successfully. To guarantee this properties all TM systems need to perform two important tasks – conflict detection and version management.

Conflict detection performs the task of detecting whether two concurrent transactions conflict with each other. A conflict occurs when two or more transactions access the same data and at least one is a writer. Conflicts may be resolved by aborting one of the transactions and restoring its pre-transactional state in order to maintain atomicity. A transaction that executes without conflicts can commit, releasing isolation by making the transaction's state visible. Conflict detection can be done *eagerly*, by inspecting every memory access; or *lazily*, by deferring the detection until commit time.

Version management handles the way in which the system stores both old (original) and new (transactional) versions of the same logical data, maintaining isolation. Version management can also be implemented either eagerly or lazily. Eager systems put new values in-place and old values are kept in an auxiliary structure, while lazy systems store new values in separate buffers and old values are kept in-place. In either case, old values need to be restored on a transactional abort, and new values need to be made visible to the rest of the system on a transactional commit.

TM systems can be implemented in software (STM) [41, 75], hardware (HTM) [3, 37, 40, 57], or a combination of both hardware and software [28, 49, 56]. Large-scale adoption of STM systems has been hindered for long due to severe performance penalties arising out of the need for extensive instrumentation and book-keeping in order to detect conflicts. Hybrid systems, despite offering hardware support, are likely to be significantly slower than HTMs [17]. This thesis focuses on HTM systems, which can deliver performance comparable to fine-grain locking. However, HTM systems require non-trivial hardware changes and are limited due to hardware space constrains.

1.3 Problem Statement

This thesis addresses several issues present in HTM systems. These can be categorised under three heads: data version management, contention management, and performance in initial real-world HTM implementations.

1.3.1 Issues in Data Version Management

The first issue tackled in this thesis is that traditional version management schemes, eager or lazy, fail to efficiently handle two versions (old and speculative) of the same logical data. This results in a number of inefficiencies, including additional data movement when transactional operations take place, making workloads susceptible to performance degradation. Solutions that allow efficient handling and access to both versions of the same logical data in eager and lazy version management schemes are necessary.

1.3.2 Issues in Contention Management

Workloads that experience contention – circumstances where transactions abort due to data dependencies – usually suffer from noticeable performance degradation. Speeding up transactions can potentially change their contention characteristics, and consequently improve performance. This defines the second issue addressed in this thesis. Transactions that experience contention tend to access the same data repeatedly. This fact opens an opportunity to study potential benefits to be had when applying a prefetching technique for TM. By prefetching data that may experience locality of reference transactional execution times can be improved.

In the presence of data conflicts transactions may abort, i.e., the results of speculative execution are discarded. This leads to wasted work, expensive rollbacks of application state, and inefficient utilisation of computational resources. While conflicts due to concurrent accesses to shared data cannot be completely eliminated, mechanisms to avoid starting a transaction when it is likely to fail are necessary for maximising computational throughput. The third issue addressed in this thesis targets the problem of blindly allowing transactions to start execution in the presence of contention, which is clearly suboptimal.

1.3.3 Issues in Initial Real-world HTM Implementations

The fourth issue is related to initial implementations of HTM systems that are starting to be widely available. Such systems employ simple policies that are easy to incorporate in existing multi-core chips. However, this simplicity comes at the cost of no inherent forward progress guarantees and susceptibility to certain performance pathologies. The likelihood of pathological behaviours and their impact on performance remains unclear. Efficient techniques to provide forward progress guarantees and to ameliorate performance pathologies, while still retaining implementation simplicity, are needed to make these systems appealing.

1.4 Thesis Contributions

In order to address the issues described in the previous section, this thesis makes the following contributions:

- **A reconfigurable data cache to improve version management.** We introduce a reconfigurable L1 data cache architecture that is able to manage efficiently two versions of the same logical data. The *Reconfigurable Data Cache (RDC)* has two execution modes: a 64KB general purpose mode and a 32KB TM mode. 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. We explain how these operations solve existing version management problems in both eager and lazy version management schemes. Our experiments show performance as well as energy-delay improvements compared to state-of-the-art baseline HTM systems; with a modest area impact.

- **Speeding up transactions through prefetching.** We investigate potential gains to be had when lines in the write-set – the set of speculatively updated cache lines – of a transaction are prefetched when it begins execution. These lines are highly likely to be referenced again when an aborted transaction re-executes. We also demonstrate that high contention typically implies high locality of reference. Prefetching cache lines with high locality can, therefore, improve overall concurrency by speeding up transactions and, thereby, narrow the window of time in which such transactions persist and can cause contention. We propose a simple design to identify and request prefetch candidates; and show performance gains in applications with high contention.

- **Transaction abort prediction.** We introduce a hardware mechanism to avoid speculation when it is likely to fail, using past behaviour of transactions and locality in conflicting memory references to accurately predict conflicts. The prediction mechanism adapts to variations in workload characteristics and enables better utilisation of computational resources. We demonstrate that HTMs that integrate this mechanism exhibit reductions in both wasted execution time and serialisation overheads when compared to prior work.

- **Techniques to improve initial real-world HTM implementations.** We show that protocols that merely guarantee livelock freedom may not be the most efficient. We investigate in depth the performance implications of a number of existing livelock mitigation and avoidance techniques that must be used in available HTM implementations in order to guarantee forward progress. Our study shows that these techniques impose a significant performance cost. To minimise this cost we introduce a number of novel techniques, in hardware and software, that retain the simplicity of current HTM designs while effectively ameliorating performance costs of existing techniques.

1.5 Thesis Organisation

Chapter 2 discusses additional background on transactional memory with emphasis on HTM systems design dimensions. Chapter 3 introduces our work on the reconfigurable data cache, including a design description, implementation details of the resulting HTM systems and evaluation. Chapter 4 presents a mechanism that makes prefetching effective for transactions. Chapter 5 contains the description of a hardware abort prediction mechanism that preempts transaction executions that are likely to fail. We explain how the prediction mechanism is able to make informed decisions, and provide an extensive evaluation using single-application and multi-application workloads. Chapter 6 highlights potential performance issues present in initial real-world HTM implementations, and describes a set of simple techniques that aim to enhance performance of such systems while retaining implementation simplicity. Chapter 7 concludes this dissertation.

2 Background on Hardware Transactional Memory

Hardware Transactional Memory (HTM) [40] offers performance comparable to fine-grain locks while, simultaneously, enhancing programmer productivity by largely eliminating the burden of managing access to shared data. Recent usability studies support this claim [18, 71], suggesting that TM can be an important tool for building parallel applications. With TM, programmers simply demarcate sections of code – called transactions – where synchronisation occurs, as shown in Figure 2.1, and the TM system guarantees correct execution by providing the following properties: atomicity, isolation, and serialisability.

Atomicity means that either all or none the instructions inside a transaction appear to be executed. Having *isolation* means that none of the intermediate state of a transaction is visible outside of the transaction – i.e., memory updates are not visible to other threads during the execution of a transaction. Finally, *serialisability* requires the execution order of concurrent transactions to be equivalent to some sequential execution order of the same transactions [38].

```
atomic {
    if ( foo != NULL ) a.bar();
    b++;
}
```

Figure 2.1: Group of instructions representing a *transaction*.

TM systems achieve good performance by allowing transactions to execute without acquiring locks, assuming that no other transaction is concurrently accessing the same data. Throughout a transaction's execution the memory addresses that are read are added to a *read-set*, and the ones that are written are added to a *write-set*. Transactions execute speculatively, i.e., a transaction execution may fail if the TM system detects data conflicts with other concurrent transactions. This is achieved by comparing the read and write sets of concurrent transactions, which allows to perform fine-grain read-write and write-write conflict detection. If a conflict is found, one of the conflicting transactions has to be aborted, the execution state is then *rolled back* to the point where the transaction started, and the transaction is *retried*. Otherwise, if no conflicts where found, the transaction *commits* successfully.

Using large transactions simplifies parallel programming because it provides ease-of-use and good performance. First, like coarse-grain locks, it is relatively easy to reason about the correctness of transactions. Second, to achieve a performance comparable to that of fine-grain locks, the programmer does not have to do any extra work because the TM system will handle that task automatically. There are three key design dimensions that determine how the properties of atomicity and isolation are implemented in a HTM system: the version management scheme, the conflict detection policy, and the way conflicts are resolved.

2.1 Version Management

Transactional systems must be able, at least, to deal with two versions of the same logical data. A new (transactional) version and an old (pre-transactional) version. The way in which these versions are stored in the system determines the version management scheme. The old version is used in case a transaction fails to commit, to perform a roll back to restore pre-transactional state. Updates to memory can be handled either eagerly or lazily.

In *lazy version management*, updates to memory are done at commit time [19, 36]. New values are saved in a per-transaction store buffer, while old values remain in place. This guarantees *isolation* because the speculative updates are not visible by other threads until the transaction commits, at which point the updates are made visible. In contrast, *eager version management* applies memory changes immediately and the old values are stored in a software undo log [3, 12, 57, 90]. If the transaction aborts, the undo log is used to restore memory state. Note that in order to grant isolation in eager TM systems, transactionally modified variables must be locked, and therefore cannot be accessed until the owner either commits or aborts the transaction. This can derive into classic deadlock situations, thus eager systems require contention management mechanisms that, when detecting a potential deadlock cycle break it by choosing a victim to abort and roll-back.

Each version management scheme has its own advantages and disadvantages. Eager versioning systems have higher overhead on transaction abort because they have to restore the memory changes from a software undo log. In contrast, lazy versioning aborts have a smaller overhead since no speculative updates were visible. However, a lazy scheme has a higher performance penalty at commit time, at which point all transactional updates have to become visible.

2.2 Conflict Detection

Conflict detection can be performed either taking a lazy (optimistic) [19, 36] or an eager (pessimistic) [3, 12, 57, 90] approach. Systems with eager conflict detection check possible data dependency violations as soon as possible, checking for conflicts on every memory access during transaction execution. In contrast, lazy conflict detection assumes that a transaction is going to commit successfully and waits until the transaction finishes its execution to detect possible conflicts. Figure 2.2 illustrates how both approaches work – example inspired from [80].

Eager conflict detection attempts to minimise the amount of wasted work in the system by detecting and resolving conflicts as soon as possible, however, such attempts to reduce wasted work are not always successful. This happens due to a limitation in eager systems; it addresses potential conflicts caused by an offending access to a shared location, at this point the system has to decide which transaction will apply the conflict resolution policy,

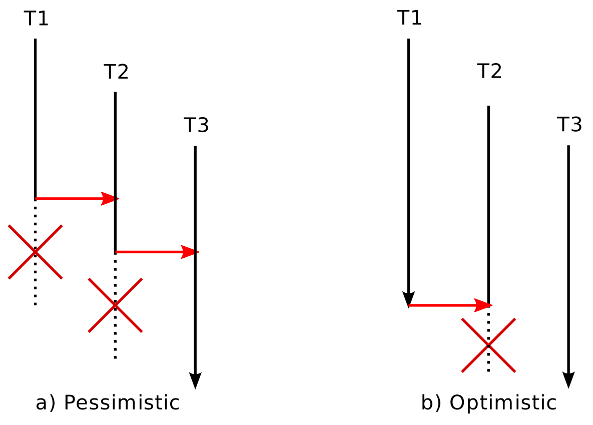


Figure 2.2: Pessimistic and optimistic conflict detection.

but it does not have all the necessary information to make the optimal decision and the prediction is sometimes wrong [14], as can be seen in Figure 2.2a. On the other hand, lazy conflict detection deals with conflicts that are unavoidable in order to allow a transaction to commit; and as a consequence, it is more robust under high contention [77]. Though lazy conflict detection systems guarantee forward progress – because a transaction only aborts to allow another transaction to commit – individual threads waste substantial computational resources due to aggressive speculation.

Eager conflict detection systems are easier to integrate in existing multi-cores because they piggyback on the already existing cache coherence protocol to perform the task of conflict detection [21, 24]. Basic extensions are sufficient to implement a simple eager conflict detection scheme. For this reason, initial widely available real-world HTM implementations are using this approach [43]. However, simplicity comes at the cost of no forward progress guarantees and susceptibility to severe performance penalties. On the other hand, lazy schemes need to detect conflicts at commit time, requiring an additional specific mechanism to compare the write set of the committing transaction against concurrently running transaction's read and write sets to detect conflicts.

2.3 Synergistic Combinations

We introduced two ways to deal with data version management and two ways to perform conflict detection. Intuitively, eager version management, where memory updates are done while the transaction is executed, is commonly used with eager conflict detection to ensure that only one transaction has exclusive access to write a new version of a given address. In contrast, lazy version management is usually combined with lazy conflict detection, doing both tasks (conflict detection and memory updates) at commit time.

However, these are not the only two options. Some of the first TM proposals used lazy version management with eager conflict detection [3, 69]. In addition, other proposals split the monolithic task of conflict detection and adopt an approach that detects conflicts while the transaction is still active (i.e., at every memory access), but resolves them when the transaction is ready to commit [62, 85]. The second generation of HTMs focused on flexible mechanisms such as detecting write-write conflicts eagerly and read-write conflicts lazily [77], detecting and resolving conflicts eagerly or lazily depending on the application [60], or providing protocols that can handle simultaneous execution of eager and lazy transactions [52].

3 Efficient Version Management: A Reconfigurable L1 Data Cache

3.1 Introduction

Three key design dimensions impact system performance of hardware transactional memory (HTM) systems: conflict detection, conflict resolution and version management [14]. 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. In this chapter 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 [23], making the commit process much more critical than the abort process. However, newer studies that present larger and more representative workloads [18] 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 3.2 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 3.3 we discuss how the inclusion of the RDC affects HTM systems and how it improves both eager and lazy versioning schemes, and in Section 3.4 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 make visible 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 3.5 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 3.6 we evaluate the performance and energy effects of our proposed HTM systems that use the RDC. We find that, for the STAMP benchmark suite [18], Eager-RDC-HTM and Lazy-RDC-HTM achieve average performance speedups of 1.36× and 1.18×, respectively, over 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.

3.2 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.* [74] 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.

3.2.1 Basic Cell Structure and Operations

Similar to prior work [74], in RDC two bit-cells are used per data bit, instead of one as in traditional caches. Figure 3.1 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 [67], 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 bits), 512 e-cells are placed side by side and are connected to the same word lines (WL).

In Table 3.1 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,

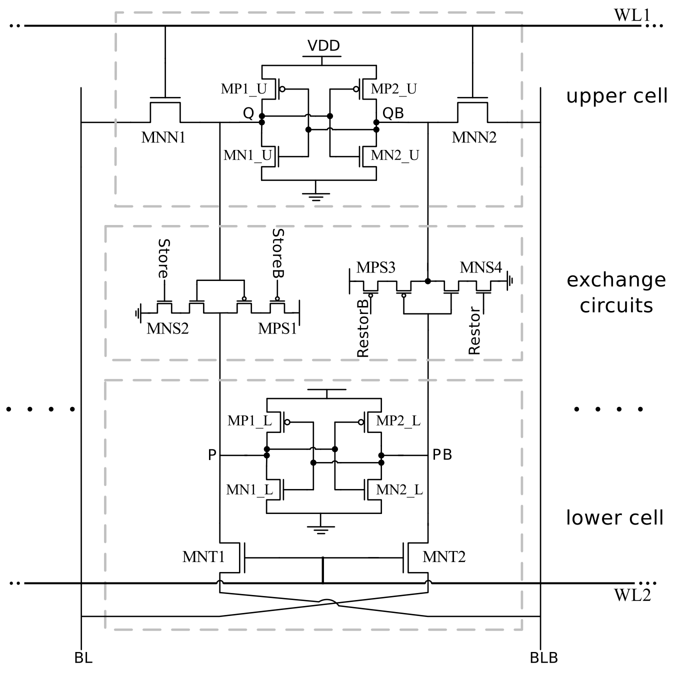


Figure 3.1: Schematic circuit of the e-cell. A typical cell design is extended with an additional cell and exchange circuits.

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

Table 3.1: Brief descriptions of the RDC operations.

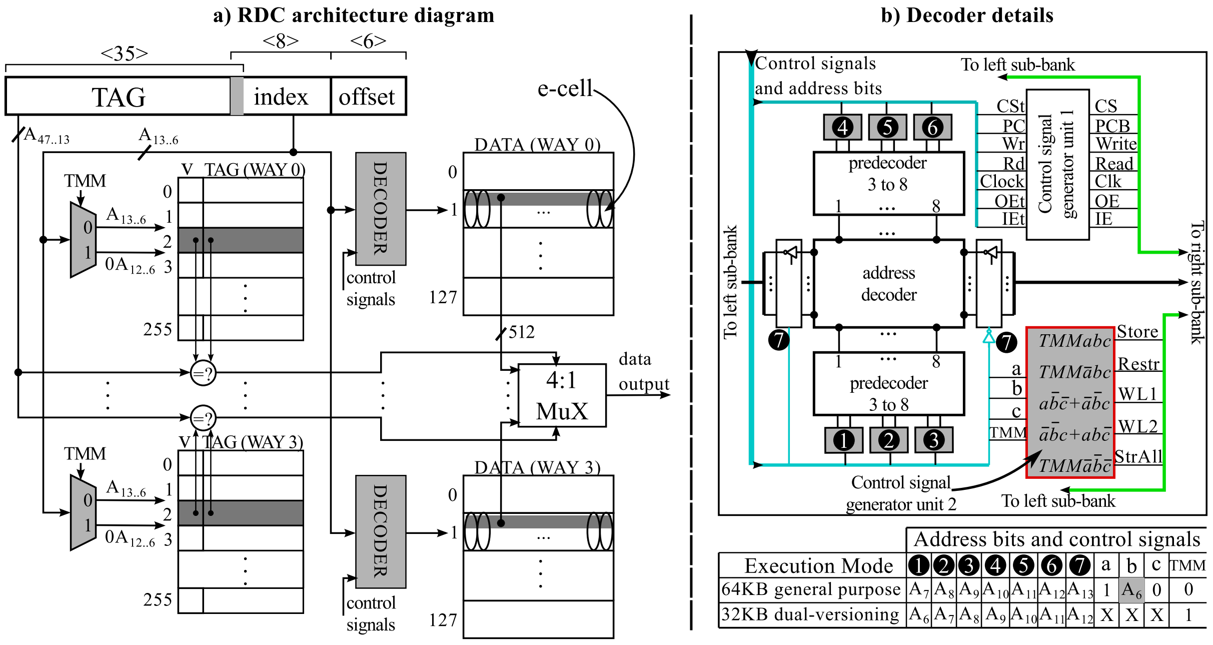
which acts as an inverter to invert Q to P; the lower cell keeps the value of P when `Store` is inactive, and it inverts P to PB, so that PB has the same value as Q. Similarly, to restore data from a lower cell to its corresponding upper cell, we activate `Restore`. Finally, `ULWrite` is used to write 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 show in our evaluation, because most of the components of the cache are not involved in this operation.

3.2.2 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 3.2 shows an architectural diagram of RDC, the decoder details and its associated signals, which change depending on the execution mode. The diagram considers 48-bit addresses and a 4-way cache organisation 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 Table 3.1, to perform typical read and write operations as in any general purpose cache. Figure 3.2a 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 3.2b it can be seen how the seven most significant bits of the index are



	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

Figure 3.2: (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.

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 Table 3.1. 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 3.2a, 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 3.2b), 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.

3.3 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.

3.3.1 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 3.3 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 3.3a). We

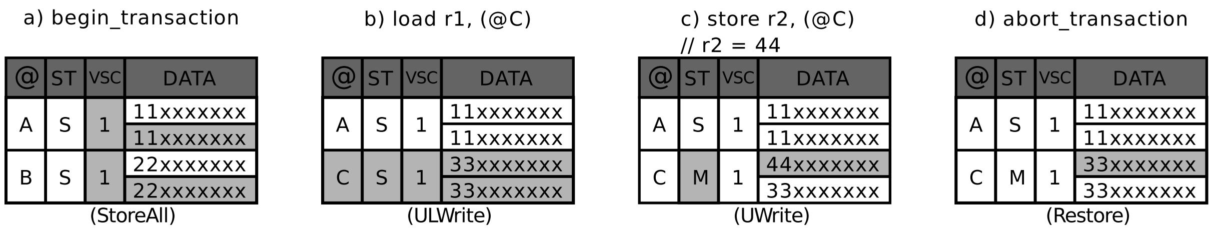


Figure 3.3: A simple protocol operation example, assuming a 2-entry RDC. Shaded areas indicate state changes. (a) Creation of the shadow-copies, in the lower cells, at the beginning of a transaction — (b) A load operation that modifies both the upper and the lower cells in parallel (ULWrite) — (c) A 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.

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 3.3b).

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 [57, 90]. 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 3.3a) or during its execution (Figure 3.3b) 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 3.3c, 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 3.6.2.

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 [19, 85], 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, 19, 85], repeated transactions that modify the same blocks are not required to write-back, resulting in significant performance gains, see Section 3.6.3, 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.

3.3.2 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.

3.3.3 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 3.3d, 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 minimised, 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

Component	Description
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

Table 3.2: Base eager systems configuration parameters.

proposals [19, 85] often use multiple levels of the memory hierarchy to track transactional state, making the re-fetch cost more significant.

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 3.3d. 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 [14], and (3) it alleviates pressure in the memory hierarchy.

3.4 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.

3.4.1 Eager-RDC-HTM

Eager-RDC-HTM extends LogTM-SE [90], 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.

Table 3.2 summarises 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 logging mechanism that stores old 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.

Component	Description
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

Table 3.3: Base lazy systems configuration parameters.

3.4.2 Lazy-RDC-HTM

Lazy-RDC-HTM is based on a Scalable-TCC-like HTM [19], 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. Table 3.3 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 `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 [19, 85] to minimise the overhead of virtualisation techniques [22, 69]. Although our proposal is compatible with virtualisation 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.

3.5 Reconfigurable Data Cache Analysis

We use CACTI 5 [81] 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 Table 3.2). We construct, for one way of the RDC and one way of a typical 64KB SRAM, Hspice transistor level net-lists that include all the components, such as the complete decoder, control signal units, drivers, and data cells. We simulate and optimise both structures with Hspice 2003.03 using HP 45nm Predictive Technology Model [2] 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. Table 3.4 shows the energy costs for typical SRAM and RDC operations.

In Figure 3.4 we show the layouts [1] 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 Table 3.5 we show the expected area impact of our RDC design on two commercial chips: IBM Power7 [46, 47], which uses the same technology node as our baseline systems and has large out-of-order cores, and Sun Niagara [48, 72], 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

Table 3.4: Typical SRAM and RDC energy consumption per operation.

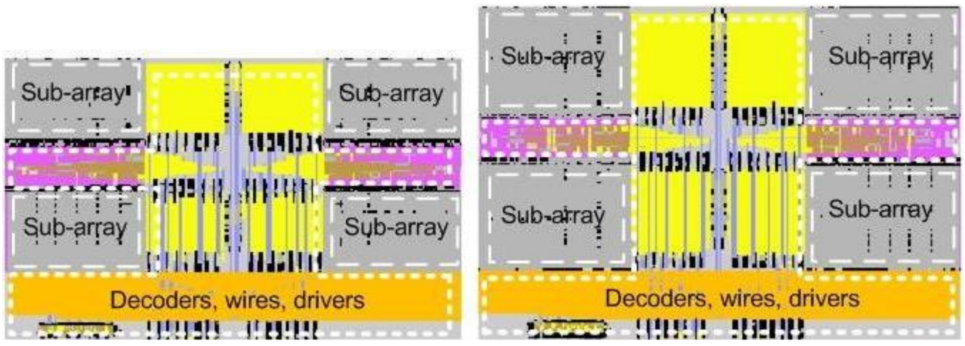


Figure 3.4: Typical 64KB SRAM (left) and RDC (right) layouts. Showing one sub-bank, address decoders, wires, drivers, and control signals. The second symmetric sub-banks are omitted for clarity.

	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%

Table 3.5: Expected area impact of our RDC design on two commercial chips: the RDC increases die size by less than 0.3%.

3.6 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 [18]. 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.

3.6.1 Simulation Environment

For Eager-RDC-HTM and LogTM-SE we use a full-system execution-driven simulator, GEMS, in conjunction with Simics [53, 55]. The former models the processor pipeline and memory system, while the latter provides functional correctness in a SPARC ISA environment. For the evaluation of Lazy-RDC-HTM and Scalable-TCC we use M5 [6], 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.

3.6.2 Performance Results for Eager-RDC-HTM

Figure 3.5 shows the execution time breakdown for LogTM-SE with 64KB L1D, which serves as our baseline, for Eager-RDC-HTM, and for an Idealised eager versioning HTM. The Idealised 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 normalised 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

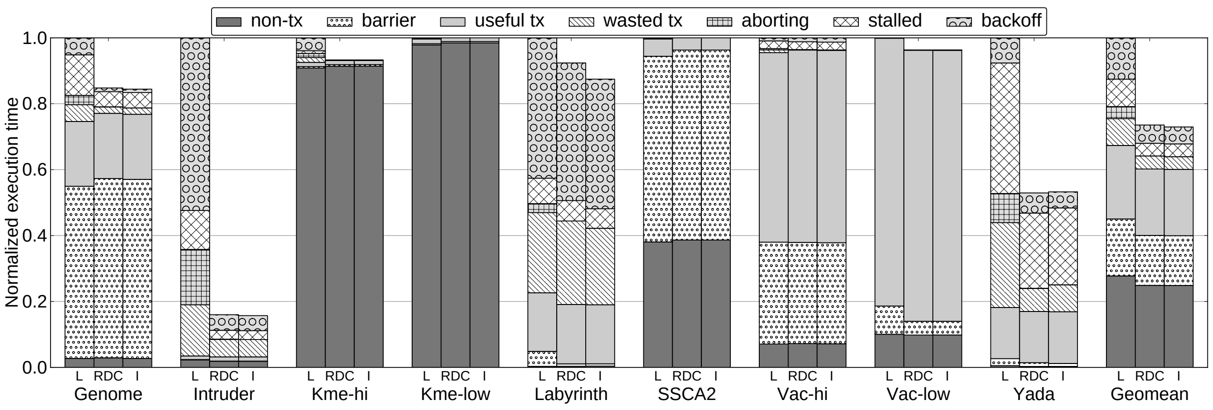


Figure 3.5: Normalised execution time breakdown for 16 threads, in eager systems.
L – LogTM-SE; **RDC** – Eager-RDC-HTM; **I** – Idealised eager HTM

		LogTM-SE			Eager-RDC-HTM				
Benchmark	Commits	%AB	%AB Conf.	Unrolled Entries	%AB Conf.	%AB Entries	Unrolled	%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

Table 3.6: 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.

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 3.5, 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 Idealised 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 Table 3.6 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. Table 3.6 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.

Table 3.6 also 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 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 Idealised eager versioning HTM, because the two-folded abort process is a hybrid solution that minimises the use of the log.

In Figure 3.6, 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,

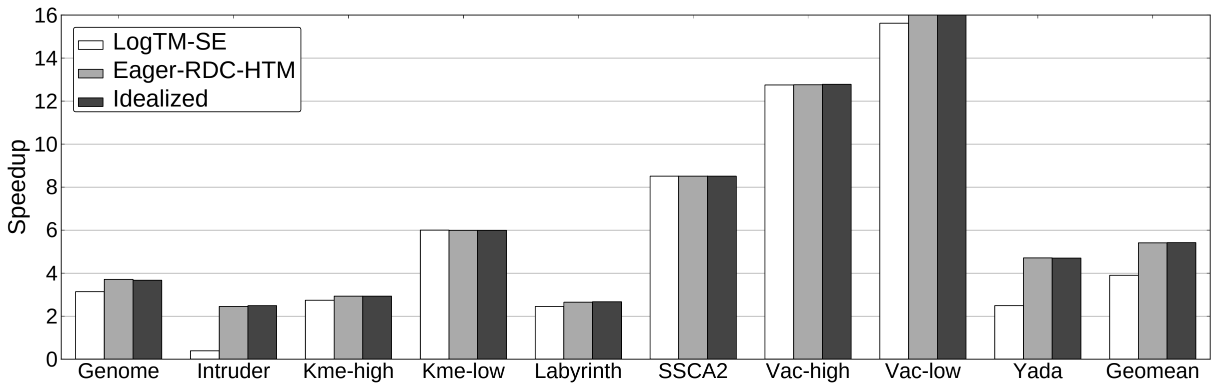


Figure 3.6: Speedup of 16-threaded applications compared to single-threaded LogTM-SE.

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.

3.6.3 Performance Results for Lazy-RDC-HTM

Figure 3.7 shows the execution time breakdown for the lazy HTM systems that we evaluate, namely Scalable-TCC (with 64KB L1D), Lazy-RDC-HTM, and Idealised lazy HTM. The Idealised 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 3.7 are normalised to Scalable-TCC 32-threaded execution, and they are split into seven parts, namely *Barrier*, *Commit*, *Useful*, *StallCache*, *Wasted*, *Wasted-Cache*, 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 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

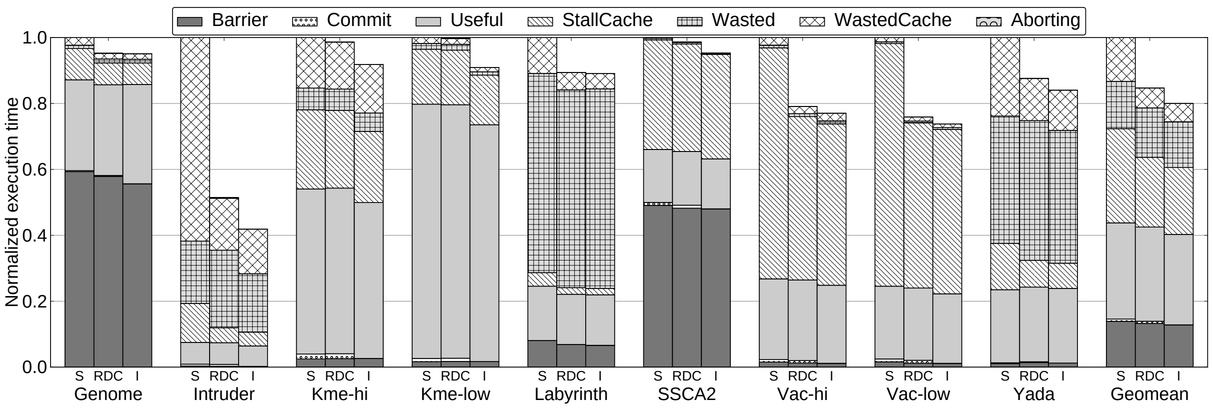


Figure 3.7: Normalised execution time breakdown for 32 threads, in lazy systems.
S – Scalable-TCC; **RDC** – Lazy-RDC-HTM; **I** – Idealised lazy HTM

	Scalable-TCC				**Lazy-RDC-HTM**					
Benchmark	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

Table 3.7: 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.

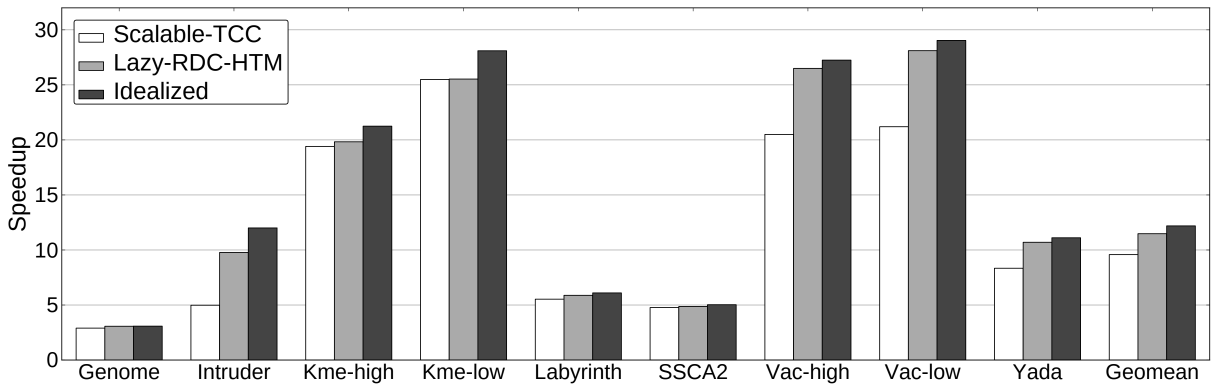


Figure 3.8: Speedup of 32-threaded applications compared to single-threaded Scalable-TCC.

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 Table 3.7 (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 3.8 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 3.9, 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.

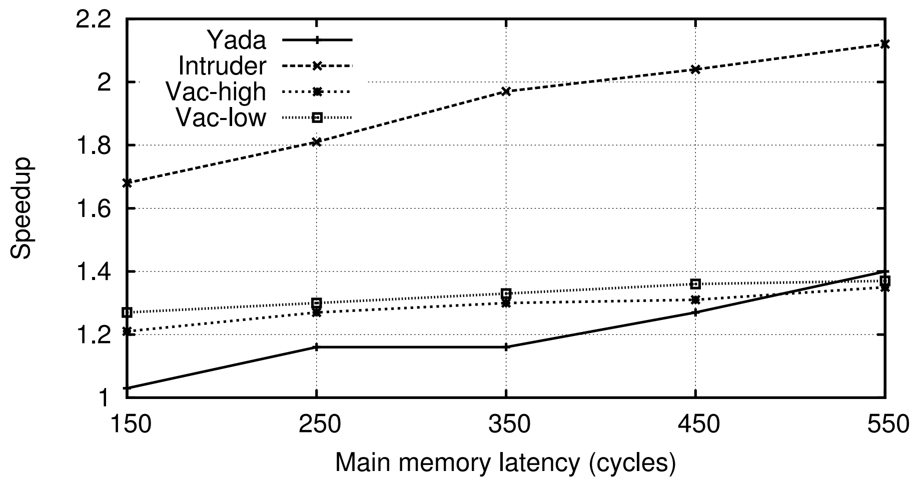


Figure 3.9: Speedup variation with respect to main memory latency in lazy systems.

Also, for latencies higher than 350 cycles, all applications obtain better results.

3.6.4 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, Table 3.4. 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 Table 3.5, 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 [26], and we present, in Table 3.8, 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.

	Eager HTM Systems					Lazy HTM Systems				
	L1D Power (mW)			Power	EDP	L1D Power (mW)			Power	EDP
Benchmark	LogTM-SE	RDC HTM	Speedup (×)	Inc. (×)	Ratio (×)	STCC	RDC HTM	Speedup (×)	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

Table 3.8: Power consumption comparison of L1D caches and Energy Delay Product (EDP) comparison of entire systems.

3.7 Related Work

Ergin *et al.* [33] 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.* [74] 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 [40], 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) [37] 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 [19] 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 [57] implementation only logs cache-lines targeted by stores and detects conflicts on coherence requests. LogTM-SE [90] 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 [51] to minimise 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.

3.8 Summary

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 utilise 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.

4 Transactional Prefetching: Narrowing the Window of Contention

4.1 Introduction

The ever-widening disparity between the speed at which a processor core can process data and the speed at which the memory hierarchy can supply it has led to a myriad of techniques that aim at overlapping data access latency with some form of useful work. Prefetching is one such technique where, by predicting memory references likely to occur in the near future, data is fetched into structures close to the core before it is needed. Various prediction techniques have been employed, targeting frequently encountered patterns in memory references. However, Hardware Transactional Memory (HTM) [39] presents a scenario where a new form of prefetching may be invoked that complements and, in certain scenarios, allows more effective latency hiding than standard techniques.

Several implementations of HTM use first-level caches to isolate speculative state, preserving a consistent state by pushing clean (old) cache lines to second-level caches and beyond [19, 36, 60, 62, 85]. Transactions execute speculatively and any data races detected by the HTM system are typically resolved by forcing one or more of the conflicting transactions to abort. When a transaction aborts speculative state must be discarded and the transaction must be re-executed. To do so, all speculatively modified lines in the first-level cache are invalidated. Subsequent references to such lines during re-execution will miss in the first-level cache and retrieve a clean version of the line from deeper levels of the memory hierarchy. Thus, data transfer latencies delay transactional execution. In scenarios with moderate to high contention this can result in extended transaction execution times, application slow-down and a higher probability of contention. We observe that while a technique like runahead execution [32, 58] could be advantageous here, the hardware requirements for runahead execution and transactional execution are similar (support for checkpointing and dependency tracking) and thus would need to be duplicated in hardware.

In this chapter we investigate potential gains to be had when lines in the write-set – *the set of speculatively updated cache lines* – of a transaction are prefetched when it begins execution. These lines are highly likely to be referenced again when an aborted transaction re-executes. Interestingly, high contention typically implies high locality of reference. Moreover, in Section 4.2 we show that this locality of reference is not limited to re-executions of a particular transaction invocation and persists even when a new invocation of the transaction occurs. These observations have motivated the design of hardware prefetching mechanisms described in this study. These mechanisms are able to track important write-set lines and are brought into play upon aborts and new transaction starts to prefetch lines that would be required by the transaction during its execution.

The benefits from prefetching write-set lines are expected to be most noticeable in lazy versioning systems like TCC [19, 36]. This is so because, unlike eager versioning designs, they do not restore clean values when speculation fails, and rely upon deeper levels of the memory hierarchy to provide consistent data. However, eager versioning designs like LogTM [90] will benefit from prefetches that are initiated when a new instance of a transaction first begins. In this case a part of the write-set may not be present in the cache when the transaction starts execution, particularly when the contention is high. This effect not only improves execution times but also narrows the window of contention improving concurrency overall.

The rest of this chapter is organised as follows. Section 4.2 provides strong evidence of the locality of reference that exists between multiple invocations of a given transaction for a variety of transactional workloads. This also motivates the hardware structures which are described in detail in Section 4.3. Section 4.3 also describes the operation of the prefetch mechanism. Section 4.4 presents potential performance gains that can be achieved when such prefetching is enabled. We evaluate several transactional prefetching configurations based on the design presented in Section 4.3, including an idealised variant. Section 4.5 puts our contributions in perspective of prior work done in prefetching and HTM.

4.2 Motivation

To make a case for prefetching in transactions we have investigated the behaviour of several workloads in the STAMP benchmark suite [18]. The goal of this analysis was to quantify the locality of reference that exists in write-sets across different invocations of the same atomic block or transaction. We recorded all stores issued by each transaction from one thread of each application, tracking the number of transaction invocations that reference each distinct cache line address. We then ranked accessed locations on the basis of frequency of such references for all invocations of each transaction. We choose to concentrate on write-sets for two reasons – first, such lines are likely to get invalidated due to coherence actions or aborts and, second, the read-modify-write nature of common transactions results in a significant overlap with the read-set. The non-overlapping part of the read-set typically sees less contention and is, therefore, likely to be found in the private cache hierarchy.

Figure 4.1 presents several plots (one for each workload included in the study) that show the number of distinct addresses that can cover a certain fraction of the total number of memory references generated by all invocations of a certain transaction over the duration of the application. For each plot the x-axis is in logarithmic scale and shows the number of distinct addresses, N. The y-axis plots cumulative reference count, C, (for the N most frequently referenced addresses) normalised to the total number of references issued. In other words, if we can track and prefetch a certain number, N, of the most frequently referenced addresses then we can potentially satisfy a fraction, C, of stores in transactions. Moreover, it can be inferred from the read-modify-write behaviour of common transactions that these prefetches would also satisfy a significant portion of loads issued by the transaction.

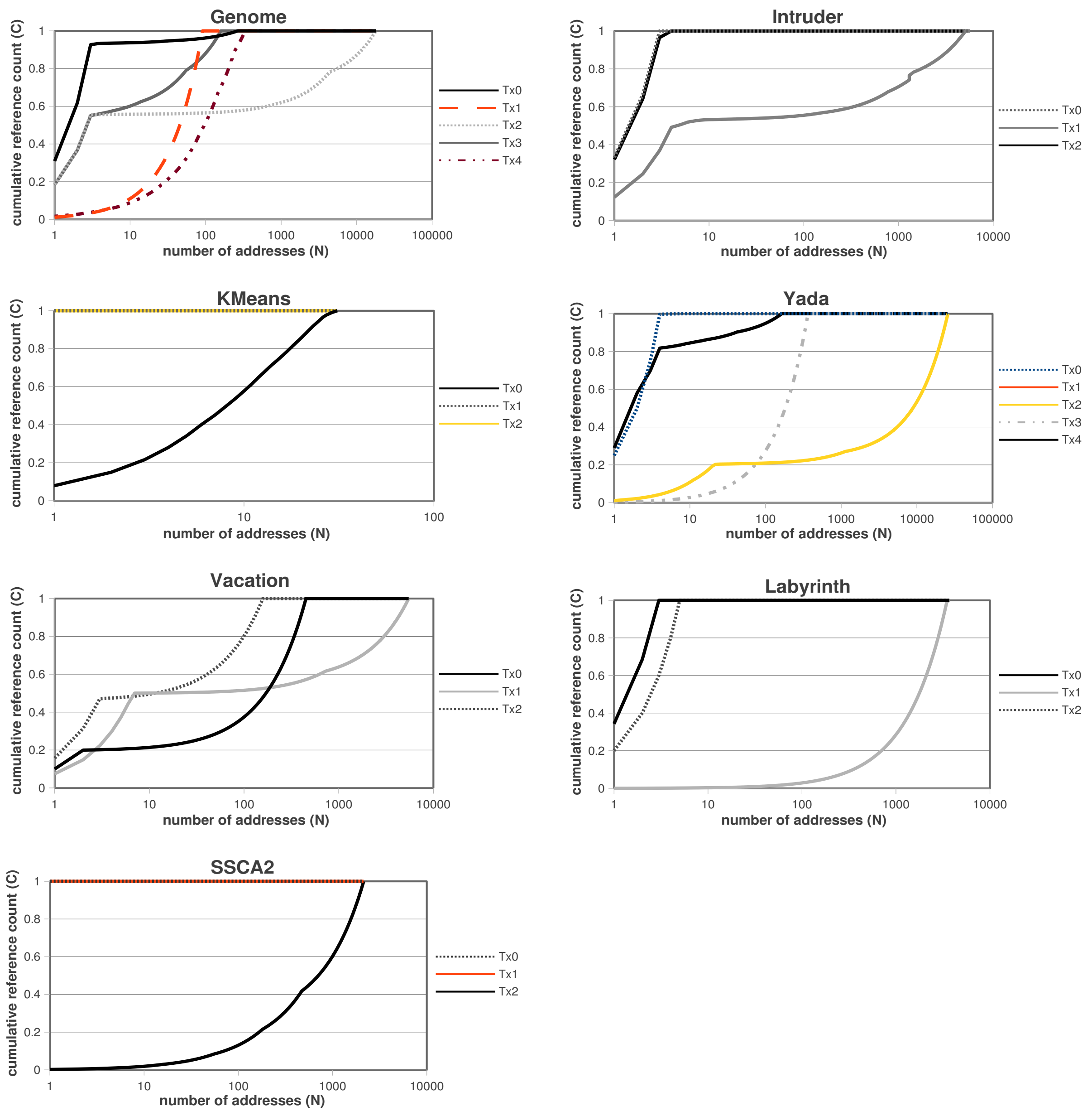


Figure 4.1: Locality of reference across transaction invocations.

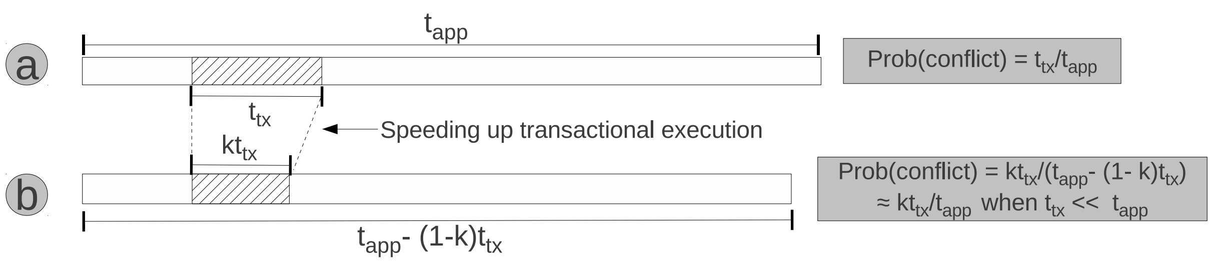

a
t_{app}
Prob(conflict) = t_{tx}/t_{app}
t_{tx}
Speeding up transactional execution
kt_{tx}
b
Prob(conflict) = $kt_{tx}/(t_{app} - (1-k)t_{tx})$
$\approx kt_{tx}/t_{app}$ when $t_{tx} << t_{app}$
$t_{app} - (1-k)t_{tx}$

Figure 4.2: Narrowing the window of contention: effect on conflict probabilities.

Some transactions have almost no locality of reference, like Tx2 from kernel 1 in SSCA2, a workload with little contention. The linear rise (note that the x-axis is logarithmic) in cumulative reference count is indicative of this fact. A similar case occurs in Labyrinth, where concurrency is limited but the nature of work results in the different invocations of the same transaction updating very different locations. However, for applications like Intruder, Genome, KMeans and Yada one notices saturation or very low growth in cumulative reference count beyond 16 or 20 addresses, indicating strong locality of reference.

The optimistic nature of TM usually provides good performance when workloads have little contention. However, when contention is high overheads of managing and restoring speculative state grow and increase application execution times. Therefore, to improve HTM design one must aim at minimising overheads when running applications with moderate to high contention. Besides direct improvements in transaction execution times, prefetching data can potentially improve overall concurrency by narrowing the window of contention for transactions. Figure 4.2 shows a very simplified view of how this might occur. We view a conflicting access as an event that can occur with equal likelihood at any point during the lifetime of a thread which might intermittently execute transactions. In such a case the probability of contention for the transaction can be represented by the time the thread spends executing the transaction expressed as a fraction of its lifetime. It can be seen that shortening the duration of a transaction reduces the probability of encountering a conflict. Moreover (not shown in the figure) this reduced probability also results in fewer aborts and consequent re-executions (which degrade overall contention even further). This results in fewer conflicting accesses being generated in the system.

Prior work [59] has found that containing transactional stores in dedicated hardware buffers can mitigate overheads associated in reading back lines invalidated on an abort. However, our prefetching technique provides improvements in performance for fresh transaction invocations as well.

As single-thread performance growth stagnates, running applications with inherently limited parallelism in a multithreaded fashion will be a natural recourse to extract maximum benefit from core count scaling. For such applications in our study (Genome, KMeans, Yada, Intruder) we see that significant locality of reference exists. If we track the 16 most frequently accessed addresses for each transaction we can typically cover more than 60% of the references issued.

4.3 Design

We subdivide the design into three components – the first which infers locality, the second which manages prefetches and the third which trims prefetch lists. The subsections below describe the structure and behaviour of each component. These components are instantiated for each core in a chip-multiprocessor.

4.3.1 Inferring Locality

To decide which cache line addresses are most suitable for prefetching one must first get a measure of the associated locality. The key problem that arises when one attempts to track locality traits of arbitrary memory locations is that of maintaining a history of memory references until there is enough to infer useful behavioural characteristics. While the history is being recorded there might not be any notion of relative importance of different addresses, resulting in seemingly very large storage requirements or extremely long delays in making inferences. A trade-off must be made that keeps the design simple yet responsive. We choose to do so by employing one or two bloom filters [11] to track memory access history for one or more invocations in the past. Performance evaluations presented later will show performance differences between designs with one and two bloom filters. The single filter design uses a two-step iterative refinement mechanism to learn high-locality cache line addresses one transaction at a time. The two-filter design (described later in Section 4.4.3) uses a 3-step mechanism using the bloom filters in a ping pong fashion.

When employing a single-bloom filter, two invocations of a transaction are required to learn prefetch candidates. Figure 4.3 shows the key elements of the proposed mechanism. During the first invocation, cache line addresses in the write set are added to the bloom

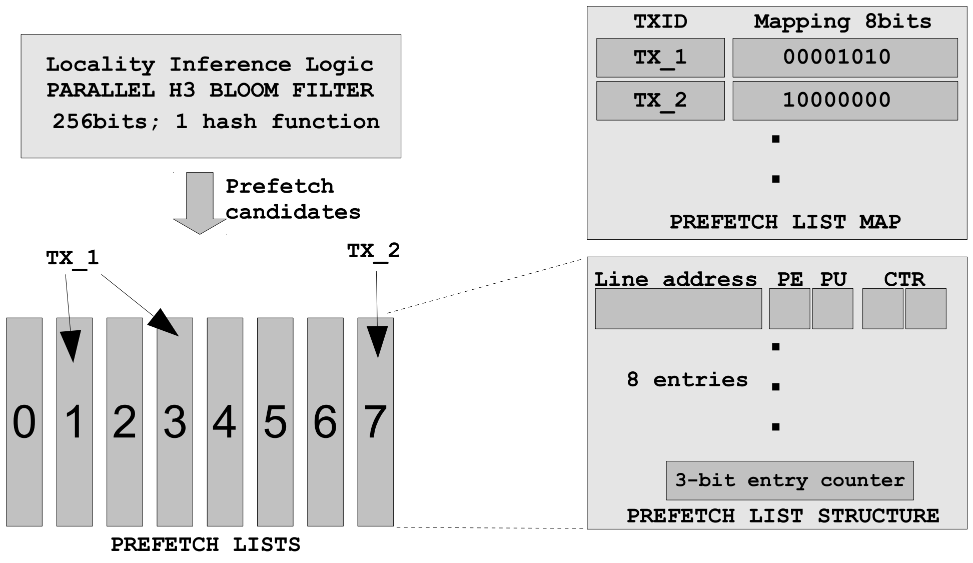


Figure 4.3: Transactional Prefetching: Key components.

filter. During the second invocation, cache line addresses targeted by stores are checked for presence in the bloom filter. If a positive match is found the address is added to one of the prefetch candidate lists. Either free lists are used or the lists allocated to the least recently invoked transaction are freed, as explained in Section 4.3.2.

Locality inference is not initiated for transactions as long as they have prefetch resources allocated to them. Training is aborted if two invocations of another transaction are seen and a watchdog timer has been triggered. This prevents seldom executed transactions from permanently blocking access to locality inference structures. We employ parallel bloom filters employing high-quality H3 hash functions, which have been found, in prior work [72], to be suitable for hardware implementation. The evaluation includes results for both real and perfect bloom filters. Note that the performance (false-positive rate) of these filters is not as critical to performance as when using such filters for conflict detection.

Training in Parallel

It is conceivable that the bloom filters can be used to train on more than one transaction simultaneously. This would involve inserting *(address, transaction id)* tuples instead of just addresses into the bloom filters. Since it is non-trivial to selectively delete entries from a bloom filter, such a design must consider the cost of false positives, transaction invocation frequencies and overall responsiveness. We do not study the concept further.

4.3.2 Managing Prefetches

Prefetch candidates produced through locality inference are stored in one or more of several prefetch lists. For the purposes of this study we have 8 lists, 8 entries each. Thus we can support 8 distinct transactions or atomic blocks. If there are fewer transactions which require more than 8 prefetch entries, two or more lists can be chained together. This is managed by the Transactional Prefetch List Map (TPLM), as shown in Figure 4.3. This is a structure with 8 entries. Each entry contains a TXID (transaction identifier) field and an 8-bit map with high bits indicating prefetch lists allocated to the transaction. Chaining lists together provides flexibility in dealing with transactions of different sizes. When more than 8 transactions exist or no prefetch lists are available we employ an LRU scheme to release resources for the least recently invoked transaction. Prefetches are issued when a transaction begins and has prefetch lists associated with it. In our experiments with lazy conflict resolution designs, it is safe to not regard prefetches as transactional accesses.

Each entry in the prefetch list contains the cache line address, a PE (prefetch enable) bit, a PU (Prefetch Useful) bit and a 2-bit counter. The PE bit is set when the corresponding line is invalidated or evicted from the cache or when a transactional store updates it. Lines with PE bit set to 0 are not prefetched. Transactional commits reset all PE bits. PE bits are also reset when a cache line fill occurs and a transactional update to the line has not yet been issued. All PU bits are reset when a transaction begins. The PU bit is set when a transactional store targets the corresponding cache line, indicating that the address still retains locality.

4.3.3 Trimming Prefetch Lists and Transactions

The two bit counter for each prefetch candidate is set to 4 when the entry is first created. On transaction commits the counter is decremented for all entries in the prefetch list for which PU bit is not set. If this counter reaches 0 the line is not prefetched any more. If all entries for a certain transaction have counts set to 0, the resources (prefetch lists and TPLM) are released for use by other transactions. This is easily achieved by associating a 3-bit counter with each prefetch list. It tracks the number of active prefetch candidates in the list. It is incremented when entries are added to the prefetch list after training and decremented every time a prefetch candidate is trimmed. When the counter is decremented to zero the corresponding bit in the list allocation bit-map in the TPLM is reset. When all

Component	Description
Cores	32 in-order 2GHz Alpha cores, 1 IPC
L1 Caches	32KB 4-way, 64B lines, 1-cycle hit
Bloom filter	256-bit, parallel H3, 1 hash function
L2 Cache	1MB/bank 8-way, 64B lines, 10-cycle hit
Memory	4GB, 150-cycle latency
Interconnect	2D mesh, 2 cycles per hop
Directory	full-bit vector sharers list, 10-cycle directory latency

Table 4.1: Simulation parameters.

bits in the list allocation bit map of a transaction are set to zero, the entry can be reused. The next invocation of such a trimmed transaction will be eligible for locality inference, when prefetch lists will be rebuilt.

4.4 Evaluation

In this section we evaluate the performance of transactional prefetching. We use as baseline Scalable-TCC, a state-of-the-art lazy HTM system. We first describe the simulation environment that we use, then we present our preliminary results.

4.4.1 Simulation Environment

For the evaluation we use M5 [6], an Alpha 21264 full-system simulator. We modify M5 to faithfully model the Scalable-TCC proposal to operate in a chip multi-processor (CMP) with private L1 caches and a banked L2 cache. The design is configured to avoid rare overflows of transactional data from private caches. These are handled using a special overflow buffer. In our experiments only a few such events are noticed. Scalable-TCC has an always-in-transaction approach and employs lazy conflict detection and resolution at commit time, transactional updates are kept in private buffers (caches) to maintain isolation. Note that the prefetch mechanism is invoked only for transactions defined in the application source code. Table 4.1 summarises the system parameters that we use, with one level of private cache and a 2D mesh network connecting the shared L2 banks,

Benchmark	Input parameters
Genome	-g4096 -s128 -n524288
Intruder	-a10 -l32 -n8192 -s1
KMeans	-m15 -n15 -t0.05 -i n32768-d24-c16
Labyrinth	-i random-x96-y96-z3-n384.txt
SSCA2	-s13 -i1.0 -u1.0 -l3 -p3
Vacation	-n8 -q40 -u90 -r1048576 -t32768
Yada	-a20 -i ttimeu10000.2

Table 4.2: Evaluated STAMP benchmarks and input parameters.

resembling the Scalable-TCC proposal. Our proposed transactional prefetching scheme is implemented on top of the baseline HTM. This detailed simulation model , denoted as TP (Transactional Prefetching), employs one 256-bit H3 bloom filter. In addition, we also simulate an idealised model that at the beginning of a transaction prefetches all the lines that have been speculatively written by that transaction in the past. These prefetches are considered to be serviced instantaneously. We name this model PA (Prefetch All).

We use the STAMP benchmark suite [18] to evaluate our proposal. Table 4.2 lists the evaluated workloads and input parameters. We exclude the application Bayes from our evaluation, because this application has non-deterministic exiting conditions leading to severe load imbalance between threads, which makes comparison between different systems inconclusive.

4.4.2 Performance Results

Figure 4.4 shows the execution time breakdown for the HTM systems that we evaluate, namely Scalable-TCC (S), Transactional Prefetching (TP), and Prefetch All (PA). The results in Figure 4.4 are normalised to Scalable-TCC 32-threaded executions, and they are split into six parts, namely Barrier, Commit, Useful, StallCache, Wasted, and WastedCache. The component *Useful* is defined as one cycle per instruction plus the number of memory accesses per instruction multiplied by the L1D hit latency; the component *StallCache* is defined as the time spent waiting for an L1D cache miss to be served. Analogously, for aborted transactions we define Wasted and WastedCache.

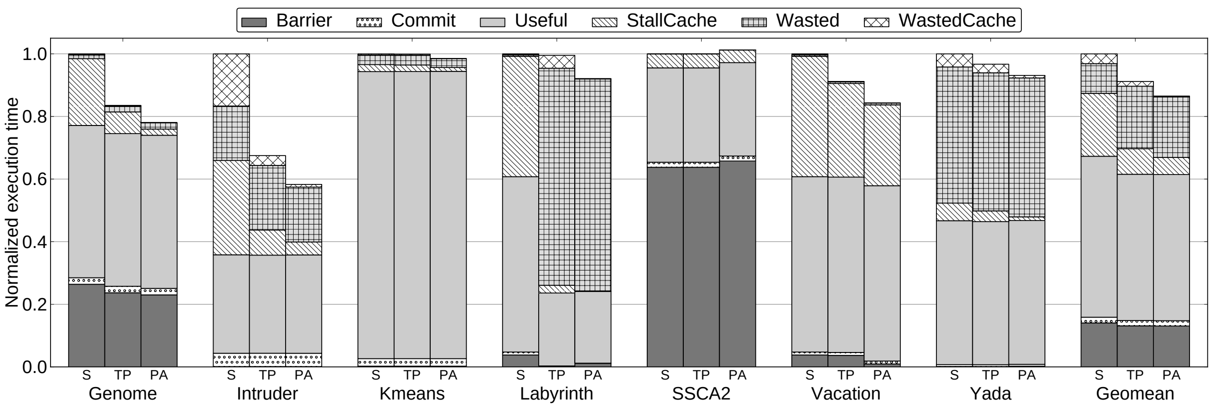


Figure 4.4: Normalised execution time breakdown for 32 threads.
S – Scalable-TCC; **TP** – Transactional Prefetching; **PA** – Prefetch All

Intruder shows remarkable improvement when prefetching is enabled (a speedup of more than 30%). It is a highly contended application exhibiting significant locality across various transaction invocations. In this scenario prefetching data results in substantial shortening of transaction lifetimes. The components, StallCache and WastedCache, show major reductions, as can be seen in Figure 4.4. We highlight this application because in our opinion it is an important workload that is representative of applications with limited concurrency. Such multithreaded applications will gain importance as parallelization is expected to become the only source of performance scaling.

Genome shows moderate contention and a significant amount of locality for most transactions (see Figure 4.1). It shows two distinct phases during execution – a short early high contention phase followed by a longer phase with low to moderate contention. The benefits of prefetching accrue in the first phase, yielding an 16% improvement over the baseline.

Yada is another application with moderate contention (see Figure 4.1). Prefetching lines improves performance, though not by much (3%). One of the reasons for this is limited tracking resources at the prefetcher. Yada has large transactions, and the number of prefetched addresses constitutes a small fraction of the memory references generated.

Vacation has large transactions, there is very little contention and transactions are read dominant. The dominant transaction (shown as Tx1 in Figure 4.1) has good locality of reference and the TP configuration is able to take advantage of this, yielding an improvement of about 15% over the baseline (as seen in Figure 4.6).

KMeans exhibits short phases with some degree of locality. This is evident from the

Benchmark	%Useful	%Trimmed	%Cache improvement	Prefetches per commit
Genome	92.88	81.36	60.07	0.02
Intruder	99.87	11.11	64.55	4.28
KMeans	41.64	99.54	1.55	0.2
Labyrinth	100.00	0.00	82.74	2.27
SSCA2	0.00	99.49	0.00	0
Vacation	98.04	14.66	14.74	0.03
Yada	98.97	38.64	36.66	2.26
Mean	88.56	40.88	37.19	1.3

Table 4.3: Statistics of Transactional Prefetching for evaluated workloads.
Legend: **%Useful** — Percentage of useful prefetches compared to issued prefetches; **%Trimmed** — Percentage of trimmed entries compared to added in prefetch lists; **%Cache improvement** — Percentage improvement of total cache service time compared to Scalable-TCC; **Prefetches per commit** – Average number of prefetches issued per committed transaction.

number of trimmed and useful prefetches as shown in Table 4.3. However, transactions do not appear to have a dominant effect on execution time in this application. We, therefore, do not notice any appreciable deviation in execution times across various configurations.

SSCA2 is a highly concurrent application with little contention and almost no locality across transactions (see Figure 4.1). Hence, prefetching is not expected to play a role here, and as shown in Table 4.3 our proposed prefetch mechanism issues just 35 prefetches spread over more than 100,000 transaction invocations. Moreover, these prefetches get trimmed from the lists rapidly (as indicated by the high (99.49%) percentage of trimmed prefetches, see Table 4.3). Though Labyrinth repeatedly accesses a large set of addresses, it executes a very small number of transactions (less than ten instances of each defined transaction), leading to negligible performance gains for the TP configuration. Moreover, due to Labyrinth's lack of parallelism and sensitivity to transaction interleaving, some configurations exhibit increased contention and therefore more wasted work, as can be seen in Figure 4.4.

Figure 4.5 shows the scalability chart for the evaluated workloads using 32 threads on 32 cores. Intruder has a remarkable boost in scalability reaching 15.3× with TP, a promising result for an application that is known to have difficulties to scale. Noticeable improvements can be also seen in Genome, Vacation and Yada, while applications like

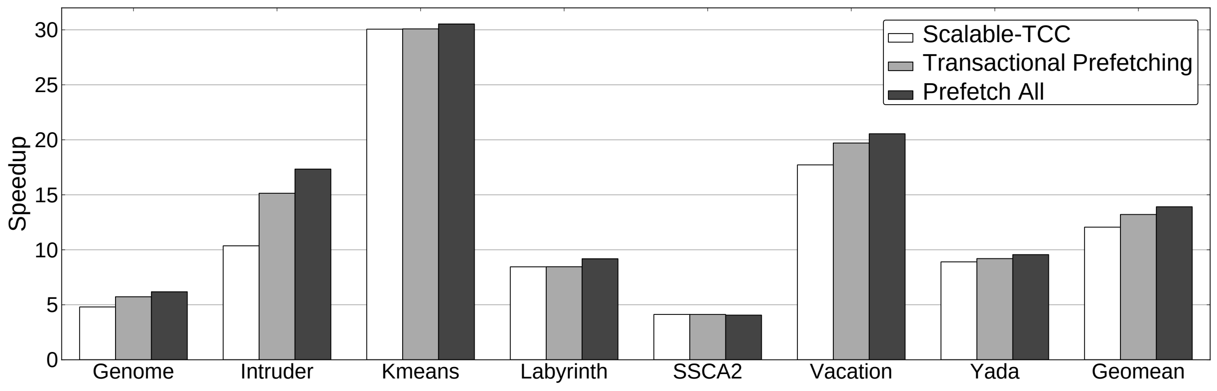


Figure 4.5: Scalability for 32 threaded workloads.

SSCA2 and Labyrinth remain flat due to their transactional characteristics.

Overall, as shown in Table 4.3 our transactional prefetching mechanism successfully infers locality from the evaluated workloads, achieving more than 90% utilisation of issued prefetches for all applications except KMeans, where locality is high, but appears in short phases. Moreover, our design is able to detect scenarios where prefetching is not useful, for example in applications like SSCA2, and does not issue useless prefetches for such scenarios. In general, as can be observed in Table 4.3, if the usefulness of prefetches is low then the number of issued prefetches per committed transaction is rather small as well.

Figure 4.6 shows relative cumulative execution times for each transaction defined in code. Only successful commits have been considered. From these numbers it is possible to estimate the impact of transactional prefetching. For each transaction, two bars are shown – the left one corresponds to execution time seen with the baseline design and the right one corresponds to execution time with prefetching enabled. It is instructive to compare these numbers to those shown in Figure 4.1. We can see that applications which show high locality (Genome, Intruder, Yada, Vacation) also see an improvement in execution times. The improvement moreover is proportional to the degree of locality seen – for example, in Intruder most transactional accesses target only a few addresses and hence, we see a larger improvement than that seen in Yada. SSCA2 and KMeans do not show much improvement since there is little locality.

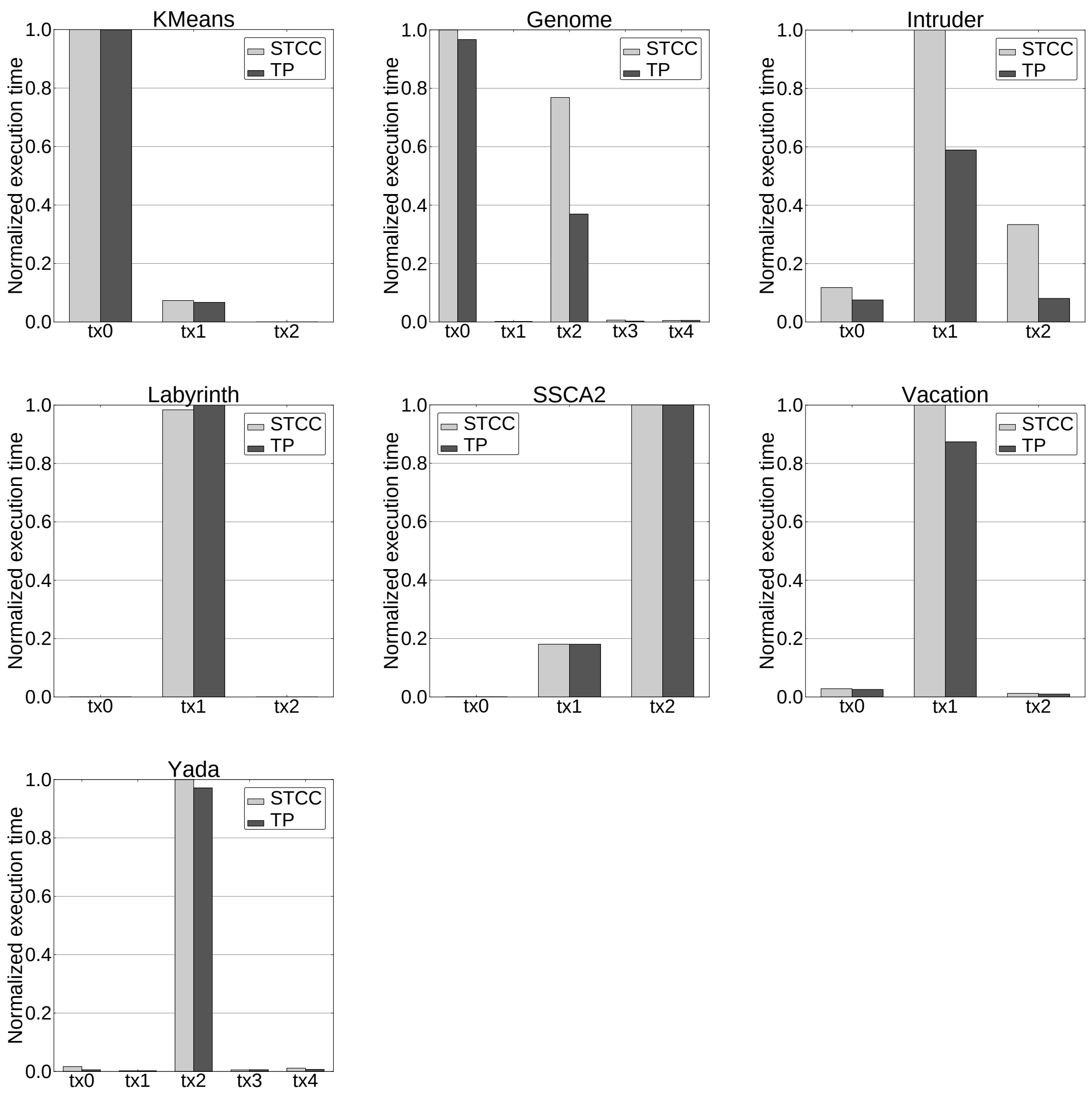


Figure 4.6: Impact on transaction execution times.

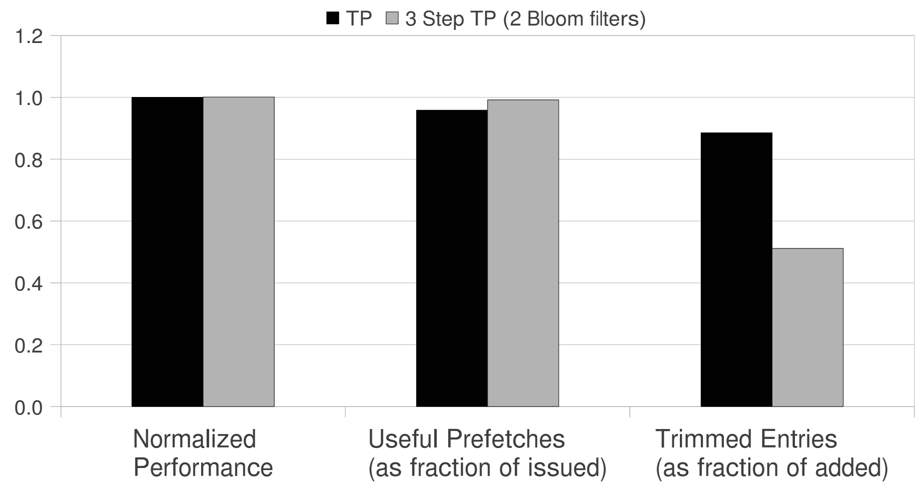


Figure 4.7: Impact of the 2-filter 3-step approach.

4.4.3 Two Filter Approach

More accurate learning of prefetch candidates can be performed by employing two bloom filters in a ping-pong fashion. We test this approach by using a 3-step interative refinement design. The first step inserts write-set cache line addresses in one (BFx) of the two filters (BFx and BFy). The next invocation of the transaction triggers the second step wherein written lines that are found in BFx are inserted in BFy. The third invocation starts the final step of the learning process, filling prefetch candidate lists based on written lines that are found in BFy. As before, we train one transaction at a time, releasing training resources on completion. This approach enables more accurate learning at the cost of responsiveness (it takes longer to train). Figure 4.7 shows how this approach compares against the single filter approach in terms of overall performance and the fraction of useful and trimmed prefetches. Although there is no appreciable difference in performance the two filter approach generates more accurate prefetches, as indicated by a larger fraction of useful prefetches and fewer trims.

4.4.4 Sensitivity Analysis

Bloom filter configuration has little impact of performance of transactional workloads. We varied bloom filter sizes, ranging from 128 bits to 1024 bits and also implemented perfect signatures. There is remarkable consistency in execution times across different filter

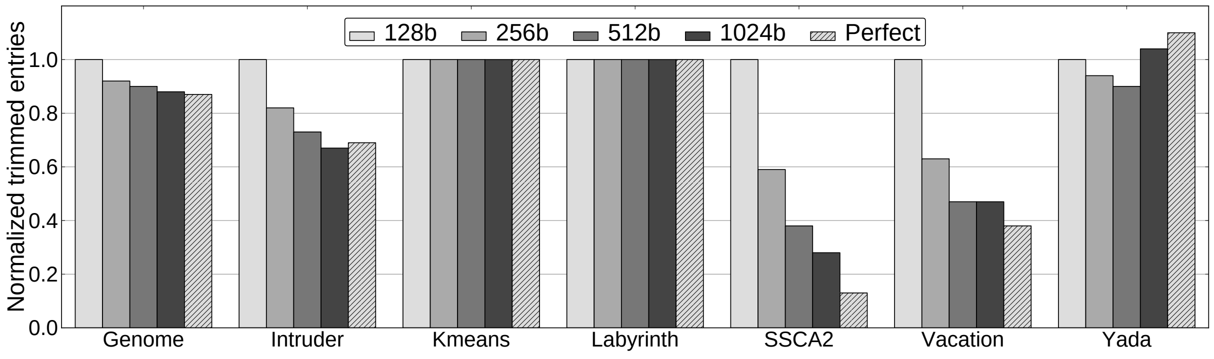


Figure 4.8: Impact of bloom filter size on trimmed entries.

configurations. This is because few additional prefetches arising from increased false positives with small bloom filter sizes have negligible impact on performance and are quickly trimmed from prefetch lists. Figure 4.8 shows that smaller filters result in more trimmed entries. However, in the case of Yada, variations in behaviour induced by transaction interleaving cause minor deviation in the number of trimmed entries (with a 2% spread in execution times).

4.5 Related Work

Although the first proposal by Herlihy and Moss [40] appeared in 1993, research in TM gained momentum with the introduction of multicore architectures. Two early HTM proposals, TCC [37] and LogTM [90], explore two very different points in the HTM design space. TCC defines a lazy conflict resolution design where transactions execute speculatively until one tries to commit its results and causes the re-execution of any concurrent conflicting transaction. LogTM describes an eager conflict resolution design that employs coherence to detect conflicts as soon as they occur and are resolved by asking the requester to retry (with a way to break occasional deadlocks through software intervention). Since then a lot of work has been done targeting a host of different issues that arise when transactional applications run on multicores. Bobba *et al.* [14] categorised pathologies that can arise in fixed policy HTM designs and degrade scalability and performance. The paper pointed out performance bottlenecks that can arise out of limited commit bandwidth in lazy conflict resolution designs and overheads due to excessive aborts in eager resolution designs. Several designs since then have targeted improved scalability in lazy conflict resolution systems through various means – making write-set commits more fine-grained [19, 65, 66] and ensuring conflicting transactions do not interfere with an on-going commit [62, 85]. Others have attempted to reduce abort overheads in both eager and lazy conflict resolution systems – by allowing eager systems to utilise deeper levels of the memory hierarchy to buffer old values [51] and by having caches with special SRAM cells that can store two versions of the same line simultaneously [5]. Yet others have attempted to incorporate the best of both eager and lazy policies in one design – at the granularity of application phases [60], at the granularity of transactions [52], and at the granularity of cache lines [83]. There exist studies that have attempted to insulate the coherent cache hierarchy from adverse effects of repeated aborts [59]. These varied attempts at reducing overheads involved in shared data accesses by cooperating threads have motivated the design effort in this work. This chapter, however, presents a study and design that is largely orthogonal to the various design approaches discussed above. It uses the fact that transactions show locality of reference which can be utilised to improve the speed at which they can complete updates to shared data, thereby improving speed and reducing contention.

Several prior studies have developed ideas regarding cache line prefetching [45, 76] and investigated various prefetching schemes based on detecting cache-miss patterns in non-transactional workloads. This chapter, unlike prior work, describes a scheme that does not rely upon the existence of a simple pattern (like a stride) in the memory reference stream. It can learn arbitrary sets of cache line addresses as long as they show locality of reference across multiple invocations of the same section of code. Thus, this proposed technique is expected to be complementary to others. Moreover, with this technique prefetches can be issued earlier than in other techniques. Chou *et al.* [20] present epoch-based correlation prefetches which utilise special hardware and software support structures to detect prefetch trigger events and manage prefetch candidates. Our work presents a simpler, less expensive interface to manage and trigger prefetches using low complexity per-core hardware.

4.6 Summary

This chapter highlights the importance of prefetching data in the new context of hardware transactional memory. Since transactions are used to annotate parts of multithreaded algorithms where concurrent tasks share information, it is important that they run as fast as possible to improve overall scalability of the application. Moreover transactions are clearly demarcated sections of code and thus can be targeted by techniques, such as the one proposed, that attempt to utilise any locality of reference that may exist within such codes. Our technique, using relatively modest hardware support shows improvements for most transactional workloads we have analysed, with substantial gains of up to 35% under high contention (for intruder).

In the future we would like to enhance this technique and apply it to other scenarios to accelerate generic blocks of code that exhibit high locality of reference across invocations. We feel that critical sections and synchronisation operations could also benefit from such prefetching. The observation that high contention is indicative of high locality makes this technique potentially advantageous in mitigating the impact of data-sharing bottlenecks in multithreaded applications. We also wish to study interactions when this technique is combined with other forms of prefetching, using the insights so acquired to develop synergistic techniques that further improve the design to speed up both transactional and non-transactional code.

5 HARP: Hardware Abort Recurrence Predictor

5.1 Introduction

The problem of extracting thread level parallelism through speculative execution has received a lot of attention from both industry and academia [39, 68]. In particular, Hardware Transactional Memory (HTM) [40] offers performance comparable to fine-grained locks while, simultaneously, enhancing programmer productivity by largely eliminating the burden of managing access to shared data. Recent usability studies support this thesis [18, 71], suggesting that Transactional Memory (TM) can be an important tool for building parallel applications. For these reasons, HTM is receiving increasing attention from the industry [24, 25, 29], and IBM has released their first chip with built-in HTM support, the BlueGene/Q [87]. More recently, Intel has published ISA extensions (TSX) that provide support for basic HTM and lock elision, with the intention of supporting these in upcoming products [43].

An HTM system allows concurrent speculative execution of blocks of code, called transactions, that may access and update shared data. However, in the presence of data conflicts transactions may abort, i.e., the results of speculative execution are discarded. This results in wasted work, expensive rollbacks of application state, and inefficient utilisation of computational resources. While conflicts due to concurrent accesses to shared data cannot be completely eliminated, mechanisms to avoid starting a transaction when it is likely to fail are necessary for maximising computational throughput. Moreover, in scenarios where multiple scheduling options are available, having such mechanisms can expose additional parallelism and improve resource utilisation.

While single application performance is still important, systems where multiple parallel applications coexist are expected to become increasingly common in the near future. The performance of HTM in scenarios with abundant transactional threads is still an open question, and solutions that provide efficient utilisation of computational resources and good performance are required for TM to gain wide acceptance. In the past, considerable work has been done on contention management, but mostly in the field of Software TM (STM) [4, 30, 73]. These proposals typically react *after* aborts happen, without trying to avoid future conflicts. Conversely, a few HTM proposals exist that try to avoid execution of possibly conflicting transactions [8, 10, 91]. However, these solutions do not provide full hardware support and rely on expensive and specialised software runtime routines and data structures. Moreover, the efficacy of these proposals in scenarios with multiple concurrently executing applications is unclear.

In this chapter, we introduce Hardware Abort Recurrence Predictor (HARP), a comprehensive hardware proposal that identifies groups of transactions that are likely to be executed concurrently without conflicts. Our proposal allows other threads or applications to execute when the expected duration of contention is long, providing better throughput when running several applications, and potentially higher parallelism when several threads of the same application are available for scheduling. Moreover, HARP dynamically chooses a contention avoidance mechanism based on expected duration of contention, in order to maximise resource utilisation, while minimising the amount of wasted work due to transaction aborts. HARP avoids software overheads by using simple hardware structures to record transactional characteristics. More specifically, we notice strong temporal locality in contended addresses in transactional applications. By detecting when conflicting locations change, we can identify *when* contention is likely to dissipate.

To evaluate HARP, we compare it against “Bloom Filter Guided Transaction Scheduling” (BFGTS) [8], a state-of-the-art transaction scheduling technique, and LogTM [57], a well established HTM design. Our evaluation includes single-application setups, comprising a scenario with the same number of threads as cores, and a scenario with more threads than cores. We provide insights on when using more threads can extract additional parallelism, and show that HARP outperforms LogTM and BFGTS on average by 109.7% and 30.5% respectively. Moreover, we are the first to study the performance implications of a transactional multi-application setup where, again, our technique outperforms the other evaluated proposals. In addition, we show that HARP is significantly more accurate in terms of predictions and resource utilisation for all the evaluated setups. Compared to BFGTS, HARP has on average 42% and 55% lower abort rates for single-application and multi-application workloads respectively.

5.2 Related Work

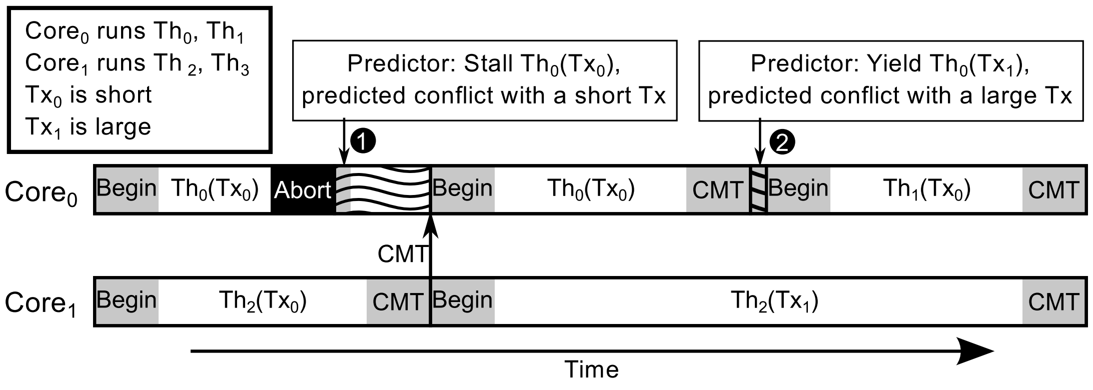
Initial efforts on Software TM (STM) contention managers by Scherer and Scott use a set of heuristics to abort transactions and choose backoff duration when facing a conflict [73]. Further developments focused on user-level support to reduce contention, by either using runtime metrics like commit rate or dynamically discovering pairs of transactions that should not be executed in parallel [4, 30, 79]. More recently, work by Maldonado *et al.* [54] explores kernel-level TM scheduling support. They define several scheduling strategies, ranging from a simple yielding strategy to a more elaborate scheduler based on queues, each having its advantages but none standing out as a clear winner for the set of workloads evaluated. All proposals mentioned above are reactive – imposing measures *after* conflicts happen without trying to avoid future conflicts.

In the field of HTM there has been less research on this area. Exponential backoff, as introduced in LogTM [57], is the most common contention management mechanism adopted in HTM designs. This was later used by Bobba *et al.* [14] for a thorough analysis identifying several performance pathologies present in HTM systems, including some that are closely related to contention management issues. The solutions proposed were not investigated in depth as it was not the focus of the paper.

Adaptive Transaction Scheduling (ATS) by Yoo and Lee [91] proposes queueing transactions in a centralised hardware queue if the amount of contention seen surpasses a preset threshold. A metric named contention intensity is maintained per thread. If this intensity surpasses a preset threshold, transactions are queued into a centralised hardware queue and dispatched one at a time serialising their execution. When the contention intensity decreases below the threshold, transactions are allowed to bypass the queue and execute in parallel again. ATS has little impact on performance when contention is low, and ensures single global lock performance for contended scenarios with small hardware and software requirements. However, serialising all transactions when contention intensity increases can be overly pessimistic, as not all transactions have to be highly contended. Moreover, like backoff-based policies, this mechanism is reactive and takes action *after* contention is already present in the system.

Blake *et al.* were the first to introduce proactive mechanisms to manage contention. Proactive Transaction Scheduling (PTS) is one such technique [10]. PTS employs a global software graph structure that maintains the confidences of conflict, with nodes representing transactions and edges representing the confidence level of a conflict reoccurring in the future. In addition, per-transaction statistics such as the read- and write-set in the form of Bloom filters are also kept in software. PTS queries the global graph at the beginning of a transaction to form a decision whether to serialise against an already running transaction, and uses the per-transaction statistics to dynamically update the global conflict graph. PTS can schedule more optimistically than ATS, thus attaining better performance. However, PTS needs to query a global data structure at the beginning of each transaction and update it when committing or aborting, incurring significant overheads.

Bloom Filter Guided Transaction Scheduling (BFGTS) [8] outperforms PTS by employing a hardware accelerator and better Bloom filter manipulations using a metric termed *similarity* – a measure of memory locality present throughout different executions of a transaction. If two transactions with high similarity conflict, the conflict is likely to be persistent. However, this approach may not be accurate because two transactions could conflict very infrequently while still having high similarity, especially if they perform a large number of reads over the same locations. BFGTS is largely implemented using (1) software data structures that store confidences of conflict, per-transaction Bloom filters, and similarity values; and (2) runtime routines that execute when the system serialises, commits, or aborts a transaction. These routines can be larger than the transaction itself, and may not be compatible with arbitrary transactional codes (e.g., different languages).



Core$_0$ runs Th$_0$, Th$_1$
Core$_1$ runs Th$_2$, Th$_3$
Tx$_0$ is short
Tx$_1$ is large
Predictor: Stall Th$_0$(Tx$_0$), predicted conflict with a short Tx
Predictor: Yield Th$_0$(Tx$_1$), predicted conflict with a large Tx
❶
❷
Core$_0$
Begin
Th$_0$(Tx$_0$)
Abort
Begin
Th$_0$(Tx$_0$)
CMT
Begin
Th$_1$(Tx$_0$)
CMT
CMT
Core$_1$
Begin
Th$_2$(Tx$_0$)
CMT
Begin
Th$_2$(Tx$_1$)
CMT
Time

Figure 5.1: Example of efficient use of computational resources.

Per-core hardware support includes a list of transactions running in remote cores, an additional 2KB cache, and a Bloom filter to infer memory locality. This hardware performs a prediction in a few cycles at the beginning of a transaction, but cache misses can increase prediction latency.

5.3 Overview and Motivation

5.3.1 Overview Example

Figure 5.1 illustrates how abort prediction enables efficient utilisation of computational resources with a simple example. It shows two cores, each executing two threads from the same application. Each thread has two transactions, where the first is short (Tx_0) and the second is long (Tx_1).

The example assumes an initial state where software threads Th_0 and Th_2 are both allowed to execute Tx_0 concurrently and eventually transaction Tx_0 in Th_0 aborts, meaning that $Core_0$ mispredicted the conflict. An HTM system without abort prediction support would now blindly try to re-execute the transaction, possibly leading to more conflicts and inefficient resource utilisation. However, if the system is aware of contention it can proactively take steps to avoid it. At time ❶, $Core_0$'s predictor decides to stall the transaction because it predicts a conflict is likely to happen with a short transaction. Thus, in this case, waiting until the short transaction finishes makes sense. When $Core_1$ commits its transaction (Tx_0), its predictor allows the execution of the next transaction (Tx_1) of the same thread Th_2, and the stalled execution in $Core_0$ can be resumed with the approval of its

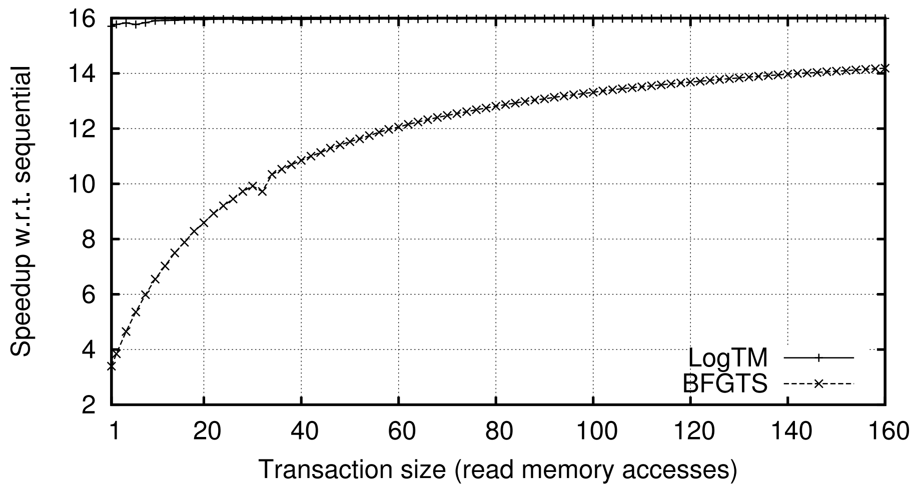


Figure 5.2: Overheads of evaluated systems at different commit throughputs. Eigenbench with varying transaction sizes, 128K iterations and 16 cores.

predictor. $Core_0$ can now successfully commit its transaction, but when trying to move on to the next transaction (Tx_1), the predictor preempts the thread because a conflict is predicted using past history (explained in depth later). Now, at time ❷, the conflict is against a transaction known to be long, so the system decides to yield the thread Th_0, and Th_1 is granted permission to start execution. The example ends with both running transactions committing in parallel. Note that if Th_0 had not yielded and Tx_1 is contended,$Core_0$ would have probably wasted time or even experienced a series of aborts until $Core_1$'s transaction commits, whereas with abort prediction support a different transaction has executed and committed meanwhile.

5.3.2 Why Do We Need a Hardware Solution?

Previous techniques rely on software components in their designs. To understand the overheads imposed by such components and the prediction mechanism in general, we perform an experiment using Eigenbench [42], a flexible exploration tool for TM systems. We configure Eigenbench to have no contention and to maximise total transactional execution time.

We evaluate LogTM and BFGTS using its best performing configuration. Figure 5.2 shows our experiments on a range of transaction sizes (smaller transactions demand higher commit throughput). The smallest transaction size evaluated performs one read operation

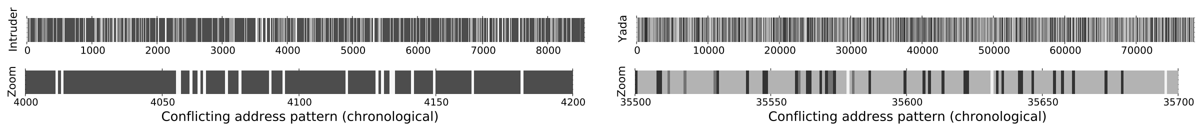


Figure 5.3: Chronological distribution of conflicting addresses for a transaction of interest in Intruder (left) and Yada (right). The x axis represents cumulative abort count. Each different grey scale level represents a different conflicting address.

and a small amount of work with the read data. Since there is no contention, LogTM scales almost linearly with any transaction size. BFGTS experiences a notable performance degradation with small and medium size transactions. Even with relatively large transactions (more than 100 reads) the performance gap under no contention is significant. The hardware accelerator of BFGTS performs a quick decision at the beginning of each transaction, however, having to interrupt the normal flow of execution on every commit (and abort) to execute additional code is the main cause of the slowdown seen in the chart. With a hardware solution we aim to minimise these overheads and deliver performance close to LogTM in uncontended scenarios.

5.3.3 Detecting Conflict Recurrence

An efficient abort prediction mechanism needs to track transaction characteristics in order to anticipate when conflicts are going to happen. It must also possess the capability to detect when conflicts dissipate. To this end, we introduce the use of *conflict lists*. A transaction's conflict list contains the last few conflicting addresses that triggered an abort; locality in such addresses is an indication that contention between two transactions is recurring in nature. These lists can be of small size, thus suitable for a hardware approach such as ours where the amount of information that can be kept is limited. To motivate this design choice, we show a study done using two of the most contended applications of the STAMP benchmark suite [18]: Intruder, a network packet intrusion detection program, and Yada, a Delaunay mesh refinement algorithm. For both applications we have looked at the history of conflicting cacheline addresses that cause an abort. More specifically, we monitored one transaction of interest (long and contended) for one of the executed threads.

Figure 5.3 shows two bars for each application with the chronological distribution of conflicting addresses that triggered an abort for the studied transaction. Each upper bar shows the entire sampling, while the lower bars show a magnified view of a representative region. Each address has a different grey scale level associated. The x axis quantifies the total number of aborts seen so far, each being triggered by a conflicting address. For better visualisation, ten addresses are considered for Intruder and five for Yada, enough to cover more than 98% of the total number of aborts. As can be seen, conflicting addresses present high temporal locality, with a dominant address in both cases. These addresses with high locality are easy to capture with the proposed conflict lists.

A conflict between two transactions is likely to be persistent if one of the transactions accesses an address present in the conflict list of the other transaction, and it has likely dissipated otherwise. For example, in applications where contention is data dependent, like Yada, two concurrent transactions may conflict when operating over the same subset of data (addresses), and the conflict will likely dissipate when one of the transactions starts operating over different data (i.e, the transaction does not access addresses present in the other transaction's conflict list). Similarly, if contention is due to accessing a data structure, like in Intruder, conflicts might be present depending on which sections or nodes (addresses) of the data structure are accessed by concurrent transactions. We expect this observation to hold true for most TM use cases, as such conflicts are often unavoidable in parallel programs.

5.3.4 HARP Versatility

HARP is largely decoupled from specific HTM conflict detection and management protocols, requiring just the knowledge of conflicting addresses that trigger an abort. This information is, typically, easy to gather in most designs. Lazy conflict detection has been found to make a system more robust under high contention [18, 77]. This is because one transaction aborts only because another transaction has successfully committed. Though a lazy system as a whole makes progress, individual threads waste substantial computational resources due to aggressive speculation. Simpler HTM implementations tend to use eager conflict detection – e.g., implementations based on extensions to traditional cache coherence protocols.

A mechanism like HARP that aims to (a) prevent concurrent execution of conflicting transactions, (b) provide low abort rates, and (c) swap potentially conflicting transactions for useful work; which makes an eager system become robust under high contention. In addition, eager systems present the following advantages: (a) can benefit from fast lo-

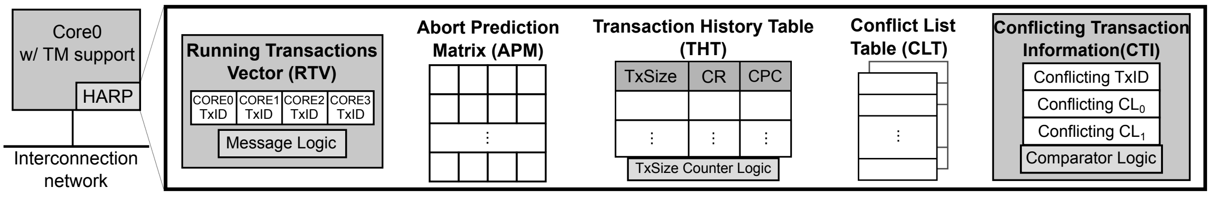


Figure 5.4: HARP extensions to a TM-aware core. Assuming a 4-core system for the RTV and a 2-way CLT. The APM, THT, and CLT have the same number of entries.

cal commits, and (b) eager conflict detection lets HARP take informed decisions earlier regarding the course of execution. For these reasons we frame our study in eager systems.

A hardware approach like HARP transparently provides support for arbitrary transactional codes (i.e., different languages or compilers), which may not be compatible in a software-based approach with specialised routines. In addition, HARP does not need to interrupt the normal flow of execution on the core on every commit and abort as previous techniques require [8, 10]. Finally, HARP's prediction latency and bookkeeping operations are not affected by inherent overheads present in software routines, e.g., cache misses.

5.4 HARP Design and Operation

This section first describes the set of per-core hardware structures necessary to implement HARP, followed by a detailed explanation of its operation. We conclude with a step-by-step execution example.

5.4.1 HARP Hardware Structures

Figure 5.4 illustrates the necessary per-core hardware structures to implement HARP. These structures track important information about current and past transactional executions. The Running Transactions Vector (RTV) has as many entries as cores and tracks a list of transactions currently running on remote cores. Each entry stores a static identifier (i.e., the program counter) of a remote transaction (if any) termed *TxID*'s. The Abort Prediction Matrix (APM), Transaction History Table (THT), and Conflict List Table (CLT) are tagless structures with the same number of entries, which are indexed by TxID. The APM contains a 2-bit saturating counter in each cell. Each counter indicates the confidence of conflict between two transactions. The THT and the CLT store past information from previously

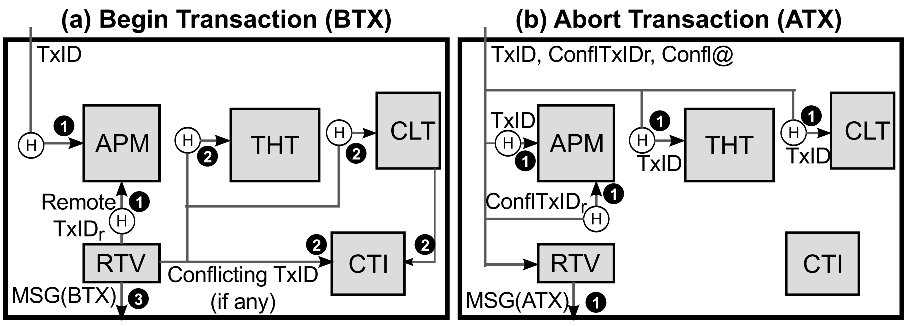


Figure 5.5: Schematic communication overview between HARP hardware structures. A subset of the bits from the TxID (PC) are used to index the APM, THT, and CLT – denoted as *H* (hash function) in the figure.

executed instances of the transactions. Each entry of the THT contains the following per-transaction information: (a) the *average size* (TxSize) of committed instances, (b) a 4-bit saturating counter that indicates the *contention ratio* (CR), and (c) a 4-bit saturating counter indicating the number of *consecutively predicted conflicts* (CPC) by HARP. The CLT contains conflict lists stored in a set associative manner. Each entry of a set stores an address of the transaction's conflict list (last few addresses that caused an abort). Finally, a few additional registers and some glue logic is necessary. These registers, collectively called Conflicting Transaction Information (CTI), are used to store the TxID and conflict list of a possibly conflicting transaction upon a predicted conflict.

Figure 5.5 shows a communication overview between HARP structures during transactional operations. At the beginning of a transaction (Figure 5.5a) a prediction is performed. ❶ The RTV and APM are used to determine if a remote transaction has a high confidence of conflict with the transaction starting locally. If a conflict is found to be likely, ❷ information about the conflicting transaction is gathered from the THT to decide whether to stall or yield the thread. Additionally, the conflict list is read from the CLT and stored in the CTI. Otherwise, if no conflict is predicted, ❸ a non-blocking message is sent through the coherent interconnect to inform remote cores to update their RTVs, and the transaction starts its execution.

On transaction abort (Figure 5.5b), after the speculative state is rolled back, ❶ the confidence of future conflict between the two transactions is incremented in the APM, statistics in the THT and the conflict list in the CLT are updated, and a message is sent to inform remote cores to update their RTVs. On transaction commit, the previously conflicting TxID (if any) stored in the CTI is used to update the confidence of future conflict, the average

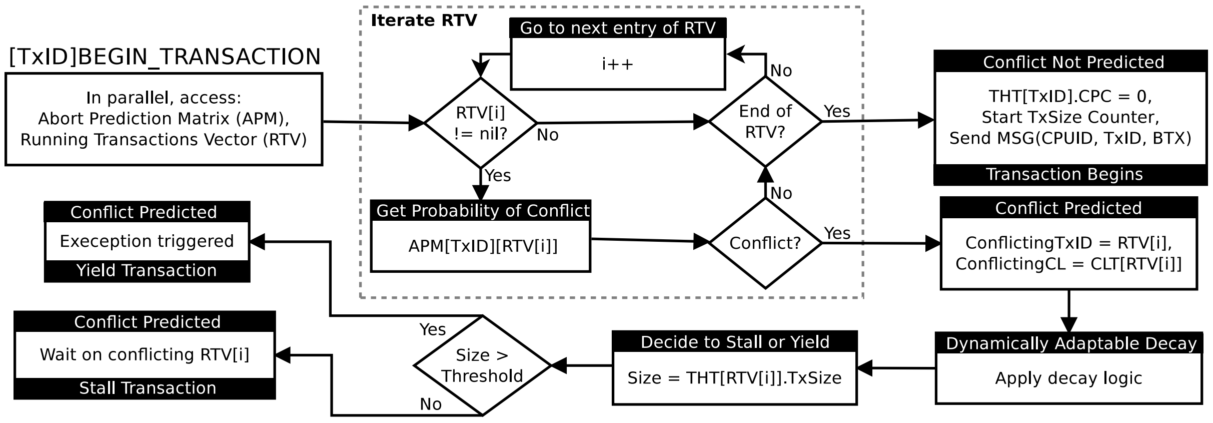


Figure 5.6: Flowchart depicting the process of performing a prediction in HARP for a certain transaction *TxID*.

transaction size is updated in the THT, and a message is sent to inform remote cores.

5.4.2 HARP Operational Details

Performing a prediction

Figure 5.6 details with a flowchart the process of predicting whether a transaction *TxID* will conflict or not. HARP iterates over the RTV until a conflict is found or the end of the RTV is reached (conflict not predicted). The APM is indexed by $TxID$, the corresponding row of the matrix can be seen as the set of confidences that $TxID$ might conflict with remote transactions. To know if a conflict with a remote transaction $TxID_r$ is likely to happen, $TxID_r$ is used to index by column, obtaining the cell with the confidence of conflict. The confidences are represented using 2-bit saturating counters, where the two upper states predict conflict and the two lower states predict no conflict. If a conflict is not predicted, the transaction can start its execution. Otherwise, if a conflict is predicted, HARP uses the local knowledge stored in the THT and CLT to infer the transactional characteristics of the remote conflicting transaction. The conflicting transaction identifier and its conflict list are stored in the CTI to later adjust confidences of conflict at commit time. If the size of the conflicting transaction exceeds a threshold, an exception is thrown and its handler will yield the thread in a similar way `pthread_yield()` does. Otherwise, HARP will stall the execution until the conflicting transaction is no longer running. Note that the CTI registers are part of the thread context, i.e., they are saved and restored on a context switch.

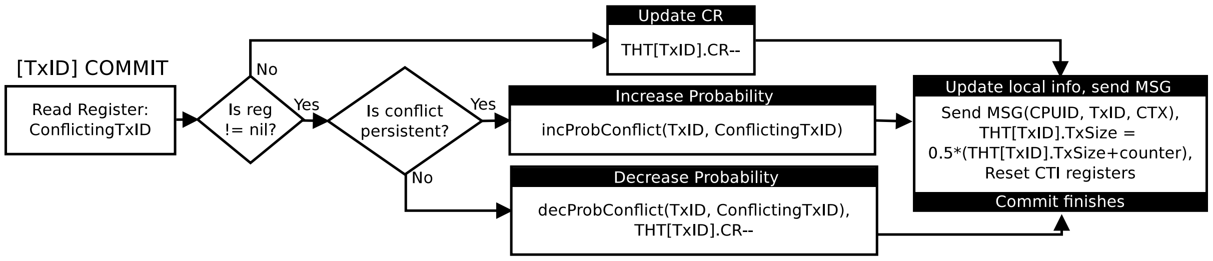

[TxID] COMMIT
Read Register: ConflictingTxID
Is reg != nil?
No
Yes
Is conflict persistent?
Yes
No
Update CR
THT[TxID].CR--
Increase Probability
incProbConflict(TxID, ConflictingTxID)
Decrease Probability
decProbConflict(TxID, ConflictingTxID), THT[TxID].CR--
Update local info, send MSG
Send MSG(CPUID, TxID, CTX), THT[TxID].TxSize = 0.5*(THT[TxID].TxSize+counter), Reset CTI registers
Commit finishes

Figure 5.7: Flowchart depicting the process of performing a commit in HARP for a certain transaction *TxID*.

Identifying persistent conflicts and committing

We can distinguish between two kinds of running transactions: (a) the ones that start without predicting any conflict, and (b) those that execute after stalling or yielding due to a prediction (serialised). If the transaction was serialised, it has valid CTI data in the registers. Throughout the execution of a serialised transaction, the memory requests are compared against the addresses in the conflict list (CTI registers) of the previously predicted conflicting transaction. This is a crucial point to learn if a conflict has dissipated or is still present. If the transaction accesses an address present in the CTI conflict list, it means that the conflict is potentially persistent, and the transaction had a chance to execute simply because a potentially conflicting transaction instance was not concurrently running; in this case, the confidence of conflict is increased at commit time. If the transaction does not access an address in the CTI conflict list, it means that the conflict between the two transactions is perhaps no longer present, and the confidence of conflict is decreased. Additionally, at commit time the average transaction size and the contention ratio (CR) are updated, the CTI registers are also cleared. A flowchart describing the process is shown in Figure 5.7.

Aborting a transaction

When a transaction aborts due to a conflict, the aborting core increases the confidence of conflict between the two transactions in the APM. The contention ratio (CR) in the THT is incremented, and the transaction's conflict list is updated in the CLT with the conflicting address. Since conflict lists can have repeated elements, the replacement policy is simple. There is no need to do a look up before replacing; instead, an LRU bit decides which entry is replaced. The broadcast message sent when a transaction aborts is slightly larger, it also contains the core identifier and TxID of the remotely conflicting transaction, and the conflicting address. In this manner, besides remote cores updating their RTVs, the remotely conflicting core can also update the confidence of conflict and the conflict list of the remotely conflicting transaction in its local structures. These remote updates on abort are important because they make a transaction aware of a potential conflict and a conflicting address.

Non-blocking communication

When a core starts or exits (commits or aborts) a transaction, communication with remote cores is necessary to keep the RTVs updated. This communication is done via small broadcast messages that include the core identifier, the TxID, and the action being performed (e.g., committing). These messages are non-blocking, which can lead to outdated information in remote cores for a small window of time, but this is not a correctness issue and far less critical to performance than adding synchronisation. The number of such messages is small when compared to coherence messages (∼1% on average in our simulations). Moreover, a large number of simultaneous messages implies a high commit rate, where HARP would not need to interfere. In high contention scenarios, HARP serialises conflicting transactions, which reduces the number of messages. These facts suggest that communication is not a limiting factor for the design to scale (see Section 5.5.6 for related evaluation).

During the process of predicting a conflict, committing, or aborting, all information is available locally. Such a distributed approach eliminates synchronisation overheads between cores and contention when accessing the hardware structures. Note that in order to predict a conflict there must be at least one transaction running on the system. Hence a deadlock scenario where all predictors repeatedly predict conflict cannot occur.

ALGORITHM 5.1: Dynamically adaptable decay algorithm.

```
if THT[TxID].CPC >= THT[TxID].CR then
    decProbabilityConflict(TxID, ConflictingTxID);
    THT[TxID].CPC = 0;
else
    THT[TxID].CPC++;
end
```

Dynamically adaptable decay

The decay targets transactions where contention varies with time, allowing them to execute optimistically faster when contention dissipates. As shown in Figure 5.6, the decay is applied after a conflict is predicted and implements a simple algorithm as shown in Algorithm 5.1. If the number of consecutively predicted conflicts by HARP is at least equal to the transaction's contention ratio, the confidence for the recently predicted conflict is decremented and the CPC counter is reset. Otherwise, the CPC counter is increased. This enables transactions that commit often to decrement their confidences of conflict faster, while contended transactions will need to predict a larger number of consecutive conflicts in order to see their confidences of conflict decremented by the decay. As contention increases, the chances to apply the decay decrease at a faster rate, since having a large number of consecutive predicted conflicts is increasingly unlikely.

Execution example

Figure 5.8 presents a self-contained step-by-step example of HARP's operation.

5.5 Evaluation

In this section, we evaluate HARP by first describing our simulation environment and methodology. We also include an overview of the hardware costs associated to our design. Then we present the main experimental evaluation using single-application and multi-application setups, followed by sensitivity analyses with respect to the most relevant parameters.

5.5.1 Simulation Environment

To evaluate HARP we compare it to two HTM baselines, LogTM [57], a well established system; and a state-of-the-art transaction scheduling technique: Bloom Filter Guided Transaction Scheduling (BFGTS) [8]. In our experiments, both HARP and BFGTS use the LogTM architectural framework for basic TM support. We use the M5 full-system simulator [6]. This simulator was made publicly available by the BFGTS authors [9], thus assuring the BFGTS baseline is faithfully modelled. Queueing delay and resource contention in the

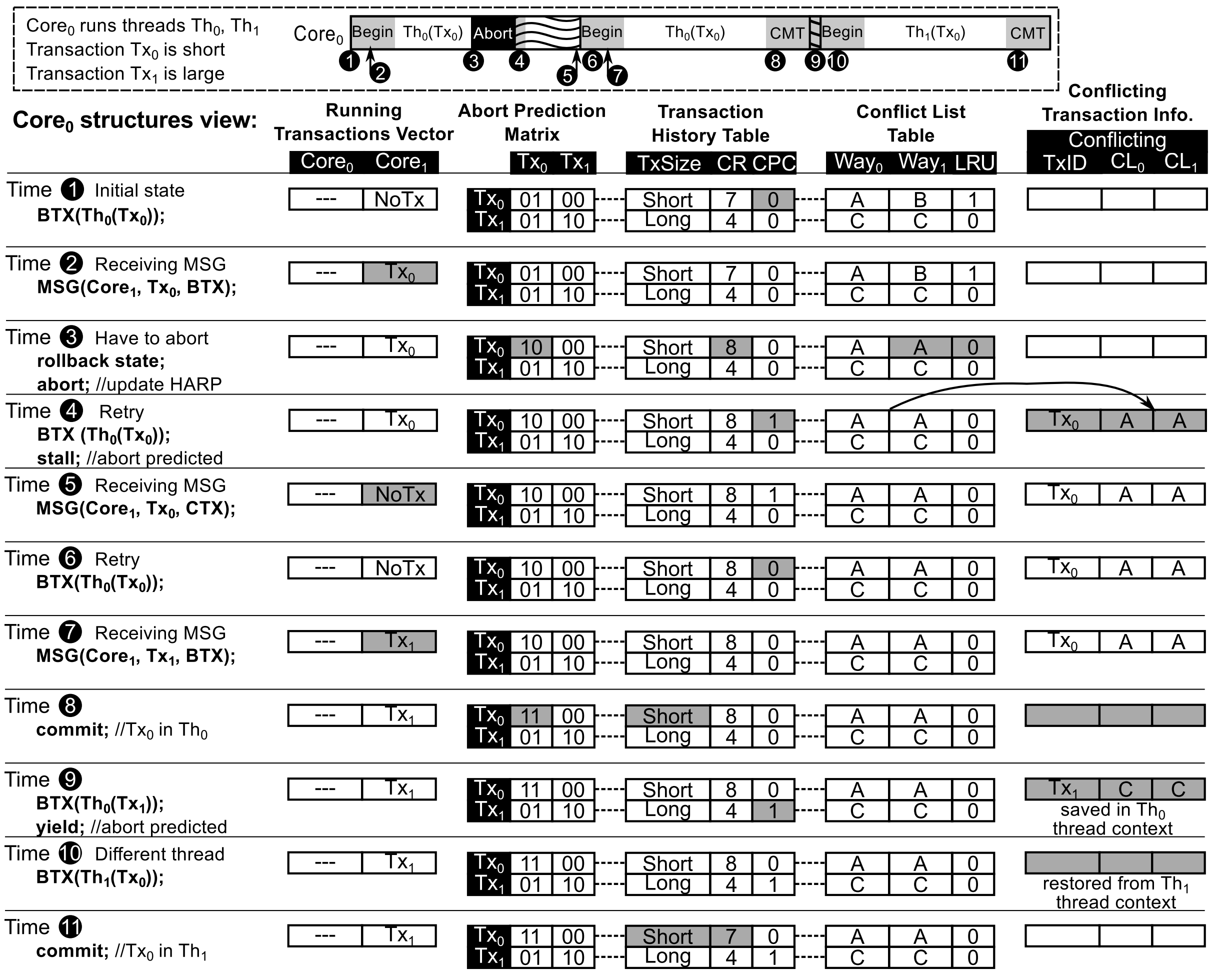


Figure 5.8: HARP execution diagram for a two core system. The box at the top depicts a sequence of events for $Core_0$, matching those presented in Figure 5.1. The rest of the figure shows changes in $Core_0$'s HARP hardware structures at each step (shaded areas), outgoing messages are not shown. The transaction begin at time ❶ triggers the predictor, since no other transactions are running on the system, it can start normally. At time ❷ a remote message from $Core_1$ is received and the RTV is updated accordingly. At time ❸ the transaction aborts due to a conflict with Tx_0 running on $Core_1$. At time ❹ the transaction tries to restart, but this time the RTV is not empty, a conflict is predicted and the CTI registers populated. Since the conflict is predicted against a transaction marked as "short" in the THT, the execution is stalled. Later, at time ❺, a message is received indicating that the conflicting transaction has finished, allowing $Core_0$ to retry again and start ❻. At time ❼, a message is received indicating $Core_1$ started to execute Tx_1, updating the RTV. At time ❽, the running transaction in $Core_0$ commits with valid CTI information because it was serialised. In this example, we consider that during the execution address A was touched, making the previously predicted conflict potentially persistent, so the confidence of conflicting again in the future is increased. At time ❾, $Core_0$ tries to start Tx_1, but a conflict is predicted with a large remotely running transaction, yielding the current thread. Note that before yielding, the CTI info is populated and will be saved as part of the thread context when yielding. At time ❿, a new thread Th_1 is granted execution, restores CTI information (null in this example), and starts executing Tx_0. The transaction commits at time ⓫, updating local information.

Component	**Description**
Cores	16 in-order 2GHz Alpha cores, 1 IPC
L1 Caches	64KB 2-way, private, 64B lines, 1-cycle hit
L2 Cache	16MB 16-way, shared, 64B lines, 32-cycle hit
Memory	4GB, 100-cycle latency
Interconnect	Shared bus at 2GHz
Linux Kernel	Modified v2.6.18
HARP	64 entries for APM, THT, and CLT
Structures	2 addresses per conflict list
BFGTS	2048bit signatures for BFGTS commit routines
Structures	2KB 16-way confidence cache, 64B lines, 1-cycle hit

Table 5.1: Simulation parameters.

memory subsystem and in added structures has been accounted for. The simulation parameters are detailed in Table 5.1.

We use the best performing BFGTS configuration, which skips most calculations in software routines when there is low contention. HARP's prediction cost is modelled as one cycle per lookup in the APM, i.e., 15 cycles in the worse case. Lower prediction cost can be achieved by fetching the entire row of the APM, filtering the columns of interest, and using a set of comparators in parallel – trading hardware footprint for prediction latency. The transaction size threshold that decides when to stall or yield is set to half the average time it takes the kernel to perform a context switch in our system. Note that after stalling, the transaction is not guaranteed to execute as a new abort could be predicted. This transaction size threshold allows for at least two consecutive stalls before having a penalty larger than yielding.

We use the STAMP [18] benchmark suite with nine different benchmark configurations. Table 5.2 describes the input parameters used and the number of transactions defined in each benchmark. The suffixes "-High" and "-Low" provide different conflict rates. We exclude *Bayes* from our evaluation because of its non-deterministic exiting conditions, leading to inconclusive results due to high runtime variability, as noted by many researchers [8, 13, 18]. *Labyrinth* is modified to do the grid copy outside the transaction,

Benchmark		Input parameters	Num Tx
Genome	(G)	-g4096 -s32 -n524288	5
Intruder	(I)	-a10 -l32 -n8192 -s1	3
KMeans-High	(K)	-m15 -n15 -t0.05 -i random50000_12	3
KMeans-Low		-m40 -n40 -t0.05 -i random50000_12	3
Labyrinth	(L)	-i random-x96-y96-z3-n128.txt	3
SSCA2	(S)	-s15 -i1.0 -u1.0 -l3 -p3	3
Vacation-High	(V)	-n8 -q10 -u80 -r65536 -t131072	1
Vacation-Low		-n2 -q90 -u98 -r65536 -t131072	1
Yada	(Y)	-i ttimeu10000.2	6

Table 5.2: STAMP input parameters and number of transactions.

Hardware structure	Equation of cost	Cost (bytes)
Running Transactions Vector	16 entries × (1 TxID/entry × 48 bits/TxID)	96
Abort Prediction Matrix	64 entries × (64 counters/entry × 2 bits/counter)	1024
Transaction History Table	64 entries × ((1 counter/entry × 16 bits/counter) + (2 counters/entry × 4 bits/counter))	192
Conflict List Table	64 entries × ((2 addresses/entry × 48 bits/address) + 1 LRU bit/ entry)	776
Conflicting Transaction Information	(1 register × 48 bits/register) + (2 registers × 64 bits/register)	18
HARP Total Storage	Sum of the above	**2.06 KB**
BFGTS Total Storage	RTV-like structure (96 bytes) + Additional confidence cache (2 KB) + Bloom filter (2048 bits)	**2.34 KB**

Table 5.3: HARP and BFGTS hardware costs for one core.

as done by other researchers, otherwise any concurrency is effectively precluded.

5.5.2 Comparison of Hardware Costs

Table 5.3 shows the storage requirements for HARP and BFGTS. Implementing HARP requires an additional storage of 2.06KB on each core, roughly 3% of a 64KB L1 cache. HARP requires less storage than BFGTS. This is because BFGTS uses an additional 2KB cache to speedup accesses to its software data structures. Moreover, a cache needs additional logic (e.g, tags), not considered in this comparison.

5.5.3 Evaluation Methodology

Our evaluation includes three different system setups: (a) a setup with a single-application using the same number of threads as cores, (b) a setup with a single-application where four threads are assigned to each core, and (c) a setup with two different applications where one thread of each application is assigned to each core, i.e., two threads per core each from a different application (multi-application workloads). While single-application performance is still critically important, we believe that for TM to be widely accepted, it also needs to deliver good performance in such multi-application scenarios. In fact, as parallel programming becomes ubiquitous, future systems would have several multithreaded applications running concurrently in the common case. To the best of our knowledge, we are the first to study multi-application transactional scheduling in an HTM environment.

For the first setup where the same number of threads as cores is used, it is inefficient to yield threads when aborts are predicted. In order to compare BFGTS and HARP fairly, we disable the yield option for this particular setup. This can be accomplished by letting the kernel scheduler notify the hardware when yielding is not useful, as the scheduler would have the knowledge to make such decision. We expect such operating system support to be present in an HTM system. For the multi-application setup, we had to modify the design of BFGTS because the original proposal was not able to deal with multiple applications. In addition, we allow BFGTS to yield. Originally the library would not yield when the number of threads is not larger than the number of cores for a particular application; but we observed that yielding judiciously benefits BFGTS when threads from different applications are available.

$$\text{Efficiency ratio} = \frac{\text{useful\_tx (cycles)}}{\text{useful\_tx+wasted\_tx+abort recovery+stall/yield/backoff+BFGTS commit (cycles)}} \tag{5.1}$$

For each setup, our evaluation includes an execution time breakdown, scalability analysis, and statistics for the evaluated workloads. Execution time breakdowns are normalised to LogTM, and the following components are shown – 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 (*abort recovery*), time spent due to contention management handling (*stall/yield/backoff*), and time spent by BFGTS in the software commit routine. Prediction cost was not visible in charts and it is attributed to other components based on prediction outcome, e.g., to *useful-tx* if the transaction starts and commits. The statistics that we show include a metric that captures how effective contention management is in BFGTS and HARP. This metric, shown in Equation 5.1, is

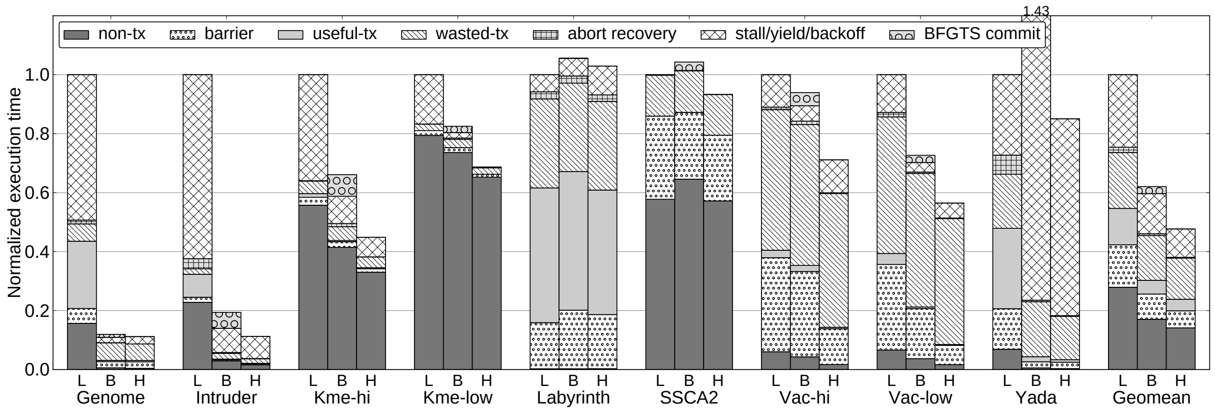


Figure 5.9: Normalised execution time breakdown for 16 threads in single-application workloads.
L – LogTM; **B** – BFGTS; **H** – HARP

an efficiency ratio that compares the amount of useful cycles with the inherent design overheads due to bad predictions and serialisation costs that lead to inefficient resource utilisation.

5.5.4 Single-Application Results

One thread per core

Figure 5.9 presents the execution time breakdown for the evaluated workloads. Overall, the backoff strategy employed by LogTM fails to manage contention and exhibits a large amount of wasted work and serialisation overheads (backoff time) when compared to BFGTS or HARP. Dynamically avoiding the execution of transactions that are likely to fail improves performance and scalability by over 2× on average (see Figure 5.10), while abort rates diminish by 6×, as shown in Table 5.4. These are clear indicators that proposals like BFGTS and HARP are likely to have a significant impact when applied to any HTM system.

Performance improvements of HARP when compared to BFGTS are due to (a) comprehensive hardware support, yet with a smaller hardware footprint than BFGTS (see Section 5.5.2), thus avoiding software data structures and runtime routines; and (b) greater prediction accuracy by focusing only on addresses that actually cause contention. HARP performs better than BFGTS for all the evaluated workloads, attaining 30.5% performance improvement on average.

As can be seen in Figure 5.9, the BFGTS commit routine accounts for a significant

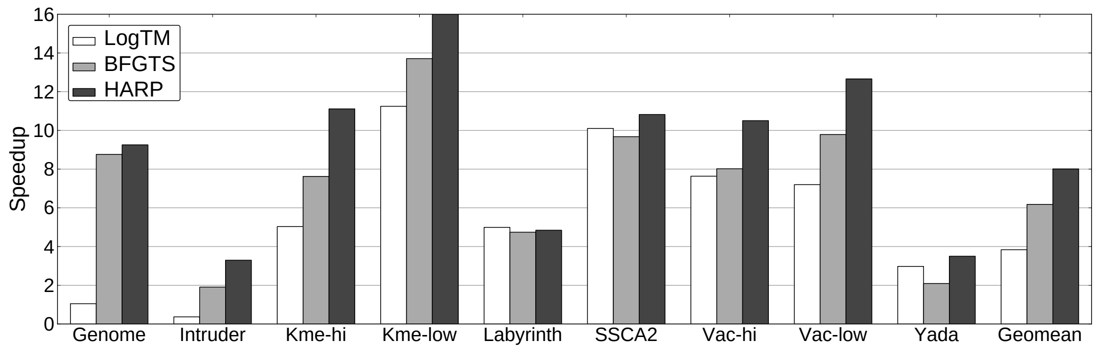


Figure 5.10: Speedup of 16-threaded executions compared to sequential execution.

amount of the execution time in workloads with small transactions like *Intruder* (27%) and *KMeans-High* (11%). This is because the time spent in the routine, which is used to adjust confidences of conflict, is constant and cannot be amortised when executing short transactions. Hence, in general, workloads with small transactions are penalised using BFGTS. However, HARP use of conflict lists results in a small, fixed maintenance cost that does not depend on workload characteristics. Having a better transactional scheduling policy and fewer aborts can also reduce non-transactional and barrier time. By executing only those transactions that are likely to commit, interactions with non-transactional code are minimised, e.g., the number of stalls when trying to access transactionally modified data is reduced. In addition, fewer aborts can reduce overall load imbalance, as it happens in textitVacation.

Regarding higher prediction accuracy, HARP offers promisingly low abort rates (see Table 5.4), obtaining near-linear speedup in *KMeans-Low*. Moreover, these improvements in abort rate are not due to overserializing transactions; as our efficiency ratio demonstrates, HARP is 1.27× more efficient than BFGTS in terms of useful computational cycles. This indicates that the conflict lists and the dynamically adaptable decay quickly adjust the confidences of conflict in accordance with actual contention levels that are present at any given time. In fact, in workloads like *KMeans* and *Yada* where contention varies with time, the decay allows to optimistically execute transactions faster when necessary – e.g., in *Yada* BFGTS overserializes transactions that could run in parallel (note the large stall time), but HARP decay logic detects this fact, allowing parallel execution while maintaining a lower abort rate.

Benchmark	**Abort Rate (%)**			**Efficiency Ratio**	
	LogTM	BFGTS	HARP	BFGTS	HARP
Genome	65.3	3.6	3.7	0.64	0.65
Intruder	70.2	14.6	7.3	0.12	0.17
KMeans-H	23.9	9.9	5.3	0.20	0.34
KMeans-L	13.0	3.9	0.5	0.39	0.89
Labyrinth	15.5	7.8	12.7	0.35	0.36
SSCA2	0.0	0.0	0.0	0.83	1.00
Vacation-H	11.6	7.0	2.4	0.79	0.79
Vacation-L	10.0	3.2	1.2	0.87	0.89
Yada	56.8	6.6	5.0	0.13	0.18
Geomean	**11.3**	**3.3**	**1.9**	**0.38**	**0.48**

Table 5.4: Benchmark statistics for evaluated systems.

Four threads per core

We execute the benchmarks with 64 threads, pinning 4 threads to each core. Both BFGTS and HARP present similar execution time breakdowns for all the benchmarks when compared to their 16-threaded executions, as can be seen in Figure 5.11. HARP attains an average speedup of 25.8% over BFGTS due to no software runtime overheads and less serialisation (stall and yield time) as a result of better predictions, with average abort rates of 4.1% for BFGTS and 2.8% for HARP.

However, an interesting point is to determine if such an overcommitted system is beneficial by comparing these workloads to their 16 threaded counterparts. Workloads with few transactions are not likely to benefit from an overcommitted system. This is the case of *Vacation*, which only has one transaction defined in the code, hence less room for improvement when switching to a different thread. Also workloads like *SSCA2* and *KMeans-Low* where contention is minimal cannot scale further, and the overheads of managing additional threads can hurt scalability – e.g., in *SSCA2* there is a significant loss of scalability from 10× to 3.5× (see Figure 5.12).

Yada exhibits significant benefits for all the evaluated systems when using 64 threads, as Figure 5.12 shows. *Yada* has the largest number of transactions (six). Moreover, its transactions are large with moderate contention. With these characteristics it is easier to

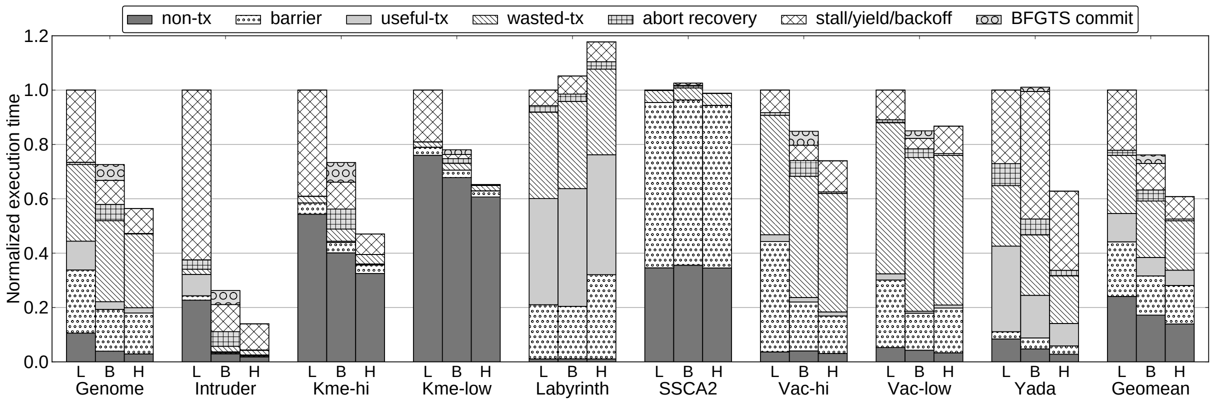


Figure 5.11: Normalised execution time breakdown for 64 threads in single-application workloads. **L** – LogTM; **B** – BFGTS; **H** – HARP

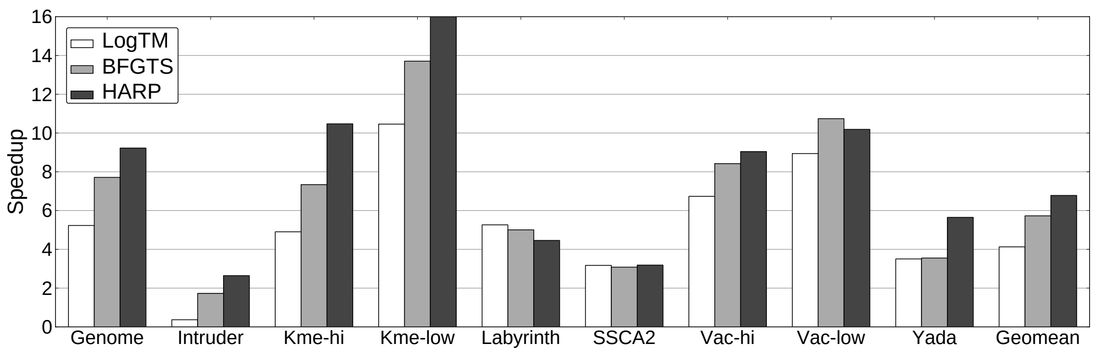


Figure 5.12: Speedup of 64-threaded executions compared to sequential execution.

Benchmark	**Abort Rate (%)**			**Efficiency Ratio**	
	LogTM	BFGTS	HARP	BFGTS	HARP
Genome	19.0	3.8	3.5	0.62	0.71
Intruder	70.1	14.4	8.7	0.11	0.14
KMeans-H	23.4	9.6	5.2	0.19	0.30
KMeans-L	15.2	3.4	0.5	0.40	0.88
Labyrinth	13.9	11.8	13.0	0.38	0.37
SSCA2	0.0	0.0	0.0	0.82	1.00
Vacation-H	9.0	7.3	6.2	0.77	0.76
Vacation-L	6.3	2.8	3.0	0.88	0.82
Yada	57.3	36.8	25.2	0.24	0.31
Geomean	**9.4**	**4.1**	**2.8**	**0.40**	**0.50**

Table 5.5: Benchmark statistics for evaluated systems.

find additional parallelism when switching between different threads, because the chances of executing a non-conflicting transaction are higher. In addition, large transactions help amortise yield time costs. *Yada* is the only benchmark that significantly improves its efficiency ratio when using 64 threads (see Table 5.5), from 0.18 to 0.31 for HARP. Our results suggest that large transactional codes, with medium or large transactions, may be necessary to benefit from overcommitted setups. This is likely to become a common case as more transactional applications become available.

5.5.5 Multi-Application Results

In this setup, each core executes two threads from different applications. This scenario will be increasingly common in the future as parallel programming becomes pervasive. As in setups evaluated earlier, we use STAMP but only consider the '-High' versions of *KMeans* and *Vacation*, and evaluate all the possible combinations of 2 applications out of the 7 possible, which amounts to 21 different workloads. The workloads are named with the initials of each application, the legend is in Table 5.2 – e.g., the workload 'GL' executes *Genome* and *Labyrinth*. To make accurate measurements, we synchronise the two applications at the beginning of their parallel sections. When an application reaches the

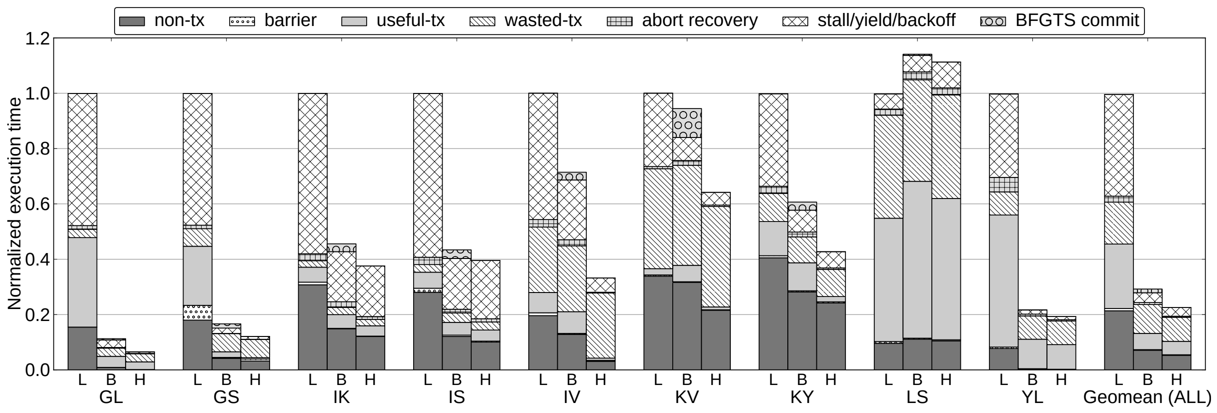


Figure 5.13: Normalised execution time breakdown for multi-application workloads. **L** – LogTM; **B** – BFGTS; **H** – HARP

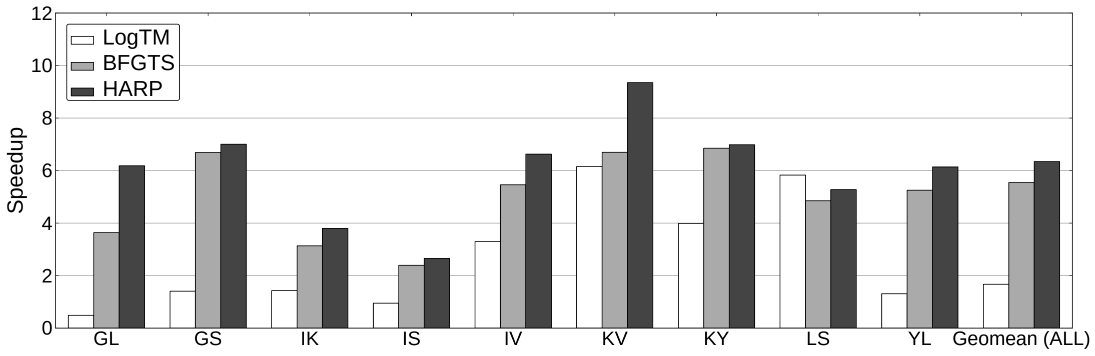


Figure 5.14: Speedup compared to single core execution.

end of its parallel section, that application is no longer considered for execution. Similarly, when a core finishes all of its threads (applications), that core is considered to be available for other tasks, and hence does not contribute to the execution time. To measure scalability, the slowest core is considered.

Figure 5.13 shows the execution time breakdown and Figure 5.14 the scalability results. We show a representative selection of 9 workloads, plus the geometric mean which considers the 21 evaluated workloads. LogTM fails to deliver good performance, experiencing a large number of aborts and high backoff overheads. Thus, policies that cannot dynamically decide what is the best course of action are not suitable for future systems where parallel applications might be dominant. However, BFGTS and HARP deliver higher performance because they can swap potentially wasted computation for potentially useful work.

HARP performs better than BFGTS for all the evaluated workloads, achieving a 29.5%

Benchmark	**Abort Rate (%)**			**Efficiency Ratio**	
	LogTM	BFGTS	HARP	BFGTS	HARP
GL	90.1	32.4	3.7	0.28	0.46
GS	34.8	2.9	1.1	0.54	0.81
IK	46.9	21.4	15.2	0.09	0.09
IS	43.2	17.9	14.7	0.11	0.10
IV	37.6	25.6	3.1	0.40	0.79
KV	17.1	11.6	2.8	0.57	0.86
KY	23.2	8.3	4.4	0.29	0.54
LS	0.0	0.0	0.0	0.36	0.37
YL	94.2	41.5	3.1	0.39	0.45
Geomean (ALL)	**24.1**	**7.3**	**3.3**	**0.38**	**0.47**

Table 5.6: Benchmark statistics for evaluated systems.

improvement on average. This is due to four main reasons. First, BFGTS is overly pessimistic in general, leading to a larger serialisation time (stall and yield). We observe a notably larger number of predicted conflicts in *GL*, *GS*, *KV*, *KY*, and *IV*; in the latter BFGTS predicts 4× more conflicts. Second, HARP makes better predictions than BFGTS; as Table 5.6 indicates, even though HARP predicts a lower number of conflicts, it still attains remarkably better abort rates. Hence, HARP allows for increased parallel execution of transactions while keeping lower abort rates. Third, BFGTS decides whether to stall or yield depending on the number of cache lines touched by the transaction, which we find is less accurate than HARP's approach that uses actual execution time. Finally, as observed before, small transactions (*Intruder* and *KMeans*) penalise BFGTS performance by increasing the software commit routine time.

Labyrinth and *Intruder* have lower scalability and significantly larger execution time than *KMeans* and *SSCA2*. Hence, scalability for *IK*, *IS*, and *LS* tends to be close to that seen in *Labyrinth* and *Intruder* for single-application (Figure 5.10). However, for combinations where the execution time is more evenly distributed, like *IV* and *KY*, we can observe how scalability is significantly higher than the one reported for *Intruder* and *Yada* respectively. *YL* achieves 6.1× speedup, higher than both *Yada* and *Labyrinth* when executed as single applications.

5.5.6 Sensitivity Analysis

System parameters

We evaluate our technique changing two major system parameters. First, we modified the size of HARP hardware structures to have no collisions (i.e., two different TxID's mapping to the same entry) for the multi-application setup, since for single-application no collisions were found. Our results with no collisions did not show any significant changes in the abort rates of the affected multi-application workloads. This is because very few collisions were present in the first place, one in *GS* and one in *GY*.

Second, we looked into conflict lists size sensitivity. Throughout our evaluation, we have used conflict lists of size 2. We evaluate single-application workloads with conflict lists of size 1 and 4. Low contention applications like *SSCA2* are not affected by the conflict lists size, due to their low conflict rates. High contention applications like *Labyrinth*, *Yada*, and *Intruder* did not experience significant variation either due to a single dominant conflicting address, as shown in Figure 5.3. However, '-High' versions of *KMeans* and *Vacation* present moderate contention and show a significant drop in performance when using conflict lists of size 1. This is because they have a larger set of conflicting addresses, with no dominant address, which makes HARP schedule too optimistically. Overall, we find that conflict lists of size 2 offer the best trade-off between performance and hardware cost.

Communication and prediction overheads

We expect uncontended scenarios demanding high commit throughput to expose communication and prediction overheads. We repeat the experiment from Section 5.3, adding HARP and a version of HARP that stores and maintains the THT and CLT structures in software (HARP-SW). Eigenbench [42] is configured to have no contention and to maximise total transactional execution time. Figure 5.15 shows our evaluation on a range of transactional sizes, smaller transactions provide higher commit rates. The smallest transaction size evaluated performs one read operation and a small amount of work with the read data. Under such conditions LogTM attains almost perfect speedup since the workload is fully parallel. HARP experiences a 7% slowdown for the smallest transaction size, due to communication and prediction latencies not being amortised. However, HARP rapidly closes the gap in performance with respect to LogTM, confirming that broadcast messages

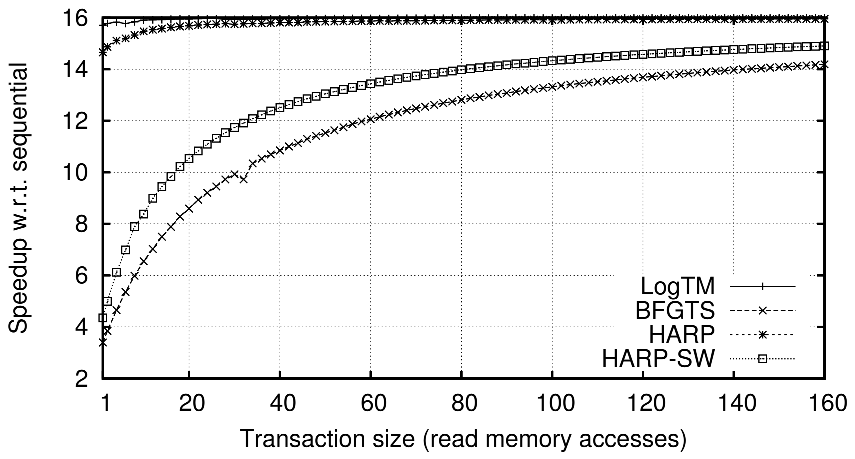


Figure 5.15: Communication and prediction overheads of evaluated systems at different commit rates. Using Eigenbench with varying transaction sizes, 128K iterations and 16 cores.

do not hinder scalability. In contrast, both HARP-SW and BFGTS have a severe performance drop, mainly due to additional code executed at commit time, which can make executed transaction several times larger. HARP-SW remains slightly better than BFGTS because its software operations are simpler.

Multi-application using four applications

We also evaluate a multi-application setup using four applications concurrently, which amounts to 35 different workloads. HARP again outperforms BFGTS by 20.3% on average, and attains scalability similar to that seen in the two application setup, 6.5×. In this scenario collisions did not affect performance either.

5.6 Summary

In spite of much research, HTM performance is susceptible to degradation when contention is present. Moreover, parallel programming is becoming the norm, and systems with several parallel applications will be increasingly common. Techniques that minimise the amount of wasted work due to misspeculation and maximise computational resource utilisation are necessary for TM to gain wide acceptance.

This work proposed HARP, a hardware mechanism that efficiently predicts future conflicts and avoids speculation when the probability of contention is high. The resources thus freed are, when it is deemed advantageous, utilised to schedule possibly non-conflicting codes, thereby improving concurrency and throughput. The design provides seamless support for both single-application and multi-application scenarios. Our investigation has shown that HARP outperforms, by a substantial margin, both LogTM, a popular HTM proposal, and BFGTS, the state-of-the-art proactive transaction scheduling scheme prior to this work. This is achieved with modest hardware support comprising three simple tagless structures in each core. Since HARP does not rely on software runtimes and data structures, it presents little management overhead, while simultaneously keeping the architecture relatively independent of the software that runs on it. In addition, HARP predictions can be leveraged to implement aggressive power saving schemes when no useful computation can be scheduled. We see this area as a potential direction for future work.

6 Techniques to Improve Performance in Requester-wins HTM

6.1 Introduction

There is an exigent need for high-productivity approaches that allow control of concurrent accesses to data in shared memory multithreaded applications without severe performance penalties. This has led researchers to look seriously at the concept of transactional memory (TM) [39, 40]. TM allows the programmer to demarcate sections of code – called transactions – which must be executed atomically and in isolation from other concurrent threads in the system. The TM system detects and resolves conflicts, i.e. circumstances when two or more transactions access the same shared data and at least one modifies it.

Although Transactional Memory has been an active research topic for almost a decade [3, 13, 37, 44, 52, 78, 90], bare-bones support for hardware transactional memory (HTM) is only just appearing. Large-scale adoption of software-only approaches has been hindered for long by severe performance penalties arising out of the need for extensive instrumentation and book-keeping to track transactional accesses and detect conflicts without hardware support. Intel TSX extensions and IBM BlueGene-Q are now testing the waters with hardware TM [43, 87]. Restricted Transactional Memory (RTM) as described in Intel TSX specifications appears to be a requester-wins HTM where transactions abort if a conflicting remote access is seen while executing a transaction. Transactions may also abort when hardware resource limitations (e.g. cache capacity) or exceptional hardware events (interrupts) are encountered. In this study we are primarily concerned with the nature of the "requester-wins" conflict resolution policy and not with conditions arising out of lack of hardware resources or exceptions. The authors do not have access to implementation details of Intel RTM and, thus, the results presented must be seen in the more general context of requester-wins HTM designs.

Requester-wins HTMs are easy to incorporate in existing chip multiprocessors [21, 24]. Conflict detection and resolution mechanisms in such systems do not require any global communication except that which naturally arises from the need to impose cache coherence. Each core tracks accesses made by transactions that run locally. This could be done using cache line annotations indicating lines that have been read or written. Some implementations may choose to employ read/write set bloom filters for the purpose. Either way, the requester-wins policy has no inherent forward progress guarantees since a local transaction aborts whenever it receives a conflicting coherence request for a line in its read or write sets. This susceptibility to livelock is well-known [14]. However, the likelihood of livelocks in such systems and their eventual impact on performance has not been investigated in depth. Livelocks may persist for a while but eventually get broken due to varying delays in real-world systems. When this occurs they may manifest themselves as degradation in application execution times or system throughput.

Figure 6.1a shows how two transactions may livelock. Both transactions read data that is eventually written by the other. Executions of the two transactions may interleave such that no progress is made at either thread. However, cyclic dependencies between concurrent transactions are not the only sources of livelock. A potentially more pathological livelock behaviour exists – Figure 6.1b – where multiple read-modify-write transactions may continually abort each other (i.e., friendlyfire [14]). The livelock occurs because the aborted transaction issues a conflicting access upon re-execution which then aborts the transaction that was allowed to proceed.

The aim of this study is to show that protocols that merely guarantee livelock free-

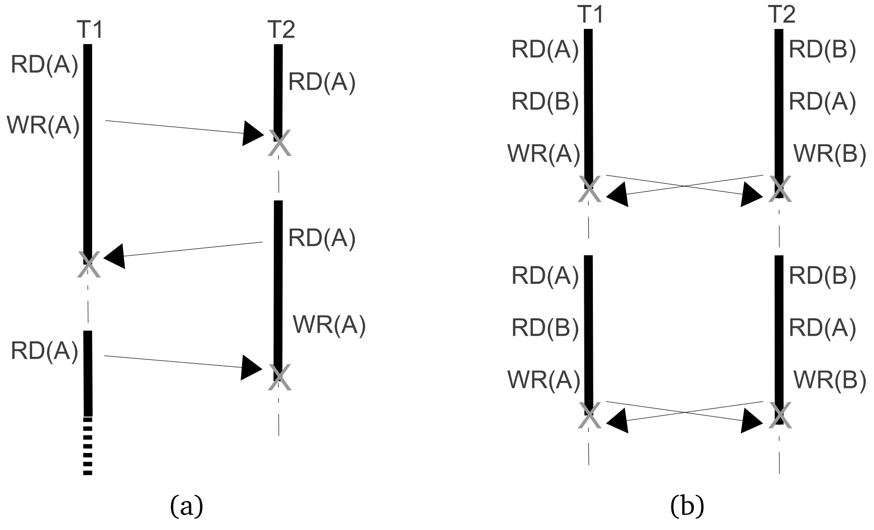


Figure 6.1: Two livelock scenarios in requester-wins HTM

dom may not be the most efficient. The performance impact of livelock mitigation and avoidance techniques should be looked at in more depth. HTM systems should incorporate a set of such techniques in a manner which allows resolution of these livelock conditions as soon as possible and with the lowest associated performance cost. This chapter investigates performance implications of a number of existing strategies like exponential backoff [57], serial irrevocability as implemented in GCC `libitm` since version 4.7.0, and hourglass [50]. Our study shows that there is a substantial cost in terms of performance imposed by these strategies. With an aim to minimise this cost, we propose some novel techniques, in hardware and software, which are well suited to requester-wins HTM designs. Four new techniques for mitigation of livelocks are presented – two are implemented in software, requiring only simple interfaces for reading information provided by the hardware; and two that are implemented in hardware with simple core-local additions.

Our analysis of relative merits of these proposed techniques shows that deficiencies of requester-wins HTMs can be ameliorated effectively for a variety of transactional workloads. One of our aims is to make system programmers using HTM aware of the severity of livelocks and the performance cost imposed by various mitigation and avoidance techniques. This would help them decide what mitigation techniques to choose. This chapter also aims to convey to processor architects the importance of simple hardware mechanisms to mitigate the impact of livelocks. In summary, it sheds light on the following concerns:

- How severe can livelocks be in requester-wins HTMs?
- What are the performance costs associated with existing livelock mitigation techniques?
- Can new techniques (hardware-only or hybrid) be designed to reduce performance degradation while retaining the simplicity of requester-wins HTM?

The rest of this chapter is organised as follows. Section 6.2 shows that livelocks frequently block forward-progress in several transactional workloads running on requester-wins HTMs. It also shows that existing livelock mitigation and avoidance strategies (back-off, serial irrevocability and hourglass) leave a large performance gap between observed performance and performance achievable by a livelock-free HTM. In Section 6.3 we describe in detail four new techniques to improve performance by mitigating livelock conditions. Section 6.4 introduces our experimental methodology, and Section 6.5 provides experimental evidence that highlights the efficacy of our new techniques. In Section 6.6 we discuss about related work. Finally, in Section 6.7 we conclude with final thoughts and a summary of insights gathered from this study.

6.2 Motivation

Our experiments with a variety of workloads – which include the STAMP benchmark suite [18], *water* and *radiosity* from SPLASH2 [89] and two microbenchmarks (*deque* and *btree*) – show that most of them consistently livelock when running on requester-wins HTM without any livelock mitigation strategy. Data in Table 6.1 lists the workloads and their susceptibility to livelock on a variety of scenarios. The results have been gathered on a simulated 8-core machine. A suffix '-h' after the workload name indicates high contention parameters have been used. A suffix '+' indicates larger datasets. These livelocks occur due to two or more concurrent threads entering a pattern of continuous aborts, eventually preventing any forward progress as other threads either wait perpetually at a barrier or enter livelock themselves. We executed each application multiple times with randomised delays added to the main memory access latency (± 5%), so as to create different thread interleavings.

Application	Live-lock	Operation where livelock occurs	Application	Live-lock	Operation where livelock occurs
Deque	Yes	Deque access	Water	Yes	Counter increment
Btree	Yes	Btree insertion	Radiosity	Yes	Counter increment
Genome	Yes	Hashtable insertion	Genome+	Yes	Hashtable insertion
Intruder	Yes	Task queue access	Intruder+	Yes	Task queue access
KMeans-h	Yes	Matrix access	KMeans-h+	Yes	Matrix access
Labyrinth	No	–	Labyrinth+	Yes	Vector access
SSCA2	Yes	Vector access	SSCA2+	Yes	Vector access
Vacation-h	Yes	Counter increment	Vacation-h+	Yes	Counter increment
Yada	Yes	Heap removal	Yada+	Yes	Heap removal

Table 6.1: Livelocks in applications.

We have also identified the kind of operation that triggers a livelock condition for each workload. The results in Table 6.1 indicate that livelocks are a serious problem for a variety of common operations in different data structures. Without appropriate mitigation strategies, in software or hardware, the use of transactions in such a system may be rendered impractical. This leads us to the next question we attempt to answer: what are the costs of various existing livelock mitigation strategies?

6.2.1 A Look at Existing Techniques

We now briefly describe existing software techniques for livelock mitigation and avoidance that can improve overall performance in requester-wins HTMs. We will concentrate on three techniques: *exponential backoff*, introduced as an HTM contention manager in LogTM [57]; *serial irrevocability*, previously used in software TM proposals [34, 88] and now also used in GCC as the default fallback mechanism upon repeated aborts; and *hourglass*, which provides a more relaxed form of serialisation than serial irrevocability [50].

Serial Irrevocability

This is a fallback mode in case a hardware transaction fails to commit after retrying several times. This mode could be chosen because of contention or hardware resource limitations.

ALGORITHM 6.1: Simplified begin and commit transaction function wrappers to implement serial irrevocability.

```
void beginTransaction()
  while true do
      TX_BEGIN(offset to fallback path);                        /* ISA begin instruction */
      if serialLockCanRead() == false then                 /* adds serial lock to the read set */
          abortTransaction();       /* there is another thread in irrevocable mode */
      else
          return;                                                 /* execute transaction */
      end
      fallback_path:                                            /* fallback path on abort */
      if retryCount < MAX_RETRIES then
          retryCount++;
          if serialLockCanRead() == false then
              waitForSerialLockCanRead();                 /* wait for irrevocable thread */
          end
                                                    /* retry transactional execution */
      else
          break;                                            /* use serial irrevocable mode */
      end
  end
  acquireSerialLockWriter();                              /* aborts other transaction */
  return;                                        /* execute in serial irrevocable mode */

void commitTransaction()
  if serialLockCanRead( ) == false then
      releaseSerialLockWriter();                  /* this was an irrevocable execution */
  else
      TX_COMMIT();                                              /* ISA commit instruction */
  end
  return;                                          /* successfully executed transaction */
```

The number of retries before entering this mode is usually configurable. The mode works by aborting any concurrent transactions in the system through the acquisition of a global multiple-reader-single-writer lock as a writer. This also ensures that no other transaction in the application can begin execution, allowing the irrevocable transaction to be executed without interference from other threads. The algorithm for starting and committing transactions is shown in Algorithm 6.1. The call to `beginTransaction()` returns success after either starting the transaction in serial irrevocable mode or in the usual hardware-supported TM mode. On a transactional abort the architectural state is restored and execution is resumed from the fallback code path. The `commitTransaction()` routine ensures that the transaction releases the serial lock if it was running irrevocably. Otherwise, it will execute the supported ISA instruction to commit a transaction. This implementation resembles the one that can be found in the new *libitm* library in GCC to provide TM support.

Randomised Exponential Backoff

Exponential backoff has been used in other domains as a collision avoidance strategy wherein backoff duration is chosen randomly from a range of durations that grows exponentially larger as the number of failures increases. Backoff has the potential to reduce chances of repetitive conflicting patterns that occur. However, it does not guarantee forward progress. Exponential backoff has been evaluated in the context of contention management options available in software transactional memory [73]. However, even though backoff strategies have also been evaluated in HTM designs [14, 57], their impact on performance in HTM systems as prone to livelock as requester-wins remains unclear.

Hourglass Contention Manager

Liu and Spear [50] define toxic transactions as those that have aborted consecutively a number of times due to conflicts. To deal with these toxic transactions they propose the *hourglass* contention manager, where such transactions try to grab a global token, preventing new or aborted transactions from starting. This gives the toxic transaction a better chance of committing after acquiring the token, although it is not guaranteed to commit. This mechanism is less drastic than serial irrevocability as it allows transactions that are already running in the system to proceed when the token is acquired.

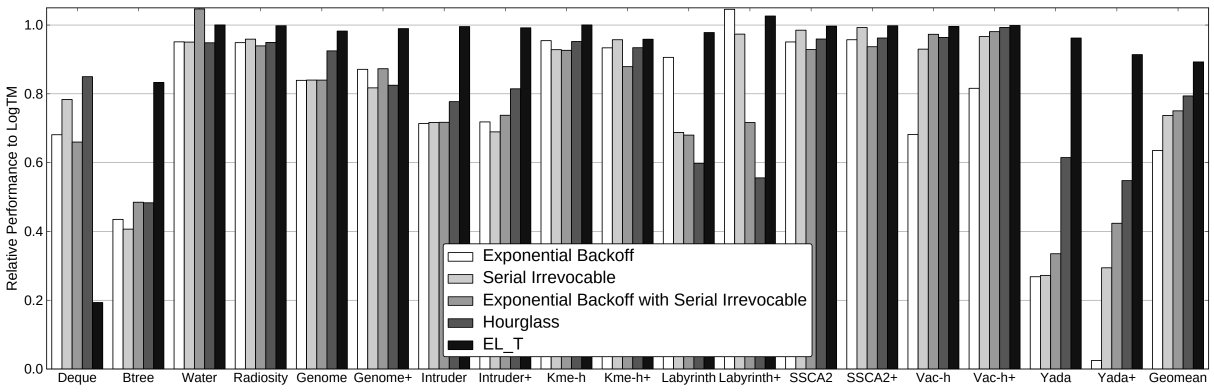


Figure 6.2: Performance of existing livelock mitigation techniques relative to LogTM

6.2.2 Brief Analysis of Existing Techniques

We now show how these existing techniques perform on a requester-wins HTM with respect to a well-known reference like LogTM [57]. This will allow us to estimate the gap in performance between such a requester-wins system and a proposal that implements a more complex strategy. LogTM is a requester-stalls design that uses a scheme for conservative deadlock avoidance. It introduces a timestamp in all coherence messages (thereby prioritising older transactions in the system) and extends the coherence protocol with support for *nacks* (negative acknowledgements) that allow transactions to be stalled upon conflict instead of aborting them. The conservative deadlock avoidance scheme works by keeping a *possible cycle* bit in each core, set when a request with an older timestamp is *nacked*. Once this bit is set, the transaction must abort as soon as it receives a *nack* response with an older timestamp.

We have also included a second HTM design point, a lazy-versioning eager-resolution HTM based on the EL_T design described Bobba *et. al* [14]. It uses the L1 caches to buffer speculative updates and resolves conflicts eagerly using timestamp priorities attached to coherence messages – aborting if the remote transaction has higher priority or responding with a *nack* (negative acknowledgement) otherwise. Note that like LogTM, EL_T also requires protocol support for *nacks* and a mechanism to assign priorities to transactions. Thus, these systems turn out to be more complex than requester-wins designs, where the coherence protocol is not modified.

Figure 6.2 shows relative performance normalised to LogTM (higher is better), for *exponential backoff*, *serial irrevocability*, a scenario where these two techniques are combined, and *hourglass*. The relative performance of the *EL_T* design is also shown. For reasons mentioned earlier, the LogTM design is more complex than simpler requester-wins HTMs that will soon be widely available, and thus should be considered as an upper bound on performance achievable by requester-wins designs. As seen in Figure 6.2, the EL_T HTM design, which has been included here primarily to add another relevant HTM design point for comparison, performs well under most workloads due to its ability to prioritise transactions, but presents significant degradation in performance under high contention. Overall, it turns out to be around 12% worse than LogTM. Among software techniques, exponential backoff performs 40% worse than the baseline, being inefficient even under mild contention. Serial irrevocability is a good choice for uncontended applications like *SSCA2*. However, when contention is present its performance drops significantly. Overall we see that the performance offered by these two techniques and their combination is on average 27% worse than LogTM for this set of applications. On the other hand, the hourglass contention manager fares much better, particularly when contention is present by reducing serialisation overheads. Overall, it achieves 20% less performance than LogTM. In general, we observe that under high contention there is a marked susceptibility to a much greater degradation in performance. In our opinion, this observed performance gap is large enough to merit a search for solutions to close it. Our solutions presented in the following section, therefore, attempt to do so while retaining the simplicity of requester-wins HTMs.

6.3 Proposed Techniques to Improve Requester-wins HTM Designs

In this section we will describe four novel techniques to improve performance in requester-wins designs by mitigating pathological scenarios like the ones shown in Figure 6.1. These techniques attempt to bridge the gap in performance highlighted in the previous section. We first introduce two software-based techniques that are simple to implement in upcoming HTM designs. Later we look at two additional techniques that require simple core-local hardware support, but retain the requester-wins nature of the HTM.

ALGORITHM 6.2: Simplified begin and commit transaction function wrappers to implement serialise on conflicting address (SoA).

```
void beginTransaction()
  while true do
      TX_BEGIN(offset to fallback path);                    /* ISA begin instruction */
      return;                                              /* execute transaction */
      fallback_path:                                      /* fallback path on abort */
      if thread->has_lock is valid then                  /* already holding one lock */
          releaseAddressLockWrite(thread->has_lock);                 /* avoid cycle */
          thread->has_lock = invalid
      end
      address = getConflictingAddress();          /* hardware provides the address */
      index = hash(address)
      if address is invalid then
          continue;                    /* abort not related to a data conflict, retry */
      end
      acquireAddressLockWrite(index);       /* try to grab lock related to address */
      thread->has_lock = index
                                                                          /* retry */
  end

void commitTransaction()
  TX_COMMIT();                                             /* ISA commit instruction */
  if thread->has_lock is valid then                 /* executed with an acquired lock */
      releaseAddressLockWrite(thread->has_lock);  /* release, others can proceed */
      thread->has_lock = invalid
  end
  return;                                    /* successfully executed transaction */
```

6.3.1 Serialise on Conflicting Address (SoA)

It has been shown in prior work [61, 83] that conflicts are usually caused by a small number of contended addresses with large fractions of data accessed by typical transactions seeing no contention at all. Thus, a large number of transactions in code are prone to see conflicts only on a few addresses. Moreover, it is not very complicated in hardware to determine this address when a conflict occurs, since, in requester-wins HTM designs cores abort when they receive coherence requests that carry the address. This information could be passed on to the runtime through interfaces similar to the ones already implemented in production devices. For example, Intel TSX supplies information about the nature of aborts through the *EAX* register, among other things.

Our approach utilises the additional bits from the *RAX* register to feed the address of the conflicting cache line onto the runtime. Using this additional information, the runtime is able to identify potential hotspots of contended cache lines and rely on locks to execute one transaction after another, with relatively few transactions requiring a fallback to the more drastic form of serialisation enforced through serial irrevocability. Algorithm 6.2 shows the necessary steps to implement this proposal. Note that for the sake of clarity, in this algorithm we do not include the necessary checks to have serial irrevocability (described in Section 6.2.1).

The approach works by trying to acquire a lock from an array of locks using a hashed version of the conflicting address as index. If another thread has already acquired the lock for that address, the current thread waits. This approach allows threads which are likely not to contend with each other to proceed, while threads that conflict on the same addresses serialise. We only allow each thread to acquire a single lock to avoid cyclic dependencies. Therefore, the number of locks concurrently in use is small, lower or equal than the number of executing threads. This approach is able to deal quite effectively with livelock scenarios produced by common read-modify-write transactions, similar to the one shown in Figure 6.1b. However, the scenario in Figure 6.1a would still require serial irrevocability to ensure forward progress.

6.3.2 Serialise on Killer Transaction (SoK)

Our second proposal is a software technique that stalls restarted transactions until the offending transaction (i.e. the transaction whose request caused the abort) completes.

ALGORITHM 6.3: Simplified begin and commit transaction function wrappers to implement serialise on killer transaction (SoK).

```
void beginTransaction()
  acquireLockArrayRead(my_id);     /* acquire local lock for reading, blocks writers */
  while true do
      TX_BEGIN(offset to fallback path);                        /* ISA begin instruction */
      return;                                                  /* execute transaction */
      fallback_path:                                          /* fallback path on abort */
      killer_id = getKillerID();           /* hardware provides conflicting thread id */
      if killer_id is invalid then
          continue;                      /* abort not related to a data conflict, retry */
      end
      clearedForDeadlock = false;        /* indicates if has been cleared for deadlock */
      while !lockArrayCanWrite(killer_id) do          /* wait until killer thread is done */
          if !clearedForDeadlock then                     /* ensure we will not deadlock */
              acquireGlobalLock();         /* check a vector of adjacencies atomically */
              if isCyclePossible(killer_id, my_id) then      /* detects cycles, defined below */
                  releaseGlobalLock();                    /* cannot wait, would deadlock */
                  break;                                                      /* retry */
              else
                  killers_vector[my_id] = killer_id;           /* will wait, update vector */
                  clearedForDeadlock = true;        /* do not do the deadlock check again */
              end
              releaseGlobalLock();                                /* deadlock check done */
          end
      end
      acquireGlobalLock();
      killers_vector[my_id] = -1;              /* my killer has finished, update vector */
      releaseGlobalLock();
                                                                              /* retry */
  end

void commitTransaction()
  TX_COMMIT();                                                 /* ISA commit instruction */
  releaseLockArrayRead(my_id);                      /* release racers waiting on the lock */
  return;                                          /* successfully executed transaction */

bool isCyclePossible(int killer_id, int my_id)
  if killers_vector[killer_id] == -1 then return false ;        /* killer not waiting, no cycle */
  if killers_vector[killer_id] == my_id then return true ;      /* killer waiting for me, cycle */
  return isCyclePossible(killers_vector[killer_id], my_id);                /* recursive call */
```

As in the previous solution, this is a hardware-assisted software mechanism that requires the identity of the conflicting thread (a.k.a. *killer*) to be passed from the hardware to the runtime at the time of an abort. This scheme is of special interest in requester-wins systems because restarted transactions are likely to abort their killers when restarting.

Algorithm 6.3 shows how the idea is implemented. Before a transaction begins its execution, it reads a multiple-reader-single-writer lock from a vector of locks indexed by the thread identifier. This read operation stalls writers if they try to write to the lock. When a transaction aborts, before it is allowed to restart, it checks whether it has permissions to write to the killer's lock. Note that the killer only releases write permissions on the lock after it has committed the transaction. If it does not have permissions to write to the lock, then the killer is still executing the transaction and the aborted transaction must wait. Cyclic dependencies may arise causing deadlock. The approach avoids this by ensuring that the wait is deadlock free through a check for potential cycles using a vector (*killers_vector*) that maintains dependencies. Accesses to this vector are protected by a global lock. This guarantees that only one among a group of conflicting transactions is allowed to proceed. Since the lock on this structure serialises accesses to it, when cyclic dependencies exist the design resolves it by allowing the last transaction in a cyclic dependency chain to detect the condition and avoid a potential deadlock by not waiting on its killer. Other transactions in the now cycle-free dependency chain wait. This solution has the advantage of guaranteeing forward progress as long as transactions can execute in hardware, avoiding the use of the serial irrevocable mode in livelock scenarios.

Note that a potential corner case may arise in which a transaction is waiting for a transaction that is not its actual killer, e.g., a transaction (*Tx-a*) aborts and before checking whether it has to wait, the killer transaction finishes and a new transaction (*Tx-b*) starts execution. This is likely to be an uncommon scenario, and it does not pose any deadlock or starvation problems. Deadlocks cannot occur because aborted transactions wait on their killer's thread identifier, so when the new *Tx-b* finishes the aborted *Tx-a* will restart. Starvation problems have not been encountered, but could be easily solved by adding fairness to the lock implementation.

6.3.3 Delayed Requester-Wins (DRW)

Our first hardware-based design makes conflicting requests wait for a bounded length of time before applying the requester-wins policy. This technique can be implemented locally at the core without changing communication protocols or messaging. Basically, it attempts to capture the benefits of the requester-stall policy (i.e. resolving conflicts through stalls rather than aborts) while avoiding the complexity introduced by negative acknowledgements (*nacks*) in the coherence protocol. To this end, LogTM's protocol introduces *nacks* as well as a special kind of *unblock* message to inform the directory that a coherence transaction has failed due to conflicts and should be *cancelled*, i.e. the coherence state reverted to its original state with no updates to the bit-vector of sharers. As opposed to LogTM, coherence requests in our Delayed Requester-Wins (DRW) design always complete successfully – perhaps with some additional latency – and thus there is no need to extend the protocol with new messages. Delaying coherence messages has been explored in the past in the context of memory consistency for scalable shared-memory multiprocessors [35].

DRW allows the exclusive owner of a cache line to buffer conflicting requests and thus delay responses until a later point in time. On the requester's side, the cache miss that resulted in a conflict simply appears to be a longer latency miss, and the execution naturally stalls at this point until the memory reference completes. Delayed conflicting requests queued at the exclusive owner's cache are considered either when the transaction ends (commits or aborts) or when an associated timeout expires. DRW uses timeouts to conservatively break temporary deadlocks situations that may appear when transactions exhibit circular dependencies. Timeouts are a simple solution to break cycles and they can be implemented locally at the core level. On the other hand, LogTM's deadlock avoidance mechanism requires the addition a global timestamp (which all threads agree upon) to every coherence request and response, increasing the size of every network message that traverses the communication fabric.

Transactions with buffered conflicting requests are allowed to execute as long as they are able to make forward progress. When a transaction with buffered requests experiences an L1 cache miss, the timer is started. If the cache miss completes within the *timeout latency*, the timer is stopped since the transaction has made forward progress while buffering remote requests, which means that no cycle has been formed yet. The timer is thus reset to its initial value and will be started again in subsequent misses. Otherwise, if the timer expires while a local miss is still pending, the buffered conflicting requests begin to be serviced normally in a requester-wins fashion, triggering an abort and thus breaking any temporary cycle. If the transaction eventually commits, all conflicts are successfully resolved and the requests are serviced with the new committed data.

An important aspect in DRW is the *timeout latency*, i.e. the value at which the timer is started on a cache miss. Ideally, the timer should not expire unless a cyclic dependency (transient deadlock) has occurred, and similarly it should expire as soon as the cycle has been formed. In order to set the timer accurately, DRW keeps a table that associates a different timeout latency to each atomic block of code (indexed by the PC of the *begin-tx* instruction). The value used for each atomic block is adaptable, and it moves between a range of values, in our experiments, from 64 to 1024 cycles. Commits that successfully resolve conflicting requests by delaying the response do not update the value in the latency table. On commits without conflicts, the latency is halved in order to keep the reaction time to potential deadlocks short when contention is low. Upon timeout expiration (i.e. on abort), the latency is doubled. In this way, if conflicts are encountered again after the transaction restarts, a larger window of time is given to remote transactions so that they have a better opportunity to reach commit (i.e. service the buffered conflicting requests) before the local offending transaction aborts due to the timeout.

6.3.4 WriteBurst: Buffering of Store Misses (WB)

Buffering transactional stores has been shown to be beneficial in both eager and lazy systems [59, 84]. In the case of a requester-wins HTM, the ability to delay completion of possibly conflicting transactional stores until close to commit time and then releasing them into the coherent cache hierarchy in a burst can improve parallelism by reducing the window of time in which a transactional write to a line may see a conflict. Remote readers can now access lines in a non-conflicting manner and writers that are close to commit have a better chance of acquiring ownership over the write-set before being aborted by a remote reader. If resources are sufficient to buffer all store misses until commit, this technique allows for a form of lazy conflict detection (*committer-wins*) [37], which provides stronger forward progress guarantees and can enhance concurrency by allowing readers to commit before a conflicting writer [77, 82, 85]. However, unlike the latter lazy systems, in our scheme there is no notion of *committer*, because it is always possible for a transaction to abort after it has reached the commit instruction while it is draining its buffered store misses; but since transactions generally write only a few cache lines, the time required to drain the buffers is generally short and the requirements to buffer all store misses are not excessive.

Our model implements the idea by utilising the L1 miss status holding registers (MSHRs) to buffer store misses. Stores to exclusively owned lines are store hits and thus can complete as usual in cache. Our scheme is applied upon stores to shared lines – *upgrade* misses – which result in a message sent to the directory requesting the invalidation of all other privately cached copies. Lines that are absent in the L1 are prefetched non-exclusively if targeted by a speculative store (the L1 cache uses a write-allocate policy). Once the line is allocated in L1, the store is buffered in the MSHR and henceforth treated as an upgrade miss.

Our design leverages the L1 cache entry itself to keep the speculative updates, and the request is buffered in the MSHR. Since the data is present in the L1 cache in shared (S) state, only a minor behavioural change in the cache controller is needed to allow speculative stores that target S state lines to update the cache entry before write permissions are actually acquired. Per-byte dirty bits in cache to track dirty words are not needed since no merging with other versions occurs. An MSHR is allocated for the upgrade miss and the SM bit is set for the entry, but the request for ownership is not immediately sent to the L2. The issue of these upgrade messages to the L2 directory is deferred until (a) the transaction reaches the commit instruction, or (b) all MSHRs are in use. The MSHR keeps track of such entries by maintaining a special *Buffered* bit. Subsequent local loads to lines with buffered MSHR entries simply obtain the data from the cache and add the line to the read set (e.g. set the *speculatively read* bit), as usual. Remote load misses get the non-speculative version from the L2 cache, since the directory remains unaware of the speculative writes at the private cache. If an invalidation is received for a line with a buffered MSHR, then the transaction is aborted and all buffered MSHRs are discarded.

For applications with large write sets, the number of MSHRs is likely to be insufficient to buffer all store misses. When a new store miss finds all MSHR entries occupied, the design triggers a draining process which sequentially issues buffered upgrade request for all entries. To prevent drained speculative writes that have completed in cache to repeatedly expose the transaction against conflicts with restarted readers, our design incorporates a simple Bloom filter [11] called *conflict set signature*. This filter is used to conservatively

		Proposal Name	Abbr.	Livelock-free?	Requires hardware?	Hardware Description
Existing		Exponential Backoff	B	No*	No	—
		Serial Irrevocability	S	Yes	No	—
		Hourglass	H	Yes	No	—
		EL_T	T	Yes	Yes	timestamps and nacks in coherence
		LogTM	L	Yes	Yes	timestamps and nacks in coherence, possible cycle detection, priority for older writers
Proposed	SW	Serialise on Address	SoA	No*	Minor	provide conflicting address to runtime
		Serialise on Killer	SoK	Yes	Minor	provide killer id to runtime
	HW	Delayed Req-wins	DRW	No*	Yes (local)	timeout counters, buffer for requests
		WriteBurst	WB	No*	Yes (local)	L1 MSHR buffered bit + logic, conflict set sig.

* Serial irrevocability, Serialise on Killer, or Hourglass must be employed to guarantee forward progress.

Table 6.2: Overview of techniques and their characteristics.

encode write-set addresses that have seen conflicts with remote transactions. Note that only store hits or drained misses from the MSHRs are added to the write set of the transaction (i.e. set the SM bit in cache). Every time a transaction aborts due to a conflict on a write-set address, the address is added to the conflict set signature. Subsequent restarts of the transaction will most likely fill up all MSHRs again, though in this case the conflict set signature will predict those MSHRs entries whose draining should be avoided for as long as possible. In this way, when MSHRs are insufficient, store misses to thread-local and non-contended data (*contamination* misses [86]) are drained first, thus minimising the aforementioned risk of cross-fire between concurrent writers and readers. The conflict set signature is always cleared on commit and thus it only records information about previous restarts of the same dynamic transaction instance.

6.3.5 Overview of Existing and Proposed Techniques

Table 6.2 shows all described techniques with a summary of their properties and required hardware changes to implement them. Exponential backoff, serial irrevocability, and hourglass do not require any kind of hardware additions, our proposed software-based techniques require minor changes to provide core local information to the runtime, while our proposed hardware-based techniques need simple hardware additions that are core-local

Benchmark	**Input parameters**		**Txs.**
	Small	**Medium (+)**	
Genome	-g256 -s16 -n16384	-g512 -s32 -n32768	5
Intruder	-a10 -l4 -n2048 -s1	-a10 -l16 -n4096 -s1	3
KMeans-h	-m15 -n15 -t0.05 -i n2048-d16-c16	-m15 -n15 -t0.05 -i n16384-d24-c16	3
Labyrinth	-i random-x32-y32-z3-n96	-i random-x48-y48-z3-n64	3
SSCA2	-s13 -i1.0 -u1.0 -l3 -p3	-s14 -i1.0 -u1.0 -l9 -p9	3
Vacation-h	-n4 -q60 -u90 -r16384 -t4096	-n4 -q60 -u90 -r65536 -t4096	1
Yada	-a20 -i 633.2	-a10 -i ttimeu10000.2	6
Deque	100K ops., 1K dummy work		1
Btree	100K ops., 20K preloads, 25% ins.		2
Water	64 molecules		7
Radiosity	-batch		32

Table 6.3: Workload input parameters and number of transactions defined in the source code.

and retain the requester-wins nature of the HTM. Both LogTM and EL_T require coherence changes that affect communication between cores. Most proposed techniques can experience livelock conditions due to contention, so they should be executed in conjunction with a contention livelock-free technique like serial irrevocability, serialise on killer, or hourglass.

6.4 Simulation Environment and Methodology

6.4.1 Workloads

We use the STAMP benchmark suite as workloads to drive our experiments. These workloads provide significant diversity in behaviour and are expected to be good examples of transactional use cases and programming style. We choose to exclude the application *bayes* from our analysis due to large variability in execution times, which are very sensitive to random interleavings. In addition to STAMP, we include four workloads that have been used in a number of TM studies in the past, *water* and *radiosity* from SPLASH2 [89], and two microbenchmarks – *deque* and *btree*. Table 6.3 lists the command line parameters used in experiments in this chapter. We simulate both small and medium datasets for STAMP

Component	Description
Cores	8 in-order 2GHz x86 cores, 1 IPC for non-memory instructions
L1 I&D Caches	64KB 8-way, private, 64B lines, 1-cycle hit latency
L2 Cache	8MB 16-way, shared, 64B lines, 12-cycle uncontended hit latency
L2 Directory	L2-Directory Full-bit vector sharer list; 6-cycle latency
Memory	4GB, 100-cycle latency DRAM lookup latency
Interconnect	2D Mesh, 64-byte links, 1-cycle link latency
Exponential Backoff	Randomized exponential backoff with saturation after *n* steps. Backoff range $[1..2^n]$ where n is 8; Backoff multiplier factor: 117
Hourglass	4 retries before becoming toxic (best observed results)
Serial Irrevocability	8 maximum number of retries before executing in serial mode
Serialise on Address Serialise on Killer	The rw-lock implementation was stripped from the Linux kernel and is very similar to the one used in GCC to implement serial irrevocability.
Delayed req-wins	Timeout latencies (min/max): 64/1024 cycles
WriteBurst	Number of MSHR to buffer store miss information: 32

Table 6.4: Architectural and system parameters.

workloads, following the recommended input parameters [18].

6.4.2 Simulation Environment

All experiments in this chapter have been performed using the GEM5 simulator [7]. TM support that had been stripped from Ruby [55] upon integration into GEM5 has been plugged back in for the purposes of this study. The setup uses the *timing simple* processor model in GEM5. The memory system is modelled using Ruby. A distributed directory coherence protocol on a mesh-based network-on-chip is simulated. Each node in the mesh corresponds to a processing core with private L1 instruction and data caches and a slice of the shared L2 cache with associated directory entries. Table 6.4 describes key architectural parameters used in the experiments, as well as parameters used in the evaluated livelock avoidance mechanisms. For each workload - configuration pair we gathered average statistics over 5 randomised runs designed to produce different interleavings between threads. For LogTM, we used the hybrid resolution policy that prioritises older writers by allowing

Benchmark	Max. Occupancy	Avg. Occupancy	%Commits without VC
Btree	1	1.00	99.99
Genome+	1	1.00	99.99
Labyrinth	32	6.84	66.20
Labyrinth+	433	378.98	56.00
Vacation-h	1	1.00	99.99
Yada	9	1.58	96.20
Yada+	19	1.65	95.50

Table 6.5: Victim cache statistics for evaluated workloads on committed transactions. Numbers have been averaged over 5 simulated runs with 8 cores using the exponential backoff configuration.

their write requests to abort younger transactions [14].

To isolate our study from the effects of aborts caused by hardware resource limitations (e.g. cache capacity), our design includes an ideal transactional victim cache which is able to hold any number of speculatively modified cache lines when they are evicted from the L1 data cache while a transaction is executing. This allows transactions with large footprints to commit entirely in hardware, without having to resort to software fallback mechanisms. When a memory reference inside a transaction misses in the L1 cache but hits in the transactional victim cache, a penalty of only one extra cycle over the L1 hit time is applied. The transactional victim cache is flushed on abort and its contents drained to the L2 cache on commit. Evictions of speculatively read lines are also tolerated by our design, which uses perfect read signatures to track read sets. Such lines are not placed in the transactional victim cache and so they need to be fetched back from the L2 if need be.

Table 6.5 shows usage of the victim cache (VC) for the simulated workloads. We do not show data for workloads that do not make use of the victim cache during their execution. Even though we use an unbounded victim cache, as can be seen in the table, the number of lines that go into the victim cache is very small for all the workloads, with the exception of *labyrinth*. Half of the workloads do not use the victim cache at all, and for those that use it, the maximum occupancy reached by the victim cache stays below 20 cache lines except in *labyrinth*. Moreover, the percentage of transactions that commit without using the victim cache at all is high. Thus, designs that have replacement policies with some priority for transactional data, or that incorporate transactional bookkeeping in deeper levels of the memory hierarchy (private L2 caches) will likely be able to execute the transactions defined in these workloads entirely in hardware.

6.4.3 HTM Support in the Coherence Protocol

We have introduced minor changes in one of several coherence protocol implementations available in GEM5. The primary intent is to make a few simple changes that permit buffering of speculative updates in the private L1 cache without maintaining an undo-log. This brings the model as close in function as possible to requester-wins HTM implementations that may soon be available. We extended a typical MESI directory protocol available in the GEM5 release to support silent replacements of lines in *E* (exclusive) state. This is implemented via *yield* response messages that are sent by a former L1 exclusive owner to the L2 directory in response to a forwarded request for a line that is no longer present (after it was silently replaced). Through this feature, the protocol is then able to integrate speculative data versioning in private L1 caches at no extra cost. When a transaction aborts, it simply flush-invalidates all speculatively modified lines in its L1 data cache, which will eventually appear as silent E replacements to the directory. When it commits, it makes such updates globally visible by clearing the speculatively modified (SM) bits in L1 cache. To preserve consistent non-speculative values, transactional writes to M-state lines that find the SM bit not asserted must be written back to the L2 cache. These fresh speculative writes are performed without delay in L1 cache while a consistent copy of the data is simultaneously kept in the MSHR until the writeback is acknowledged (required in case of forwarded requests). Furthermore, transactional exclusive coherence requests (TGETX) must be distinguished from their non-transactional counterparts (GETX) both by L1 cache and L2 directory controllers. For TGETX, the L1 exclusive owner must send the data to both the L1 cache requester and the L2 cache (in order to preserve pre-transactional values), whereas for GETX requests it is sufficient with a cache-to-cache transfer, and in these cases the L2 directory expects no writeback.

The design also provides support for early release of addresses from the read-set of a transaction. This allows improved scalability in scenarios where a transaction may read a global data structure while intending to modify only a small part of it. For example, in the application *labyrinth* the global grid structure can be released after a local copy has been created within the transaction.

Component	**Abbrev.**	**Description**
Non-transactional	non-tx	Time spent execution non-transactional code
Barrier	barrier	Time spent waiting at barriers
Useful-Transactional	useful	Time spent executing transactions that commit
Wasted-Transactional	wasted	Time spent executing transactions that abort
Waiting in serial lock	wait-serial	Time spent waiting for an irrevocable transaction to complete
Waiting in address lock	wait-address	Time spent waiting for a conflicting transaction on the same address to complete
Waiting for killer	wait-killer	Time spent waiting for our killer transaction to complete
Serial irrevocable	serial	Time spent executing an irrevocable transaction
Token useful	token	Time spent in useful transactions with the token (hourglass)
Backoff	backoff	Time spent performing exponential backoff
Stall	stall	Time spent waiting for a memory request to complete in LogTM, or by the delayed requester-wins conflict resolution

Table 6.6: Various components in execution time breakdown plots.

6.4.4 Experiments and Metrics

We use execution time breakdowns to identify possible sources of overhead and compare them across the studied mechanisms. Execution times account for memory system effects by allowing the cache hierarchy and locality characteristics of the application to affect the metric. Execution breakdowns are broken down into several components listed in Table 6.6 based on the number of cycles spent performing the corresponding activity in all the cores. Some components are present only in certain configurations. Tables of results also show different statistics depending on the evaluated proposal, and include abort rates which indicate the fraction of transaction executions that result in aborts. This metric, when looked at in conjunction with execution time, provides a better picture of the efficacy of various contention and livelock mitigation techniques evaluated. Finally, we also use execution times for different techniques normalised to single-thread execution time to compare their scalability.

6.5 Evaluation

We first evaluate existing techniques in depth to identify possible sources of overhead. Later we evaluate our proposed software-based and hardware-based techniques. Finally, we conclude with a scalability comparison for the evaluated proposals.

6.5.1 Evaluation of Existing Techniques

Figure 6.3 compares the performance (execution times) of the existing techniques – exponential backoff (B), serial irrevocability as implemented in GCC (S), a design that combines both exponential backoff and serial irrevocability (BS), and hourglass (H). LogTM execution times have been used as the basis for normalisation, with the breakdown for the configuration shown using the bar marked *L*. In BS, serialisation occurs when a transaction fails to commit even after having retried 8 times applying an exponential backoff. For a description of breakdown components see Table 6.6.

Serial irrevocability imposes a performance cost because any parallelism among concurrent transactions is precluded. Frequent entries into this mode may result in severe performance degradation. Exponential backoff alone performs badly too. From the figure it is clear that when contention is present (for example in applications like *deque*, *btree*, *genome*, *intruder* and *yada*), just relying on serial irrevocability or exponential backoff can result in performance degradation ranging from 20% to about 40% in *intruder*, 2-2.5× in *btree* and several times (3-4×) in *yada*. Even a small portion of time in serial irrevocable mode results in significant time spent by other threads waiting for the irrevocable execution to finish (wait-serial). This overhead is expected to become worse as thread count increases. Though the combination of exponential backoff and serial irrevocability (BS) performs marginally better, all three livelock mitigation techniques perform comparably. Hourglass contention manager shines here being 6.8% better than BS. However, note that a performance gap of 26.8% can be seen between the baseline (LogTM with conservative deadlock avoidance using timestamp priorities) and the best existing technique (hourglass).

Table 6.7 shows some key metrics for different existing techniques evaluated in this section. The column *%Saturation* indicates the percentage of backoff events where backoff had saturated. Note that we use exponential backoff where the range of possible backoff

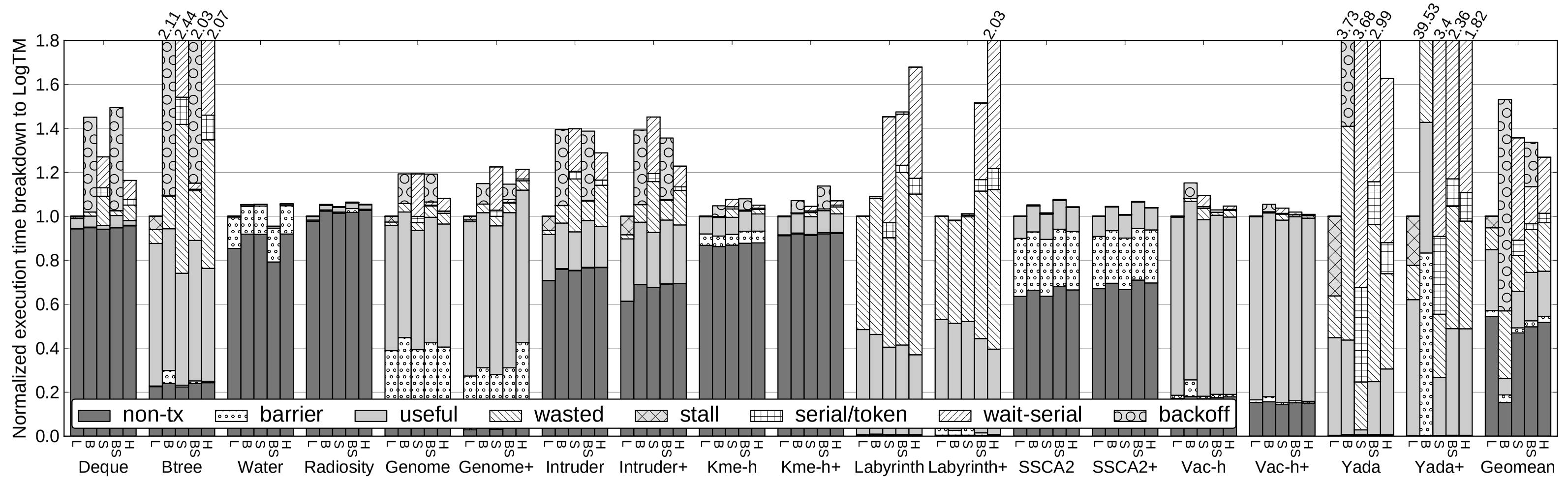


Figure 6.3: Relative performance of existing livelock mitigation techniques for 8 core runs.
L – LogTM; **B** – Exponential backoff; **S** – Serial irrevocable; **BS** – Exponential backoff and serial irrevocable; **H** – Hourglass

Benchmark	%Saturation	%Irrevocable/Token			%Token aborts	%Abort Rate			
	B	S	BS	H	H	B	S	BS	H
Deque	0.12	67.71	0.19	43.45	19.7	36.2	95.2	65.7	78.0
Btree	4.74	11.83	0.62	8.38	20.6	14.7	65.9	25.0	44.5
Water	0.00	1.19	0.00	2.08	6.3	2.2	13.9	2.1	16.8
Radiosity	0.47	0.12	0.00	0.05	26.7	0.4	2.4	0.5	1.2
Genome	2.52	1.39	0.15	0.38	44.8	4.6	16.3	6.4	8.3
Genome+	3.51	0.70	0.06	0.17	34.5	2.5	8.4	3.1	3.5
Intruder	0.61	9.45	0.08	1.31	79.9	14.6	59.8	18.3	44.6
Intruder+	1.05	9.92	0.07	2.32	53.9	10.6	60.9	14.2	38.1
KMeans-h	0.00	8.59	0.00	3.61	29.3	6.0	53.0	7.1	31.2
KMeans-h+	0.18	6.29	0.04	4.52	34.9	7.3	43.6	11.3	38.8
Labyrinth	3.65	18.37	1.74	7.88	36.9	34.5	71.9	50.0	61.9
Labyrinth+	0.95	0.42	2.85	8.89	37.3	30.4	32.0	47.2	63.2
SSCA2	0.00	0.08	0.00	0.12	0.0	0.1	2.2	0.1	1.1
SSCA2+	0.00	0.03	0.00	0.06	0.3	0.1	0.9	0.1	0.6
Vacation-h	8.04	0.87	0.00	0.46	1.1	2.1	12.3	1.7	5.0
Vacation-h+	11.83	0.41	0.01	0.17	0.0	0.8	6.8	1.2	1.8
Yada	33.86	46.42	12.26	13.72	34.8	45.3	90.9	71.9	62.8
Yada+	90.17	29.24	5.40	10.42	32.6	80.9	83.1	52.6	55.7

Table 6.7: Key metrics for existing techniques.

periods stops growing after a certain number of consecutive aborts. We find that *yada* and *btree* experiences this event often, a sign of contention being persistent, and that larger backoff periods might be beneficial in this particular workloads. The columns under the head *%Irrevocable/Token* indicate the percentage of transactions that ran irrevocably (as a fraction of the total number of committed transactions) when using serial irrevocability as fallback (configurations S and BS), or the percentage of transactions that acquired the global token when using hourglass contention management (configuration H). The last three columns under the head *%Abort Rate* show the percentage of aborts encountered in each configuration as a fraction of the total number of transaction starts (including restarts) – `aborts/(aborts+commits)`. We observe that high-contention workloads running on configuration S (without backoff) enter in irrevocable mode far more often than when using it in conjunction with backoff (configuration BS). This is expected since backoff pre-empts immediate restart of transactions that are likely to abort their killers. Thus the use of backoff is recommended, specially in contended scenarios. Though hourglass outper-

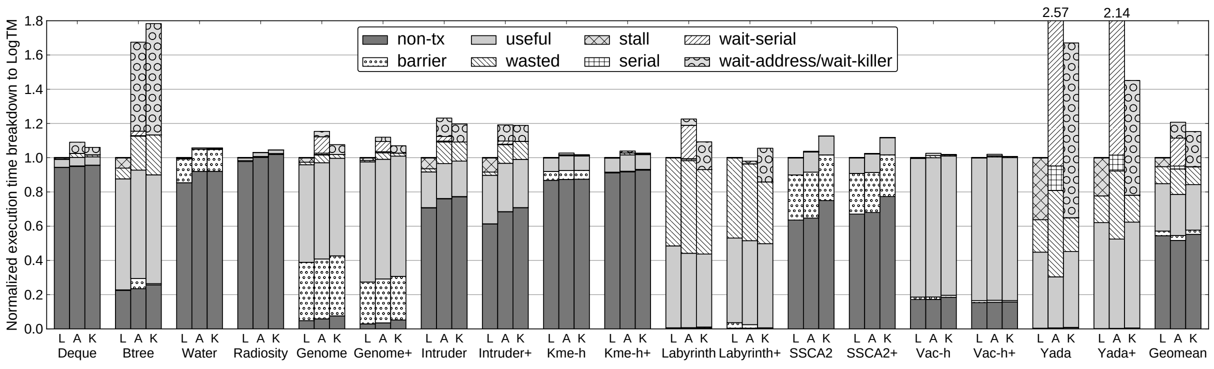


Figure 6.4: Relative performance of proposed software-based techniques for 8 core runs.
L – LogTM; **A** – Serialise on conflicting address (SoA); **K** – Serialise on killer transaction (SoK)

forms all other techniques, it is susceptible to performance degradation under contention, particularly if transactions are large. The column *%Token aborts* indicates the abort rate for transactions that executed while holding the hourglass token, a large number of aborts in this mode may cause a penalty similar or even larger than that of serial irrevocability.

6.5.2 Evaluation of Proposed Techniques

Software-based Techniques

Figure 6.4 compares the performance of software based techniques proposed in this chapter. The numbers are again normalised using LogTM as a baseline. This allows us to compare visually the improvements over techniques discussed in the previous section. Data has been presented for two configurations – serialise-on-conflicting-address (SoA) and serialise-on-killer-transaction (SoK).

We notice modest improvement in overall performance using either technique over existing techniques (shown in Figure 6.3). SoK performs the best, reducing the performance gap from the baseline to 15.3%, being slightly better than SoA. However, we see that both these techniques suffer when contention is high and transactions are large. This is evident from the execution times for *btree* and *yada*. In such cases, these techniques turn out to be substantially slower (1.6× - 2.5×) than LogTM. Note that in the case of *btree* SoA performs better than SoK, while the opposite trend is seen in the case of *yada*. In *btree* there is some overlap expected among conflicting addresses since a tree is being accessed. This is however not the case in the mesh-refinement algorithm used by *yada*. Previous work [61]

Benchmark	**%Irrevocable**	**%Aborts cycle**	**%Abort rate**		
	SoA	SoK	SoA	SoK	LogTM
Deque	0.00	3.98	44.1	31.2	9.5
Btree	0.29	3.08	20.0	22.2	7.9
Water	0.00	4.38	2.1	3.7	0.6
Radiosity	0.01	8.06	0.9	0.7	0.5
Genome	0.45	2.61	8.8	4.0	3.1
Genome+	0.15	2.28	3.7	1.7	1.1
Intruder	0.78	6.96	25.7	23.1	15.4
Intruder+	0.26	6.52	17.5	16.4	11.8
KMeans-h	0.00	11.46	8.7	4.5	0.2
KMeans-h+	0.04	6.03	16.1	6.1	0.2
Labyrinth	0.77	0.27	32.2	26.4	27.9
Labyrinth+	0.00	0.90	28.0	23.6	30.2
SSCA2	0.00	2.55	0.2	0.1	0.0
SSCA2+	0.00	3.39	0.1	0.1	0.0
Vacation-h	0.00	0.00	1.7	0.9	0.6
Vacation-h+	0.00	0.00	0.9	0.5	0.2
Yada	5.13	1.42	50.4	14.8	32.3
Yada+	2.86	1.26	36.5	10.3	18.1

Table 6.8: Key metrics for software-based proposed techniques.

has shown that *yada* typically has a very large number of conflicting addresses that do not show much repetition. Moreover, contention in *yada* tends to occur among groups of threads working on the same region of memory. Hence, SoK with its per-thread locks fits this case well.

Table 6.8 shows statistics for the evaluated software-based techniques. From the table we can see that, in fact, the percentage of transactions executed in serial irrevocable mode is substantially lower in SoA when compared to existing techniques. The column labelled *%Aborts cycle*, shows the percentage of aborted transactions that are allowed to restart without waiting on the killer transaction because a cyclic dependence would occur otherwise. Note that this value stays relatively low for all workloads, keeping additional

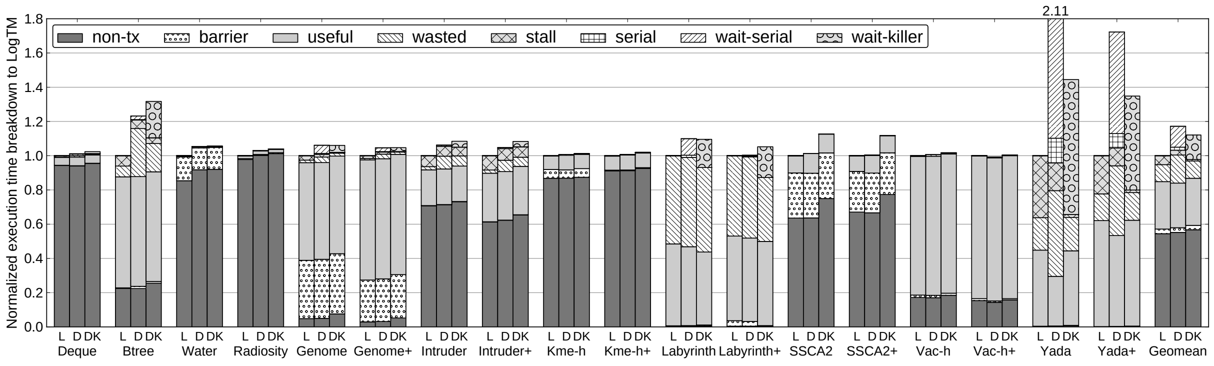


Figure 6.5: Relative performance of delayed requester-wins technique for 8 core runs.
L – LogTM; **D** – Delayed requester-wins with serial irrevocability; **DK** – Delayed requester-wins with SoK

aborts that might occur due to non serialised transactions low. In fact, the abort rates for *yada* in SoK are substantially lower than in LogTM, however, LogTM still performs better because with large transactions the overheads of serialising grow rapidly.

SoA significantly reduces the number of transactions that run in irrevocable mode when compared to existing techniques, which translates into lower overheads waiting for the serial lock. This is evident upon comparing numbers in Table 6.8 to those in Table 6.7. Time waiting on transactions executing on the same conflicting address is generally small (wait-address), although this overhead remains visible in *intruder* and *btree* since these are benchmarks with a larger number of read-modify-write transactions that conflict on a small set of addresses.

Hardware-based Techniques

Figure 6.5 shows relative performance of two new livelock mitigation techniques based on the delayed-requester-wins (DRW) mechanism. Since DRW does not guarantee forward progress we must have some form of software fallback to break persistent livelocks. The first DRW-based scheme (bar D in Figure 6.5) uses serial irrevocability as fallback while the second scheme uses SoK as fallback (bar DK). A version with hourglass as fallback was evaluated yielding lower performance (results not included); serial irrevocability and SoK are more efficient at bypassing short hotspots of high contention.

Performance differences between the two techniques are most noticeable in applications with large transactions or with moderate to high contention. Notice the large wait time due to serial irrevocability in *yada*. The drastic improvement in performance over

Benchmark	%Commits w. timeout		%Unexpired timers		%Irrevocable	%Abort Rate	
	DRW-S	DRW-SoK	DRW-S	DRW-SoK	DRW-S	DRW-S	DRW-SoK
Deque	32.72	21.49	71.85	74.54	0.09	24.0	13.9
Btree	12.84	9.44	49.67	46.58	0.27	24.7	15.3
Water	0.89	2.67	84.21	91.43	0.00	1.5	0.9
Radiosity	0.30	0.22	96.27	96.71	0.01	0.6	0.4
Genome	1.24	0.71	49.19	47.30	0.28	6.5	3.0
Genome+	0.57	0.29	52.12	48.56	0.07	2.5	1.3
Intruder	12.70	11.25	69.87	71.41	0.16	16.8	10.8
Intruder+	9.10	7.89	75.57	76.54	0.09	12.3	8.0
KMeans-h	4.29	2.72	73.34	65.21	0.00	1.7	1.8
KMeans-h+	4.21	3.66	79.45	78.23	0.00	1.3	1.2
Labyrinth	0.58	0.29	35.29	50.00	1.17	34.7	26.3
Labyrinth+	0.14	0.00	25.00	0.00	0.28	31.8	24.5
SSCA2	0.17	0.10	98.99	100.00	0.00	0.0	0.0
SSCA2+	0.07	0.05	99.69	99.07	0.00	0.0	0.0
Vacation-h	0.60	0.20	84.83	90.91	0.00	1.4	0.9
Vacation-h+	0.22	0.05	72.58	68.75	0.00	0.5	0.3
Yada	11.73	1.71	42.36	26.83	6.12	54.6	13.9
Yada+	6.34	1.37	39.72	30.80	2.89	38.7	10.0

Table 6.9: Key metrics for delayed requester-wins in conjunction with serial irrevocability and SoK.

serial irrevocability when using SoK is the result of improved parallelism since only those transactions that actually conflict wait. Other contended applications like *intruder* and *genome* obtain the best results seen so far, being only a few percent behind LogTM. In *yada*, DRW helps some transactions to commit that would otherwise have to abort, while SoK ensures that aborted transactions do not abort their killers upon restart. *Btree* also benefits substantially from DRW, experiencing a considerable performance boost with respect to previous evaluated techniques. Table 6.9 shows the percentage of commits that had active timeouts, which delayed (buffered) conflicting requests from remote cores instead of aborting the local transaction. *Intruder*, *yada* and *btree* benefit substantially from this fact. The head *%Unexpired timers* shows that applied timers tend to be cancelled before they expire, allowing the transaction to continue execution. Table 6.9 also shows that abort rates obtained for DRW-SoK, which are considerably lower with respect to other proposals across all workloads.

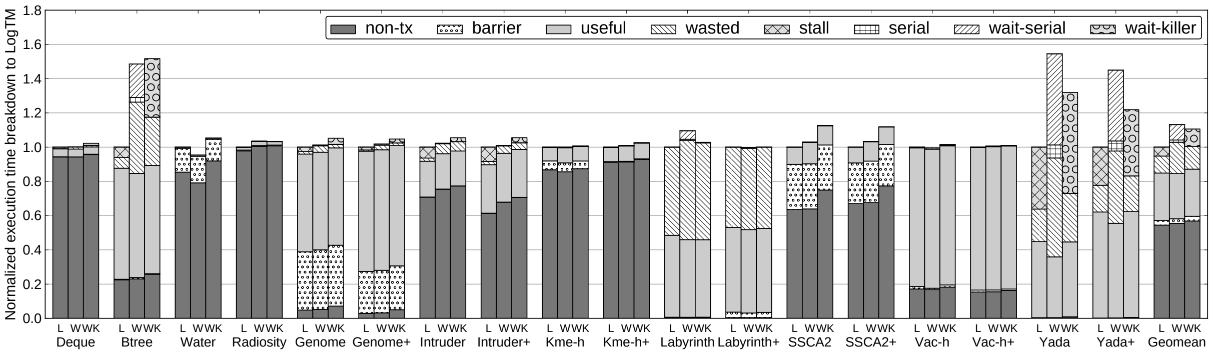


Figure 6.6: Relative performance of the WriteBurst technique for 8 core runs.
L – LogTM; **W** – WriteBurst with serial irrevocability; **WK** – WriteBurst with SoK

The use of timestamp priorities and reductions in wasted execution time due to the possibility to retry conflicting accesses (effectively stalling a lower priority transaction) still allows LogTM to perform significantly better under contention. However, even though DRW does not use additional coherence messages or timestamps, it has an average performance close to that seen in LogTM. Using SoK as fallback we observe a performance gap of about 12.1%, which can be largely attributed to the results obtained in *yada* and *btree*, as other workloads perform considerably closer to LogTM.

Figure 6.6 presents an execution time breakdown for the WriteBurst technique. Two versions have been evaluated: one with serial irrevocability (bar W) and one with SoK (bar WK) as fallback mechanism to guarantee forward progress. Again, a version with hourglass as fallback was evaluated (not shown) delivering slower performance.

In workloads where buffering stores can hide conflicts between transactions – by shrinking the window of time in which a transaction is susceptible to abort due to remote readers – using serial irrevocability proves to be slightly better (*btree*, *genome*, and *intruder*). This is due to the fact that transactions can restart immediately as long as they do not reach the threshold to execute in irrevocable mode, and under low contention this is beneficial. However, if contention is still present, serial irrevocability again imposes a severe performance penalty, see *yada*. When using WriteBurst in conjunction with SoK as a fallback mechanism, there is a slight penalty in applications where restarted transactions may now not conflict due to the WriteBurst mechanism. However, it proves to be much more effective for large transactions with moderate to high contention (*yada*). Overall, this approach is only 10.5% slower than LogTM.

Benchmark	**Max. Stores Buffered**		**Avg. Stores Buffered**		**%Irrevocable**	**%Abort Rate**	
	WB-S	WB-SoK	WB-S	WB-SoK	WB-S	WB-S	WB-SoK
Deque	3	3	2.89	2.89	0.04	27.3	25.8
Btree	11	12	3.42	3.42	2.40	40.5	24.7
Water	2	2	1.86	1.87	0.00	1.9	1.8
Radiosity	18	15	1.10	1.11	0.00	0.8	0.6
Genome	12	11	1.68	1.72	0.02	4.8	3.0
Genome+	12	11	1.51	1.52	0.02	2.1	1.3
Intruder	17	17	2.21	2.19	0.02	13.0	12.0
Intruder+	19	20	1.79	1.78	0.01	10.5	8.5
KMeans-h	2	2	1.69	1.69	0.00	3.9	3.5
KMeans-h+	2	3	1.73	2.38	0.00	2.9	3.1
Labyrinth	32	32	7.17	7.15	0.58	30.7	27.1
Labyrinth+	32	32	13.46	13.56	0.14	26.8	26.9
SSCA2	2	2	1.15	1.14	0.00	0.2	0.1
SSCA2+	2	2	1.10	1.10	0.00	0.1	0.1
Vacation-h	8	8	1.57	1.57	0.00	1.1	0.8
Vacation-h+	5	5	1.48	1.48	0.00	0.4	0.4
Yada	31	32	6.13	6.84	3.13	42.4	14.9
Yada+	32	32	6.26	6.66	1.93	31.5	10.4

Table 6.10: Key metrics for the WriteBurst mechanism in conjunction with serial irrevocability and SoK.

Table 6.10 provides information about the maximum and average number of buffered stores per committed transaction. As can be seen, *labyrinth* and *yada* exhausted the buffer capacity for some transactional executions, and maintain a relatively high average number of buffered stores. *Btree*, *radiosity*, *Intruder* and *genome* also have a higher number of maximum stores buffered when compared to the rest of the workloads, but their average usage is low. Serial irrevocability, as observed in the breakdown, is only used by *btree* and *yada*, where contention is still an issue.

This high usage of the buffers in *labyrinth* and *yada* may imply that these workloads can benefit from larger buffering capacity, and that they would also be sensitive to a lower number of MSHRs. We ran experiments using WB-SoK with 16 and 64 MSHRs, and observed that only *labyrinth* and *yada* experienced changes in performance compared to the results gathered using 32 entries. When 16 MSHRs are available, *labyrinth* and *labyrinth+* see performance drops of 7.3% and 5.4%, while *yada* and *yada+* drop by 9.0% and 5.2%

respectively. On the other hand, when the number of registers is set to 64, *yada* and *yada+* improve by 5.0% and 6.8% respectively, while both *labyrinth* and *labyrinth+* show roughly the same performance levels of the 32-MSHR configuration. We observed that the substantial improvement seen in *yada* is due to its maximum usage of 60 MSHR, whereas *labyrinth* uses around 40 entries.

6.5.3 Performance Overview of Proposed Techniques

In this section we compare relative performance of the introduced techniques in software and hardware. Figure 6.7 compares scalability for 8-core runs using a subset of the configurations we have discussed earlier.

We show the best performing existing technique, hourglass, which has significant drops in performance under contended scenarios, but can be a good choice when contention is low. Overall, the proposed hardware schemes perform better than their software counterparts, this is specially noticeable in contended applications like *btree*, *intruder* and *yada*. In applications where contention is mild like in *water*, *radiosity*, *SSCA2*, or *vacation*; SoA and SoK present competitive performance, being on par or even slightly better than hardware proposals, e.g., *SSCA2*. LogTM, plotted as the last bar, performs the best; especially under contention (*btree* and *yada*), where timestamp priorities become more useful, though the proposed schemes can achieve similar performance for most workloads.

This comparison highlights the need for basic livelock mitigation techniques in hardware (specially in contended scenarios), if not full-fledged forward progress guarantees which may be better implemented in software. As long as hardware techniques can effectively limit the need for software intervention, the performance cost associated with providing strong progress guarantees in software would be manageable.

6.6 Related Work

HTM proposals in the literature have typically provided forward progress guarantees using transaction priorities (through timestamps, for example) [14, 57] or lazy contention management [37, 62, 85]. However, the simplicity with which requester-wins HTMs [24, 25, 43] can be incorporated in hardware has resulted in such HTMs being the first ones to be widely accessible. As we have shown in this study, such designs tend to be susceptible

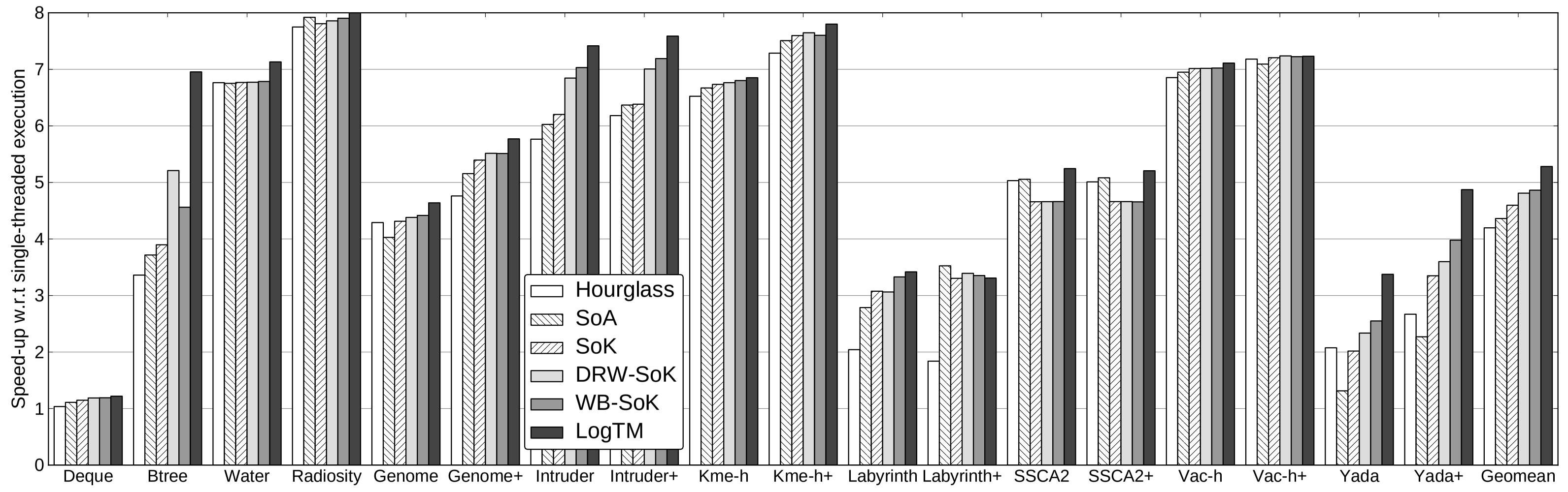


Figure 6.7: Relative performance of 8 core runs using existing, software, and hardware techniques with respect to single-threaded execution.

to performance degradation through transient or persistent livelocks. To the best of our knowledge, prior work has only noted this fact in passing, without presenting in-depth analyses of its performance implications or evaluating solutions that enhance forward-progress properties of requester-wins HTMs.

However, we would like to point out the connection between livelock mitigation and contention management. Extensive research into management of conflicting transactions has been undertaken in both HTM and STM. Designs which manage contention better are also less susceptible to livelock. In the area of STM, a variety of ways in which transactional conflicts could be handled have been evaluated. STM implementations allow great freedom in contention management policy design. Scherer and Scott [73] have evaluated a range of such options – Polite, which employs exponential backoff in a manner similar to our implementation; Karma and Eruption, which prioritise transactions based on the amount of work they have done; Kindergarten, where transactions accessing an object of contention take turns; and Polka, where exponential backoff and Karma are used together. Prioritised contention management policies (like Karma), with appropriate instrumentation in code, are relatively simple to implement in software. However, hardware implementations necessitate mechanisms to award and transport such priorities among processing units and, more importantly, mechanisms to notify and respond to decisions based on their use. The key attraction of requester-wins HTM in hardware design is the lack of any such requirement. The cache hierarchy and protocols do not change, changes local to processing units being sufficient to determine and rectify conflicts.

Entry into the serial irrevocable mode in GCC aborts all concurrent transactions and prevents new ones from starting. Toxic transactions [50] present a less drastic way – hourglass – to allow transactions that repeatedly abort due to conflicts to complete successfully. The mechanism requires such transactions to become toxic, i.e. prevent new ones from being scheduled (or re-executed upon abort) by acquiring a token. This gives a chance to concurrent non-conflicting transactions to complete successfully. We have included this strategy in this study, showing that is quite effective – being the best contendant amongst existing techniques.

Dolev et al. have proposed CAR-STM [30], which provides, in software, two methods to mitigate the adverse effects of conflicts. It maintains, for each processing core, a queue of transactions to be serialised on that core. An aborted transaction is rescheduled by queueing it on the conflicting core. In addition to this mechanism, a predictive scheduling approach assigns transactions to cores with which they are likely to conflict. However, this mechanism views transactions as tasks to be scheduled and thus, imposing scheduling overheads particularly in high contention scenarios. Another predictive approach is used in the Shrink contention manager described in [31].

In the context of HTMs, prior work [14] has identified several pathological conditions that can beset certain contention management policies. Requester-wins systems are inherently eager conflict resolution systems and suffer from pathologies that such systems are susceptible to. However, the absence of transaction priorities swaps starvation problems for increased risk of livelocks. For example, the requester-wins design treats reads and writes at an equal footing, thus avoiding the problem of starving readers/writers. However, the livelock risk, termed "friendly fire" in Bobba's paper, is present. Our study aims to estimate the likelihood of this risk and presents some new techniques to mitigate or avoid it.

Hybrid approaches have also been investigated. In particular, Hybrid-NOrec [27] describes the implementation of a hybrid TM system on best-effort HTM. The design allows software and hardware transactions to co-exist, although concurrency among such transactions is restricted rather severely. High-performance variants of this approach require the ability to issue non-transactional loads from within a transactional context.

Further research in HTM has investigated the use of reactive and proactive scheduling strategies [10, 91] to enhance parallelism and limit speculation when it is likely to fail. These proposals track conflicts between transactions and use this information in the future to predict contention and decide whether or not to stall a transaction when a transaction-begin primitive is encountered. Dependence-aware TM [70] tracks dependencies between concurrent transactions, supplying uncommitted data to dependent transactions and ensuring that commits occur in proper order. Cyclic dependencies are broken by aborting one of the transactions when a cycle is detected. These proposals tend to rely on HTMs that are more sophisticated and significantly more complex than a requester-wins design and, hence, are unlikely to be adopted soon by hardware vendors.

6.7 Summary

In this section we summarize key results and insights gathered during this study. These can be categorised under two heads:

6.7.1 For Programmers

Livelocks present a real and rather severe problem in requester-wins best effort HTMs. Even when cyclic dependencies may not arise among transactions, performance degradation due to transient livelocks may still occur because of repeated conflicts between an *aborter* and a restarted *abortee*. Exponential backoff is quite effective at mitigating adverse effects of livelocks. However, it does not guarantee freedom from livelocks. It must be used in conjunction with serial irrevocability to ensure forward progress. However, the TM runtime should not be very eager when deciding to enter serial irrevocability as this can potentially create pathological situations wherein applications with little contention may show severe performance degradation due to frequent serialisation because of the contention created by the serialisation mechanism itself. As we show in this study, serialisation should be done in stages. Initially using less severe techniques like Hourglass, SoA or SoK which permit much greater levels of parallelism before falling back to serial irrevocability.

6.7.2 For Architects

Bare-bones requester-wins HTM support, while being a good, low-complexity way of introducing practical TM in the real world, is not safe from livelocks even in lightly contended scenarios. While software strategies can prevent livelocks from precluding forward progress, they can also impose a performance penalty which in several cases is rather steep. Simple hardware mitigation strategies are quite useful in this context. By delaying conflict resolution, the architectural simplicity of requester-wins HTM designs can be retained while simultaneously mitigating the possibility of livelock and overheads associated with it. As we have shown in this study, this can be easily done by deferring processing of conflicting coherence requests (DRW) or delaying when writes are injected into the memory hierarchy (by buffering store misses). While such schemes may not guarantee freedom from livelock, they prove to be quite effective in avoiding them in many transactional use cases.

7 Conclusions

During the time frame of this thesis, hardware transactional memory has moved from a heavily researched topic to initial real-world implementations that are starting to be widely available in common commodity hardware [16, 43]. Hardware manufacturers are testing the waters with these initial implementations, which favour simplicity of integration into existing architectures, offering bare-bones support for transactions. Proposed HTM implementations by the academic community tend to be more complex in order to obtain higher performance or to provide additional compelling features that might be missing in initial real-world implementations. As TM gains awareness from mainstream programmers, future iterations of hardware support for transactions might progressively incorporate some of the many contributions described in academic research.

In this thesis, we have particularly focused on techniques to improve concurrency in HTM systems.

In Chapter 3, we propose a reconfigurable data cache (RDC) architecture able to handle two versions of the same logical data, with the objective to improve both eager and lazy version management schemes. The RDC has two execution modes: a 64KB general purpose mode and a 32KB TM mode. The latter mode allows the RDC to gracefully manage both old and new values within the cache. We explain the benefits that the RDC offers in version management schemes and how these translate in performance improvements and energy savings.

In Chapter 4, we demonstrate that transactions that experience contention tend to have high locality of reference. We exploit this fact by proposing a prefetching mechanism tailored to work in the context of hardware transactional memory. We propose a simple hardware design that successfully identifies prefetch candidates and that quickly adapts to changing contended addresses. We show that prefetching cache lines with high locality can improve overall concurrency by speeding up transactions and, thereby, narrow the window of time in which such transactions persist and can cause contention. This technique provides improvements for most transactional workloads we have analysed, with substantial gain in contended applications.

In Chapter 5, we present a hardware abort recurrence predictor (HARP) that proactively identifies transactions likely to fail. HARP dynamically chooses a contention avoidance mechanism based on the expected duration of contention, maximising computational resource utilisation, while minimising the amount of wasted work due to transaction aborts. We provide a detailed design description based on simple hardware structures, that yield a smaller hardware footprint with respect to prior work. The design provides seamless support for both single-application and multi-application scenarios, and our experimental results show that HARP outperforms previous state-of-the-art proposals. In terms of future work, HARP predictions can be leveraged to implement aggressive power saving schemes when no useful computation can be scheduled.

Finally, in Chapter 6, we study the performance and forward progress issues present in protocols employed in initial read-world HTM implementations. Hardware vendors have chosen low complexity approaches, which provide bare-bones support by doing minimal modifications to existing chip multiprocessors. We study how this protocols behave in several transactional use cases, and find that persistent and transient livelocks conditions are likely to occur. Our evaluation of existing livelock mitigation and avoidance techniques shows that they impose a performance penalty which in several cases is rather steep. We then propose a set of hardware and software techniques that retain the simplicity of these initial designs while enhancing their robustness towards livelock conditions, improving overall HTM performance. Fortunately, future work in this area can now benefit from HTM enabled chips, which can greatly simplify future research by freeing researchers from slow and complex simulation platforms for certain studies.

7.1 Future Work

Some of the contributions described in this thesis may be further extended, more specifically, we believe that transactional prefetching (Chapter 4) and abort prediction (Chapter 5) offer clear future research directions that might be worth exploring.

Regarding transactional prefetching, in this thesis, we have not studied the acceleration of generic blocks of code that may present high locality of reference, such as critical sections or synchronisation operations; nor interactions with other forms of prefetching in order to develop synergistic combinations to speed up both transactional and non-transactional code. The mechanisms described in this document may be applicable to generic demarcated sections of code where locality of reference can be exploited.

We believe that abort prediction is going to play a major role in future implementations of transactional memory. Our approach proposes a new hardware structure that employs conflicting addresses to determine if conflicts are likely to happen in the future. However, implementations that leverage existing branch prediction hardware may be more appealing to hardware vendors. While transaction conflicting patterns might differ substantially from branching patterns, simple extensions to branch predictors might be sufficient to provide a good initial low-complexity approach to transaction abort prediction. In such a scenario, instructions that start a transaction could be treated as branch instructions and the predictor can determine whether the transaction is allowed to execute or a branch to a fallback execution path is taken.

An additional line of future work would include the study of interactions present when combining together some of the proposed techniques. We have not investigated this further because we see the different proposals as orthogonal techniques that might be employed depending on the characteristics of the base HTM system. It is important to note that the initial real-world implementations that we discuss in Chapter 6 are a good target for most of the contributions of this thesis. For example, the use of lazy version management that invalidates speculative state upon abort makes these systems good targets for transactional prefetching, in addition, abort prediction is likely to have a positive impact due to the fact that eager conflict detection is employed and conflicts are likely to be common due to the requester-wins policy. Finally, the techniques that we propose in Chapter 6 have proven to be effective at ameliorating livelock conditions and improving overall performance.

8 Publications

The contents of this thesis led to the following publications:

Adrià Armejach, Azam Seyedi, Rubén Titos Gil, Ibrahim Hur, Adrián Cristal, Osman S. Unsal and Mateo Valero, **Using a Reconfigurable L1 Data Cache for Efficient Version Management in Hardware Transactional Memory**, In Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT'2011).

Adrià Armejach, Anurag Negi, Adrián Cristal, Osman S. Unsal and Per Stenstrom, **Transactional Prefetching: Narrowing the Window of Contention in Hardware Transactional Memory**, In the 7th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'2012).

Anurag Negi, **Adrià Armejach**, Adrián Cristal, Osman S. Unsal and Per Stenstrom, **Transactional Prefetching: Narrowing the Window of Contention in Hardware Transactional Memory**, In Proceedings of 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'2012).

Adrià Armejach, Anurag Negi, Adrián Cristal, Osman S. Unsal, Per Stenstrom and Tim Harris, **HARP: Adaptive Abort Recurrence Prediction for Hardware Transactional Memory**, In Proceedings of the 20th International Conference on High Performance Computing (HiPC'2013).

Adrià Armejach, Rubén Titos Gil, Anurag Negi, Osman S. Unsal and Adrián Cristal, **Techniques to Improve Performance in Requester-wins Hardware Transactional Memory**, In ACM Transactions on Architecture and Code Optimization (TACO), Volume 10 Issue 4, Article no. 42, December 2013.

The following publications are related but not included in this thesis:

Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, **Adrià Armejach**, Adrián Cristal, Osman S. Unsal, Tim Harris and Mateo Valero, **EazyHTM: Eager-Lazy Hardware Transactional Memory**, In Proceedings of the 42nd International Symposium on Microarchitecture (MICRO'2009).

Azam Seyedi, **Adrià Armejach**, Adrián Cristal, Osman S. Unsal, Ibrahim Hur and Mateo Valero, **Circuit design of a dual-versioning L1 data cache for optimistic concurrency**, In Proceedings of the 21st ACM Great Lakes Symposium on VLSI (GLSVLSI'2010).

Azam Seyedi, **Adrià Armejach**, Adrián Cristal, Osman S. Unsal, Ibrahim Hur and Mateo Valero, **Circuit design of a dual-versioning L1 data cache**, In the VLSI Journal of Integration, Volume 45 Number 3, June 2012.

Azam Seyedi, **Adrià Armejach**, Adrián Cristal, Osman S. Unsal and Mateo Valero, **Novel SRAM bias control circuits for a low power L1 data cache**, In Proceedings of Norchip Conference (NORCHIP'2012)

References

[1] The Electric VLSI Design System. `http://www.staticfreesoft.com`. Cited on page: 28

[2] Predictive technology model. `http://ptm.asu.edu/`. Cited on page: 28

[3] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 316–327, February 2005. DOI: 10.1109/HPCA.2005.41. Cited on page: 4, 11, 13, 23, 38, 87

[4] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers*, pages 4–18, January 2009. DOI: 10.1007/978-3-540-92990-1_3. Also appears in Springer-Verlag Lecture Notes in Computer Science volume 5409. Cited on page: 60, 61

[5] Adrià Armejach, Azam Seyedi, Rubén Titos-Gil, Ibrahim Hur, Osman S. Unsal, Adrián Cristal, and Mateo Valero. Using a reconfigurable L1 data cache for efficient version management in hardware transactional memory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 361–371, October 2011. DOI: 10.1109/PACT.2011.67. Cited on page: 57

[6] Nathan Binkert, Ronald Dreslinski, Lisa Hsu, Kevin Lim, Ali Saidi, and Steven Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006. DOI: 10.1109/MM.2006.82. Cited on page: 30, 49, 72

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, pages 1–7, August 2011. DOI: 10.1145/2024716.2024718. Cited on page: 105

[8] G. Blake, R.G. Dreslinski, and T. Mudge. Bloom Filter Guided Transaction Scheduling. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 75–86, February 2011. DOI: 10.1109/HPCA.2011.5749718. Cited on page: 60, 61, 62, 67, 72, 74

[9] Geoffrey Blake. *A Hardware/Software Approach for Alleviating Scalability Bottlenecks in Transactional Applications*. PhD thesis, University of Michigan, 2011. Cited on page: 72

[10] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 156–167, December 2009. DOI: 10.1145/1669112.1669133. Cited on page: 60, 62, 67, 121

[11] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. ISSN 0001-0782. DOI: 10.1145/362686.362692. Cited on page: 46, 102

[12] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 24–34, June 2007. DOI: 10.1145/1273440.1250667. Cited on page: 11

[13] Colin Blundell, Arun Raghavan, and Milo M. K. Martin. RetCon: Transactional Repair without Replay. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 258–269, June 2010. DOI: 10.1145/1815961.1815995. Cited on page: 74, 87

[14] Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 81–91, June 2007. DOI: 10.1145/1250662.1250674. Cited on page: 12, 15, 25, 56, 61, 88, 93, 94, 106, 118, 121

[15] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July 1999. ISSN 0272-1732. DOI: 10.1109/40.782564. Cited on page: 1

[16] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 225–236, June 2013. DOI: 10.1145/2485922.2485942. Cited on page: 123

[17] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 69–80, June 2007. DOI: 10.1145/1250662.1250673. Cited on page: 4

[18] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, September 2008. DOI: 10.1109/IISWC.2008.4636089. Cited on page: 9, 15, 17, 30, 43, 50, 59, 65, 66, 74, 90, 105

[19] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 97–108, February 2007. DOI: 10.1109/HPCA.2007.346189. Cited on page: 11, 23, 25, 27, 28, 38, 42, 57

[20] Yuan Chou. Low-Cost Epoch-Based Correlation Prefetching for Commercial Applications. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 301 –313, December 2007. DOI: 10.1109/MICRO.2007.23. Cited on page: 57

[21] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40, 2010. DOI: 10.1145/1755913.1755918. Cited on page: 12, 88

[22] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. Tradeoffs in transactional memory virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, October 2006. DOI: 10.1145/1168857.1168903. Cited on page: 28

[23] JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. The common case transactional behavior of multithreaded programs. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February 2006. DOI: 10.1109/HPCA.2006.1598135. Cited on page: 15

[24] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 39–50, December 2010. DOI: 10.1109/MICRO.2010.40. Cited on page: 12, 59, 88, 118

[25] C Click. Azuls experiences with hardware transactional memory. In *HP Labs - Bay Area Workshop on Transactional Memory*, 2009. Cited on page: 59, 118

[26] S. Cosemans, W. Dehaene, and F. Catthoor. 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*, 44(7):2065–2077, 2009. ISSN 0018-9200. DOI: 10.1109/JSSC.2009.2021925. Cited on page: 36

[27] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: a case study in the effectiveness of best

effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 39–52, 2011. DOI: 10.1145/1950365.1950373. Cited on page: 121

[28] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, 2006. DOI: 10.1145/1168918.1168900. Cited on page: 4

[29] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, March 2009. DOI: 10.1145/1508284.1508263. Cited on page: 59

[30] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, pages 125–134, August 2008. DOI: 10.1145/1400751.1400769. Cited on page: 60, 61, 120

[31] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, pages 7–16, August 2009. DOI: 10.1145/1582716.1582725. Cited on page: 121

[32] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing*, pages 68–75, 1997. DOI: 10.1145/263580.263597. Cited on page: 42

[33] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Early register deallocation mechanisms using checkpointed register files. *IEEE Transactions on Computers*, 55(9): 1153–1166, 2006. ISSN 0018-9340. DOI: 10.1109/TC.2006.145. Cited on page: 37

[34] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*,

21(12):1793–1807, 2010. ISSN 1045-9219. DOI: 10.1109/TPDS.2010.49. Cited on page: 91

[35] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990. DOI: 10.1145/325164.325102. Cited on page: 100

[36] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004. DOI: 10.1109/MM.2004.91. Cited on page: 11, 42

[37] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102, June 2004. DOI: 10.1145/1028176.1006711. Cited on page: 4, 38, 56, 87, 101, 118

[38] Tim Harris, Adrián Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional Memory: An Overview. *IEEE Micro*, 27(3):8–29, 2007. DOI: 10.1109/MM.2007.63. Cited on page: 9

[39] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010. ISBN 1608452352, 9781608452354. Cited on page: 3, 41, 59, 87

[40] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993. DOI: 10.1145/165123.165164. Cited on page: 3, 4, 9, 37, 56, 59, 87

[41] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003. DOI: 10.1145/872035.872048. Cited on page: 4

[42] Sungpack Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal tm characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 1–11, 2010. DOI: 10.1109/IISWC.2010.5648812. Cited on page: 64, 84

[43] Intel Corporation. Transaction Synchronization Extensions (TSX). In *Intel Architecture Instruction Set Extensions Programming Reference*, pages 506–529, February 2012. `http://software.intel.com/file/41604`. Cited on page: 12, 59, 88, 118, 123

[44] Syed Ali Raza Jafri, Gwendolyn Voskuilen, and T. N. Vijaykumar. Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 521–534, 2013. DOI: 10.1145/2451116.2451173. Cited on page: 87

[45] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual International Symposium on Computer Architecture*, pages 364–373, 1990. DOI: 10.1145/325164.325162. Cited on page: 57

[46] Ronald N. Kalla and Balaram Sinharoy. Power7: IBM's Next Generation Power Microprocessor. *IEEE Symposium of High-Performance Chips (Hot Chips 21)*, 2009. Cited on page: 28

[47] Ronald N. Kalla, Balaram Sinharoy, William J. Starke, and Michael S. Floyd. Power7: IBM's Next-Generation Server Processor. *IEEE Micro*, 30(2):7–15, 2010. DOI: 10.1109/MM.2010.38. Cited on page: 28

[48] Poonacha Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *Micro, IEEE*, 25(2):21–29, 2005. ISSN 0272-1732. DOI: 10.1109/MM.2005.35. Cited on page: 28

[49] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, pages 209–220, March 2006. DOI: 10.1145/1122971.1123003. Cited on page: 4

[50] Yujie Liu and Michael Spear. Toxic transactions. In *TRANSACT '11 6th Workshop on Transactional Computing*, February 2011. Cited on page: 89, 91, 93, 120

[51] Marc Lupon, Grigorios Magklis, and Antonio 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*, pages 293–302, September 2009. DOI: 10.1109/PACT.2009.19. Cited on page: 38, 57

[52] Marc Lupon, Grigorios Magklis, and Antonio González. A dynamically adaptable hardware transactional memory. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 27–38, 2010. DOI: 10.1109/MICRO.2010.23. Cited on page: 13, 57, 87

[53] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002. ISSN 0018-9162. DOI: 10.1109/2.982916. Cited on page: 30

[54] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-877-3. DOI: 10.1145/1693453.1693465. Cited on page: 61

[55] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005. ISSN 0163-5964. DOI: 10.1145/1105734.1105747. Cited on page: 30, 105

[56] Mark Moir. Hybrid transactional memory, July 2005. PDF: `http://www.cs.wisc.edu/trans-memory/misc-papers/moir:hybrid-tm:tr:2005.pdf`. Unpublished manuscript. Cited on page: 4

[57] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the International*

Symposium on High-Performance Computer Architecture, pages 254–265, February 2006. Cited on page: 4, 11, 23, 38, 61, 72, 89, 91, 93, 94, 118

[58] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *Proceedings of The Ninth International Symposium on High-Performance Computer Architecture*, pages 129–140, 2003. DOI: 10.1109/HPCA.2003.1183532. Cited on page: 42

[59] A. Negi, R. Titos-Gil, M.E. Acacio, J.M. Garcia, and P. Stenstrom. Eager Meets Lazy: The Impact of Write-Buffering on Hardware Transactional Memory. In *International Conference on Parallel Processing (ICPP)*, pages 73–82, 2011. DOI: 10.1109/ICPP.2011.63. Cited on page: 45, 57, 101

[60] Anurag Negi, M. M. Waliullah, and Per Stenström. LV*: A low complexity lazy versioning HTM infrastructure. In *ICSAMOS*, pages 231–240, 2010. Cited on page: 13, 42, 57

[61] Anurag Negi, Adrià Armejach, Adrián Cristal, Osman S. Unsal, and Per Stenström. Transactional Prefetching: Narrowing the Window of Contention in Hardware Transactional Memory. In *Proceedings of the 21th International Conference on Parallel Architectures and Compilation Techniques*, pages 181–190, 2012. DOI: 10.1145/2370816.2370844. Cited on page: 97, 112

[62] Anurag Negi, Rubén Titos-Gil, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Pi-TM: Pessimistic Invalidation for Scalable Lazy Hardware Transactional Memory. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, pages 1–12, 2012. DOI: 10.1109/HPCA.2012.6168951. Cited on page: 13, 42, 57, 118

[63] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992. ISSN 1057-4514. DOI: 10.1145/130616.130623. Cited on page: 2

[64] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3:26–29, September 2005. ISSN 1542-7730. DOI: 10.1145/1095408.1095418. Cited on page: 1

[65] Seth H. Pugsley, Manu Awasthi, Niti Madan, Naveen Muralimanohar, and Rajeev Balasubramonian. Scalable and reliable communication for hardware transactional memory. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 144–154, October 2008. DOI: 10.1145/1454115.1454137. Cited on page: 57

[66] Xuehai Qian, Wonsun Ahn, and Josep Torrellas. ScalableBulk: Scalable Cache Coherence for Atomic Blocks in a Lazy Environment. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 447–458, 2010. DOI: 10.1109/MICRO.2010.29. Cited on page: 57

[67] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolic. *Digital integrated circuits - A design perspective*. Prentice Hall, 2ed edition, 2004. Cited on page: 17

[68] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305, December 2001. Cited on page: 59

[69] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, June 2005. DOI: 10.1109/ISCA.2005.54. Cited on page: 13, 28

[70] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 246–257, 2008. DOI: 10.1109/MICRO.2008.4771795. Cited on page: 121

[71] Christopher Rossbach, Owen Hofmann, and Emmett Witchel. Is transactional memory programming actually easier? In *Proceedings of the 8th Workshop on Duplicating, Deconstructing, and Debunking*, June 2009. PDF: http://www.cs.utexas.edu/users/rossbach/pubs/wddd09-rossbach.pdf. Cited on page: 9, 59

[72] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing Signatures for Transactional Memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society. DOI: 10.1109/MICRO.2007.24. Cited on page: 28, 47

[73] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, pages 240–248, July 2005. DOI: 10.1145/1073814.1073861. Cited on page: 60, 61, 93, 120

[74] Azam Seyedi, Adrià Armejach, Adrián Cristal, Osman S. Unsal, Ibrahim Hur, and Mateo Valero. Circuit design of a dual-versioning L1 data cache for optimistic concurrency. In *Proceedings of the ACM Great Lakes Symposium on VLSI*, pages 325–330, 2011. DOI: 10.1145/1973009.1973074. Cited on page: 17, 37

[75] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995. DOI: 10.1145/224964.224987. Cited on page: 4

[76] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pages 42–53, 2000. DOI: 10.1145/360128.360135. Cited on page: 57

[77] Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 136–146, June 2009. DOI: 10.1145/1542275.1542299. Also available as TR 939, Department of Computer Science, University of Rochester, September 2008. Cited on page: 12, 13, 66, 101

[78] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 139–150, June 2008. DOI: 10.1109/ISCA.2008.17. Cited on page: 87

[79] Nehir Sonmez, Tim Harris, Adrián Cristal, Osman S. Unsal, and Mateo Valero. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, pages 1–10, May 2009. DOI: 10.1109/IPDPS.2009.5161032. Cited on page: 61

[80] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional

memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150, February 2009. DOI: 10.1145/1504176.1504199. Cited on page: 11

[81] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. *HP Laboratories, Palo Alto, Tech. Rep*, 20, 2008. Cited on page: 28

[82] Rubén Titos, Manuel E. Acacio, and Jose M. Garcia. Speculation-based conflict resolution in hardware transactional memory. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, pages 1–12, May 2009. DOI: 10.1109/IPDPS.2009.5161021. Cited on page: 101

[83] Rubén Titos-Gil, Anurag Negi, Manuel E. Acacio, José M. García, and Per Stenstrom. ZEBRA: a data-centric, hybrid-policy hardware transactional memory design. In *Proceedings of the International Conference on Supercomputing*, pages 53–62, 2011. DOI: 10.1145/1995896.1995906. Cited on page: 57, 97

[84] Ruben Titos-Gil, Anurag Negi, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Eager beats Lazy: Improving Store Management in Eager Hardware Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2012. ISSN 1045-9219. Cited on page: 101

[85] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 145–155, 2009. DOI: 10.1145/1669112.1669132. Cited on page: 13, 23, 25, 28, 42, 57, 101, 118

[86] MM Waliullah and Per Stenstrom. Removal of conflicts in hardware transactional memory systems. *International Journal of Parallel Programming*, 2012. DOI: 10.1007/s10766-012-0210-0. Cited on page: 103

[87] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136, 2012. DOI: 10.1145/2370816.2370836. Cited on page: 59, 88

[88] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 285–296, June 2008. DOI: 10.1145/1378533.1378584. Cited on page: 91

[89] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd annual International Symposium on Computer Architecture*, pages 24–36, 1995. DOI: 10.1145/223982.223990. Cited on page: 90, 104

[90] Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 261–272, February 2007. DOI: 10.1109/HPCA.2007.346204. Cited on page: 11, 23, 25, 38, 42, 56, 87

[91] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 169–178, June 2008. DOI: 10.1145/1378533.1378564. Cited on page: 60, 61, 121