Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y
Tecnología de Computadores

Técnicas Hardware para Sistemas de Memoria Transaccional de Alto Rendimiento en Procesadores Multinúcleo

Tesis Doctoral

Autor:
José Rubén Titos Gil

Directores:
José Manuel García Carrasco
Manuel Eugenio Acacio Sánchez

Murcia, Septiembre de 2011

Resumen

En el último lustro hemos sido testigos de un punto de inflexión fundamental en la historia de la computación. Durante las últimas cuatro décadas, los avances en la escala de integración han permitido doblar el número de transistores integrados en un chip cada 18 meses, lo cual ha posibilitado un extraordinario aumento en las prestaciones de los procesadores, cuyas aplicaciones han revolucionado nuestra sociedad en menos de medio siglo. Por una parte, la progresiva miniaturización de los transistores ha reducido su tiempo de conmutación, permitiendo construir procesadores capaces de operar a frecuencias de reloj cada vez más altas. Por otro lado, los arquitectos de computadores han empleado los transistores adicionales disponibles en el chip para diseñar cauces de ejecución más y más sofisticados, capaces de explotar el paralelismo a nivel de instrucción existente en los programas secuenciales. Así, la combinación de los progresos tanto en el proceso de integración como en la arquitectura de los procesadores ha sostenido varias décadas de crecimiento exponencial en el rendimiento de los sistemas informáticos, un logro sin precedentes en la historia de la tecnología que no obstante ha tocado a su fin.

Las limitaciones impuestas por el consumo energético y la capacidad de refrigeración del chip, unido a los crecientes costes de diseño y verificación de arquitecturas monoprocesador cada vez más complejas, han provocado un estancamiento en las mejoras de rendimiento de los sistemas monoprocesador en los últimos años. La industria ha encontrado una salida a esta crisis por medio de la fabricación de chips que incluyen más de un procesador por chip, los llamados *procesadores multinúcleo* o CMPs. De esta forma, las hojas de ruta de la mayoría de fabricantes proyectan un rápido aumento en el número de núcleos integrados en un solo chip, con el propósito de incrementar las prestaciones mediante la explotación del paralelismo a nivel de tarea. Este cambio de paradigma hacia arquitecturas paralelas trae consigo nuevas oportunidades, pero también difíciles

retos que hacen de estos tiempos un momento apasionante para la investigación en Arquitectura de Computadores.

El giro hacia las arquitecturas multinúcleo ha situado el problema de la programación concurrente al frente de la investigación en computación, pues la gran barrera que impide aprovechar plenamente la capacidad de cómputo agregado de un CMP es precisamente su complejidad de programación. El auge de los multinúcleos está haciendo que la programación paralela se generalice, empujando a los programadores hacia un paradigma de programación poco conocido. Desafortunadamente, la programación concurrente es una tarea más compleja que la secuencial, pues un programa paralelo es indudablemente más difícil de diseñar, escribir y depurar que su versión secuencial: Los diferentes hilos de un programa paralelo necesitan comunicarse para llevar a cabo la tarea de manera coordinada. En este escenario cooperativo, orquestar la ejecución de los hilos con el fin de garantizar la corrección del programa al tiempo que se consigue un alto grado de eficiencia y se mantiene una elevada productividad de programación, supone un complicado reto. Los modelos tradicionales de programación multi-hilo recurren a primitivas de bajo nivel como los cerrojos para proteger las estructuras de datos compartidas, garantizando el acceso de hilos en exclusión mutua. En este contexto, la complejidad de la sincronización basada en cerrojos hace de la programación paralela una tarea propensa a errores, especialmente cuando se utilizan cerrojos de granularidad fina con el fin de extraer mayor rendimiento.

Sin lugar a dudas, el compromiso entre la facilidad de programación y rendimiento impuesto por el modelo de sincronización basado en cerrojos sigue siendo uno de los retos clave para los programadores y arquitectos de computadores de la era multinúcleo. La Memoria Transaccional (TM) ha sido propuesta como un modelo de programación conceptualmente más sencillo que puede ayudar a incrementar la productividad de los programadores al eliminar la compleja tarea de razonar sobre la corrección de un programa que usa cerrojos de grano fino. Al utilizar transacciones para sincronizar el acceso a los datos compartidos, los programadores no necesitan preocuparse por posibles entrelazados en la ejecución de las secciones críticas que provoquen la aparición de interbloqueos o den lugar a un resultado incorrecto. Así pues, TM aborda el compromiso entre rendimiento y productividad: Desde el punto de vista del programador, TM facilita la programación paralela al favorecer un estilo de sincronización de grano grueso, manteniendo la promesa de que aún con dicho estilo es posible alcanzar prestaciones comparables a las de los cerrojos de grano fino. El sistema TM subyacente, por su parte, trata de sacar el mejor partido del paralelismo existente

en la aplicación gracias a la ejecución especulativa de transacciones, utilizando los mecanismos oportunos para detectar conflictos entre transacciones concurrentes, con el fin de mantener en todo caso las propiedades de las transacciones.

Los sistemas TM pueden implementarse completamente en software, en hardware o usando una combinación de ambos. Una aproximación software permite la posibilidad de ejecutar aplicaciones transaccionales en sistemas existentes, con un alto grado de flexibilidad a un bajo coste. Por contra, la implementación de los mecanismos necesarios en software impone una sobrecarga demasiado alta, provocando que las soluciones basadas en cerrojos todavía sean superiores cuando el rendimiento es importante. A pesar de las ventajas del enfoque software, es incuestionable la necesidad de implementaciones rápidas para que el nuevo paradigma basado en transacciones alcance un uso generalizado. De hecho, dada la abundancia de transistores disponibles en los chips actuales, uno de los retos más importantes para los arquitectos de computadores hoy día es entender qué abstracciones pueden mejorar la productividad del desarrollo de software paralelo, y a partir de ahí introducir el soporte hardware apropiado para realizarlo. Las transacciones son un buen candidato como abstracción en el ámbito de la sincronización de programas de memoria compartida. Esta tesis se centra en la implementación en hardware de los mecanismos que proporcionan un control de concurrencia optimista con estrictas garantías de atomicidad y aislamiento, con el objetivo de alcanzar altos niveles de rendimiento a un coste razonable en términos de complejidad en el diseño global.

El trabajo recogido en esta tesis abarca diferentes aspectos dentro del espacio de diseño de los sistemas hardware de memoria transaccional (HTM), en el contexto de un procesador multinúcleo construido sobre una red de interconexión escalable. Esta tesis identifica ineficiencias críticas que impactan el rendimiento de los diferentes enfoques HTM, y propone mecanismos para solventar dichas limitaciones. En esta disertación consideramos tanto sistemas HTM de política ansiosa como aquellos diseñados bajo el enfoque perezoso, y afrontamos las sobrecargas en el rendimiento que son inherentes a cada política.

Quizá la contribución más relevante de esta tesis es ZEBRA, un sistema HTM de política híbrida que adapta su comportamiento en función de las características dinámicas de la carga de trabajo. Gracias a la selección de una u otra política en los mecanismos transaccionales básicos (gestión de versiones y resolución de conflictos), ZEBRA consigue combinar las ventajas de los enfoques ansioso y perezoso en un mismo diseño. Nuestra propuesta combina la buena concurrencia en situaciones de contención inherente a la política perezosa, con los *commits* completamente paralelos propios de la política ansiosa, y lo hace a un coste

hardware reducido gracias a que trabaja con una granularidad semejante al de los mecanismos básicos de TM: la línea de caché. Se trata de un diseño HTM híbrido y adaptable que mejora en rendimiento a los enfoques estáticos y de políticas prefijadas. En lugar de percibir la contención como una característica de una transacción –i.e. bloque de código–, nosotros la entiende como una característica de los datos accedidos dentro de cada transacción. Nuestra observación de que los datos contendidos conforman una fracción relativamente pequeña del conjunto de escritura de una transacción justifica la decisión de incorporar estructuras hardware para gestionar de forma eficiente dichos datos en el caso común. En el proceso, nuestra propuesta auna las bondades de ambas políticas tradicionales de diseño HTM, con cambios muy modestos en la arquitectura y el protocolo de coherencia. Así, ZEBRA soporta *commits* instantáneos para transacciones que no acceden datos contendidos, y permite la compartición por parte de múltiples lectores y escritores sobre datos contendidos. En esta tesis mostramos tanto cualitativa como cuantitativamente que nuestro diseño puede aprovechar el paralelismo existente mejor que los diseños de política invariable, igualando o superando al diseño de política fija que mejor se adapta a las características de cada carga trabajo: ZEBRA exhibe prestaciones similares a los de un sistema HTM puramente ansioso en aquellos casos en los que la elevada tasa de commits limita el rendimiento, al tiempo que supera claramente tanto a sistemas ansiosos como a perezosos en aplicaciones donde la contención es un factor dominante. En resumen, ZEBRA obtiene la menor desviación sobre el mejor rendimiento medido para un conjunto diverso de cargas de trabajo, corroborando nuestra afirmación de que es un diseño robusto y menos susceptible a condiciones patológicas.

En esta tesis abogamos por la adaptación de las políticas de gestión transaccionales a la naturaleza de los datos, ya que reconocemos que la asunción de que todos los datos accedidos dentro de una transacción poseen las mismas características conduce a soluciones sub-óptimas. No obstante, pese a que la predeterminación de la política en tiempo de diseño puede resultar en un sistema HTM que no rinde bien en ciertos rangos del espectro de cargas de trabajo, la mayor sencillez de implementación de un sistema HTM de política fija hace que sea importante considerar los problemas específicos de cada enfoque y encontrar técnicas para solucionarlos. Así pues, esta tesis investiga los factores limitantes del rendimiento de aquellos sistemas HTM de política fija que utilizan un protocolo de coherencia de directorio distribuido para detectar conflictos entre transacciones sobre una red de interconexión escalable, y desarrolla soluciones que evitan la degradación de prestaciones.

En lo que se refiere a los sistemas HTM ansiosos, proponemos una técnica que

previene la formación de un cuello de botella en el mecanismo de detección de conflictos, que tiene lugar en el controlador de directorio durante situaciones de elevada contención en el acceso a un dato compartido. El esquema propuesto separa el mantenimiento de la coherencia y la detección de conflictos, permitiendo que dicha detección también pueda ser realizada directamente por el directorio y no exclusivamente por las cachés privadas, como viene siendo tradicional. Nuestra propuesta solventa un escenario patológico que afecta los sistemas HTM ansiosos, al conseguir una detección de conflictos independiente de los mecanismos de coherencia y por tanto más rápida y con menores demandas sobre la red de interconexión, lo que a su vez hace posible que el sistema pueda reaccionar y resolver dichos conflictos con mayor celeridad. La evaluación experimental muestra que gracias a nuestra técnica los sistemas HTM ansiosos también logran comportarse de manera eficiente cuando la contención es alta en el acceso a un mismo dato, conduciendo no solo a un menor número de transacciones abortadas, sino también a una menor latencia de acceso a memoria a dichos datos conflictivos. Nuestros experimentos muestran una reducción tanto del tiempo de ejecución como del uso de la red con respecto a un sistema LogTM-SE con firmas perfectas. En particular, la eliminación del cuello de botella en el directorio es responsable de una ganancia en prestaciones muy significativa para aquellas cargas de trabajo que sufren mucha contención sobre un número reducido de líneas de caché. Además, en comparación con sistemas tipo LogTM-SE que utilizan firmas reales con un coste hardware similar al de nuestro diseño, éste reduce la degradación de rendimiento causada por la aparición de falsos positivos ya que prácticamente los elimina en su totalidad. Nuestro novedoso esquema para el mantenimiento de los conjuntos de lectura y escritura transaccionales aprovecha la naturaleza intrínseca del directorio para codificar de forma global dichos conjuntos, asociando cada dirección de memoria con las transacciones que la acceden, en lugar de mantener un conjunto de direcciones –a menudo redundante– en cada núcleo de ejecución. Esto permite una codificación global más eficiente y permite la eliminación de falsos positivos en los propietarios transaccionales de un bloque, al utilizar la propia información de directorio sobre los compartidores de dicho bloque. En definitiva, defendemos la idea de que aumentar el rol del directorio para incluir la funcionalidad de detección de conflictos es una evolución natural de sus responsabilidades dentro de un sistema HTM ansioso.

En el dominio de los sistemas HTM con resolución de conflictos perezosa, esta tesis presenta π-TM, un sistema HTM que aplica el concepto de invalidación pesimista para conseguir combinar la detección temprana de conflictos con la res-

olución perezosa, sin necesidad de lastrar la ejecución transaccional común para salvaguardar la corrección ante posibles condiciones de carrera. π-TM detecta conflictos de forma ansiosa sin caer en la penalización de acceder al directorio en el caso común para garantizar una ejecución correcta, y alcanza un esquema de *commits* verdaderamente escalable al permitir que transacciones no conflictivas puedan completar su ejecución exitosa de forma verdaderamente paralela. En esta tesis mostramos como la información pertinente para la detección de conflictos se encuentra disponible en el tráfico de coherencia generado durante la ejecución de una transacción, y por tanto puede utilizarse para detectar conflictos antes de que la transacción llegue a su fin. Si se utiliza apropiadamente, esta información posibilita un diseño sencillo que soporta *commits* paralelos de transacciones no conflictivas, mientras mantiene el comportamiento optimista de las transacciones perezosas que les permite continuar su ejecución más allá de accesos conflictivos. Desafortunadamente, el trabajo previo en esta dirección que encontramos en la literatura ha introducido otra forma de pesimismo en un escenario más crítico si cabe, forzando que cada nuevo acceso transaccional a una línea de caché deba pasar obligatoriamente por el directorio, lo cual supone una degradación sustancial del rendimiento tal y como demostramos en nuestra evaluación. El sistema π-TM, en cambio, se libra de esta penalización en la mayor parte de escenarios, cuando resulta prudente hacerlo, recurriendo a una forma de pesimismo mucho más liviana que no lastra la ejecución en el caso común. Además, esta tesis también subraya la importancia de incorporar mecanismos de adaptación en el diseño para tolerar características cambiantes en las cargas de trabajo, como forma no sólo de obtener mayor rendimiento sino también de conseguir un sistema más robusto.

Otra contribución importante de la tesis es nuestro análisis del impacto que la utilización de búfers de escritura no coherentes tiene en los sistemas HTM, tanto ansiosos como perezosos. Esta tesis demuestra que el uso de una optimización estructural tan común como los búferes de escritura juega un rol primordial en el rendimiento global de una implementación HTM, revelando que las diferencias de rendimiento observadas entre sistemas con políticas ansiosas y perezosas se reducen sustancialmente. La sorprendente convergencia de ambos enfoques en términos de prestaciones desvelada en esta tesis desmitifica la percepción generalizada de que los sistemas HTM perezosos son la elección más eficiente desde el punto de vista del rendimiento. Con este trabajo demostramos las ineficiencias causadas por el almacenamiento de los valores especulativos en estructuras coherentes como las cachés privadas. Si bien no abogamos por el uso de exclusivo de búferes de escritura para la gestión de versiones de datos en las

transacciones –pues las restricciones de área y energía limitan severamente su utilidad–, sí señalamos la importancia de disponer de este tipo de búferes para dar un soporte TM eficiente en el caso común.

Índice

Resumen en español 3

Resumen 19

Índice 21

Lista de figuras 25

Lista de tablas 29

1 Introducción 31

2 Trabajo relacionado 43

3 Metodología de evaluación 65

4 Un esquema de detección de conflictos basado en el directorio 93

5 ZEBRA: Un sistema HTM de política híbrida adaptable a los datos 127

6 π-TM: Invalidación pesimista para la escalabilidad de los sistemas HTM perezosos 155

7 Influencia de los búferes de escritura en el rendimiento de los sistemas HTM 181

8 Conclusiones y vías futuras 207

Bibliografía 230

A mis padres y hermanos

Agradecimientos

Se cumplen ahora cinco años desde que, pedaleando por Dallas Road frente al océano, decidiese dejar aquella preciosa isla y regresar a Murcia con el fin de comenzar unos estudios de doctorado. Sin duda alguna, en aquel momento no era consciente de lo que significa emprender una carrera investigadora en la Universidad, de los grandes sacrificios que a menudo conlleva, ni tampoco de las enormes satisfacciones que reporta. Ajeno por completo al mar de vivencias que me habría de deparar, me adentré en este largo camino que es la tesis, durante algún tiempo tan oscuro y sinuoso que no era capaz de atisbar su final. Afortunadamente, han sido muchos los que me han acompañado a lo largo de esta aventura, compañeros de viaje con los que he compartido muchas risas y no pocas preocupaciones. Entre todos me han transmitido el ánimo y la motivación para seguir día a día dando los pasos que finalmente me han traído hasta aquí y hasta ti, que lees estas palabras de reconocimiento.

Nada de lo que estás leyendo habría sido posible –literalmente– sin mis padres, Alfredo y Ana María. Gracias a ellos, a la educación que me han brindado, a los valores que me han transmitido y sobre todo al cariño incondicional que me han dado siempre. No tengo palabras para expresarles toda mi gratitud, pero valgan estas líneas como una muestra de mi reconocimiento hacia ellos, por su amor infinito. Junto a mis padres, mis hermanos son los otros grandes responsables de que yo haya llegado hasta aquí, pues ellos han sido el espejo en el que me he mirado. Sin ellos, ni esta tesis existiría ni yo sencillamente sería quien soy. Ellos hicieron con su brillantez y su esfuerzo que nuestro poco común apellido fuese un sinónimo de trabajo bien hecho. Confieso que ser el menor de los hermanos Titos acarreó ciertas ventajas en mis años de colegial, y por qué no decirlo, también una gran responsabilidad, así que espero haber estado a la altura. Gracias, Encarni, Salvi y Alfredo, por contagiarme vuestro apetito insaciable por querer conocer el mundo.

A mis amigos les debo el haber llegado hasta aquí sin perder –completamente– la cordura. Ellos han sido un pilar fundamental para mí todos estos años, me han apoyado en los momentos difíciles y me han ayudado a despejarme cuando el estrés de la tesis más me nublaba la vista. A mis jumillanos, especialmente a Puri, Miguel, Hermo, Carlos y Cati, gracias por estar siempre ahí, dispuestos a escucharme. A Miguel le deberé siempre el privilegio de ser haber sido becario FPU. A mis informáticos, Luisa, Óscar, Ana, Javi y María Ángeles, gracias por confiar siempre en mí, y por aguantar mis agobios en este último tramo de escritura de la tesis. A mis amigos al otro lado del charco, especialmente a Kate, mi inspiración durante tantos años, gracias por compartir conmigo tantas aventuras y regalarme vivencias extraordinarias, por tierras cercanas y lejanas. A todos ellos y los demás amigos que me han estado a mi lado todos estos años, muchas gracias.

Sin lugar a dudas, el sitio donde más tiempo he pasado estos años es la universidad y el apoyo recibido de mis amigos y compañeros del departamento ha sido crucial para hacer estos largos días más amenos. La ayuda de Ricardo, Alberto, Dani, Juan Manuel, Chema y Toni durante los años de doctorado ha sido vital, resolviendo mis dudas y compartiendo risas en los descansos.

Mi estancia en el Barcelona Supercomputing Center en el otoño de 2009 supuso una experiencia muy enriquecedora, tanto en lo profesional como en lo personal. Me alegro mucho de haber estado allí esos cuatro meses y de haber conocido a gente tan especial. Mi agradecimiento es para Adrián Cristal, por darme la oportunidad de visitar el BSC, así como para todos los que conforman el grupo BSC-Microsoft Research.

My stay in Göteborg last fall was most definitely an enriching experience. Thanks to professor Per Stenström for giving me the opportunity to work as a part of his team at Chalmers. I truly learnt a lot from all the people at the CSE department, particularly from my TM colleague and friend Anurag. Thanks Negi for the exciting debates about TM, and the endless discussions about life.

I would like to thank Dr. Rubén González and Dr. M.M. Waliullah for reviewing this thesis and providing useful comments.

Finalmente, quiero agredecer a mis directores, José Manuel y Manolo, el apoyo prestado y la confianza depositada en mí todos estos años. Gracias por darme la oportunidad de realizar la tesis en este departamento y en este campo que tanto me gusta.

Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y
Tecnología de Computadores

Hardware Techniques for High-Performance Transactional Memory in Many-Core Chip Multiprocessors

PhD Thesis

By
José Rubén Titos Gil

Advised by
José Manuel García Carrasco
Manuel Eugenio Acacio Sánchez

Murcia, September 2011

Abstract

Owing to the incessant improvement in process technology, nowadays billions of transistors can be integrated on a single die, providing computer architects with huge amounts of silicon resources. Pressed by power and energy constraints, and instigated by ever-increasing design and verification costs, microprocessor vendors have responded to the crisis of stalled sequential performance by manufacturing chips that contain multiple processing cores, known as chip multiprocessors or CMPs. Most microprocessor road-maps today project rapid growth in the number of cores integrated on chip in an attempt to provide increasing performance through thread level parallelism.

The rise of multicores has brought the problem of effective concurrent programming of such systems to the forefront of computing research, presenting both immense opportunities and enormous challenges. Programmers must change the way they create applications and turn their focus to multi-threaded applications which can take full advantage of the computational resources available in multicore hardware. Unfortunately, concurrent programming is a far more difficult task than sequential programming, because the different threads of a parallel program need to communicate in order to cooperatively carry out the task to completion. A key challenge is guaranteeing correctness while simultaneously maintaining high efficiency and productivity. Traditional multithreaded programming models use low-level primitives such as locks to guarantee mutual exclusion and protect shared data. In this context, the complexity of lock-based synchronization makes parallel programming an error prone task, particularly when fine-grained locks are used to extract more performance.

The trade-off between programming ease and performance imposed by locks remains one of the key challenges to programmers and computer architects of the multicore era. Transactional Memory (TM) is as a conceptually simpler programming model that can help boost developer productivity by eliminating

the complex task of reasoning about the intricacies of safe fine-grained locking. By using transactions to safely access shared data, programmers need not reason about the safety of interleavings or the possibility of deadlocks to write correct multithreaded code. Hence, TM addresses the performance-productivity trade-off by not discouraging programmers from using coarse-grain synchronization, since the underlying system can potentially achieve performance comparable to fine-grained locks by executing transactions speculatively.

In spite of the advantages that software-based approaches to TM offer in terms of overall system complexity, the reality is that fast implementations of transactional programming constructs are necessary for TM to gain widespread usage. Indeed, given the abundance of transistors available in today's chips, one of the priorities for computer architects today is to understand which abstractions can enhance the productivity of parallel software development and then introduce the appropriate hardware support to realize it. Transactions are a good candidate for such an abstraction, and this thesis focuses on the hardware mechanisms that provide optimistic concurrency control with stringent guarantees of atomicity and isolation, with the intent of achieving high-performance across a variety of workloads, at a reasonable cost in terms of design complexity.

This thesis identifies key inefficiencies that impact the performance of several hardware implementations of TM, and proposes mechanisms to overcome such limitations. In this dissertation we consider both eager and lazy approaches to HTM system design, and address important sources of overhead that are inherent to each policy. This thesis presents a hybrid-policy, adaptable HTM system that combines the advantages of both eager and lazy approaches in a low complexity design, by selecting the appropriate policy at the granularity of cache lines.

Furthermore, this thesis investigates the overheads of the simpler, fixed-policy HTM designs that leverage a distributed directory-based coherence protocol to detect data races over a scalable interconnect, and develops solutions that address some performance degrading factors. For eager systems, we propose a mechanism to prevent the directory controller from becoming a bottleneck in the conflict detection mechanism during situations of high contention. For lazy systems with early conflict detection, we present a solution that unburdens transactional execution from the penalty of accessing the directory in the common case to guarantee correctness, while providing true commit parallelism for non-conflicting transactions. It also demonstrates that common structural optimizations such as store buffers play a major role in determining the overall performance of an HTM implementation, bridging the performance gap between eager and lazy designs.

Contents

Extended abstract in Spanish 3

Table of contents in Spanish 11

Acknowledgments 15

Abstract 19

Contents 21

List of Figures 25

List of Tables 29

1 Introduction 31
1.1 The Era of Multicores 31
1.1.1 The Concurrency Revolution 33
1.2 The Challenges of Parallel Programming 34
1.2.1 Drawbacks of Lock-Based Synchronization 35
1.3 The Transactional Abstraction 35
1.3.1 High-Performance Transactional Memory 36
1.4 Thesis Motivation and Contributions 38
1.5 Thesis Organization 41

2 Background and Related Work 43
2.1 Fundamentals of Transactional Memory 43
2.2 Cache Coherence Protocols 46
2.3 Basic Mechanisms in Hardware Transactional Memory 50

2.3.1 ISA Extensions 50
2.3.2 Transactional Book-keeping 51
2.3.3 Data Versioning 52
2.3.4 Conflict Detection 54
2.3.5 Conflict Resolution 55
2.3.6 Transaction Commit 56
2.3.7 Transaction Abort 57
2.4 An Overview of Hardware Transactional Memory Research 58

3 Evaluation Methodology 65
3.1 Simulation Tools 65
3.2 Metrics and Methods 67
3.3 Workloads 69
3.3.1 Workload Characterization 73
3.4 CMP Architecture 75
3.5 Baseline HTM systems 76
3.5.1 Eager-Eager HTM Overview 76
3.5.2 Lazy-Lazy HTM Overview 78
3.6 Validation of Simulation Framework 79
3.6.1 Basic setup 79
3.6.2 Interference from the Operating System 80
3.6.3 Starvation of Writer Transactions 82
3.6.4 Barrier Synchronization Overhead 83
3.6.5 Level-2 Cache Conflict Misses 84
3.6.6 Summary of Changes 86
3.6.7 Relative Performance of Lazy-Lazy HTM Systems 86
3.6.8 Workload Speedup on Single Global Lock 90

4 A Directory-Based Scheme for Detection of Transactional Conflicts 93
4.1 Introduction 93
4.2 Motivation 96
4.2.1 Decoupling conflict detection from cache coherence 97
4.2.2 Reducing traffic generated during stalls 99
4.2.3 Reducing false positives of signatures 101
4.2.4 Avoiding broadcast on L2 misses 102
4.3 Background on Conflict Detection 103
4.3.1 Conflict Detection on Evicted Lines 104
4.3.2 Silent Replacements and Sticky States 105

4.4 Directory-Based Conflict Detection . . . 106
4.4.1 Transactional status . . . 106
4.4.2 Transactional meta-data . . . 107
4.4.3 Conflict detection logic . . . 110
4.4.4 Propagation and update of transactional meta-data . . . 111
4.4.5 Awareness to priority and deadlock detection . . . 111
4.4.6 Clearing transactional meta-data . . . 112
4.4.7 Dealing with races . . . 113
4.4.8 Reducing meta-data propagation . . . 114
4.5 Evaluation . . . 114
4.5.1 Experimental Setup . . . 115
4.5.2 Performance Analysis . . . 117
4.5.3 Traffic Considerations . . . 122
4.6 Concluding Remarks . . . 125

5 ZEBRA: A Data-Centric, Hybrid-Policy HTM System 127
5.1 Introduction . . . 127
5.2 Background . . . 130
5.3 Design and Operation . . . 132
5.3.1 Conceptual Overview . . . 132
5.3.2 Protocol behavior . . . 136
5.3.3 Resetting C bits . . . 140
5.4 Evaluation . . . 140
5.4.1 Experimental Setup . . . 140
5.4.2 Workload Characteristics . . . 142
5.4.3 Performance Analysis . . . 145
5.4.4 Analysis of C-bit reversion mechanism . . . 150
5.4.5 Traffic Considerations . . . 152
5.4.6 Key inferences . . . 153
5.5 Concluding Remarks . . . 154

6 π-TM: Pessimistic Invalidation for Scalable Lazy HTM 155
6.1 Introduction . . . 155
6.2 Background . . . 159
6.3 π-TM: An Adaptable Lazy HTM Design with Parallel Commits . . . 161
6.3.1 Baseline eager detection-lazy resolution . . . 161
6.3.2 π-TM: Pessimistic invalidation of contended lines . . . 163
6.3.3 Adaptable π-TM . . . 164

6.3.4 Protocol support for π-TM 167
6.4 Evaluation 171
6.4.1 Early Conflict Detection Performance 172
6.4.2 Comparison with other designs 177
6.5 Concluding Remarks 179

7 Implications of Store Buffering on the Performance of HTM Systems 181
7.1 Introduction 181
7.2 Background 185
7.3 Buffering speculative updates in L1 caches 188
7.3.1 Lazy HTMs 188
7.3.2 Eager HTMs 192
7.4 Use of store buffers 193
7.5 Evaluation 195
7.5.1 Lazy HTM Results 197
7.5.2 Eager HTM Results 201
7.5.3 Eager vs. Lazy: Relative performance 204
7.6 Concluding Remarks 205

8 Conclusions and Future Ways 207
8.1 Conclusions 207
8.2 Future Ways 210

Bibliography 213

List of Figures

2.1 Lazy version management. 45
2.2 Eager version management. 45
2.3 Policies for conflict detection. 46
2.4 The cache coherence problem [39]. 47
2.5 State transition diagram for a MESI protocol [112]. 49
2.6 SRAM cells augmented with circuitry for flash-clear and conditional flash-clear [19]. 53

3.1 Architecture of the Simics-GEMS simulation framework. 66
3.2 Organization of a tile and a 4×4 tiled CMP. 75
3.3 Hardware overview of baseline EE and LL HTM systems. 77
3.4 Workload scalability on LogTM, with and without OS interference. . . 81
3.5 Workload scalability on LogTM, without writer starvation. 83
3.6 Workload scalability on LogTM, with ideal hardware barriers. 84
3.7 Workload scalability on LogTM after solving L2 conflict misses. 85
3.8 Evolution of scalability for the baseline EE system. 87
3.9 Relative workload scalability of both LL systems. 87
3.10 L2 bank mapping policies. 89
3.11 Performance of EE and LL-STCC systems with L2 mapping policies. . 90
3.12 Workload scalability using a single global lock for synchronization. . . 91

4.1 Example of busy states in a directory protocol state machine. 97
4.2 Cache-based vs. directory-based conflict detection. 99
4.3 Messages generated on a write-read conflict in cache-based vs. directory-based conflict detection. 100
4.4 Block diagram of the new directory organization. 108
4.5 Transactional directory internals. 109

4.6 SRAM cells used to track transactional accessors in the directory . . . 109
4.7 Relative performance of "magic" directory-based vs. ideal cache-based conflict detection. 118
4.8 Transactional cycle breakdown in baseline vs. directory-based schemes. 119
4.9 Relative performance of realistic directory-based vs. cache-based conflict detection. 121
4.10 Effect of false positives on performance for various signature schemes and sizes. 123
4.11 Network message breakdown and network flits. 124

5.1 Behavioral differences between different HTM design points. 130
5.2 ZEBRA – Salient architectural features. 133
5.3 Write handling at the private cache. 135
5.4 ZEBRA – Key protocol actions. 137
5.5 Support for new transitions at L1 (left) and L2-directory (right) controllers. 139
5.6 Relative sizes of conflict sets for STAMP applications. 142
5.7 Proportion of C-bit lines over time. 143
5.8 Contention over time. 144
5.9 ZEBRA vs. fixed-policy HTM designs. 145
5.10 ZEBRA vs. idealized lazy designs. 146
5.11 Deviation from best observed performance. 146
5.12 ZEBRA – Policy distribution at commit. 147
5.13 Microbenchmark analysis. 151
5.14 C-bit microbenchmark analysis. 152
5.15 Network traffic. 152
5.16 Normalized network message count. 153

6.1 Problems with detecting conflicts early in lazy designs. 158
6.2 Key protocol action: Baseline eager conflict detection in lazy HTMs. . 162
6.3 Key protocol actions: Conflict detection in π-TM. 164
6.4 High level behavior of π-TM. 165
6.5 Hardware structures that enable mode switches. 167
6.6 Supporting transitions at L1. 168
6.7 Access handling at directory. 170
6.8 A comparison of early conflict detection schemes. 172
6.9 Network messages. 173
6.10 Network flits. 173

6.11 Miss rates. 174
6.12 Adaptable π-TM: Mode distribution for transactions 176
6.13 π-TM: Comparison with other designs. 177

7.1 Store buffer model from this chapter. 183
7.2 *Downgrade miss*: Redundant cache-state changes when a transaction eventually commits. 190
7.3 *Contamination miss*: Invalidations that could be avoided using a store buffer. 192
7.4 Lazy design points: L1 data cache miss rates. 197
7.5 Lazy design points: Downgrade and contamination misses. 198
7.6 Lazy design points: execution time breakdown. 199
7.7 Lazy design points: Commit and arbitration delays. 200
7.8 Eager design points: execution time breakdown 202
7.9 Eager design points: transactional time breakdown 203
7.10 Policy performance comparison: eager vs. lazy 204

List of Tables

3.1 Components: Execution time breakdown 68
3.2 Benchmarks and inputs used in the simulations. 70
3.3 STAMP Workload Characteristics. 70
3.4 Count, read and write set sizes for the transactions in STAMP. 73
3.5 System parameters. 76

4.1 HTM configurations evaluated in Chapter 4. 115
4.2 Specific parameters of the DirCD system. 116
4.3 Inputs to the additional workloads used in Chapter 4. 116
4.4 Number of aborted transactions for the three systems in Figure 4.7. . . 119

5.1 Specific parameters of the ZEBRA system. 142
5.2 HTM configurations evaluated in Chapter 5. 142
5.3 ZEBRA – LWB and OVB utilization. 148

6.1 HTM configurations evaluated in Chapter 6. 171
6.2 Contention statistics. 175
6.3 Mode-switch threshold values. 175

7.1 Store buffer configurations. 195

Chapter 1

Introduction

The 2000s have witnessed a major inflection point in the history of computing. For the past four decades, Moore's law has fuelled an extraordinary boost in processor performance by doubling the on-chip transistor count approximately every two years: smaller transistor feature sizes allowed higher and higher clock frequencies, and computer architects made use of the newly available silicon resources to design more and more sophisticated pipelines that better exploit the instruction level parallelism (ILP) present in sequential programs. The synergistic combination of advances in semiconductor technology and micro-architectural techniques has sustained decades of exponential growth in uniprocessor performance, an unprecedented achievement which has recently come to an end. A fundamental paradigm shift towards parallel architectures is taking place in front of our eyes, bringing about new opportunities and challenges that make this an exciting time to be in Computer Architecture.

1.1 The Era of Multicores

Transistor scaling has been the leading force that has driven the rapid growth in microprocessor performance for the past decades. In every technology generation, transistor integration doubled, circuits were faster, and power consumption stayed the same [24]. With twice as many transistors available on each new generation, architects had plenty of resources to create more complex micro-architectural techniques, include larger and more sophisticated cache hierarchies as well as exploit transistor speed to increase frequency. Unfortunately, as

transistors approach atomic dimensions, new challenges have appeared that make it increasingly difficult to continue this virtuous trend. Despite continuing miniaturization, issues such as increased threshold voltage to control leakage, and limited supply-voltage scaling, decrease the performance benefits of transistor scaling [48].

While Moore's law is still alive and well [61, 111] –we can expect transistor counts to keep doubling at least for a few more generations to come–, power and cooling concerns have now become first-class limiting factors that restrict further raises in the clock frequency. This has caused the abandonment of micro-architectures with very-deep instruction pipelines [62]. Low power efficiency, high design complexity and expensive verification costs have also brought to a stop the efforts to design very aggressive wide issue superscalar processors with large instruction windows, despite the fact that some amount of ILP still remains to be exploited. Even the investment of the on-chip real state in ever larger caches is reaching the point of diminishing return [42]. Consequently, it has become increasingly difficult to continue to improve the performance of sequential processors [131] due to what has become to be known as *the three walls*: Power, Memory, and ILP Walls [12].

In response to stalled sequential performance, power and energy efficiency constraints and sky-rocketing design and verification costs, industry has found a way out of the looming crisis by manufacturing chips that include more than one processor, connecting them through a shared memory [50]. These novel single-chip, parallel computers are known as "chip multiprocessors" (CMPs) or "multicore" systems[1]. IBM pioneered the release of a dual-core processor in 2002 [135], though it was not until 2005 when major vendors such as Intel and AMD began introducing their multicore products into the personal computing market [1, 66]. Most chip-makers today are producing microprocessors with two to eighteen processing cores [23, 47, 125, 129]. In less than a decade, multicores have become ubiquitous, powering a wide span of systems that range from server [51], desktop, and laptop computers [2, 126], to gaming devices [76] and smart phones [10]. Road-maps project rapid growth in the number of cores integrated on chip in an attempt to provide increasing performance through thread-level parallelism (TLP). Some researchers predict that future chips could contain thousands of cores [12]. Furthermore, power and energy efficiency constraints have not only been responsible for this transition to the multicore era,

[1]The term *manycore* is also employed when the number of processing cores is large.

but are now also forcing hardware architects to move from complex multicore processors to simplified manycore processors [84].

1.1.1 The Concurrency Revolution

This shift to mainstream parallel architectures has brought the problem of effective concurrent programming of such systems to the forefront of computing research, presenting both immense opportunities and enormous challenges. The rise of multicores is quickly making parallel programming become widespread, pushing programmers towards an unfamiliar programming paradigm. Despite the fact that multiprocessor systems have existed for a long time, multi-threaded software development has not been much of a focus in mainstream software development. Instead, multiprocessors were of interest only to the small community of supercomputing, and so was parallel programming, which was mostly ignored by software vendors, and not widely investigated nor taught. As a matter of fact, most software development over time has been predicated on single-core hardware, and the collective knowledge of software developers across organizations has been based primarily on single processor hardware platforms.

Now that the *free lunch* is over [131], software developers must change the way they create applications to fully leverage multicore hardware. By turning their focus to multi-threaded applications, developers can take full advantage of the newly available computational resources and deliver software that meets the demands of the world. The problem is that writing efficient and correct parallel code is a difficult task that involves orchestrating the concurrent execution of the parts to improve performance while at the same time guaranteeing correctness. Complex and hard-to-find, software defects unique to multi-threaded applications such as race conditions and deadlocks can quickly derail a software project [31]. Software engineering tools have yet to simplify the programming for these shared-memory architectures in order to make the new hardware resources accessible to the common programmer. In order to avert a software crisis, developers must adapt and improve such tools to make them better suited for parallel multicore software development [144]. The reality is that software has not matured enough to take advantage of the number of cores that are already available in today's systems, and the vast majority of applications are still single-threaded [54].

1.2 The Challenges of Parallel Programming

Concurrent programming is a far more challenging task than sequential programming: A parallel program is undoubtedly more difficult to design, write, and debug than its sequential counterpart. Orchestrating the concurrent execution of the parts to improve performance while at the same time guaranteeing correctness is by no means an easy task. Parallel software is harder to design than sequential or single-threaded software because parallelism cannot be abstracted away [8]: Parallelism is not a local property of software, but requires restructuring code and data in often counter-intuitive ways. Designing algorithms that can be split into parallel tasks, balancing the workload among the available processors, or communicating and managing shared data between different processors are only some of the many factors that make parallel programming a complicated endeavour. Programmers need to reason carefully about possible interactions of their threads when running concurrently, and not doing so may result in programs that are incorrect, perform poorly, or both. To add insult to injury, parallel programs are very hard to debug due to the combinatorial explosion of possible execution orderings: Parallel programs often produce non-deterministic results, making it harder to prove programs correct, and their bugs are often elusive and notoriously difficult to find and fix, because of the difficulty to reproduce the exact same execution (i.e. interleaving of threads, etc.) that leads to a race.

The different threads of a parallel program need to communicate in order to carry out the task cooperatively to completion. For this matter, two popular types of general-purpose communication abstractions exist, which provide a link between the software (programming model) and the hardware (physical implementation). Threads can exchange information by sending messages (*message passing* model), or by merely accessing and modifying shared memory locations (*shared memory* model). This thesis focuses on shared memory, as is widely regarded as a more intuitive model than message passing for the development of parallel programs [72], and nowadays is the prevalent model in most CMPs [58]. By offering a single physical view of the memory to all processors, this model makes the move into the parallel realm less daunting to sequential programmers, as they are well accustomed to such abstraction.

In the context of shared memory architectures where concurrent tasks process shared data, guaranteeing correctness while maintaining efficiency and productivity is one key challenge. Parallel thread execution requires synchronization for accessing shared data. Programmers are responsible for ensuring that concurrent accesses to shared data structures are correct, and often rely on mutual exclusion

mechanisms to protect these critical sections, so that no more than one thread can simultaneously enter the same critical section and access the same shared data.

1.2.1 Drawbacks of Lock-Based Synchronization

Traditional multi-threaded programming models use low-level primitives such as locks to guarantee mutual exclusion. Unfortunately, the complexity of lock-based synchronization makes parallel programming an error prone task, particularly when fine-grained locks are used to extract more performance. At one end, heroic programmers seeking performance try to minimize the amount of resources (data objects) that are protected by the same lock, so that different threads accessing different data do not have to serialize their execution unnecessarily, thus enabling maximum concurrency. However, the use of fine-grain locks adds more programming complexity, since programmers must be careful to acquire them in a fixed, predetermined order so as to avoid deadlocks. At the other end, common programmers seeking productivity (correctness) choose to reduce the complexity of reasoning (i.e. likelihood of deadlock) by using fewer locks with coarser granularity, where each lock is responsible for protecting larger critical section, at the cost of sacrificing performance. Though programmers can also include deadlock detection mechanisms in their programs, to try and recover from deadlocks, this alternative also adds substantial complexity.

As if deadlocks were not enough, locking brings about other undesired situations like priority inversion (when a high priority thread is unable to acquire a lock because a lower priority thread is holding it), convoying (when a lock holder is de-scheduled from execution, impeding others to progress) and lack of fault tolerance (when a lock holder modifies data and then crashes, causing the whole program to fail). Furthermore, locking break the abstraction principle, as programmers using a module need to be aware of the locks it uses, to ensure that the program still follows the predetermined locking order that prevents deadlock. Therefore, locks jeopardize the code composability property, as two individually correct modules can deadlock when combined together.

1.3 The Transactional Abstraction

The trade-off between programming ease and performance imposed by locks remains one of the key challenges to programmers and computer architects of the multicore era. Transactional Memory (TM) [57, 60] has been proposed

as a conceptually simpler programming model that can help boost developer productivity by eliminating the complex task of reasoning about the intricacies of safe fine-grained locking. TM inherits the concept of *transaction* from the database community, and applies it to the domain of shared-memory programming in an attempt to simplify the task of thread synchronization. Transactions in the multi-threaded programming world are blocks of code that are guaranteed to be executed atomically and in isolation with respect to all other code. At a high level, the programmer or compiler annotates sections of the code as atomic blocks or transactions. The underlying system then executes these transactions speculatively in an attempt to exploit as much concurrency as possible. TM systems generally employ an optimistic approach to concurrency control in order to let multiple transactions execute in parallel, while still preserving the properties of atomicity and isolation. Therefore, the TM system attempts to make best use of available concurrency in the application while guaranteeing correctness. By using transactions to safely access shared data, programmers need not reason about the safety of interleavings or the possibility of deadlocks to write correct multi-threaded code. Hence, TM addresses the performance-productivity trade-off by not discouraging programmers from using coarse-grain synchronization, since the underlying system can potentially achieve performance comparable to fine-grained locks by executing transactions speculatively. In addition to addressing such critical trade-off, TM addresses other limitations of lock-based synchronization. Transactional code is robust in the face of both hardware and software failures, as the system can always rollback the speculative updates to its pre-transactional state in case a thread crashes inside a transaction. Unlike locks, transactions are composable, and they can be safely nested without any risk of deadlocks [17].

1.3.1 High-Performance Transactional Memory

Transactions are a promising abstraction that could ease parallel programming and make it more accessible to the common programmer. Transactional semantics can be entirely supported in software, hardware, or using a combination of both. According to this, we can classify TM systems into software transactional memory (STM), hardware transactional memory (HTM), and hybrid transactional memory systems. STM implementations [44, 59, 83, 120] allow running transactional workloads on existing systems without requiring special hardware support, providing a great degree of flexibility at little cost. Unfortunately, implementing the necessary mechanisms entirely in software imposes too high an overhead

and thus STM systems do not fare well against traditional lock based approaches when performance is important. For this new paradigm to be a viable alternative to locks, the key mechanisms that provide transactional semantics must be implemented at the architectural level.

Hybrid TM systems [14, 26, 41, 75, 123, 134] attempt to combine both the speed and flexibility by using simple hardware to accelerate performance-critical operations of an STM implementation. In this way, hybrid implementations of TM rely on some kind of software intervention to execute transactions, though they minimize the overheads of providing transactional semantics in comparison to a software-only solution. Hybrid TM models use the STM as a backup to handle situations where the hardware cannot execute the transaction successfully [57].

Transactional semantics can also be supported largely in hardware [9,27,29,55, 88, 92, 153], allowing for good performance with varying degrees of complexity which change considerably from one HTM proposal to another, depending on what kind of transactions the TM system is capable of committing without resorting to fallback mechanisms. Simple HTM schemes [30,36,60] adopt a "best-effort" solution that cannot not guarantee that all transactions will eventually commit successfully using hardware support alone, mostly because of the limitations imposed by the hardware structures involved. More sophisticated HTM proposals [9, 55, 92] address this limitation in transaction size, guaranteeing that certain "bounded" transactions can be entirely executed in hardware, no matter the transaction's footprint. These proposals typically behave in the same way as best-effort ones as long as hardware structures are sufficient, and then fall back to additional hardware mechanisms to maintain transactional properties on resource overflow. However, neither bounded nor best-effort solutions can commit transactions that encounter events that are too complicated to handle in hardware, like context switches, page faults, I/O, exceptions or interrupts [63], and in such circumstances the transaction is invariably aborted. Even more elaborated HTM schemes have been designed [9,107] to handle all transactions in hardware, ensuring that the same transaction is not indefinitely aborted because of its size, duration or other events it may encounter. Unfortunately, the complexity of these "unbounded" HTM designs makes them too costly for processor manufacturers to consider them in practice.

1.4 Thesis Motivation and Contributions

In spite of the advantages that software and hybrid TM systems offer in terms of overall system complexity, the reality is that fast implementations of transactional programming constructs are necessary for TM to gain widespread usage. Some processor manufacturers have already ventured into the inclusion of hardware support for TM in their latest designs [30,36,47]. Though it never became commercially available, the *Rock* processor [30] developed at Sun Microsystems was the first general-purpose chip multiprocessor with best-effort HTM capabilities. IBM has recently unveiled its Blue Gene/Q processor [47], which has become the first commercial chip to ship with TM support. Indeed, given the abundance of transistors available in today's chips, one of the most important challenges for computer architects is to understand which abstractions can enhance the productivity of parallel software development and then introduce the appropriate hardware support to realize it [42]. Transactions are a good candidate for such an abstraction, and this thesis focuses on the hardware mechanisms that provide optimistic concurrency control with stringent guarantees of atomicity and isolation, with the intent of achieving high-performance across a variety of workloads, at a reasonable cost in terms of design complexity. Some researchers argue that, for TM to become a successful programming model, the best approach for implementing transactional memory is the hybrid approach that uses a hardware-software solution [132]. The intent of this thesis is not to determine which design point is preferable, whether best-effort, bounded, unbounded or a hardware-software combination. Rather than determining the right degree of TM support that should be placed in hardware, in this thesis we aim to investigate the hardware mechanisms themselves, identify their limitations and propose adequate solutions to overcome them while keeping complexity low. Our goal is not to provide complete HTM solutions, but instead give important insights on the performance implications that different design choices have, when critical TM mechanisms are implemented in hardware.

This thesis identifies key inefficiencies that impact the performance of several hardware implementations of TM, and proposes mechanisms to overcome such limitations. In this dissertation we consider both eager and lazy approaches to HTM system design, and address important sources of overhead that are inherent to each policy. This thesis argues for a hybrid-policy, adaptable HTM design that combines the advantages of both eager and lazy approaches, by selecting the appropriate policy at the granularity of cache lines. We also investigate the overheads of the simpler, fixed-policy HTM designs that leverage a distributed

directory-based coherence protocol to detect data races over a scalable interconnect, and we develop solutions that address some performance degrading factors. For eager systems, we propose a mechanism to prevent the directory controller from becoming a bottleneck in the conflict detection mechanism during situations of high contention. For lazy systems with early conflict detection, we present a solution that unburdens transactional execution from the penalty of accessing the directory in the common case to guarantee correctness, while providing true commit parallelism for non-conflicting transactions. Furthermore, we demonstrate that common structural optimizations such as store buffers play a major role in determining the overall performance of an HTM implementation, bridging the performance gap between eager and lazy designs.

The main contributions of this thesis are summarized next:

- A *directory-based scheme for conflict detection* that decouples conflict detection from cache coherence at the directory level in order to overcome pathological situations that degrade the performance of an eager HTM system in highly contended workloads. In this work we demonstrate that the traditional cache-based approach to conflict detection introduces several sources of inefficiency when used in the context of a directory protocol, and show how under situations of high contention the directory becomes a bottleneck. Our alternative solution moves transactional bookkeeping from caches to the directory, introducing separate hardware module that acts as conflict controller and works independently of the coherence controller, leaving the protocol largely unmodified. In comparison to state-of-the-art eager HTM systems based on signatures, our proposal is not only capable of dealing with contention more efficiently, but it also minimizes the performance degradation of false positives for signatures of similar hardware cost, as well as reduces the network traffic generated by conflict detection.

- A *data-centric, hybrid-policy HTM design* that introduces policy flexibility in hardware, selecting the most appropriate policy on a per-cache-line granularity. HTM systems, in prior research, had either fixed policies of conflict resolution and data versioning for the entire system or allowed a degree of flexibility at the level of transactions. Unfortunately, this resulted in susceptibility to pathologies, lower average performance over diverse workload characteristics or high design complexity. Recognizing the fact that contention is more a property of data rather than that of an atomic code block, we develop an HTM system that allows selection of versioning and conflict resolution policies at the granularity of cache lines. We discover

that this neat match in granularity with that of the cache coherence protocol results in a design that is very simple and yet able to track closely or exceed the performance of the best performing policy for a given workload. It also brings together the benefits of parallel commits (inherent in traditional eager HTMs) and good optimistic concurrency without deadlock avoidance mechanisms (inherent in lazy HTMs), with little increase in complexity.

- An *HTM design with pessimistic invalidation that enables scalable lazy commits*. In spite of allowing better utilization of available concurrency, lazy HTM poses challenges at commit time due to the requirement of en-masse publication of speculative updates to global system state. Early conflict detection can be employed in lazy HTM designs to allow non-conflicting transactions to commit in parallel, though it has not been utilized effectively so far. Prior work in the area burdens common-case transactional execution severely to avoid some relatively uncommon correctness concerns. In this work we explore this problem, quantify its severity and develop an early conflict detection - lazy conflict resolution design. The design highlights how, with modest extensions to existing directory-based coherence protocols, information regarding possible conflicts can be effectively used to achieve true parallelism at commit without burdening the common-case. We leverage the observation that contention is typically seen on only a small fraction of shared data accessed by coarse-grained transactions. Pessimistic invalidation of such lines when committing or aborting, therefore, enables fast common-case execution.

- An *analysis of the impact of store buffering techniques in HTM performance*, which shows how straight-forward optimizations that are commonly found in modern processors play a major role in determining the overall performance. HTM systems have been studied extensively along the dimensions of speculative versioning and contention management policies. The relative performance of several design policies has been discussed at length in prior work, yet, the impact of simple structural optimizations like write-buffering had not been investigated, and performance deviations due to the presence or absence of these optimizations remains unclear. This lack of insight into the effective use and impact of these interfacial structures between the processor core and the coherent memory hierarchy forms the crux of the problem we study in this work. Our study of both eager and lazy conflict resolution mechanisms in a scalable parallel architecture notes a remarkable convergence of the performance of these two diametrically op-

posite design points when write buffers are introduced and used well to support the common case. The insights, related to the interplay between buffering mechanisms, system policies and workload characteristics, contained in this work clearly distinguish gains in performance to be had from write-buffering from those that can be ascribed to HTM policy.

- We have evaluated all the proposals presented in this thesis in a common framework using full-system simulation on a set of transactional benchmarks commonly employed in the TM literature. We have found that our proposals improve the performance of transactional applications in comparison to several state-of-the-art HTM designs.

All the contributions of this thesis have been published or are currently being considered for publication in international peer reviewed conferences [94, 95, 137–139, 141] and journals [136, 140].

1.5 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 reviews the fundamentals of transactional memory, and explores the previous work on hardware implementations of TM.
- Chapter 3 describes the methodology used in the evaluation of the different approaches presented in this thesis. This chapter discusses the architecture, tools, workloads and metrics used in the evaluation, as well as a thorough performance characterization of the baseline HTM systems used throughout the thesis.
- Chapter 4 presents a mechanism for the acceleration of eager HTM systems that are built on top of a directory-based coherence protocol.
- Chapter 5 introduces a data-centric, hybrid-policy HTM design that neatly combines the benefits of both eager and lazy approaches to HTM, at a very modest cost in complexity.
- Chapter 6 describes a design that applies the idea of pessimistic invalidation to enable scalable commits in a lazy HTM system that is capable of detecting conflicts early.

- Chapter 7 analyzes the implications of store buffering in the performance of both eager and lazy HTMs.
- Chapter 8 summarizes the main conclusions of the thesis and points out future lines of work.

CHAPTER 2

Background and Related Work

2.1 Fundamentals of Transactional Memory

Transactional Memory (TM) [57,60] has been proposed as an easier-to-use programming model that can help developers build scalable shared-memory data structures, relieving them from the burdens imposed by fine-grained locking. Under the TM model, the programmer declares *what* regions of the code must appear to execute atomically and in isolation (called transactions), leaving the burden of *how* to provide such properties to the underlying levels. The TM system then executes optimistically transactions, stalling or aborting them whenever real run-time data conflicts occur amongst concurrent transactions. The TM programming model thus replaces explicit synchronization mechanisms like locking with a more declarative approach whose aim is to decouple performance pursuit from programming productivity. The key abstraction that TM incorporates at the programming language level is the `atomic` construct, which programmers use to delimit critical sections (accesses to shared data), structuring their code into *atomic blocks* or transactions. A transaction is said to *commit* when it completes its execution successfully –confirming its speculative updates to shared memory–, while it is *aborted* or *squashed* when some condition occurs –e.g. a race with a concurrent transaction– that impedes its completion with success. To guarantee race-free execution of a transactional multi-threaded application, TM implementations must satisfy two basic properties, namely atomicity and isolation, which are inherited from the database domain.

The *atomicity* property dictates that a transaction is either executed to com-

pletion or not executed at all. If the transaction successfully commits, all of its changes are made globally visible at once. Otherwise, if the transaction aborts, all its tentative updates are discarded in order to revert the machine to its pre-transactional state, as if the transaction was never executed. To the outside world, this means that a transaction appears as an indivisible operation that cannot be partially executed. On its part, the *isolation* property requires that the intermediate (speculative) state of a partially completed transaction must remain hidden from other code. By satisfying these properties, transactions appear to execute in some serial global order, i.e. committed transactions are never observed by different processors to execute in different orders. To provide these properties, the TM system must implement two basic mechanisms, namely data version and conflict management. The policy and implementation of these two mechanisms constitutes the two fundamental dimensions of the TM design space.

Version management handles the simultaneous storage of both speculative data (new values that will become visible if the transaction commits) and pre-transactional data (old values retained if the transaction aborts). Only one of the two values can be stored *in-situ*, i.e. in the corresponding memory address, while the other needs to be placed somewhere else. The data versioning policy dictates how the system handles the storage of both versions, and it constitutes a major design point of the system. Depending on which value, old or new, gets to stay "in place" during the course of the transaction, the data version management policy can be classified as eager or lazy. Lazy versioning keeps old values *in-situ* until the commit phase, buffering speculative updates "on the side" in the meantime, as shown in Figure 2.1. Only if the transaction commits, old values are overwritten with the new ones. Since the old values stay in place, a system with lazy versioning can get rid of an aborted transaction quickly, simply by discarding the speculative values. In contrast to the lazy policy, an eager approach to versioning uses a per-thread *transaction log* to backup the old value of a memory localtion prior to each write, and then updates the memory location with the new value, as depicted in Figure 2.2. This policy makes commits fast as new values are already "in place", but leads to expensive aborts that require unrolling the log in order to restore each tentatively modified memory location with its original content.

When two concurrent transactions access the same memory location, and at least one of the accesses is a write operation, we say that there is a *conflict* or *race* between them. All TM systems implement a *conflict management* mechanism to detect and resolve such conflicts. For this purpose, the data both read and written by each transaction must be tracked. The set of data addresses that a transaction

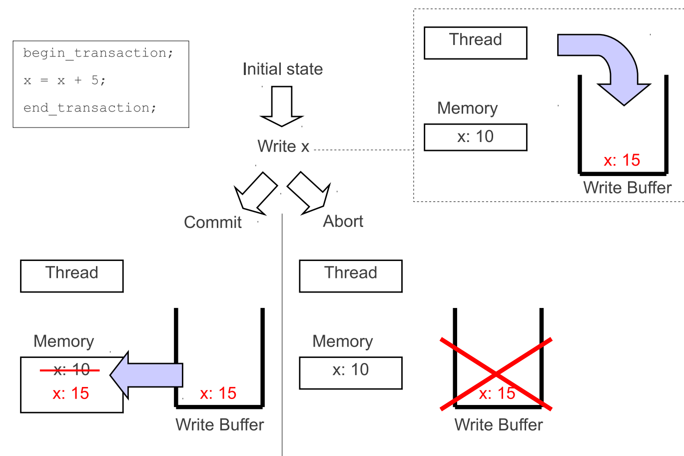


Figure 2.1: Lazy version management.

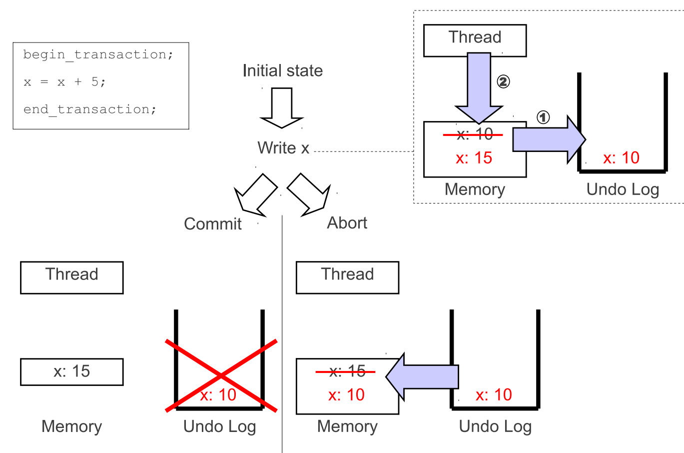
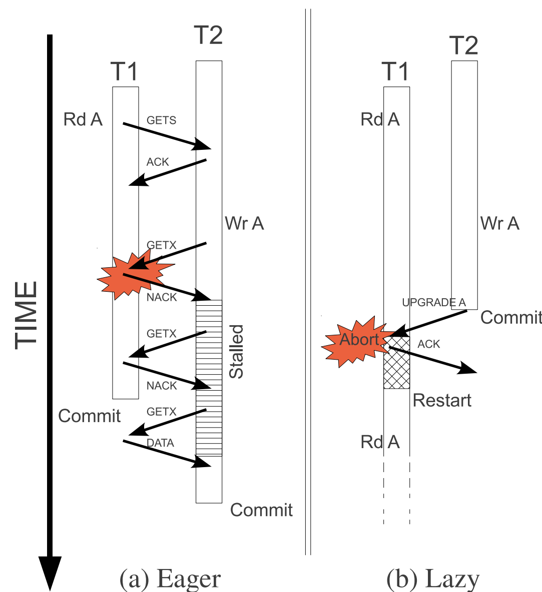


Figure 2.2: Eager version management.

modifies during its execution is known as *write set*. Similarly, the *read set* refers to the group of memory locations read by the transaction. In these terms, a conflict between two concurrent transactions happens when a transaction's write set overlaps with other concurrent transactions' read or write set. Depending on the meta-data information used for transactional book-keeping, conflict detection can take place at different levels of granularity, from objects, to cache lines to word or even byte-level addresses.

Strategies for conflict detection vary depending on when a processor examines



(a) Eager (b) Lazy

Figure 2.3: Policies for conflict detection.

the book-keeping information of its read and write sets. Figure 2.3 shows the two policies. In systems with eager detection –sometimes also referred to as pessimistic–, conflicts are detected as soon as they happen, i.e. on every individual memory access. In the opposite approach, called lazy or optimistic conflict detection, this check is delayed until transaction commit, and the resolution is generally on a committer-wins scheme. The committer transaction publishes its write set to the rest of the system, so that every other transaction can check against its read and write sets, and proceed to abort if necessary.

2.2 Cache Coherence Protocols

Cache coherence is a key hardware design concept and is a necessary part of our intuitive notion of the shared-memory abstraction [39]. Memory provides a set of locations where values are stored, and when a location is read it should return the latest value written to that position. This is how a value is communicated from the point in a program where it is calculated to other points where it is used, whether it is by the same or other threads, in a single-threaded or in multi-threaded program. It simply constitutes the fundamental property of the memory abstraction.

In shared-memory multiprocessors, communication amongst threads occurs implicitly as a result of conventional loads and stores. All processing cores read and write to the same shared address space and they eventually observe the updates by other cores, according to a particular memory consistency model [3]. Although all processors logically access the same address space, each core in a CMP usually contains one or several levels of private cache [112]. Cache hierarchies are crucial to bridge the gap between processor and memory speeds, reducing the average memory access time and bandwidth requirements by taking advantage of locality in memory accesses. Caches work by keeping a local copy of locations that the processor is likely to access soon, thereby saving the processor from having to access main memory directly. Since processors store data in their private caches to take advantage of the locality of memory accesses, several copies of those memory blocks are held in different caches at the same time. In these circumstances, if a processing core modifies a cache line stored in its private cache without notifying other cores that may have a copy of the line, they could be accessing different values for the same data, resulting in data incoherence. This problem is shown in Figure 2.4.

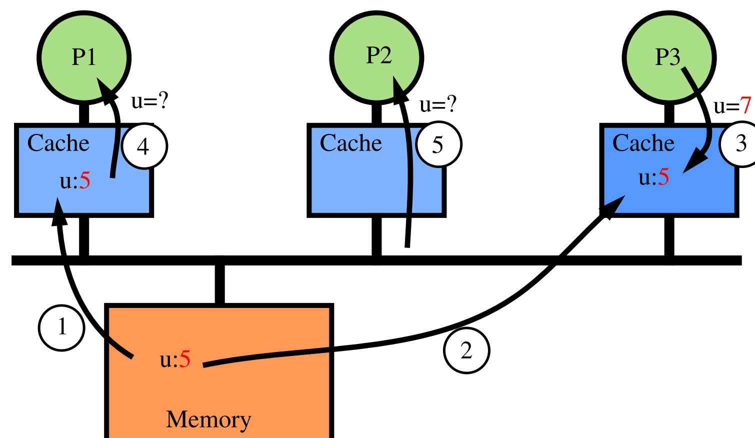


Figure 2.4: The cache coherence problem [39].

The shared memory abstraction can be provided by hardware, software, or some combination of both. One of the key mechanisms that support shared memory at the hardware level is the cache coherence protocol, which is responsible for tracking the multiple copies of each cache line that may exist in the different private caches, and ensuring that all processors see a consistent view of memory. Coherence protocols implemented in hardware make the existence of

caches transparent to the software levels. The coherence protocol is responsible for ensuring that writes to shared memory are eventually made visible to all cores, and writes to the same memory location appear to be seen in the same order by all processors [39]. There are several approaches to solve the cache coherence problem in hardware [128]. Protocols that invalidate other cached copies on a write are called invalidation-based protocols [52], while those that propagate the result of the write operation to all the copies are called update-based protocols [91]. The former are a more attractive choice for architects and have been implemented in most modern cache-coherent multiprocessor systems, as they place much lighter demands on the interconnect.

Up to now, proposed HTM systems rely on invalidation-based protocols in order to detect conflicts, and therefore deserve further attention as they are an important topic related to this thesis. Invalidation-based protocols enforce the following invariant at any point in time [112]: A cache line can be written at most by one core, or it can be read by multiple cores. This means that before a processor can write a line in its private cache, first it must acquire write permissions for that line. Because the coherence protocol only grants write permissions to one cache at a time, this line has to be previously invalidated (revoking read permission) from the other caches. Similarly, if a processing core wants to read a cache line that is found in a remote cache with write permission, the coherence protocol must previously revoke such exclusive rights before granting read-only access to the new requester.

The coherence protocol design space spans several alternatives, depending on the states of the lines stored in the private caches: Modified (M), Exclusive (E), Shared (S) and Invalid (I). Each state represents different access permissions for a line. The MESI states are most common in commercial multiprocessors [57]. However, some systems also support the Owned state (O). Coherence protocols are named after the states that a cache line can be in: MESI, MSI, MOESI, etc. Figure 2.5 shows the state transition diagram for the MESI cache coherence protocol.

A line in M state can be both read and written by the local core, and it indicates the data has been modified (i.e. the copy in the shared level is stale). A line can also be in E state, which is similar to M in terms of permissions, with the only difference that the data is clean (the copy in the shared level is not stale). If a line is in E or M state in a given cache, no other caches can contain a copy of it, or in other words, that cache has *exclusive* access to it. Lines in S state contain valid data that has not been modified and can be read, allowing multiple S copies of the line to coexist in different private caches with *shared* access, but no processor

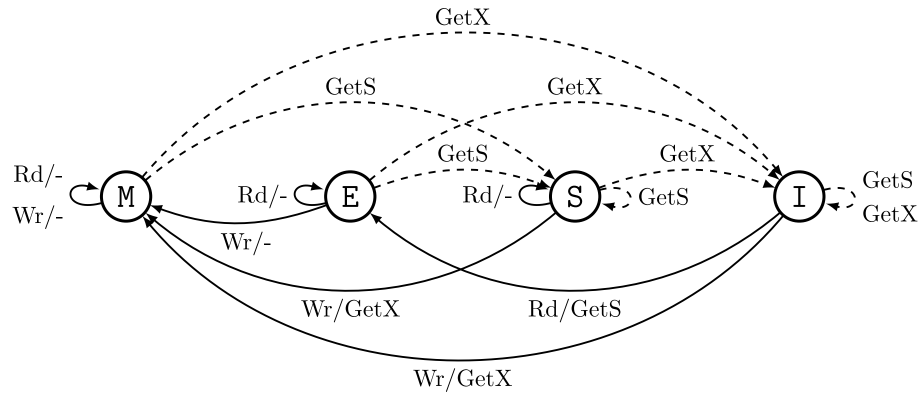


Figure 2.5: State transition diagram for a MESI protocol [112].

has permission to modify it. Finally, the data can be in I state indicating that the processor must request both data and permissions before it can read or modify it.

Buses allow for the simplest solution to the cache coherence problem. *Snooping-based* coherence protocols leverage the total order property of such kind of interconnect to maintain coherence by extending the requirements on the cache controller. As each device connected to the bus can observe every bus transaction, a cache controller can snoop the memory transactions of others, and interpret them so as to maintain its cache in a coherent state. Loads and stores are implicitly used to keep the caches coherent, as the bus serializes requests from different cores to the same cache line, maintaining consistency. Unfortunately, buses do not scale in terms of both area requirements and power consumption, and other solutions are necessary to maintain cache coherence over large-scale systems.

Systems with scalable, point-to-point interconnects, such as the many-core tiled CMPs considered in this thesis, employ *directory-based* protocols to maintain coherence over an unordered network. The coherence information (state, bit-vector of sharers) of each privately cached line can be kept in a centralized directory, or it can be distributed across the nodes of the system. In the context of the multicore architecture assumed in this thesis, L1 private caches are kept coherent through a distributed directory protocol. Each cache line is assigned a *home* tile that keeps the directory entries for the lines mapped to its shared L2 cache bank. The directory acts as serialization point for the requests issued by several cores for a given line. Cache misses are resolved by sending the request to the corresponding home L2 cache bank, whose controller determines when it must be processed and then performs the coherence actions that are necessary to

satisfy the miss. These actions may involve forwarding the request to the cache that must provide the data block, or sending invalidation messages to the sharers in case of a write miss.

2.3 Basic Mechanisms in Hardware Transactional Memory

HTM systems must identify memory locations for transactional accesses, manage the read-sets and write-sets of the transactions, detect and resolve data conflicts, manage architectural register state, and commit or abort transactions [57].

2.3.1 ISA Extensions

Identifying transactional boundaries is accomplished by extending the instruction set architecture (ISA). All HTM implementations introduce a pair of new instructions, i.e. "begin transaction" and "commit transaction", to delimit the scope of a transaction. On the one hand, the execution of the "begin transaction" instruction causes the processor to enter into "transactional mode" (usually setting some bit in the status register) and perform some common actions related to the initialization of the basic transactional mechanisms, like checkpointing the architectural registers to a shadow register file. The architectural registers and memory combined form the precise state of the processor, and therefore the register state also needs to be restored to a known precise state in case of abort. The operation of creating a shadow copy of the architectural registers at the start of a transaction is rather straightforward and can often be performed in a single cycle. On the other hand, the "commit transaction" instruction attempts to confirm the speculative updates of the transaction by publishing them to the rest of the system, and it returns the processor to non-transactional state if successful, discarding the register checkpoint.

The most straightforward step to identify transactional accesses is to leverage these two instructions that mark the beginning and end of a transaction, so that all the loads and store instructions executed while in transactional mode are implicitly considered transactional. This is the approach that most modern HTM proposals follow, including the Rock processor [30] and the systems evaluated thoughout this thesis [55, 143, 153]. Another option is to further augment the ISA with explicit "transactional load" and "transactional store" instructions, separated from their conventional counterparts. Though allowing a transaction to contain

both transactional and non-transactional accesses may complicate things, this provides increased flexibility and may aid programmers to reduce the pressure on the underlying TM mechanisms, as non-transactional accesses do not participate in data versioning nor conflict detection. The original HTM proposal by Herlihy and Moss [60] as well as the AMD Advanced Synchronization Facility (ASF) [35] are explicitely transactional designs.

Some proposed HTMs also include an "abort transaction" instruction to explicitly roll back the tentative work of transaction. This is an example of flexible design that may enable TM hardware to be applied toward solving problems beyond guaranteeing mutual exclusion during the execution of critical regions. Programmers using hardware transactions may find useful the ability to explicitly rollback execution upon a certain condition, which need not necessarily be a conflict with other transaction.

2.3.2 Transactional Book-keeping

HTM systems must track a transaction's read and write set in order to detect data races amongst concurrent transactions. Many HTMs extend the cache line metadata kept at the private cache level, with two new bits that record, respectively, whether the line has been speculatively read (SR) and/or speculatively modified (SM) during the ongoing transaction [55, 92, 143]. Such designs also support the capability to clear all the read bits in the data cache instantaneously, an action that is performed when the transaction commits or aborts. The private caches serve as a natural place to track a transaction's read and write sets, enabling low overhead tracking, although they also constrain the granularity of conflict detection to that of a cache line. Some systems propose the addition of "rename" bits to reduce the granularity for tracking to that of words or even bytes [56]. Other proposals propose adding a separate transactional cache to track the read and write sets [60]. There are also hybrid solutions that use the data cache to track the read-set while using the store buffer to track the write-set [30].

All HTM systems that leverage the private level cache to perform transactional book-keeping are susceptible of transactional *overflows* due to the cache's limited capacity or associativity. Best effort designs would automatically abort the transaction if a cache line whose SR or SM bit is set is replaced, while bounded schemes would resort to safety nets in order to keep tracking read and write sets and detecting conflicts in the presence of spilled lines [56]. Some have proposed the addition of a permissions-only cache to hold transactional metadata

for evicted lines [16], in order to support transactions of larger size before falling back to serialization as a means to allow the overflowed transaction to commit.

An alternative scheme of transactional book-keeping which does not leverage the private level cache is to use Bloom filters to conservatively summarize a transaction's data accesses into two fixed-size registers or "address signatures" [27, 153] , one for each transactional set. This avoids modifying the cache design and it is important because private caches are critical structures in the design of high performance processors. The main disadvantage of hash encoding is that false positives may signal spureous conflicts, this is, the signature may indicate that an address belongs to the transaction read and write sets when in fact it does not.

Hybrid combinations are also possible in transactional book-keeping: SR bits in cache can be used to track speculatively read lines that remain cached, while using signatures to conservatively encode only the addresses of those SR lines that have been spilled from the cache. This mixed solution reduces the population of the address signature and can help reduce the amount of false positives [96] while allowing transactions of larger size.

2.3.3 Data Versioning

Besides keeping read and write set metadata, private caches can also buffer speculatively written state in a natural way, since they are on the access path for the local processor and thus can automatically forward the latest transactional update to subsequent loads without special search. Write-back caches can be easily modified to behave as write buffers that support lazy versioning. When in transactional mode, the processor can speculatively modify a line with read-only permissions, setting the SM bit and hiding the update from the rest of the system. To the coherence protocol, the line is perceived as a shared copy of the data, and thus multiple speculative writers of the same data are possible [29, 56, 143]. From a coherence point of view, those speculative writes to a private copy of the cache line without write permissions can be seen as writes "on the side" in spite of being to the same memory location. The coherence protocol must be suitably adapted to ensure a consistent copy of the data exist in the shared levels of the memory hierarchy: If a transactional store targets a line that is already dirty, not speculatively modified, the coherence protocol must first write back the line to the shared levels and downgrade its access permissions to shared before allowing the speculative write to proceed. Because coherence protocols generally support silent invalidations of lines in shared state, the cache design can also be extended

to support conditional invalidation of those lines whose SM bit is set, so that the HTM system is able to instantaneously discard the speculative state in the event of an abort. Figure 2.6 shows the additional circuitry (in black) required to support flash-clear of SR and SM bits (left-most and middle cells) as well as gang-invalidation of SM lines (right-most cell). On commit, the system still needs to locate SM lines and publish their contents, and thus a buffer with the write-set addresses is usually maintained [56]. HTM systems that perform lazy versioning in cache are susceptible of transactional overflows due to the limited capacity or associativity of the cache, or even the address buffer used to track the write-set could fill up. In any case, speculatively written values cannot be spilled from private structures, and the system needs to either abort the transaction or it may enter a special mode of operation in an attempt to commit the overflowed transaction [56].

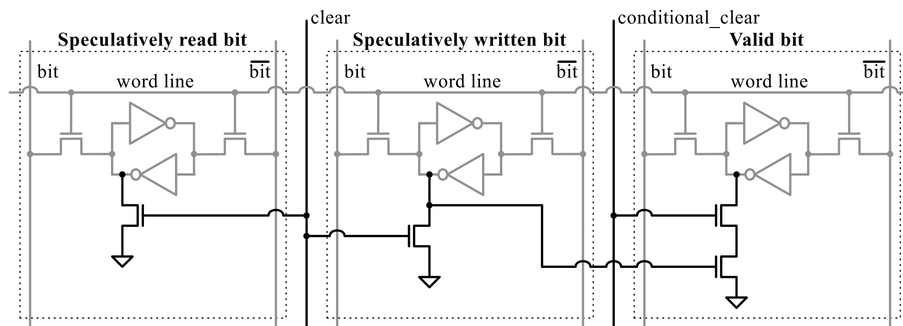


Figure 2.6: SRAM cells augmented with circuitry for flash-clear and conditional flash-clear [19].

For HTM systems with eager version management [92, 153], caches need no changes in the coherence protocol in order to support versioning since the hardware does not have any notion of speculative writes. All writes update memory, whether they occur inside or outside a transaction, and it is the responsibility of the coherence protocol to detect remote accesses to speculatively written data, and ensure no other threads or transactions observe it. Because a per-thread log in cacheable virtual memory replaces hardware buffers used in lazy versioning, systems with eager versioning can accommodate transactions of a much larger size without overflowing hardware structures. Unlike lazy systems, evictions of speculatively written data from the private caches are tolerated, and they need no special treatment from the point of view of the versioning mechanism. However, specialized hardware is required to fill this virtualized log with the old value

of each memory location that is being over-written inside a transaction. The contents of the log are simply discarded on commit, by reseting the log pointer to its initial position. On an abort, a software handler walks the log restoring the original values into memory.

2.3.4 Conflict Detection

In order to detect data conflicts, HTMs leverage the invalidation-based cache coherence protocols presented in the previous section. Cache coherence protocols allow HTM implementations to detect conflicts among concurrently running transactions at the granularity of cache lines. While unnecessary transactional conflicts may arise as a result of false sharing, for most transactional workloads this choice of granularity represents a good trade-off between design cost and performance. Most HTM design proposals choose to leverage coherence mechanisms for conflict detection.

The coherence protocol already provides mechanisms to locate the copies of a requested cache line, and thus the detection of transactional conflicts can be achieved with straightforward extensions. Let us assume for this elaboration the directory-based MESI coherence protocol that is used throughout this thesis. In such scenario, a local store to a line that is currently in S state results in a write miss, since the protocol ensures that no cache can have permissions to write the data at this point. A coherence message requesting exclusive access is sent to the directory, which in turn sends invalidation messages to the current sharers of the line (except maybe the requestor, if it is amongst the sharers). The sharers are then able to check whether the requested address belongs to their read set –by checking the SR bit in cache, the read signature, etc.– and appropriately detect a *write-read* conflict. Similarly, a local load (store) to a line that is currently in M or E state in a remote cache results in a read miss that results in a shared (exclusive) coherence request being forwarded by the directory to the cache that has the latest copy of the data, which then checks its write-set (read- and write-set) metadata to determine if a *read-write* conflict (*write-write* conflict) exists.

In snooping-based protocols, all caches observe all coherence traffic for all lines, allowing cache controllers to check for conflicts whenever a request is observed on the bus. In directory-based protocols, cache controllers only observe the coherence traffic corresponding to the lines that are currently privately cached, and thus they need modest design extensions to support TM semantics. Evictions of transactional data from private caches are possible in HTMs if the coherence protocol is modified so that the cache continues receiving coherence traffic for the

spilled line, and is then able to keep detecting conflicts on it. For speculatively read lines, this is not an issue: Directory protocols usually already implement *silent replacements* of lines in S state, allowing a private cache to simply replace an S line without reporting to the directory. Upon a subsequent write attempt, the cache receives an invalidation to a line that is no longer cached, and then it is able to check its transactional metadata and infer if a conflict exist on the spilled line. Overflows of speculatively modified data are possible in HTMs with eager versioning. In this case the directory needs to be extended with a *sticky-M* state to perform conflict detection even after replacing transactionally written data from the cache [92].

As mentioned earlier, the coherence protocol needs minor modifications to support lazy conflict detection HTM. First, the coherence controller needs to allow transactional writes to S lines without having to request exclusive access, as part of the support for lazy versioning. Furthermore, E and M lines must be downgraded to S state before a transactional store can modify the line, not only because the last consistent version of the data needs to be preserved in the shared levels (M state), but also to resume conflict detection via commit-time coherence invalidation messages. Revoking exclusive ownership to a line before the first transactional write simplifies lazy conflict detection, as it always takes place in the same fashion. A committing trasaction publishes its speculative updates by issuing exclusive ownership requests for the lines in its write set. Since all transactional accessors (whether readers or also writers) are marked as sharers of the line, they eventually receive the corresponding invalidations that are used for conflict detection. Downgrading lines in E state to S before a transactional store is served also avoids adding extra complexity to the protocol, since it enables transparent gang-invalidation of speculatively modified lines in case of abort, which appears as an instananeous silent replacement of multiple S lines. Otherwise, the protocol needs to be adapted in order to support silent replacements of lines in E state.

2.3.5 Conflict Resolution

Once an HTM system detects a conflict, it must determine how to resolve it. The conflict resolution policy constitutes another design dimension in HTM by dictating which transaction wins the conflict and is granted access to the data. The loser transaction can stall its execution, or it can be aborted: The alternatives change depending on when the conflict is detected.

In HTMs with eager conflict detection, there are several policies for resolution:

requester wins, requester aborts, or requester stalls using a scheme of conservative deadlock avoidance. The implementation of the requester wins policy is straightforward: The cache or caches that detect a conflict simply trigger abort and yield to the requester. If the conflicted data was not speculatively modified (write-read conflict), the loser responds with the appropriate invalidation acknowledgement or data message. Otherwise, the response may be delayed until the data is conveniently restored. The main drawback of this policy is that it can produce livelock scenarios. The opposite option is to abort the requester. This is acomplished by augmenting the coherence protocol with negative acknowledgements (NACK) messages, so that a cache controller that detects a conflict responds to a forwarded request or invalidation with a NACK message. On reception of a NACK response, the requester knows it has lost the conflict and can take the appropriate actions. The simplest alternative is to trigger its own abort, but this can also result in livelock. A less draconian, livelock-free solution is to stall the transaction and periodically retry the conflicting memory access until a positive response (different from the NACK) is received. In this case, cyclic dependencies amongst transactions can bring the system to a deadlock, and so the system must have a way out such possible cycles. LogTM [92] uses a simple timestamp-based scheme to conservatively detect cycles, aborting the youngest transaction to break the possible cycle.

HTM systems with lazy conflict detection must resolve conflicts when a committer seeks to commit a transaction that conflicts with one or more other transactions. The resolution policy in this scenario can abort all others, or else stall or abort the committer. In general, lazy HTMs follow a committer wins policy [22, 56] that favours forward progress and is both deadlock- and livelock-free. Unfortunately, the committer wins policy does not guarantee fairness and can result in starvation for some transactions.

2.3.6 Transaction Commit

The execution of the "commit transaction" instruction attempts to make the transaction's tentative changes permanent and visible to other processors instantaneously. Such publication is in itself a task that must occur atomically and without interference from other processors. For most HTMs, publishing speculative updates means obtaining exclusive ownership for all cache lines in the write set, and then releasing isolation over both transactional sets at once.

The implementation of the commit instruction is a straightfoward operation in eager HTMs, since writes were performed in place and therefore all write

set lines are already held in exclusive ownership. Because no further communication is required to validate the consistency of the transaction, commit then simply consists of releasing isolation over the read and write sets by clearing the read/write bits or signatures used for transactional book-keeping. This allows remote requests to fetch those cache lines that belonged in the transactional sets of the committed transaction.

As for lazy HTMs, the requirement of en-masse publication of speculative updates to shared memory at commit time poses more challenges. Committing a lazy transaction means upgrading all the lines whose SM bit is set from S to M state, issuing exclusive ownership requests for each line and waiting for the invalidations of the copies in remote caches (and possible transaction squash). This process involves global communication and is not instantaneous, and thus the committing transaction must stall (e.g. respond with a NACK message) any subsequent requests from other processors to lines that belong to its read and write sets.

2.3.7 Transaction Abort

A hardware transaction may be implicitly aborted by the conflict resolution mechanism, or the abort can be explicitely triggered from the program via an "abort transaction" instruction. Aborting a transaction means discarding all its tentative changes and return the state of the processor to the exact same state it was right before the transaction began. Book-keeping information (SM and SR bits, signatures, etc.) must always be cleared on abort, and the last step of the abort process is the restoration of the architectural registers using the checkpoint that was saved in the shadow register file at the beginning of the transaction.

Implementating the abort functionality is quite simple in lazy HTMs, since speculative writes were performed "on the side" (in private structures local to the core) and therefore the shared memory still contains consistent, pre-transactional values. Therefore, flushing the contents of the write address buffer and gang-invalidating the cache lines whose SM bit is set is sufficient to discard the updates. This operations can usually be done in hardware and take no more than a few cycles.

Eager HTMs, on the other hand, must restore each cache line in the write set with the pre-transactional value that was backed up in the transaction log. Except maybe for small transactions that write a very small number of lines, the log unroll is generally done in software, by trapping to an abort handler that accesses the log base and pointer registers, and walks the log in reverse direction

–those entries that were added last must be processed first–. No transactional conflicts should arise during this process, as the coherence protocol ensures that the lines that belong to the write set of the aborting transaction are isolated and cannot belong to any other transaction. Because aborting is an slow process in eager HTMs, isolation over the read set is usually released as soon as the abort is triggered, as it is safe for other transactions to access it while the log is unrolled.

2.4 An Overview of Hardware Transactional Memory Research

Research in HTM design has been very active since the introduction of multicores in mainstream computing. In the early nineties, Herlihy and Moss introduced Transactional Memory [60] as a hardware alternative to lock-based synchronization. Their main idea was to generalize the LL/SC primitives in order to perform atomic accesses not to one but to several independent memory locations, thus eliminating the need for protecting critical sections with lock variables. Almost a decade later, architects began to recover their interest in transactions at a hardware level. Rajwar and Goodman's Transactional Lock Removal (TLR) [106] was the first to apply the concept of transaction to the execution of lock-protected critical sections, merging the idea of Speculative Lock Elision (SLE) [105] with a timestamp-based conflict resolution scheme.

The early proposal by Herlihy and Moss was revived ten years later by Hammond *et al.*, who present Transactional Coherence and Consistency (TCC) [56] as a novel coherence and consistency model that uses continuous transactional execution. The novelty of TCC stems from its "all transactions, all the time" philosophy, where transactions are the basic unit of parallel work, synchronization, memory coherence and consistency. TCC's lazy approach contains speculative updates within private caches and lazily resolves races when a committing transaction broadcasts its write-set, employing a bus to serialize transaction commits.

In contrast to Stanford's TCC, Wisconsin's LogTM [92] explores the opposite corner of the HTM design space. Moore *et al.* take a more evolutionary approach to transactional memory in LogTM, combining transactional support with a conventional shared memory model that enables a more gradual change towards transactional systems. LogTM is a purely eager HTM system that leverages a standard coherence protocol to perform conflict detection on individual memory requests, and makes commits fast by storing old values to a per-thread log in cacheable virtual memory, which is unrolled by a software handler in case of

abort. Unlike TCC, LogTM can tolerate evictions of transactional data from caches thanks to the log, and enables conflict detection on evicted blocks through an elegant extension to the coherence protocol.

LogTM has been subsequently refined. Moravan *et al.* [93] introduce support for nested transactions, enabling both closed nesting with partial aborts and open nesting [97]. Open nesting is a programming language construct motivated by performance, which can improve concurrency by relaxing the atomicity guarantee. When an open nested transaction commits, the TM system releases its read and written data so that other transactions can access them without generating conflicts. Thanks to open nesting, otherwise-offending transactions can access the exposed data after the nested transaction commits, while the outer transaction still runs. This can enhance the degree of concurrency achieved by the flattening scheme found in LogTM, which enforces isolation until the outermost transaction commits. In [13], Baek at al. propose FanTM, a design that uses address signatures in hardware [27] to efficiently support transaction nesting.

Later on, Yen *et al.* [153] decouple transactional support from caches, removing read and write bits used for transactional book-keeping, and replacing them with *hash signatures*. This latest improvement, called LogTM-SE (*Signature Edition*), borrows the concept of Bloom filters [15] to conservatively encode a transaction's read and write set metadata. The idea of applying hash encoding towards conflict detection/thread disambiguation was first introduced into the realm of TM by Ceze *et al.* in [27] and [28]. The use of hash signatures for transactional book-keeping has been further explored by several authors. In [116], Sanchez *et al.* examine different signature organizations and hashing schemes to achieve hardware-efficient and accurate TM signatures. Quislant at al. have also studied signature organizations, basing their works in LogTM-SE. In [103], they show that locality can be exploited in order to reduce the number of bits inserted in the filter for those addresses nearby located, and reducing the number of false conflicts. More recently, the authors have studied multiset signature designs [104] which record both the read and write sets in the same Bloom filter. Yen at al. developed Notary [154], which introduces a privatization interface that allows the programmer to explicitly declare shared and private heap memory allocation, which can be used to reduce the signature size as well as the number of false conflicts arising from private memory accesses. Sanyal *et al.* exploit the same concept in [117], proposing a scheme that dynamically identifies thread-local variables and excludes them from the commit set, both reducing the presure on the versioning mechanisms and improving the scalability of such phase in lazy HTMs.

In FASTM [80], Lupon *et al.* extend LogTM with a coherence protocol that enables fast abort recovery in an otherwise eager HTM, by leveraging the private cache to buffer speculative state, effectively avoiding traps to software handlers that perform log unroll as long as speculatively modified data does not overflow the private cache level. LogTM's approximation of making commits fast has also inspired OneTM [16] [18], which uses a cache to reduce the frequency with which transactions overflow on chip resources, and proposes a simple irrevocable execution switch to handle such overflows as well as context switches, I/O or system calls inside transactions, at the cost of limited concurrency.

Bobba *et al.* propose TokenTM [21], another unbounded HTM design that uses the abstraction of tokens [85] to precisely track conflicts on an unbounded number of memory blocks and it handles both paging, thread migration and context switching, but incurs high state overhead. In [69], Jafri *et al.* improve on TokenTM and propose LiteTM, a design that maintains the same virtualization properties of TokenTM while greatly reducing the state overhead, and without sacrifying much performance. Support for transactions of unlimited duration, size and nesting depth has also been considered by proposals such as UTM [9] [78] or VTM [107], which focus on hardware schemes that provide virtualization of transactions. However, both achieve this goal by introducing large amounts of complexity in the processor and the memory subsystem. On its part, XTM [33] implements transaction virtualization support in software, using virtual memory and operating at page granularity. A similar approach is taken by Chuang *et al.* [32] in PTM, a page-based, hardware-supported TM design that combines transaction bookkeeping with the virtual memory system to support transactions of unbounded size, as well as to handle context switches and exceptions.

While it is not an issue for eager systems like LogTM, parallelism at commit is important for lazy systems when running applications with low contention but a large number of transactions. Transactions that do not conflict should ideally be able to commit simultaneously. The very nature of lazy conflict resolution protocols makes it difficult since only actions taken at commit time permit discovery of data races among transactions. Simple lazy schemes like the ones employing a global commit token [22] or a bus [55] do not permit such parallelism. The reason for limited parallelism at commit time is that the committing transaction has no knowledge of which other concurrently running transactions must abort to preserve atomicity. The TCC design [55] was later extended to scalable DSM architectures using directory based coherence. This proposal is called Scalable TCC (STCC) [29], and it employs selective locking of directory banks to avoid arbitration delays and thereby improve commit

throughput. Pugsley *et al.* [102] improve over STCC by proposing even more scalable commit algorithms that reduce the number of network messages, remove the need for a centralized agent, and tackle deadlocks, livelocks and starvation scenarios.

Another approach to improve the scalability of the commit process in lazy systems has been explored by EazyHTM [143]. Tomic *et al.* record the information pertaining to potential conflicts, which is readily available from coherence messages during the lifetime of any transaction, and use this information at commit time to allow true commit parallelism. All potentially conflicting transactions that must be aborted would be known, and committers that have not seen races can commit in a truly parallel fashion. FlexTM [122] also provides lazy conflict resolution by recording conflicts as they happen, using this information to enable distributed commits. Unlike EazyHTM, Shriraman *et al.* choose to do so in software, sacrifying progress guarantees to gain greater parallelism. Performance costs associated with software intervention and software verification challenges without watertight forward progress guarantees could limit the value of this approach. EazyHTM, on the other hand, provides parallel lazy commits in hardware and ensures forward progress, but trades off common-case performance to achieve it. FlexTM allows flexibility in policy but it does so by implementing critical policy managers in software. It provides a significant improvement in speed over software TM implementations by proposing the use of alert-on-update hardware, but the considerable cost of software intervention renders a comparison with pure HTMs moot. In the context of HTM, Shriraman and Dwarkadas [121] have also analyzed the interplay between conflict resolution time and contention management policy. They show that both policy decisions have a significant impact on the ability to exploit available parallelism and demonstrate that conflict resolution time has the dominant effect on performance, corroborating that lazy HTMs are able to uncover more parallelism than eager approaches.

With DynTM [81], Lupon *et al.* introduce a cache coherence protocol that allows transactions in a multi-threaded application run either eagerly or lazily based on some heuristics like prior behavior of transactions, at the cost of adding extra complexity at level of the coherence controller. It works at the granularity of a transaction and then develops a cache coherence protocol around it that supports multiple ways to version the same shared memory block. LV* [96], a proposal that utilizes snoopy coherence, allows programmer control over policy in hardware but with the constraint that all transactions in an application must use the same policy at any given time. The requirement of programmer-assisted

policy change is a drawback too since the same phase of an application can exhibit different behavior with varying datasets.

The mitigation of the performance penalty associated with transaction aborts has been of interest to the HTM community. Waliullah and Stenstrom study the utility of intermediate checkpoints in lazy HTM systems [146, 148], as a means to reduce the amount of work that is discarded on abort. In their scheme transactions record conflicting addresses upon abort, and use this historical information to insert a checkpoint before a memory reference predicted as conflicting is executed. If the transaction is squashed, it is rolled back to the checkpoint associated with the first conflicting access, rather than all the way back to the beginning. Reducing the penalty of abort was also considered by Armejach *et al.* [11], who propose a reconfigurable private level data cache to improve the efficiency of the version management mechanism in both eager and lazy HTMs.

The applications of data forwarding and value prediction for conflict resolution have also been explored in the context of eager HTM systems. Pant *et al.* [99] [100] observe that shared-conflicting data is often updated in a predictable manner by different transactions, and propose the use of value prediction in order to capture this predictability and increase overall concurrency by satisfying loads from conflicting transactions with predicted values, instead of stalling. In DATM [108], Ramadan *et al.* investigate the advantages of value forwarding for speculative resolution of true data conflicts amongst concurrent transactions. DATM is an eager system that discovers and tracks the data dependencies amongst concurrent transactions, allowing writer transactions to proceed in the presence of other conflicting transactional accessors, and reader transactions to obtain uncommitted data produced by a concurrent transaction, while still enforcing a legal serialized order that preserves consistency.

Hardware TM systems can suffer a series of pathological behaviours that negatively affect performance. Bobba *et al.* explore HTM design space, identifying how some of these undesirable scenarios [22] affect each kind of system depending on the choice of policies for version and conflict management. Some pathologies such as starvation have been further analysed and resolved in other subsequent works [147]. Other pathologies that affect HTM performance have been the topic of several studies. Volos *et al.* [145] investigate the interaction of transactional memory implementations and lock-based code, and discover other problematic scenarios that may arise in these circumstances. False sharing, another undesired situation that may arise in multi-threaded codes, becomes even a bigger problem when it occurs in conjunction with hardware transactional memory [92] due to the detection of conflicts at a cache line granularity. Tabba

et al. [133] propose a mechanism that takes the concepts of coherence decoupling [65] and value prediction, and combines them to mitigate the effects of coherence conflicts in transactions. The granularity of conflict detection in HTM has also been the subject of the works by Khan *et al.* [70], whose HTM proposal is able to detect conflicts at the level of objects –instead of cache lines–, which leads to a novel commit scheme as well as an elegant solution to the problem of version management virtualization.

Another kind of pathological behaviour affecting HTM performance happens when concurrent operations on data structures that are not semantically conflicting –such as two insertions in two different buckets of a hash table– result in conflicting transactions because of updates on auxiliary program data –e.g. the *size* field–. Inspired by instruction replay-based mechanisms [43], Blundell at al. propose RetCon [20], a hardware mechanism that eliminates the performance impact of such spureous transactional conflicts. RetCon tracks the relationship between input and output values symbolically and uses this information to transparently repair the output state of a transaction at commit.

Ramadan *et al.* have examined the architectural features necessary to support HTM in the Linux kernel for the x86 architecture [109] [114]. They propose MetaTM, an HTM model that contains features that enable efficient and correct interrupt handling for an x86-like architecture. Using TxLinux –a Linux kernel modified to use transactions in place of locking primitives in several key subsystems– they quantify the effect of architectural design decisions on the performance of such a large transactional workload. TxLinux, based on the Linux 2.4 kernel and thus characterized by its simple, coarse-grained synchronization structure, is used by Hoffman *et al.* in [64] to show that a minimal subset of TM features supported in hardware can simplify synchronization, provide comparable performance to fine-grained locking and handle overflows. The challenge of operating system (OS) support in HTM is also addressed Wang *et al.* [119] and Tomic *et al.* [142]. DTM [119] proposes a hardware-based solution that fully decouples transaction processing from caches, while HTM-OS [142] leverages the existing OS virtual memory mechanisms to support unbounded transaction sizes and provide transaction execution speed that does not decrease when transaction grows. A related challenge that has been addressed in the HTM literature is the support of input/output operations within transactions: Lui *et al.* [79] analyse this problem and propose an HTM system that supports I/O within transactions by means of partial commits, using commit-locks and blocking/waking-up of transactional threads.

The applicability of hardware transactional memory (HTM) has also been

considered in the context of dynamic memory management. Dragojevic *et al.* [46] demonstrate that HTM can be used to simplify and streamline memory reclamation for practical concurrent data structures. The use of HTM to aid lightweight dynamic language runtimes in evolving more capable and robust execution models while maintaining native code compatibility has been studied too. Using a modified Linux kernel and a Python interpreter, Riley at al. [110] explore the lack of thread safety in native extension modules and use features found in an HTM implementation to address several issues that impede to the effective deployment of dynamic languages on current and future multicore and multiprocessor system.

Energy efficiency has been timidly considered in HTM research. Gaona *et al.* [49] characterize the energy consumption of eager and lazy HTM systems in a common framework. In [118], Sanyal *et al.* propose a clock-gating scheme that turns off a processor dynamically when a transaction running on it is aborted.

Chapter 3

Evaluation Methodology

In this chapter, we describe the experimental methodology used for evaluating the proposals presented in this dissertation.

3.1 Simulation Tools

The methodology chosen for this evaluation is based on full-system simulation, which gives us the ability to run realistic workloads on top of an actual operating system. Unlike trace-based simulation, full-system simulation also enables the introduction of random variability on the timing model, producing distinct thread interleavings on different executions of the same program. This slight timing variations can make threads take different code paths from run to run.

We use the Wisconsin *General Execution-driven Multiprocessor Simulator* (GEMS) simulation environment [86], which is based on Wind River Simics [82]. Simics is a full-system functional simulator of multiprocessor systems that supports the SPARC instruction set architecture (ISA), amongst others. We use Simics to boot an unmodified Solaris 10 box on which we run the transactional workloads. Simics supplies an in-order processor model in which all instructions take one cycle to execute. Simics then allows an external module to register with its timing interface, so that the latency of memory access instructions can be modeled with accuracy.

While Simics is responsible for the functional correctness of the simulation framework, GEMS provides several timing modules that plug into Simics to incorporate detailed models for the fundamental components of the system. As

shown in Figure 3.1, GEMS comprises two main modules, namely *Ruby* and *Opal*: The former models memory hierarchies and uses a domain-specific language to specify coherence protocol, and the latter captures the temporal features of an out-of-order processor. A third module called *Tourmaline* allows Simics to be used as functional simulator for transactional applications, enabling near-Simics execution speeds while preserving transactional semantics. For the evaluations presented in this thesis, we model a chip-multiprocessor (CMP) composed by simple in-order processing cores [84], and hence only the Ruby module was used.

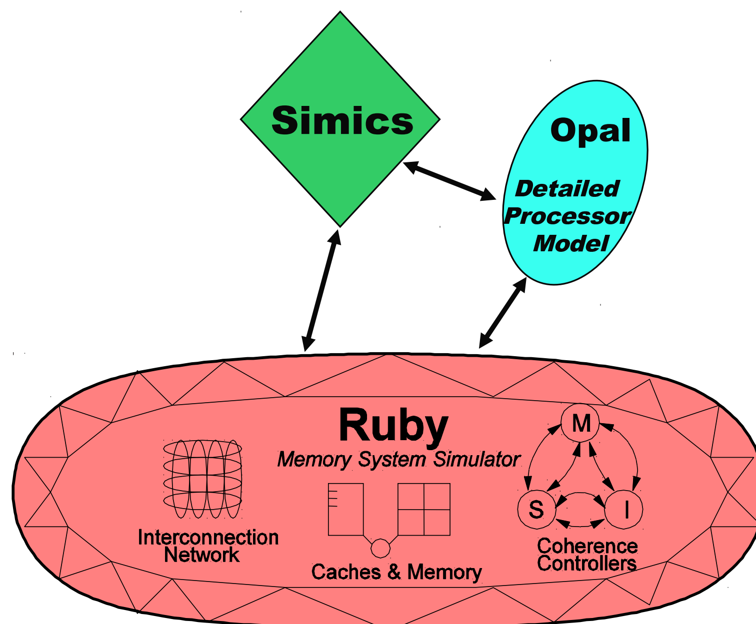


Figure 3.1: Architecture of the Simics-GEMS simulation framework.

The Ruby module offers an event-driven framework to precisely simulate a memory hierarchy that allows us to measure the effects of behavioural and structural changes to the components that conform the memory subsystem, namely L1 and L2 caches, and directory and memory controllers. Ruby models the latency of each memory request received from the functional simulator by stalling Simics until the memory hierarchy brings the requested data with appropriate permissions to the requesting processor's first level cache. As a request travels across the memory subsystem, each component introduces a given delay, measured from the moment the message is picked from its input port for processing, until the component generates a response and injects it back into the

network. All the components are connected using a detailed network model called *Garnet* [4], which features a state-of-the-art interconnect that precisely models the time required to deliver a message from one component to another. At the heart of Ruby lays the *Specification Language for Implementing Cache Coherence* (SLICC), a domain-specific language to describe the behaviour of coherence protocols in terms of their state machine. SLICC has been used to implement the cache and directory controllers for the different protocols discussed in this thesis. Furthermore, Ruby also includes the transactional components required to simulate hardware transactional memory (HTM) systems implemented on top of a CMP architecture. Both eager- and lazy-policy HTM systems are supported in the latest release of the simulator as of this thesis (2.1.1). A later subsection describes these HTM systems in more detail, as representative points in opposite corners of the design space.

Transaction boundaries (i.e., tx_begin and tx_end instructions) are delimited through Simics *magic instructions*, a special assembly instruction with the effect of a `nop` when executed on a real machine, but which is captured by Simics. Ruby leverages these hooks to be informed about the occurrence of certain events, and implements the functionality of the associated TM instruction. The macro-based style of programming that the TM benchmarks are developed under facilitates the insertion of the appropriate magic instructions that mark the borders of an atomic block. Magic instructions are also used for a variety of purposes besides wrapping the critical regions of the multithreaded program, like signaling the beginning/end of the parallel phase in the program, measuring synchronization overheads (i.e. barrier time), implementing other language-level constructs such as early release [124], or temporarily disabling the timing module for given regions of the code. For compiling our TM programs we use the widely available GNU C compiler (gcc) and the -O3 optimization level.

3.2 Metrics and Methods

The evaluation of the designs presented in this dissertation focuses mostly in two key aspects of a multiprocessor system: performance and network traffic. Our main performance metric is the execution time of the complete parallel phase in the transactional program, skipping the initialization phase (in which per-thread data structures are set up). Execution time is measured as the number of cycles counted from the instant the first thread leaves the initial barrier that marks the beginning of the parallel phase, until the cycle at which the last thread

reaches the final barrier that synchronizes them prior to join. Another metric used throughout our evaluations is the *accumulated cycle count*, which equals to the overall execution time multiplied by the number of threads that executed the program. The accumulated cycle count is used to represent the execution time divided into disjoint components, each one corresponding to sum of the cycles spent by each thread in a given state during its execution. The typical categories comprise barrier, non transactional, transactional useful, transactional aborted, rollback, back-off and commit times. In some cases a given category may be further split into finer-grain components. This execution time breakdown serves as a general and intuitive picture about the characteristics of the multithreaded application, and it also gives us some insights into the behaviour of the system. Despite useful, this metric must be interpreted with caution as it does not reflect by any means a breakdown of what the parallel application does as a whole in each cycle.

Table 3.1: Components: Execution time breakdown

Component	**Description**
barrier	wait at barriers
non_txnal	non-transactional execution (including supervisor code)
tx_useful	successful transactional execution
tx_aborted	aborted transactional execution
stall	conflicting request retries (eager only)
backoff	idle cycles after an abort
arbitration	acquisition of lazy commit permission (lazy only)
commit	cycles spent committing updates to shared memory (lazy only)
rollback	cycles spent aborting transactions (eager only)

Metrics that show how the characteristics of a workload impact overall execution time are considered too. For instance, the number of aborts per commit acts as a quantitative measurement of the amount of contention present in a given application. Transaction-level statistics such as the read- and write-set sizes of transactions are also relevant. Maximum and average occupation of key hardware structures such as the write-buffer are of interest as well. Other metrics specific to each chapter will be discussed where appropriate. We disregard a common performance metric in uniprocessor architectures, instructions-per-cycle (IPC), as it is not a representative measure of the amount of useful work performed by a multiprocessor system when running a multithreaded application [6]. Despite

the virtual removal of lock-based synchronization achieved by HTM systems, other factors still exist –such as transaction aborts or synchronization at barriers– that may artificially contribute to the IPC when indeed no progress is made, hence rendering this metric useless.

Each simulation is pseudo-randomized by introducing a variation of ±5 cycles in the response time of the accesses to main memory. This captures the non-determinism present in real systems [5] and produces runs with different thread interleavings that help reveal the different paths that can be taken by the application. In some cases, this variation in the interleaving of threads produces a different –but still correct– solution to the program, as it is the case of one of the benchmarks used. Multiple simulations are performed for each workload and the average run-time is calculated with an arithmetic mean. Error bars in our results approximate a 95% confidence interval.

3.3 Workloads

For evaluating the proposals presented in this thesis, we have selected seven transactional applications from the STAMP suite [25]: genome, intruder, kmeans, labyrinth, ssca2, vacation and yada. The program bayes was excluded since it exhibits unpredictable behaviour and high variability in its execution time [45,96]. For kmeans and vacation, both high and low contention configurations were used. Small input parameters, detailed in [25], were used for all applications. For genome, intruder, ssca2 and yada, recommended medium-sized inputs were used too, as they do not increase simulation times beyond excessive levels. In total, 13 benchmarks are used, whose inputs are summarized in Table 3.2. Configurations that use medium inputs are marked with the plus symbol in Table 3.2 and throughout this thesis. For vacation and kmeans, the medium size input was deemed irrelevant and thus excluded, as it does not yield any additional results of interest, mainly because these two benchmarks barely stress the performance of the underlying TM system due to their particular characteristics.

genome. This benchmark implements a gene sequencing program that reconstructs the gene sequence from segments of a larger gene. Given a large set of DNA segments, the algorithm will try to construct the shortest gene that can be made from them (i.e. the source genome) under the pairing constraints of the nucleotides that conform the DNA segments, according to their nucleobase (adenine, thymine, guanine and cytosine, or ATGC). Since there may be many duplicates in the relatively large number of DNA segments, a first phase in

Table 3.2: Benchmarks and inputs used in the simulations.

Benchmark	Input
genome+	-g512 -s32 -n32768
genome	-g256 -s16 -n16384
intruder+	-a10 -l16 -n4096 -s1
intruder	-a10 -l4 -n2048 -s1
kmeans-low	-m40 -n40 -t0.05 -i random-n2048-d16-c16
kmeans-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16
labyrinth	-i random-x32-y32-z3-n96
ssca2+	-s14 -i1.0 -u1.0 -l9 -p9
ssca2	-s13 -i1.0 -u1.0 -l3 -p3
vacation-low	-n2 -q90 -u98 -r16384 -t4096
vacation-high	-n4 -q60 -u90 -r16384 -t4096
yada+	-a10 -i ttimeu10000.2
yada	-a20 -i 633.2

Table 3.3: STAMP Workload Characteristics.

Workload	Trans Size	Contention	Commit rate
genome	Moderate	Moderate	Moderate
intruder	Small	High	Moderate
kmeans	Small	Low	Low
labyrinth	Large	Moderate	Low
SSCA2	Small	Low	High
vacation	Large	Low	Moderate
yada	Large	High	Moderate

the algorithm utilizes a hash set to create a set of unique segments, using a transaction to ensure atomicity during the insertion. Then threads carry out the gene matching in the second step, by removing segments from the global pool of unmatched segments and adding them to its partition of currently matched segments. The removal from the global pool is also enclosed by transactions as threads may try to remove the same segment. Compared to the locking approach, transactions simplify the parallelization of this program by eliminating the need for a deadlock avoidance in the reconstruction phase.

intruder. This benchmark emulates a network intrusion detection system (NIDS), scanning packets for matches against a known set of attack signatures.

Each processed packet subsequently goes through capture, reassembly, and detection stages. A simple FIFO queue stores the captured packets, and then threads extract one at a time (by means of a transaction) for processing. The reassembly phase is also enclosed inside a coarse grain transaction, which protects the access to the self-balancing tree that contains lists of packets that belong to the same session. The code for each phase is as simple as that with coarse-grain locks but hopefully achieves good performance through optimistic concurrency.

kmeans. K-means is an example of programs in which programmers do not use transactions for its ability to exploit thread-level parallelism (TLP) out of coarse-grained transactions, but simply to enforce mutual exclusion during the update of a global vector that contains the result of the computations. Each thread processes a partition of the objects iteratively, and transactions are used to protect the update of the cluster center that occurs during each iteration, as well as a global variable that controls the task queue. The result is that transactions are very small and short running. Optimistic concurrency can be beneficial when threads update different centers concurrently, although as the number of threads grows, so does the probability of having two threads concurrently operating on the same cluster center, hence creating frequent transaction conflicts.

labyrinth. Labyrinth implements a routing algorithm for solving the maze problem [150]. For each pair of input points, the program finds the shortest route that connects them in a three-dimensional uniform grid that represents the maze. Following Lee's algorithm, it uses a wave propagation style throughout the routing space, so that in the n-th iteration the wave is expanded to all points that can be reached in n steps from the source. When the target point is reached, the expansion stops and the path is determined by backtracking from destination to source. The main transaction of the program encloses the calculation of the path and its addition to the global grid. The development of this program under the TM programming model yields a simpler program that does not need deadlock avoidance techniques required the lock-based approach. To avoid unnecessary writes to the global grid during the expansion phase and reduce the chance for conflicts, a privatization technique is employed. Each thread creates a local copy of the global grid and uses it for the route calculation (expansion and trace-back phases). In the process of creating a private copy of the grid, transactions add the entire global grid to their read sets, causing conflicts whenever one of them attempts to add its calculated path to the global grid. Despite this privatization, this benchmark cannot exploit the available TLP effectively because, no matter how many concurrent transactions are calculating paths in a given moment, only one of them will be able to update the global grid, while the rest will have to

rollback, even if their calculated paths do no overlap with the newly-committed path. For this reason, and provided the underlying TM system supports it, the benchmark can make use of the early release programming construct [124] to remove the global grid from the transaction's read set, after the grid copy has completed. Releasing addresses from the read set requires the grid points along the found path to be validated before the path is added, to make sure that none of the selected points has become part of another path after the grid copy.

ssca2. The SSCA2 benchmark focuses on one of the four graph kernels from the *Scalable Synthetic Compact Applications 2*. The selected kernel operates on a large graph to create an efficient representation using adjacency arrays and auxiliary arrays. Threads add nodes to the graph in parallel, using transactions to protect accesses to the adjacency arrays. The optimistic approach to concurrency control suits very well this code as the large number of graph nodes leads to infrequent concurrent updates of the same adjacency list.

vacation. This application implements a travel reservation agency system based on a 3-tier architecture, and whose designed is similar to the on-line transaction processing system of SPECjbb2000 [38], which emulates a very common type of server-side Java application. The database is implemented as a set of self-balancing trees that keep track of customers and their different booked items (flights, cars, rooms). Threads emulate a number of client connections that interact with the database server to reserve, cancel or update a booking. Each session is enclosed in a coarse-grain transaction that ensures a consistent view of the database, producing a parallel TM application that is very similar to its sequential counterpart. In comparison, a traditional lock-based parallelization would require non-trivial efficient fine-grain locking schemes on the data structures that conform the database, in order to exploit the large amounts of TLP available in this program.

yada. This benchmark replaced an earlier implementation of Ruppert's algorithm for creating quality Delaunay triangulations. The algorithm takes a planar straight-line graph and returns a conforming Delaunay triangulation of only quality triangles. The data structures used for mesh refinement are a graph where triangles are kept, a set that contains boundary segments, and a task queue that holds the poor-quality triangles that need to be refined. Each iteration removes a poor-quality triangle from the work queue and retriangulates on the mesh, adding to the work queue any new skinny triangles that result from the retriangulation. Transactions protect the removal and addition to the work queue, as well as the refinement of each triangle on the global graph, in a similar way to [74].

3.3.1 Workload Characterization

Table 3.4: Count, read and write set sizes for the transactions in STAMP.

	TID0			TID1			TID2			TID3			TID4		
	#	R	W	#	R	W	#	R	W	#	R	W	#	R	W
genome+	2731	75	1	481	1	1	14911	15	4	481	12	3	879	5	2
genome	1366	66	1	241	1	1	3615	16	4	241	12	3	449	4	2
intruder+	18307	3	1	18306	27	6	18306	2	1	-	-	-	-	-	-
intruder	3754	3	1	3753	20	6	3753	3	2	-	-	-	-	-	-
kmeans-high	6144	7	2	2046	1	1	3	2	1	-	-	-	-	-	-
kmeans-low	8192	7	2	2728	1	1	4	2	1	-	-	-	-	-	-
labyrinth	97	4	1	96	144	218	1	9	3	-	-	-	-	-	-
ssca2+	1	1	1	1	1	1	93683	3	2	-	-	-	-	-	-
ssca2	1	1	1	1	1	1	47255	3	2	-	-	-	-	-	-
vacation-high	3671	70	8	206	31	6	219	44	4	-	-	-	-	-	-
vacation-low	4014	56	7	34	28	6	48	28	3	-	-	-	-	-	-
yada+	3149	8	3	3148	1	0	2449	172	75	2449	1	1	2449	6	1
yada	1322	7	2	1321	1	0	705	166	75	705	1	1	705	8	2

Table 3.4 presents a detailed characterization of the transactions in these benchmarks, obtained from a single-threaded run. Transaction count (#) as well as average read (R) and write (W) set sizes are shown for each atomic block in the code –identified by a transaction ID or TID–. Read and write set sizes are given in number of cache lines.

genome. This benchmark has three distinct phases of execution. TID0 corresponds to the first phase, and is a coarse grain transaction that protects the elimination of duplicate elements from a hash table, implemented as a linked list. Thus, it has a large read set (*Rset*) but only one line in the write set (*Wset*). This phase dominates most of the parallel execution time of the workload. TID1 to TID3 are part of the second step of the algorithm, and have much smaller data sets. TID4 belongs to the third step. We stop simulation after this phase, before the sequence string is built, as that task is performed only by one thread.

intruder. The three transactions of the benchmark are executed the same number of times as part of a loop that iterates until no elements (packets) are left for processing. TID0 is a small transaction used to extract elements from a queue. TID1 is the main transaction of the benchmark, which protects the accesses to shared data during packet processing using coarse grain synchronization, hence its larger footprint (Rset and Wset sizes).

kmeans. Most of the runtime of this benchmark is non-transactional execution. Small transactions are used inside worker threads for the computation of cluster centers. TID0 protects the update of the new centers and accesses several lines

for read and write, while TID1 only accesses one line in order to update the index in the global task queue. TID2 is only executed when the worker function is completed, to update the global delta that determines if another iteration of the algorithm is performed.

labyrinth. TID0 extracts a pair of points from a global job queue, and then TID1 attempts to find a route between the two. Virtually all the execution time is spent inside TID1, which accesses a huge number of cache lines. This is due to the privatization of the global data structure, which is copied into a local matrix. As shown in Table 3.4, the Rset is lower than the Wset because read lines belonging to the global matrix are released using the *early release* programming construct. It should be noted that the algorithm exhibits divergent executions, and depending on thread interleavings it can reach different solutions in each run (i.e. a different number of paths routed) leading to substantial deviations in execution times. Hence, improvements in its performance must be observed with caution.

ssca2. After a significant amount of time in non-transactional execution, the benchmark has a phase in which a small transaction such as TID2 is executed a huge number of times. For this benchmark, we stop simulation after all adjacency lists of all vertex have been inspected, as there are is no more synchronization required in the remaining part of the parallel execution.

vacation. Each TID correspond to a type of database transaction. TID0 are *user* queries that only perform reservations and do not update the tables, thus resulting in read-only operations for the most part. TID1 correspond to the deletion of a customer from a database table, while TID2 updates the corresponding table with a new travel item, depending on its type. The size of the database makes all three coarse grain transactions have a large Rset. Given the input arguments, TID0 is the most frequent type of transaction in both cases, particularly in the low contention configuration, while the high contention configuration executes more instances of TID1 and TID2.

yada. This benchmark has a wide variety of transactions. TID0 removes an element from a priority heap for processing, and thus it is a small transaction. TID1 and TID3 are tiny transactions to handle the delayed de-allocation of elements. TID2 constitutes the main transaction of the benchmark (triangle refinement), which accounts for the majority of the execution time. We can see in Table 3.4 how it has a huge Rset of over 150 lines, as well as a very large Wset of 75 lines, which gives an idea of its coarse granularity. In contrast, TID4 appears as a transaction with small footprint, even though it wraps a loop that accesses an array inside a coarse grain transaction. TID5, not shown in the table for clarity,

is a small read-modify-write transaction which is only executed once per thread, at the end of the parallel phase.

3.4 CMP Architecture

We choose a tiled CMP design as reference because its modular nature has made it popular in several commercial many-core designs [51,67,68] and the availability of reliable simulation models [86] makes comparison of various policies and architectural features less daunting. The basic architecture comprises several tiles overlaid over point-to-point interconnects forming a mesh-based network-on-chip. This arrangement is depicted in Figure 3.2.

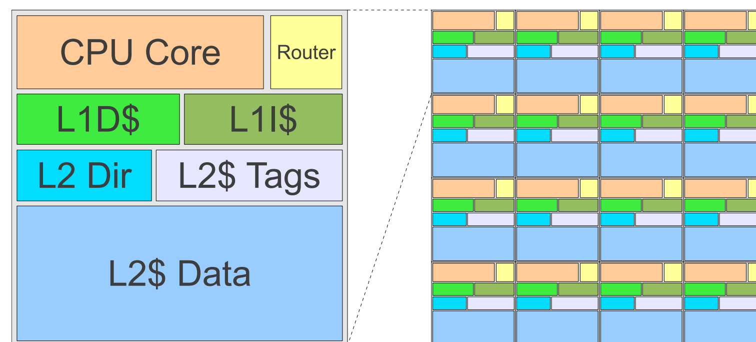


Figure 3.2: Organization of a tile and a 4×4 tiled CMP.

Each tile has a processing core, one level of private cache, a slice of the level-two cache along with its directory entries, and some routing logic. Considering that part of the appeal of CMPs is their ability to exploit TLP and provide higher throughput than a wider-issue uniprocessor while consuming less energy per operation, we have modeled the processing cores of our CMP architecture after lightweight, in-order processors. Split instruction and data caches are available at the private level, while the second level is unified, physically distributed but logically shared amongst all processing cores. Private caches are kept coherent across the unordered network through an on-chip distributed directory protocol. The L1 caches maintain inclusion with the L2 cache, trading off some on-chip capacity for lower design complexity in the coherence controllers. Each L2 bank includes the directory entries to keep cache coherence for the lines that belong to this tile. Each directory entry contains a full bit-vector to track the sharers of the line.

Table 3.5: System parameters.

Core Settings	
Cores	16, single issue, in-order
CPI non-memory instr.	1 cycle
Memory Settings	
Cache line size	64 bytes
L1 I&D caches	Private, 32KB, 4-way, split
L1 cache hit time	1-cycle latency
Write Buffer	Non-coherent, private, 128 bytes
L2 cache	Shared, 8MB, 8-way, unified
L2 cache hit time	6(tags/directory) + 6(data) cycles
Memory access time	300 cycles
Page size	4KB
Network Settings	
Topology	2-dimensional mesh
Link latency	1 cycle
Link bandwidth	40 bytes/cycle (between internal nodes)
	160 bytes/cycle (to memory)
Message size	8 bytes (control) / 72 bytes (data)
Flit size	16 bytes
Routing	Deterministic X-Y

3.5 Baseline HTM systems

The simulation infrastructure provides support for two generic designs that explore opposite corners of the HTM design space: LogTM-SE [153] as the eager-eager (EE) system of choice, and a global commit token-based, lazy-lazy (LL) system as described by Bobba *et al.* in [22]. Figure 3.3 shows a hardware overview of the simulated HTM systems. Common TM components to both EE and LL systems are shaded in blue, while EE and LL-specific components are depicted in yellow and orange, respectively.

3.5.1 Eager-Eager HTM Overview

The EE system is basically an extension to LogTM [92] in which Bloom filters –referred to as *address signatures*– are used for transactional read and write-set bookkeeping, instead of the traditional bits in private caches. LogTM performs

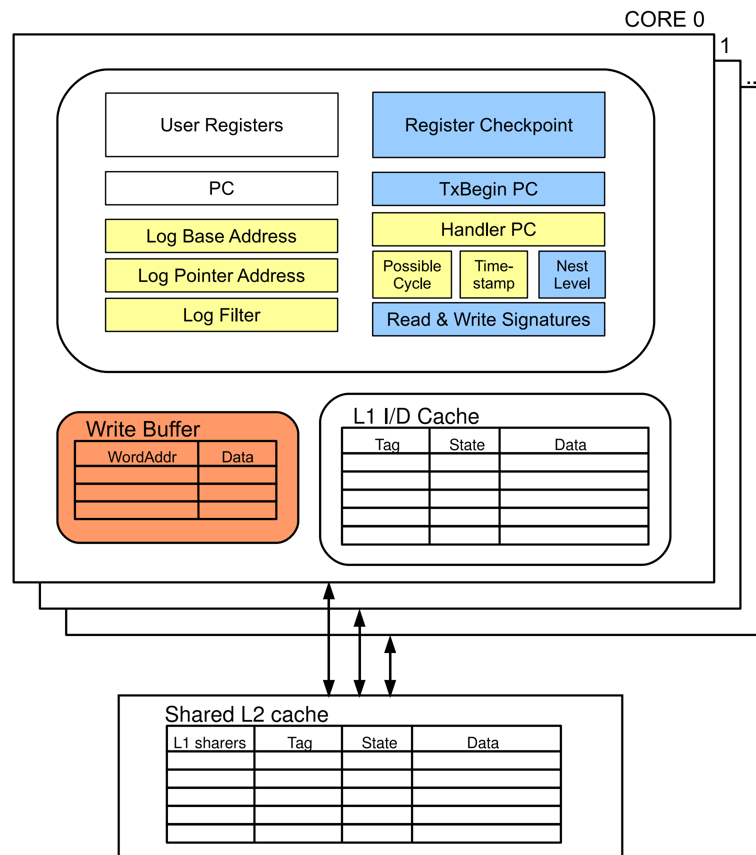


Figure 3.3: Hardware overview of baseline EE and LL HTM systems.

in-place speculative updates and uses a per-thread log in cacheable virtual memory to back-up the consistent values before the store completes. The logging hardware is responsible for copying the value of each line to the address indicated by the log pointer. A log filter maintains recently written addresses to minimize redundant logging. The basic policy for resolving conflicts in LogTM is as simple as stalling the requester, resorting to a conservative deadlock avoidance mechanism based on timestamps. A processor sets the *possible cycle* bit if it sends a negative acknowledgement (*nack*) to an older transaction upon detection of a conflict. If in turn it receives a nack from an older transaction, this represents a potential cycle and the transaction aborts. On abort it traps to a software handler, which walks the transaction log restoring the consistent values into memory, and finally restores processor state with the register checkpoint, pointing the program counter back to the beginning of the transaction. A standard MESI-style

coherence protocol is extended to support conflict detection despite evictions of transactional data from L1 or even L2 cache. L1 cache transactional overflows are elegantly handled by LogTM using sticky-states [92]. Nested transactions are trivially handled by flattening, i.e. only committing transactional state when the outer-most transaction commits, by simply keeping a counter with the current nesting level. The more uncommon L2 cache transactional evictions entail the loss of all coherence and directory information (sticky states) for the line, and are handled by extending the protocol with a new *filter check* coherence request message that is broadcast to all cores upon an L2 cache miss. Based on each core's response to the filter check, the directory is able to determine if the transactional owner/sharers and properly rebuild the sharing code, in order to retain isolation despite the L2 replacement.

3.5.2 Lazy-Lazy HTM Overview

The LL system in GEMS is modeled after designs such as Stanford's Transactional Coherence and Consistency (TCC) [56]. Unlike TCC –which is based on snoopy coherence over a shared bus–, the LL HTM flavour of GEMS is implemented on top of the tiled CMP architecture described in the previous section, and hence it leverages distributed directory coherence over an unordered network to detect conflicts. It reuses the same directory protocol and signature-check logic as the EE system, but it models a lazy approach to version management by incorporating a write buffer (see Figure 3.3) that sits between processor and L1 cache. All speculative updates are kept in this infinite-sized write buffer until transactions are granted permission to commit, so that during the execution of the transaction only loads are issued to the memory hierarchy. No space virtualization mechanisms are used, as there are no overflows of transactional data due to limited capacity or associativity of private hardware structures. The LL system then resolves conflicts when a committing transaction broadcasts its write-set. In its basic form it employs a bus to serialize transaction commits, similar to TCC. Transactions arbitrate for a global commit token using a zero-cycle bus. Once commit has been granted, a transaction issues exclusive coherence requests for the lines in its write set, effectively publishing the new globally visible values by invalidating all remote copies and aborting any transactional readers. We then augmented the simulator framework with an implementation of a more sophisticated commit scheme that closely follows the algorithm of Scalable TCC (STCC) [29]. STCC employs selective locking of directory banks to avoid arbitration delays and thereby improve commit throughput.

3.6 Validation of Simulation Framework

In this section, we present a performance characterization of the baseline HTM systems that will act as a reference points to measure the contributions presented in this thesis. We identify several pathological interactions and provide solutions to avoid the appearance of unacceptable levels of noise in our experiments. Some important insights on the simulation framework that we have learned by thorough experimentation and analysis of the obtained data are described. Our findings reveal such situations that hinder application scalability, and explain the causes of the unexpected performance degradation in the baseline HTM architecture that we observe. During this process of simulator and workload fine-tuning, we compared the results of our experiments to the STAMP suite characterization paper [25] for validation. After the incremental adjustments reported in the following paragraphs, our simulation framework was able to exhibit workload scalability figures somehow similar to Cao Min's study [25] as well as other works found in the TM literature [20, 143]. The aim is to establish a clear and reproducible simulation environment that yields the best performance numbers for the well-known baseline HTM designs used throughout this thesis, with the intent of validating the environment our proposals are evaluated upon.

3.6.1 Basic setup

Simics. We use Simics 3.0.31 to boot a Solaris 10 box with 16 processors and 4GB of memory, using the *abisko* target machine (Sun Fire 6800 server with UltraSPARC-III Cu processors). Once the system boots, we switch it to single user mode (*runlevel 1*) in order to reduce its load and keep the number of background processes to a minimum, in order to avoid interference of other processes with our experiments. Before the benchmark is executed, we use the command `psrset` to create 15 processor sets, and then `psradm` to assign each processor to a different processor set and disable interrupts (except for processor number 0).

Workloads. Following the workload setup guide found in the GEMS documentation, during the initialization of the benchmark we use Solaris' `processor_bind` to bind threads to the aforementioned processor sets, so as to avoid thread-migration in our simulations. When compiling labyrinth, we enable the use of the *early release* construct. Padding was added to some data structures in the STAMP library files (lib/random.c) in order to avoid false sharing between thread-local data. In vacation, for example, a data objects used for random number generation that ought to be private to each thread fell in

the same cache line due to consecutive malloc calls, creating false conflicts that unfairly penalized performance. In genome, calls to glibc functions `strcmp` and `strncmp` where replaced with our own thread-safe implementations, to stop lock-related system calls (`lwp_mutex_timedlock/lwp_mutex_wakeup`) from suspending the thread while inside the transaction.

GEMS. In order to isolate our performance measurements from the interference of conflicts due to false positives inherent to hash-encoding, we use *perfect* signatures – mere address lists – for the experiments presented in this thesis, except where noted otherwise. The use of perfect signatures makes possible the removal of a given address from the read set, enabling support for the *early release* programming language construct in the simulated HTM system. This allowed us to maintain the benchmark labyrinth as a relevant workload.
All simulated systems employ backoff to reduce contention after abort. In the case of the EE system, we replaced the exponential back-off performed inside the software handler (some dummy calculations on a local array) with a linear back-off algorithm whose length is directly controlled by stalling the Simics processor for the computed number of cycles, similar to the back-off implementation of the LL flavour. The same linear back-off algorithm is then used for both EE and LL systems, making the comparison between them fairer. Thanks to this modification, the noise introduced in L1 hit rates due to software back-off is removed: The original software-based back-off resulted in an artificial increase in the number of cache hits that polluted our miss rate measurements of the memory hierarchy.

3.6.2 Interference from the Operating System

Figure 3.4 shows the workload scalability of the LogTM-SE system for the STAMP benchmark suite, for configurations from 2 to 16 threads, measured as speedup over a single-threaded run. The 16-core CMP architecture described in the previous section is used for all experiments. When the thread count is lower than the number of available cores, we disconnect all unused processors (except processor 0) from the timing module in order to speed up simulation time. We verified the validity of this approximation by running experiments in which all processors were simulated regardless of the number of threads, as we obtained very similar results. Processor 0 is left idle (for OS usage) and the benchmark is run in cores 1 to n, depending on the number of threads.

For 16-thread runs, all 16 cores are used to run the benchmark, including processor 0. In certain occasions, the operating system interrupts the execution of

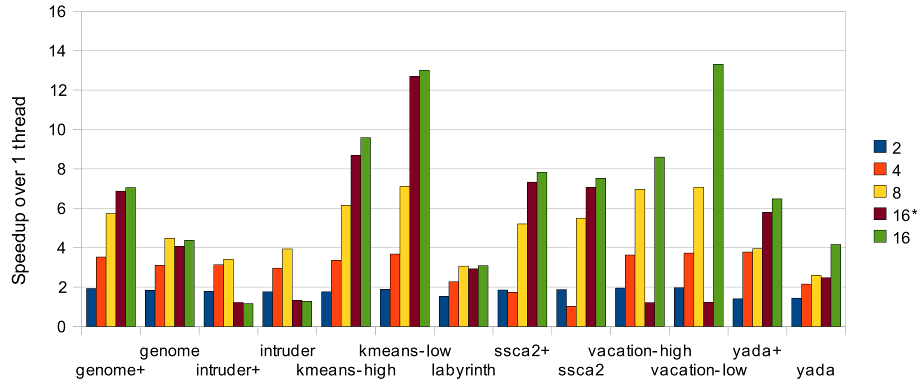


Figure 3.4: Workload scalability on LogTM, with and without OS interference.

the benchmark and regains control over processor 0 to execute some completely-unrelated code, degrading the performance of the application. As we can observe in plots 16* of Figure 3.4, such is the case of vacation: The OS interrupt happens in the middle of a transaction and takes much longer than the duration of the benchmark itself, making all other threads wait at the final barrier for millions of cycles, thus completely jeopardizing performance. For yada, the weight of the OS portion is not as drastic but it still stops the benchmark from scaling as it should.

Undesired interference caused by the OS can be dealt with in a variety of ways. Yen's approach [152] to solve this problem was to reduce the thread count to 15 when running the benchmarks on a 16-core system. We found a different solution that allowed us to maintain the number of threads as a power of 2. We decide to include an extra processor in the functional simulator, and reserve it for the exclusive use of the operating system. Hence, when running 16-thread configurations we boot Simics with not 16 but 17 processors, and then bind our user threads to processors 1 to 16, leaving processor 0 idle and available for the OS to run any service or attend any interrupt that may arise during the execution of the benchmark. From the perspective of the memory subsystem, however, we make it look as if there are still 16 cores (Simics processors 1 to 16), while processor 0 is disconnected from the timing module. The results of this modification to the initial simulation infrastructure is the complete removal of OS-induced noise in our experiments, as we can observe in the green plots of Figure 3.4. It is important to observe that in spite of these changes, execution in supervisor mode is still properly simulated by the timing module on the cores running the benchmark, as user threads still cause exceptions (i.e. TLB misses)

and traps (i.e. system calls) that need OS support. For the remainder of this chapter and throughout the thesis, the results shown for 16 threads use this configuration with an extra *idle* core to avoid such OS interference.

3.6.3 Starvation of Writer Transactions

The basic conflict resolution policy of LogTM is susceptible to a type pathological interaction amongst transactions referred to as *starving writer* [22]: When a writer transaction attempts to acquire exclusive ownership over a line that is currently in the read set other transactions, the readers can nack the requester regardless of which transaction has higher priority –i.e. older timestamp–. The writer may starve if new readers arrive before existing readers commit. The pathology can still appear if existing readers abort instead of committing, if readers are aborted one at a time: Before one reader has released read isolation over the contended line, other readers may have restarted and read the line without observing a conflict, continuously depriving the writer from the opportunity to acquire exclusive ownership. A back-off scheme reduces contention and ensures that readers eventually delay their retry long enough for the writer to complete its exclusive request, and thus guarantees that the system makes forward progress. Nonetheless, the drop in performance becomes significant as contention increases with the number of threads. We observed this writer starvation in benchmarks like intruder, yada or vacation, though the performance degradation is most visible in the latter. For 16-thread configuration without interference of the OS (green plots), Figure 3.4 shows a 4X performance drop in the speedup achieved by vacation-high (8.6) in comparison to its low contention counterpart (13.3). This difference is largely due to the appearance of the aforementioned pathology.

As proposed by Bobba *et al.* [22], this starvation scenario can be solved by a *hybrid* conflict resolution policy that allows an older writer to simultaneously abort a number of younger readers. In this case, the readers abort themselves and immediately acknowledge the exclusive request, allowing the older writer to proceed with its transactional execution. For all other conflicts, the resolution is similar to the basic policy: stall the requester and rely on conservative deadlock avoidance to ensure forward progress. The advantages of using this hybrid resolution policy are shown in Figure 3.5. Comparing it to Figure 3.4 we can observe how it greatly improves the scalability of vacation-low, rising it from 8.6 to 12.7 for 16-threads, hence closing the gap with the high-contention configuration. All other benchmarks also benefit from this change of policy, although the

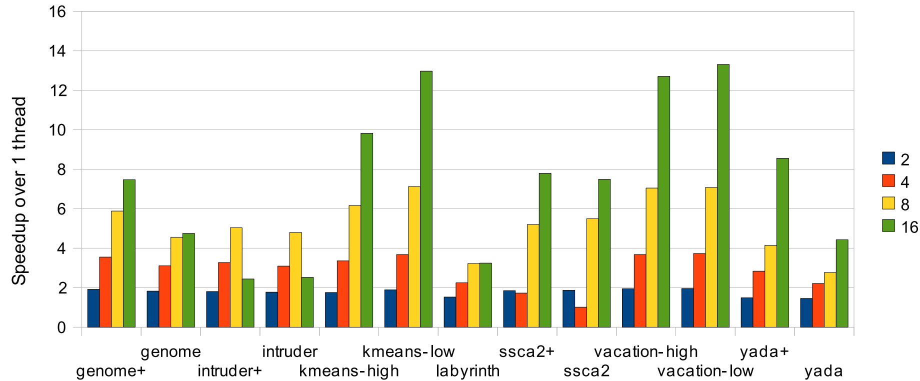


Figure 3.5: Workload scalability on LogTM, without writer starvation.

improvement is most noticeable in yada+ (from 6.5 to 8.5 for 16 threads) or intruder+ (from 3.4 to 5 for 8 threads).

3.6.4 Barrier Synchronization Overhead

In some of the STAMP benchmarks, the parallel computation is divided into several phases by means of barriers: Such is the case of genome and ssca2. When building the applications for its use with a simulator, binaries are compiled by default using Pthread barriers (`pthread_barrier_wait`). We observe in our simulations that resorting to a library call for barrier synchronization results in expensive system calls (lwp_park) that introduce significant overheads: In a sample 16-thread genome run, we observed barrier release latencies of between 70 and 130 thousand cycles for the first and last thread released the barrier, respectively, measured from the cycle on which the last thread arrived at the barrier. In a benchmark like genome, with frequent barriers and a total run-time of under 3 million cycles (16 threads), a barrier latency in the order of hundred thousand cycles makes this synchronization overhead have a significant weight in the total execution time.

Since barrier performance is critical for the scalability of benchmarks like genome, we decided to modify the simulation framework to eliminate this other source of noise from our evaluations. After all, our research does not concern barrier performance, and thus emulating ideal barriers is not only an acceptable assumption, but also a desirable feature that help isolate our experiments and performance analysis from interference that might unfairly hide the benefits of a given design. We hence modify both simulator and workload in order to

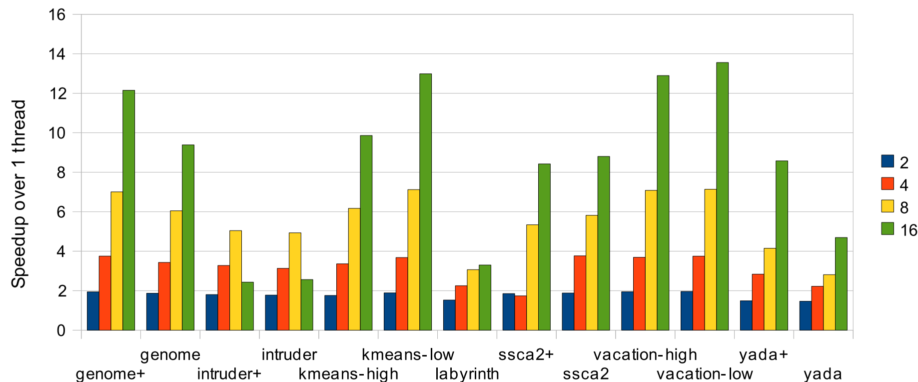


Figure 3.6: Workload scalability on LogTM, with ideal hardware barriers.

implement a zero-cycle hardware barrier. On the application side, we introduced a Simics magic call that signals the arrival to the barrier, followed by an infinite loop on which the thread is trapped and keeps spinning on a local register (`l1`). On the simulator side, we capture the new magic call and simply keep track of how many threads have reached the barrier so far. When the last thread arrives, we write a non-zero value in the `l1` register of each processor, releasing all threads from the otherwise infinite loop.

Figure 3.6 shows the scalability of STAMP after replacing the call to `pthread_barrier_wait` with a hardware barrier. The cumulative changes of previous subsections are kept from one subsection to the next (OS interference, hybrid policy). When compared to Figure 3.5, we can see how genome experiences a spectacular performance gain (genome+ raises from 7.4 to 12.1 in 16 threads) whose reason is the drastic reduction in the barrier synchronization overhead. The improvement in ssca2 is also noticeable (from 7.5 to 8.8).

3.6.5 Level-2 Cache Conflict Misses

Handling of system calls inside transactions is an open research topic [18,90,101], as non-trivial issues arise from the interaction of the operating system with transactions. Because system calls are executed in kernel mode and might modify system state, user mode rollback handlers are not applicable. In our simulation framework, all code executed in privileged mode is treated as non-transactional and its effects cannot be undone if the transaction is deemed to abort. To isolate applications from this intricacies, the STAMP benchmarks attempt to avoid system calls from taking place inside transactions. With this intent, a very simple

thread-local memory allocator is implemented in the STAMP library, whose function is to service malloc-like calls in the parallel region, providing memory from a thread-local pool without having to resort to the actual `malloc` call – and thus avoiding `mmap` system calls–.

However, this simple memory allocator is responsible for some undesired side-effects that can hurt performance as a result of the alignment of per-thread memory pools. By default, a total of 256MB are allocated in the initialization phase, and divided amongst threads. When 16 threads are used to run the benchmark, the initialization of the memory allocator *mallocs* 16 blocks of 16MB each. Since these memory blocks appear consecutively in memory, the initial portion of each block is at n times 16 MB of distance from the beginning of every other block. Regardless of the mapping policy the L2 cache, the result of this alignment is that all addresses at a given offset k within each memory block are mapped to the same set x of the same L2 cache bank b. When 16-thread configurations are run, the associativity of the L2 cache (8 ways) is not enough to contain the dynamically allocated lines for all threads, and L2 conflict misses significantly penalize the performance of the application, increasing the number of expensive accesses to main memory. For applications that make an intensive use of the dynamically allocated memory, this kind of L2 conflict misses hinder scalability to 16 threads, as is the case of labyrinth: Each thread works on a local matrix of 12KB that is dynamically allocated, and all matrices are mapped to the same (small) subset of the total number of L2 sets available across banks, resulting in incredibly high L2 miss rates (over 40%).

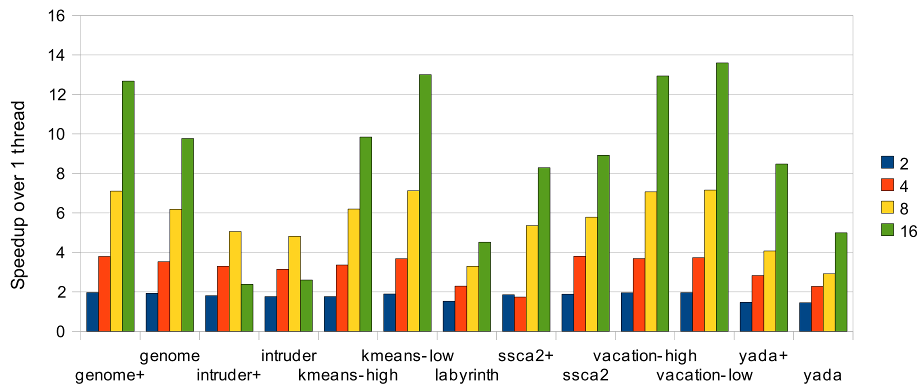


Figure 3.7: Workload scalability on LogTM after solving L2 conflict misses.

To address this pathological behaviour in the simulation framework, we

modify the memory management library of STAMP and increased the size of each block allocated during the initialization by a small amount, in order to avoid their alignment to 8MBs (the total size of the L2 cache in the modeled architecture). Blocks were displaced by an offset of 13KB, specifically targeted to avoid conflict misses amongst the local matrices used in labyrinth. With this change, the number of L2 cache misses in labyrinth was reduced to less than one fifth of its previous value, and its scalability was substantially improved, as shown in Figure 3.7. The rest of benchmarks experience less relevant improvements after addressing this issue.

3.6.6 Summary of Changes

Figure 3.8 summarizes the incremental changes made to the simulation framework, in order to obtain the best-performing EE HTM which will be used as baseline for comparison of the designs proposed in this dissertation. As we can see it, removing the pathological disturbances from the OS permits to maintain as part of our workloads a benchmark with long running transactions such as vacation. Furthermore, the change to a hybrid resolution policy that addresses the starving writer performance pathology allows vacation-high to reach its full potential, and brings considerable benefits too for applications like intruder and yada. The removal of the barrier synchronization overhead achieved by modeling a hardware barriers allows genome to show its authentic scalability, and also helps ssca2 to a lesser extent. Finally, solving the alignment problems in the thread-local, dynamically allocatable memory largely eliminates L2 conflict misses that penalized the scalability of labyrinth. Thus, the baseline EE HTM system used as reference in the evaluation sections of the following chapters includes all such changes.

3.6.7 Relative Performance of Lazy-Lazy HTM Systems

In the previous subsections we have focused on the performance of the EE system, which represents one end of the design space in hardware transactional memory. Now we present comparative performance numbers for the two LL designs considered, which explore opposite policies for the two key HTM design dimensions, version management and conflict resolution. The adjustments to the simulation environment described so far are maintained when measuring the relative performance of the LL systems, with the exception of the change in the

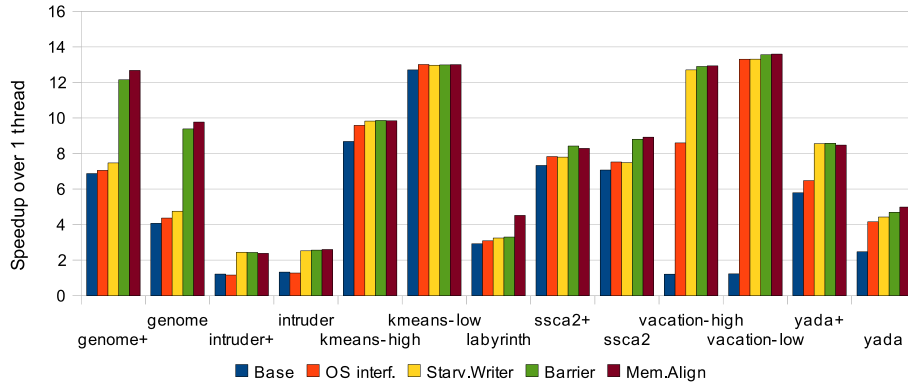


Figure 3.8: Evolution of scalability for the baseline EE system.

conflict resolution policy, which is only applicable to EE systems (the LL system always uses a *committer-wins* policy for resolving conflicts).

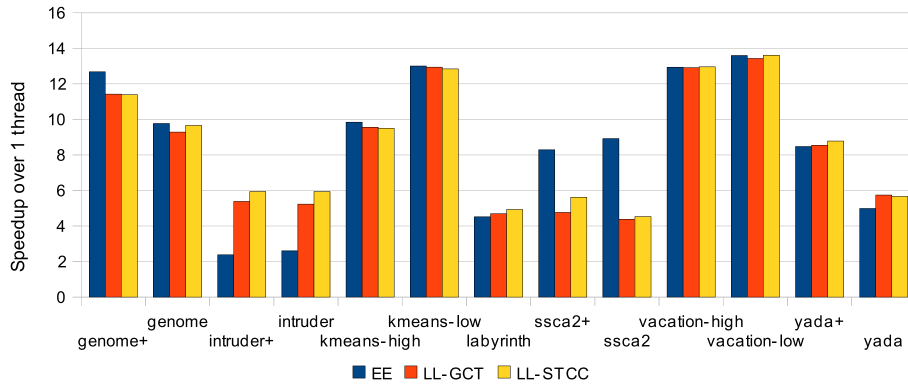


Figure 3.9: Relative workload scalability of both LL systems.

Figure 3.9 shows the speedup achieved by each considered HTM system, running 16 threads. The plot LL-GCT corresponds to the LL system that employs a global commit token for serializing commits, while the LL-STCC represents a similar LL system with a sophisticated scheme that follows the algorithm of Scalable TCC [29]. Results are normalized to the single-thread execution time of the EE system. We see in the figure how there is not a clear winner, with both policies performing similarly in most benchmarks, while outperforming each other in two of the benchmarks.

On the one hand, as exemplified by intruder, the LL approach is more efficient at exploiting parallelism in situations of high contention, for several reasons. First, resource acquisition (i.e. request of exclusive ownership over write-set lines) happens once a transaction has executed till completion and it is sure to commit its speculative updates to shared memory. Therefore, in a lazy system lines are only blocked when useful work is made, whereas in the EE approach a transaction can stop others from accessing its write set, without any guarantees that stalling them will allow itself to commit. In fact, the EE approach of *holding state before commit* leads to situations of unnecessary serialization, in which two independent transactions T_1 and T_3 –without mutual data dependencies– cannot execute in parallel because of eagerly-detected conflicts with a third transaction T_2 that *connects* them ($T_1 \rightarrow T_2 \rightarrow T_3$). Another reason of why EE systems do not perform well under contention is that there is no upper bound on the number of aborts caused by each commit, while in the LL system a committing transaction can only cause a maximum of n-1 aborts (n is number of threads).

On the other hand, for applications with a large number of small, low-contended transactions like ssca2, the EE system defeats the lazy approach, as transactions can commit instantly and in a completely local manner, without having arbitrate for a commit bus. In this case, the simple scheme of using a global token to serialize transaction commits results in a clear performance bottleneck, as this mechanism does not allow parallel commits whatsoever. The LL-STCC system employs selective locking of directory banks to avoid arbitration delays and allows transactions to commit in parallel as long as their written lines are mapped to disjoint sets of directory banks [29]. Nonetheless, STCC does not perform much better than LL-GCT. mainly because of the mapping policy to L2 cache banks that we have assumed so far.

L2 Cache Mapping Policy. The L2 cache level in this thesis is modeled as a *Non-Uniform Cache Access* (NUCA) design [71] in which are a set of cache banks distributed across the chip and connected through a point-to-point network. Although cache banks are physically distributed, they constitute a logically shared L2 cache. As such, the mapping of memory blocks to cache entries is not only defined by the cache set, but also by the cache bank. Most CMP architectures that implement NUCA caches map lines to cache banks by taking some fixed bits of the physical address of the block [71,77]. This physical mapping spreads blocks uniformly among cache banks, resulting in optimal utilization of the cache storage. In this and all the above subsections, results were obtained using a a coarse-grain interleaving of memory blocks to L2 banks –depicted in the left part of Figure 3.10–, by taking the lowest bits of the physical address to select the

L2 cache set. Under this mapping scheme, virtually every transaction in ssca2 writes to at least one common directory bank, rendering the more sophisticated mechanism of STCC useless.

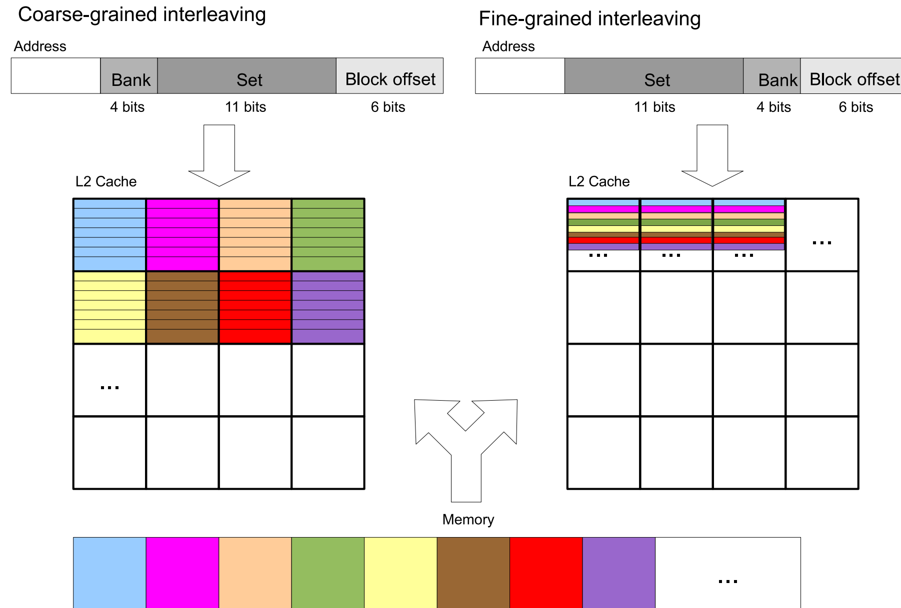


Figure 3.10: L2 bank mapping policies.

Commonly, the bits taken to select the cache bank for a particular block are the less significant ones, leading to a fine-grained interleaving of lines across L2 cache banks [112], as shown in Figure 3.10 (right). We thus adjust our simulated CMP architecture according to this alternative mapping policy, in order to observe the full benefits of the parallel commit scheme of LL-STCC. Figure 3.11 shows the speedup of both EE and LL-STCC systems, using the two alternative policies. Speedups are over the single-threaded execution on EE, with the initial interleaving at a coarse granularity (L2 banks). Starred plots correspond to the fine-grain interleaving configurations, EE* and LL-STCC*. For the majority of the workloads, altering the distribution of data across the on-chip storage only causes slight performance variations. In ssca2, on the contrary, we observe that LL-STCC now clearly outperforms the LL-GCT, as a consequence of the reduction in the commit arbitration delays achieved by the selective locking of L2 banks, which in many cases allows transactions to commit in parallel. Nevertheless, as the EE system introduces nearly no synchronization overhead

(as commits are by far the common case outcome), it still proves to be more efficient than LL-STCC, whose communication latencies with directory banks are in the critical path and account for a significant portion of the total duration of the transactions –given their small size–.

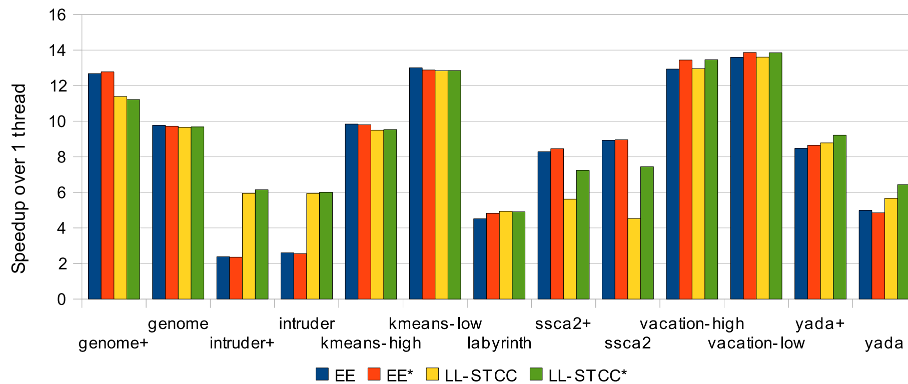


Figure 3.11: Performance of EE and LL-STCC systems with L2 mapping policies.

In summary, given the little impact that the L2 mapping policy has in the performance of the EE system –shown in Figure 3.11–, a fine-grain interleaving scheme is chosen in our evaluations for the remainder of this thesis, so as to establish a fair comparison against the LL-STCC system.

3.6.8 Workload Speedup on Single Global Lock

In order to measure the degree to which the programmer's effort is responsible for the performance achieved by each application –and thus the role played by the underlying TM system– in this subsection we measure the scalability of these workloads in our simulation framework using a single global lock –and not transactions– as synchronization method. To accomplish this, we simply replace tx-begin/tx-end instructions with pthread_mutex_lock/unlock calls, hence keeping the programming effort and complexity of reasoning the same as when transactions protect accesses to shared data. Figure 3.12 shows the scalability measurements when transactions are replaced with a single global lock.

Unlike other benchmarks [151] whose *transactified* versions have been previously used to evaluate HTM proposals [29, 92], the STAMP applications were developed while keeping in mind that the main appeal of TM for programmers is the ability to write simple parallel code with frequent use of coarse-grain

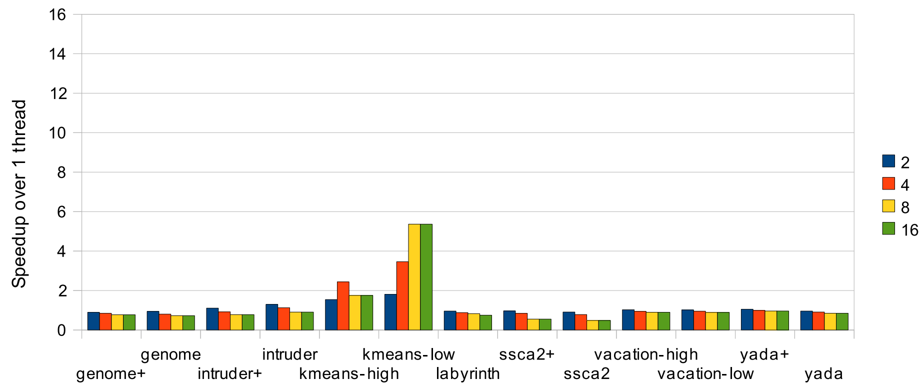


Figure 3.12: Workload scalability using a single global lock for synchronization.

transactions that perform as well as code that has been carefully optimize to use fine-grain locks. Figure 3.12 quantitatively shows that the underlying TM system largely determines the performance and scalability of these workloads, as opposed to other benchmarks [151] developed with fine-grain locking strategies, which see little or no benefit when replacing lock-base synchronization with transactions [92]. In summary, this reveals the idoneity of STAMP as the most representative set of benchmarks on which we can evaluate the proposals of this thesis.

Chapter 4

A Directory-Based Scheme for Detection of Transactional Conflicts

4.1 Introduction

The detection of conflicts amongst concurrent transactions is one of the key aspects that any HTM system must address. The decision of *when* to detect and *how* to resolve conflicts over the course of a transaction constitutes a major design dimension of the TM system. The choice of one or other policy for managing conflicts has broad implications in both performance and implementation costs of adding TM support onto a tiled CMP substrate.

The check for conflicts can be done on each individual memory request or it can be deferred until the end of the transaction. While the latter approach opens up more opportunities for parallelism by allowing the coexistence of multiple transactional reads and writes to the same line, the implementation of lazy commits in hardware is not straightforward: The commit logic must retrieve every address in the write set of the transaction and publish it –issuing coherence requests to acquire exclusive ownership– to make the updates globally visible, squashing any other concurrent accessor of the committed data.

Eager-eager (EE) HTM designs [92, 153], on the other hand, adopt a more evolutionary step towards the adoption of hardware support for TM, at the cost of sacrificing some concurrency. From the perspective of the cache controller, memory accesses generated within a transaction –including writes– are no different from the rest: Just like in the non-transactional case, exclusive ownership

must be acquired over a cache line before a transactional store can complete. In turn, a cache controller simply needs to monitor the coherence traffic and check against its local transaction's read and write sets, in order to detect if a remote transaction is attempting a conflicting memory reference. EE systems commonly resolve conflicts by stalling the requester. The coherence protocol is augmented with negative acknowledgement messages (*nacks*), sent in response to conflicting requests. A *nacked* transaction is forced to stall while it retries the conflicting memory access. Aborts are triggered when possible cyclic dependencies are detected, to avoid deadlocks. In this way, once a transaction has written a line, its access is blocked to others until commit or abort, ensuring that speculatively written values remain hidden. Similarly, a transaction impedes others from writing to lines that belong to its read set.

The eager approach to resource acquisition greatly simplifies the implementation of the commit instruction: Since all write set cache lines are acquired with exclusive ownership prior to commit, the only step left to make the tentative changes visible across the system is as simple as clearing the read and write set meta-data. Unfortunately, this strategy of booking resources *a priori* –before the transaction is certain to succeed– makes eager systems inherently less efficient at exploiting parallelism than their lazy opponents. By *holding state* before being guaranteed to commit, eager transactions create chains of conflicts through which otherwise independent transactions become unnecessarily serialized due to the transitivity of dependencies. In spite of this fundamental disadvantage, their lower implementation complexity and the modularity of their components still makes eager solutions an appealing choice. For instance, the applications of the logging circuitry go beyond the scope of TM, and could be of use in diverse domains such as fault tolerance [115].

Many HTM systems proposed to date implement eager policies [9,60,80,92,107, 153]. One common aspect of all these proposals is that they store the transactional bookkeeping information in structures that are private to the processor running the transaction. The meta-data is kept in places that are directly accessible by the private cache controller, which is conveniently modified to interpret such information and use it to check for data races. In other words, conflict detection has invariantly been performed at the private cache levels of the memory hierarchy, regardless of the bookkeeping scheme –bits in cache, signatures, etc–. This makes the most sense when private caches are able to snoop on every memory reference that takes place across the system. While this invariably happens in bus-based systems [56], it constitutes an abnormal behaviour for a directory-based protocol that maintains coherency over an unordered, point-to-

point interconnect, as that of a tiled CMP. Despite this substantially different scenario, eager conflict detection schemes for directory-based HTM systems have so far implicitly inherited the same style of *cache-level* conflict detection [92, 153].

When directories are used to maintain coherence, a cache controller generally receives requests only for those memory blocks of which it has a copy. Limited cache capacity or associativity cause replacements of active transactional lines that break the transactional status-cache residence connection upon which a basic directory protocol relies to detect conflicts. To fully harness the advantages of eager version management and enable transactions of larger footprints, previous work proposes to augment a directory coherence protocol with *sticky states* [92], so that transactions are still capable of retaining isolation over lines that are no longer privately cached. While extending a directory-based system with sticky states does not entail a substantial increase in complexity, it simply constitutes a fix to help maintain the same strategy of cache-level conflict detection found in bus-based systems. However, the particular characteristics of directory-based systems have not been thoroughly analyzed in the context of an HTM design.

Previous work on directory-based eager HTM systems has not addressed a major source of inefficiency that arises as a consequence of the directory's obliviousness to transactional status and priorities. Using the directory as a mere router that simply forwards messages to the appropriate destinations, has the implicit effect of restricting the throughput of conflict detection to the pace at which the directory can process coherence requests. This means that an HTM system can only resolve conflicts as fast as its directory can process the coherence requests that create them. Since the directory acts as the serialization point for the requests to a memory location, it cannot process new requests until it receives confirmation that the previous one has completed. Although this limitation in concurrency affects directory coherency in general, the situation is aggravated to the point of pathological performance when transactions are introduced.

In this chapter, we propose a novel approach to eager conflict detection that extends the directory logic in order to provide a fast detection scheme in tiled CMP architectures. Where previous proposals had combined these two related but distinct roles [92, 153], we propose to decouple conflict detection from cache coherence at the directory level, in order to overcome pathological situations that degrade the performance of an eager HTM system. We demonstrate that traditional *cache-based* conflict detection introduces several sources of inefficiency when used in the context of a directory protocol, and show how under situations of high contention the directory becomes a bottleneck for the conflict detection mechanism. We propose an alternative solution, a *directory-based* scheme, which

moves transactional bookkeeping from caches to the directory. By comparing our proposal with an HTM system such as LogTM-SE [153], we observe several advantages of implementing conflict detection at the directory level instead of at the cache level. These are the main contributions of our proposal:

Faster conflict detection with less traffic. Detection itself is accelerated, as conflicts are mainly detected in one hop instead of two, and the number of messages generated for the task of conflict detection is reduced. This is important for eager systems that attempt to resolve conflicts through stalls rather than aborts. If the system implements a simple retry scheme –as it is the case of LogTM-SE– the savings in network traffic grow in significance for workloads composed by coarse grain transactions with long run times.

Higher conflict resolution throughput during contention. While detecting conflicts sooner does not directly imply that conflicting transactions will serialize sooner, our design dispatches conflicting requests without forwarding additional coherence messages, and thus without blocking the directory. This allows quicker reaction to high-contention scenarios in which the same line is accessed by several conflicting transactions, and it has the potential to avoid many aborted transactions, improving performance.

More efficient transactional bookkeeping. Having each tile track its transactional addresses is not an efficient global encoding, as transactions often access the same shared data and keep redundant meta-data on their read and write sets, which in turn may lead to false positives if signatures are used for transactional bookkeeping. In contrast, our scheme extends the role of the directory not only to map addresses with cache residence, but also with transactional ownership.

Modularity. The new functionality can be introduced as a separate hardware module that acts as a directory-level conflict controller, capable of working independently of the coherence controller. Hence, the coherence protocol remains largely unmodified.

4.2 Motivation

In this section, we discuss a number of reasons that support our claim that the directory is a well-suited location for the detection of transactional conflicts in HTM systems that detect and resolve conflicts eagerly.

4.2.1 Decoupling conflict detection from cache coherence

Maintaining coherence means guaranteeing that all processors see the writes to a given location as having happened in the same order. In a bus-based system, all accesses to any location are serialized by the order in which requests appear on the bus. In a distributed system with coherent caching, such as the baseline multicore architecture of this thesis, it is the directory that acts as the serialization point for all the requests to the same memory location, since all relevant operations first come to the home tile. A common solution to ensure serialization to a location while keeping the complexity of the protocol low is to use additional directory states called *busy* states [39]. In Figure 4.1 we can observe a partial snapshot of the state machine implemented at the directory level in the baseline CMP modelled throughout this thesis. Busy states are depicted in grey colour.

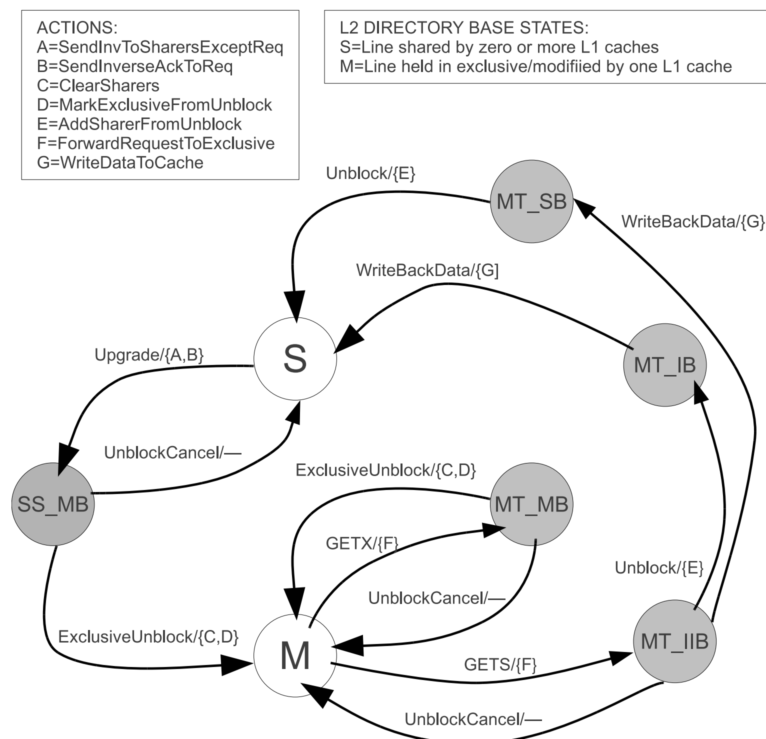


Figure 4.1: Example of busy states in a directory protocol state machine.

In this protocol, cache misses must finish with the requesting tile sending an *unblock* message to the home tile once data and acknowledgements have

been obtained. This message notifies the home tile that the in-progress miss has been completed, and allows it to process the subsequent requests for the same block. Figure 4.1 also shows *unblock cancel* messages that indicate that the coherence transaction has failed –the requestor could not obtain the appropriate data and coherence permissions– due to a transactional conflict. In these cases, the directory simply reverts the line to the initial base state. For clarity, those unblock messages sent by a requestor after it has acquired exclusive ownership are denoted as *exclusive unblocks* in the figure.

When a line is in a busy coherence state, subsequent requests that target the same block must wait until an unblock message is received from the last requestor, indicating the (perhaps unsuccessful) completion of the previous coherence transaction. Only when the line returns to a *base* state is the next queued request considered. The result of this serialization is that coherence requests targeted to the same line can pile up in the input buffers of the directory controller when a cache line experiences high contention. The situation is much worse in the context of transactions, because the directory ignores whether a given requests is conflicting, and is unaware of the priority scheme used by transactions. Oblivious to the status of the transactions running on a given moment, the directory always attends messages in a first-in-first-out (FIFO) basis. As a consequence, undesired scenarios may arise during high contention: low priority requests are serviced while high priority ones are sitting at the input buffers, when indeed the former do not produce in any useful work since their transactions will probably end up aborting as a result of the conflict. The scenario is depicted in the left part of Figure 4.2. The figure shows a transactional interleaving in which four transactions access the same cache line, which are initially in the read set of transactions running in processors P0 and P1, whose caches have shared copies of the line. We can see how a non-conflicting shared request from P3 must wait for other conflicting requests that arrived earlier at the directory, delaying unnecessarily the commit of the reader transaction. Furthermore, the directory's obliviousness to timestamps makes the high-priority exclusive request from P0 wait for a lower priority request from P2, which does not succeed in acquiring exclusive ownership.

In contrast to the behaviour of the cache-based approach to conflict detection shown in left part of Figure 4.2, the directory could handle contention more efficiently, provided it had information about the transaction's timestamp and read and write sets. In the right part of Figure 4.2 we see how a directory-based scheme of conflict detection serializes more quickly the same transactional interleaving described earlier. In this other case, the directory detects the conflict

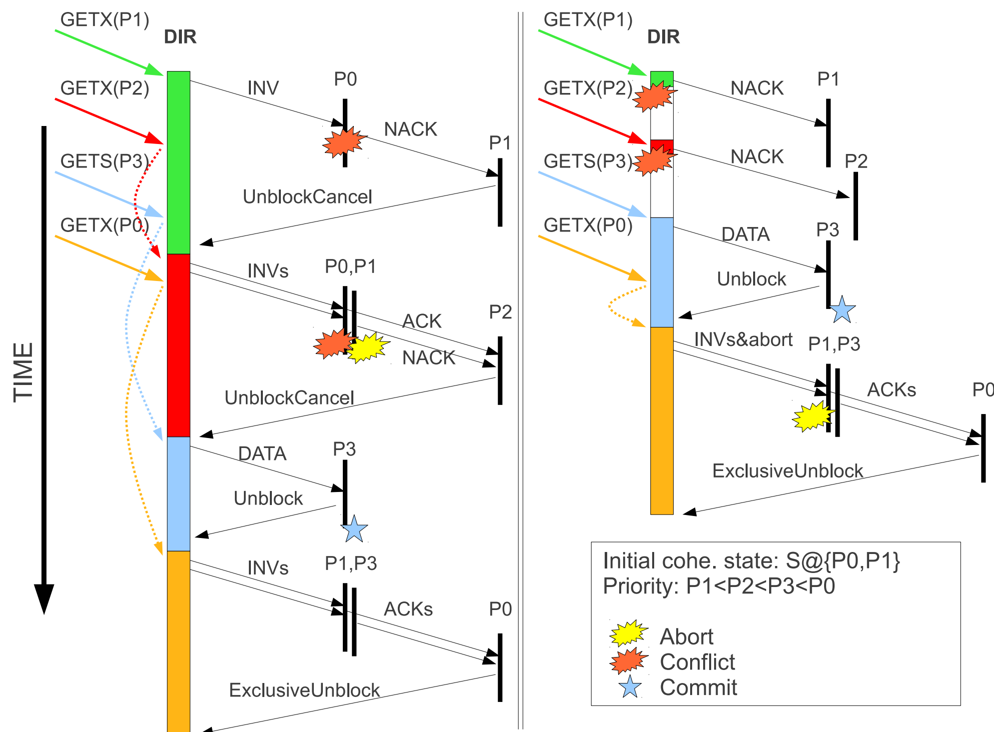


Figure 4.2: Cache-based vs. directory-based conflict detection.

itself using its own bookkeeping meta-data, and thus does not need to forward coherence requests to the private caches that have a copy of the line. As a result, the line does not need to go into a busy state and non-conflicting or high-priority requests can be dispatched without having to wait for the completion of coherence requests known not to make any useful work.

4.2.2 Reducing traffic generated during stalls

Eager systems generally attempt to resolve conflicts using a *requester stalls* policy [22, 92, 153]. Unlike other resolution policies that always abort all but one of the conflicting transactions –like *requester wins*–, the strategy of stalling the requester tries to save the computations performed up to the conflicting memory reference. Upon reception of a nack response indicating a conflict, the processor stalls, retries its coherence operation, and aborts on a possible deadlock cycle. If no cyclic dependencies are formed, this policy can successfully commit both transactions in spite of the conflict, by serializing their execution after the conflicting point.

The simplest way to implement this policy is simply to retry the conflicting coherence request a few cycles after the nack response has arrived. From the perspective of the processor, the conflicting memory access appears as a long-latency miss that takes hundreds of cycles to be serviced. The conflicting coherence request is retried once and again and again, until it succeeds in bringing the data with the right permissions, or until the deadlock detection mechanism indicates the possibility of a cyclic dependency. This is the solution adopted by the popular LogTM design. More sophisticated methods are possible, including trapping to a software contention manager, using an adaptive back-off algorithm to calculate the interval between retries, or even implementing a notification mechanism to report transaction commit/abort to stalled requestors.

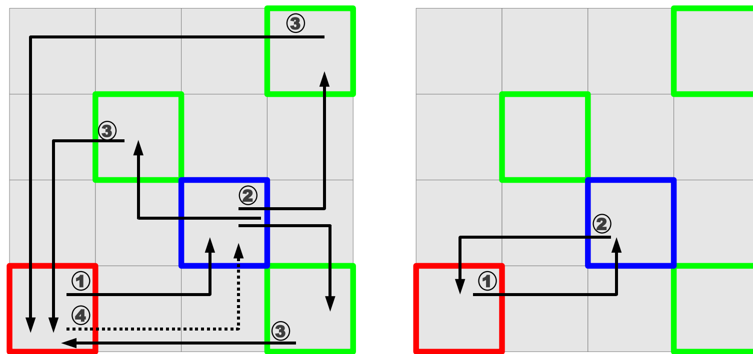


Figure 4.3: Messages generated on a write-read conflict in cache-based vs. directory-based conflict detection.

The straightforward *nack-and-retry* scheme has the advantage of its simplicity, but results in a substantial amount of network messages generated when we consider the entire length of the stall, specially if the *nacker* is a long-running transaction. This is particularly relevant for write-read conflicts, i.e. when a transaction attempts to write a line that is in the read set of one or more concurrent transactions. While on a bus-based system the number of bus transactions generated in this case is always constant, in a directory-based system there is an individual invalidation message sent from the home tile to each sharer of the line, and one negative acknowledgement response for each invalidation. In general, detecting a conflict on a cache-based approach to detection takes 2+2n messages, where n is the number of bits set in the bit-vector kept at the directory. Although all of them are control messages of small size (e.g. one flit), the scheme of repeatedly retrying the conflicting request produces an amount of traffic in

the interconnect that is not negligible. Figure 4.3 (left) illustrates all network messages generated on a write-read conflict when a cache-based style of conflict detection is employed. The tiles with a shared copy of the line are coloured in green, while the home is shown in blue and the requester in red. A dotted line differentiates the final unblock message from the initial request. On the right side of Figure 4.3 we show how by detecting conflicts at the directory, the requester transaction can observe the conflict after only two messages, independently of the number of sharers –in fact, for type of conflict, not only write-read–. Thus, a directory-based approach to conflict detection can also help address this source of inefficiency at the interconnect.

4.2.3 Reducing false positives of signatures

One of the main advantages of eager-eager systems is their ability to accommodate transactions with large footprints. The versioning hardware takes care of logging the old contents of the line before it is speculatively modified "in place". Therefore, evictions of both clean and dirty transactional data from the private cache are tolerated, provided that the cache controller can detect conflicts on spilled lines. The scheme to summarize overflowed addresses can vary from a single bit [92] to a hash signature [96]. Systems that only use signatures for bookkeeping do not need to make a distinction between cached and overflowed lines [27, 153]. However, using signatures introduces the possibility of false conflicts that arise as a consequence of their conservative encoding of addresses, i.e. the signatures encode a super-set of the addresses in the read and write sets. Address aliasing can hurt performance but does not affect correctness.

While signatures can be designed to minimize aliasing [103, 104, 116, 154], an inherent limitation of cache-based conflicting detection is that the bookkeeping meta-data is always recorded on a per-core basis. Each core keeps the set of addresses that it has accessed in a transaction, regardless of the addresses accessed by other cores. Using a data structure simile and observing the meta-data from a global perspective, the typical organization resembles an array of n hash tables, where n is the number of cores. Since the data accessed inside transactions is often shared, we can expect the elements found in different hash-tables to overlap to a certain extent. Given such locality, and considering that the number of accessed addresses is commonly larger than the number of cores in the system –specially for coarse grain transactions–, a more efficient encoding of the meta-data would avoid having the same element repeated across different sets. Instead, one hash table could track the union of all addresses, each one

mapped to a bit-vector that indicates which cores are transactional accessors of the address.

The aforementioned idea of tracking meta-data on a per-address basis can be naturally applied at the directory level, enabling a conflict detection scheme that also minimizes the impact of false positives due to its more efficient encoding. The directory already acts as a hash table that maps addresses to presence bit-vectors; extending it to track transactional ownership is straightforward. The extra overhead of having a *transactional directory* is low since a small cache per L2 bank is generally sufficient to contain all the lines accessed inside transactions which are mapped to that bank. This would remove signatures from the critical path of an L1 cache miss and thus would help reduce the impact of false positives.

4.2.4 Avoiding broadcast on L2 misses

Cache controllers need to observe coherence traffic for all lines that belong to the read and write set, even if they are no longer cached. This naturally happens on a bus-based system, whereas directory protocols need to be extended with *sticky states* [92,153] so that overflowed caches keep receiving coherence messages for replaced lines, and detecting conflicts in spite of the eviction. Basically, the solution of sticky states leverages the presence bit-vector kept at the directory to forward coherence requests only to those caches that could potentially have a conflict: The last exclusive owner of a speculatively modified line that was written back, or the sharers of a line that was silently replaced by some transactional readers. Once the line is written back to the shared level, the directory protocol has no way to distinguish a transactional line from the rest.

The problem arises when the directory entry is lost due to the unlikely but possible event of an eviction from the first level of shared cache. In the baseline tiled CMP architecture of this thesis, tiles have a directory entry for each of the cache lines in its L2 cache bank. When a line is replaced from L2, the associated directory entry is lost. Since the L1 caches maintain inclusion with the shared L2, an L2 replacement forces the invalidation of all L1 copies, rendering the directory information useless. Unfortunately, the mechanism of sticky states leverages the presence bit-vector to maintain isolation over lines that are no longer privately cached. The easy way out this problem is a best-effort approach: Aborting all possible transactional accessors of the line –as dictated by the bit-vector– before it gets replaced. However, an unbounded design like LogTM-SE [153] is able to tolerate the loss of the directory information that occurs on an L2 replacement.

LogTM-SE further extends the directory protocol in order to broadcast *filter*

check messages to L1 caches on every L2 miss, and use the result of every signature check to conservatively rebuild the directory information. Filter responses are collected by the requestor, which reconstructs the presence bit-vector with the results and piggybacks it on the unblock message sent back to the directory. For every signature that signalled a conflict, the corresponding bit of the bit-vector at the directory is set. From this point on, usual conflict detection is resumed by forwarding coherence traffic to those caches whose signatures reported to be transactional accessors of the line.

The solution adopted by LogTM-SE has a fundamental drawback: It burdens common-case execution in order to solve a rare event such as an eviction of transactional data from the on-chip storage. Each and every L2 miss causes a broadcast of filter check messages in order to maintain transactional isolation at all times. The overall latency of the miss is not generally affected because these messages are broadcast in parallel with the request to the off-chip memory, whose access time is generally much longer than the time it takes to deliver all messages, access signatures and collect the responses. However, the solution is not very efficient as it increases network traffic in all cases, even when running non-transactional codes. The reason for such broadcast is that the bookkeeping information required to detect conflicts is solely kept at the private cache level. A directory-based scheme could easily avoid this broadcast by having each tile keep summarized information about its overflowed lines. L2 misses are not a frequent event, and thus L2 evictions of transactional data are an even more uncommon situation –at least in the context of a CMP with a large, shared level L2 cache– which could be handle with simplistic solutions. For example, a single bit-vector would be enough to conservative track transactional accessors that have suffered L2 evictions. The bit-vector would then be directly used as sharing code when a new memory block is fetched to L2 cache, successfully retaining isolation over L2-evicted transactional data without the need of expensive broadcasts.

4.3 Background on Conflict Detection

The early HTM proposal from Herlihy and Moss [60] added a separate cache to track the transaction's read and write sets. Adding a new transactional cache in parallel with an ordinary data cache adds significant complexity to the optimized performance-critical data path of modern microprocessors, as it introduces an additional structure from which data may be sourced. When TM was revived a decade later by Stanford's TCC system [56], transactional bookkeeping was

accomplished by augmenting the existing private caches with transactional status bits associated to each entry. For bus-based systems like TCC, integrating the conflict detection logic into the cache controller is undoubtedly the most natural and straightforward solution because all cache controllers are able to snoop all potentially conflicting memory references issued by remote transactions. In systems of this type, coherence is maintained by having each cache controller snoop the bus and monitor the transactions, taking action if a bus transaction involves a memory block of which it has a copy in its cache. Cache is thus invariably accessed on every bus transaction in order to check for a tag hit and retrieve the coherence state bits that determine its actions. In such context, a straightforward solution to conflict detection is to incorporate the transactional status bits (SR and SM bits) along with the coherence state –as part of the cache line meta-data– and modify the state machine to interpret those bits as well, adapting the state machine behaviour (actions taken) if a conflict is detected: On a bus write, a conflict is signalled if either transactional bit is set, while on a bus read, a conflict is detected only if the transactional write bit is set. This is a rather simple change in the internals of the coherence controller, and is enough to detect conflicts on a best-effort HTM design which does not allow evictions of transactional data.

4.3.1 Conflict Detection on Evicted Lines

A simple yet conservative solution upon spills of transactional data is to enforce transaction serialization, letting the overflowed transaction write its results directly to shared memory [56]. Nonetheless, transactions of larger footprints can be accommodated without resorting to global serialization if the system is capable of keeping track of overflowed addresses, even if it does it in a summarized way. Replacements of read-set data can be tolerated regardless of the version management policy used, whereas speculatively written lines can only be spilled to the shared levels of the memory hierarchy if the system logs values before they are speculatively written [92, 153]. To retain isolation on overflowed lines, cache controllers need a way of determining if an address whose tag is not found in cache indeed belongs to the read and write sets of its transaction. The solution can be as simple as an overflow bit [92] which is set upon the first transactional spill and cleared on commit/abort. After the overflow bit has been set, any snooped bus transaction or incoming coherence message which does not correspond to a cached line must be considered conflicting. To maintain isolation after an evicted line has been re-fetched to the cache, line-fill operations always

set both transaction status bits if the overflow bit is set. In bus-based systems where all controllers observe all memory requests, this conservative solution can lead to abundant false conflicts if evictions are common. In directory-based systems, the directory acts as a filter so that not all cache controllers observe all coherence traffic, thus lowering the number of false conflicts. This number can be reduced by keeping more precise information about the spills, like an overflow signature [96] or a permissions-only cache [16].

A different solution is to remove the transactional status bits from caches and only use signatures for transactional bookkeeping [27,153]. In this case, the cache controller operates in the same fashion regardless of the presence of the line in the private cache, handling the detection of both cached and evicted lines altogether. However, this uniformity comes at a cost, as both cache and signatures have to be accessed for every bus transaction or incoming coherence request in order to obtain the inputs to the cache controller –coherence state plus transactional status–. Although the signature check can be performed in parallel with the cache access, the more or less complex logic of hash encoding –depending on the implemented scheme [116] – could make the signature response time exceed that of the cache, and thus it may negatively affect the latency of misses. False positives are another key disadvantage of hash-signatures, which arise as a consequence of their conservative encoding of addresses. Addresses that do not belong to the read and write set of the transaction may be considered as such due to aliasing, and false conflicts can be signalled when none exists causing unnecessary performance degradation. This poses a dangerous situation if the ratio of false positives becomes significant as the transaction footprint grows, as it may discourage programmers from using coarse grain synchronization, somehow jeopardizing one of the main goals of TM. In spite of these drawbacks, a signature-only scheme has two clear advantages. First, it eases the problem of transactional virtualization, as all the bookkeeping information is summarized in one large register that can be saved and restored by software, for example, in the event of a context switch. And second, they allow the introduction of hardware TM support while leaving private caches completely unmodified, an important achievement since they are highly-optimized, performance-critical structures whose design is tightly coupled with the processing core.

4.3.2 Silent Replacements and Sticky States

In a standard directory-based coherence protocol, the directory only forwards coherence messages to those caches that have a copy of the line. Only if the

protocol implements silent replacements of lines in shared state, may caches receive an invalidation message for a line that is no longer cached. Existing directory protocols commonly allow such an optimization, which directory-based HTM designs leverage to enable the detection of write-read conflicts at the private cache level at no extra cost. The case of read-write and write-write conflicts is not as simple. To handle the eviction of a transactionally dirty line, the coherence protocol must be augmented with a special write-back message so that the dirty, speculative data gets written to the shared level while keeping the directory pointed at the transactional owner, as if it still had the only copy of the line. This solution is called *sticky-M state* [92], and basically uses the directory entry to track the transactional owner in spite of the spill, so that it keeps forwarding requests for that block to the overflowed cache. The overflowed cache is then in the position to detect conflicting accesses that try to revoke its transactional ownership. The sticky states at the directory are then lazily cleared after commit, by means of a special *clean* message sent from the owner to the directory upon reception of a forwarded request that no longer causes a conflict. Somehow, the introduction of sticky states at the directory represents the fusion of coherence maintenance and conflict detection support at the directory.

4.4 Directory-Based Conflict Detection

In this section, we describe how the directory is augmented with several components in order to support the directory-level conflict detection introduced in the previous sections.

4.4.1 Transactional status

Using the directory to check for conflicts over lines that remain cached by transactional owners does not necessarily need any more information about a block than what is already stored in its directory entry. For example, let W be a transactional writer that locally caches a block B with exclusive ownership, and let R be a reader that tries to acquire non-exclusive ownership of B. When R's read request arrives at the directory, the standard protocol dictates that the request must be forwarded to W, which would then detect the conflict. However, if the directory only knew that W is executing a transaction, forwarding the request to W would be unnecessary; the directory itself could conservatively detect a conflict on B and directly send a nack response to R. A simplistic solution is to extend the

directory with a *transactional status register* that records which cores are executing a transaction at the moment. This register could be kept updated by sending explicit messages to all directories at transaction begin and commit/abort, and waiting for the corresponding acknowledgement before resuming the execution. Other schemes that avoid stalling the execution of the transaction and reduce the number of contacted directories are discussed later in this section. For now, let us assume the simplistic solution based on transaction begin/end reports.

Once the directory knows which cores are executing transactions in a given moment, it can start detecting conflicts on their behalf. Again, the simplest solution would be to perform the logical *AND* of the transaction status register and the presence bit-vector of the requested line, and interpret the result as the current transactional accessors of the line. Obviously, this would conservatively consider as transactional all privately cached lines, and while it would suffice to provide correct transactional semantics, it would result in a tremendous amount of false conflicts. Hence, the directory needs some sort of transactional bookkeeping to distinguish between cache residence and transactional ownership, in order to detect conflicts more accurately.

4.4.2 Transactional meta-data

Transactional meta-data is kept at the directory by means of a small, set-associative cache that we call the *transactional directory* (*TXDIR*). Just like the regular directory tracks cache residence, the TXDIR tracks transactional ownership. The TXDIR is accessed in parallel with the L2 directory, and the outputs from both structures are provided to the module that contains the conflict detection logic. This organization of the directory, including the newly added components, is shown in Figure 4.4.

The internal organization of the TXDIR is detailed in Figure 4.5. We can observe how it combines a cache-like structure that maintains precise meta-data for a limited number of lines, with a set of small signatures (one per core) which conservatively encode those transactional addresses that do not fit in the aforementioned buffer. Along with the *TxAccessors* bit-vector, each entry includes an additional *TxWriter* bit indicating if the line has been written (not shown in the figure, for clarity); this bit is only meaningful when only one transactional accessor exists. On each incoming request that arrives at the directory, the TXDIR cache is accessed. If a tag hit is found, the transactional accessors and writer status are taken from the entry. Otherwise, the result of the parallel signature check is selected.

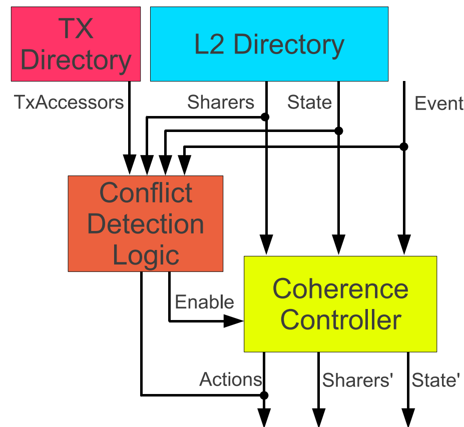


Figure 4.4: Block diagram of the new directory organization.

The TXDIR cache has a special feature: It is augmented with circuitry for flush-clearing the *TxAccessors* at the granularity of bits, as shown in Figure 4.6. Individual bits can be flush-cleared in a single cycle by enabling the corresponding clear signals, which are controlled by the conflict detection logic.

Though not shown in Figure 4.5, a fully-associative victim buffer can be optionally incorporated to the TXDIR cache to reduce the overflows due to its limited associativity. A small number of entries per TXDIR suffices to precisely track the accessors for a large number of transactional lines. In our baseline 16-core tiled CMP, for example, a 32-entry TXDIR per bank would be able to book-keep up to 512 different cache lines before resorting to a conservative scheme based on signatures. Given the fine-grain mapping policy used throughout this thesis (shown in Figure 3.11), consecutive line addresses are mapped to subsequent banks, spreading read and write set lines across all directories with the same probability. This distribution of the meta-data amongst the different banks plays a key role in minimizing the frequency of meta-data overflows at the transactional directory.

An inherent advantage of the directory-based bookkeeping is that, even when the limited capacity or associativity of the TXDIR requires the use of signatures to track transactional accessors, some false positives can be identified and properly ignored: The use of sticky states at the directory ensures that every transactional accessor is marked as a holder of the line in the presence bit-vector. Thus, the opposite scenario –transactional accessor not in *sharers*– is clearly the result of address aliasing and can be ignored. For this reason, a set of small, per-core

Figure 4.5: Transactional directory internals.

signatures (64 bits each, 1 Kbit overall) suffices to support conflict detection over an unbounded number of lines without introducing frequent false positives.

Figure 4.6: SRAM cells used to track transactional accessors in the directory

4.4.3 Conflict detection logic

As shown in Figure 4.4, the module that performs the conflict detection takes as inputs two bit-vectors –transactional accessors and sharers–, the coherence state, the event –type of request– and the identity of the requestor. Its main output is a *no-conflict* signal, which is in turn connected to the *enable* input of the coherence controller module. This signal is usually asserted, allowing the coherence controller to operate as usual. When a conflict is detected, the coherence controller is disabled so that the line's state and sharers remain unchanged, while the actions taken by the directory are directly controlled by the conflict detection module. Its simple logic is specified in Algorithm 1.

Algorithm 1 Conflict Detection Logic

```
conflict ← false
if TxAccessors(Address) ≠ {Requestor} AND TxAccessors(Address) ≠ {} then
   if Event = Write then
      if State(Address) = M then
         conflict ← true
      else
         if Priority(Requestor) < HighestPriority(TxAccessors(Address)) then
            conflict ← true
         end if
      end if
   else
      if Event = Read then
         if State(Address) = M AND TxWriter(Address) then
            conflict ← true
         end if
      end if
   end if
end if
```

The conflict controller only attempts to detect a conflict if the line has at least one transactional accessor different from the requestor. In that case, a conflict is signalled if a write request finds a line that is exclusively owned by another transaction (write-write or write-read conflict), as dictated by the M coherence state. In any other state, a conflict is immediately detected if the requestor does not have higher priority than the current transactional accessors. If it does, the directory omits the detection of the conflict and forwards invalidations to the sharers of the line, in order to support a hybrid resolution policy [22], similarly to the baseline eager-eager system evaluated throughout this thesis. Finally, shared requests cause a conflict if they find the line held in exclusive ownership

whose writer status indicates that the only transactional accessor is indeed a write transaction (read-write conflict). It should be noted that this detection logic is able to process conflicting requests to lines in busy states, effectively decoupling conflict detection from coherence maintenance.

4.4.4 Propagation and update of transactional meta-data

The fundamental drawback of detecting conflicts at the directory is that not all memory references within a transaction must go through directory, but only those that result in cache misses. Because the notification of a transactional load or store hit to the directory cannot happen instantly, conflicts still need to be temporarily detected at the cache level until the directory has knowledge of the transactional access and can take over the task. Thus, transactional loads and stores that hit in the private cache must notify the directory in order to update the TXDIR meta-data. This communication takes place asynchronously –off the critical path of the memory reference– by means of a special write-back message we call *txaccess*. Note that this new type of message is not part of the protocol and thus does not participate in the coherence mechanisms. *Txaccess* messages simply update the meta-data kept at the directory, adding the sender as a transactional accessor and appropriately setting the writer status bit (a write flag distinguishes the type of access). For memory references that do go through directory, the TXDIR must be updated after the cache miss has been successfully solved. In this case the *txaccess* report is piggybacked as a couple of flags in the final *unblock* message.

Since propagation of transactional meta-data happens asynchronously, caches must be able to detect conflicts in those cases when a request reaches the directory before it has been informed about a transactional hit at the L1 cache level. Once the directory receives the corresponding *txaccess* report it resumes the task of conflict detection, and the offended caches no longer observe conflicting traffic. We discuss several alternatives for dealing with such racing requests in a later subsection.

4.4.5 Awareness to priority and deadlock detection

As mentioned in Chapter 3, eager HTM systems that rely on a pure *requester stalls* policy are susceptible to a pathology known as *starving writer* [22]. A priority scheme is required to support the aforementioned hybrid resolution policy, which resolves write-read conflicts in favour of the requester when the writer has higher

priority than all readers. Timestamps transported in all coherence requests for deadlock avoidance [92] are now also leveraged to support such hybrid policy at the directory, which must keep a *timestamp table* and constantly update it by snooping incoming requests.

Thanks to this timestamp table, the conservative deadlock avoidance mechanism commonly employed by eager systems [92, 153] can be kept unmodified. Negative acknowledgement messages are sent from the directory on behalf of the eldest accessor of the line –including its timestamp in the response– so that caches are oblivious to the fact that most of the *nack* responses they receive are in fact sent by the directory, and not by other caches. This allows the resolution scheme at the caches to remain unchanged: A transaction aborts if it is *nacked* by an older transaction and it has its *possible-cycle* bit asserted. This bit gets set when a transaction *nacks* a request from an older transaction. Because caches no longer *nack* forwarded requests, the *possible-cycle* bit is set upon reception of a new *txnacked* message: To emulate the original behaviour, the directory sends a *txnacked* message to the eldest transactional accessor when an even older requestor gets *nacked* on its behalf.

4.4.6 Clearing transactional meta-data

Transaction begin is implicitly communicated to each directory bank on the first access to a line mapped to the bank, via *txaccess* (L1 hit) or a coherence request (L1 miss). Each directory bank snoops these messages and updates its transactional status register and timestamp table when it detects that a core has entered transactional mode. Coherence request messages are guaranteed to arrive in order, as guaranteed by an in-order processor model. *Txaccess* messages that arrive out of order (e.g. that belong to the previous transaction) are detected by observing their timestamp and properly discarded.

On the other hand, when a transaction ends –whether it is or not successfully–, the transactional meta-data kept for it at the directory must be cleared in order to release isolation over the lines in the read and write set. To this end, we introduce a second protocol-independent message called *txend*, sent by cores both at transaction commit or after the rollback has completed. When a *txend* message from core k reaches the directory, flush-clears the k-th bit *TxAccessors* of each entry in the TXDIR cache. The writer status flag of the entry is also conditionally cleared (if the k-th bit is set). The transactional status bit and overflow signature associated to the core are also cleared. Not all directory banks need to be informed about the end of the transaction, but only those whose

mapped addresses were part of the finished transaction. Thus, for every memory reference performed during the transaction, each core records the directory bank that the address is mapped to. This bit-vector of *txaccessed directories* is then used at commit/abort time to selectively issue *txend* messages. The core can continue its execution without delay, as *txend* messages do not need acknowledgement because they carry the transaction's timestamp, which is used in conjunction with the internal timestamp table kept at the directory to handle cases in which *txend* or *txaccess* messages arrive out of order.

4.4.7 Dealing with races

While a *txaccess* message traverses the network towards the directory after a cache hit, the core must be able to detect conflicts on forwarded requests that reached the home tile before the *txaccess*. Several solutions are possible to handle conflict detection in these races. The most straightforward approach is to maintain the typical transactional status bits in cache, which are used to detect conflicts on such races, as well as to identify those lines whose transactional status has been already reported to the directory –avoiding repeated *txaccess* messages on subsequent hits–. For simplicity, this is the approach we use in our design, but other solutions are possible too, since the directory already keeps accurate information about the read and write sets, and therefore it is not necessary to have such precise –redundant– meta-data at the cache level at all times. In this case, replacement of transactional lines in S state cannot be silent, as conflicts could go undetected if the L1 cache loses the transactional meta-data before the corresponding *txaccess* reaches the directory. Hence, transactional S replacements are treated much like M replacements, by sending a write-back message that updates the TXDIR meta-data, and waiting for the acknowledgement to arrive back to the L1 cache before finally de-allocating the line.

One way to detect conflicts at the cache level without having to add bits to the private caches, is to use a small *summary signature* that encodes both read and write sets, and then modify the conflict logic so that the signature is only checked if there is a potential race, i.e. a *txaccess* message on-the-fly whose destination is the same directory bank that forwarded this request. Thus, the *txaccessed directories* bit-vector acts as a first filter to distinguish non-conflicting from potentially conflicting requests. More accurate ways of filtering the check of the signature require some kind of *txaccess* acknowledgement scheme from the directory, which could be easily accomplished by means of serial numbers that are piggybacked in existing messages.

Another method consists in maintaining a separate *txaddress buffer* with the most recently accessed addresses of the transaction. This scheme decouples transactional bookkeeping from caches at the cost of increasing the amount of network messages, as now *txaccess* messages need to be acknowledged before an address can be de-allocated from the buffer. Addresses can be buffered indefinitely in order to avoid redundant communications with the directory. However, the buffer should be drained after a certain occupancy threshold in order to leave room for new addresses. Performance can suffer if at some point the buffer fills up completely, since the processor will stall on the next memory reference that results in a cache hit until space becomes available, or else violations of isolation would be risked.

4.4.8 Reducing meta-data propagation

Obviously, reporting every cache hit to the directory is an expensive solution in terms of its traffic demands. However, the performance benefits of directory-based conflict detection only apply to contended lines. Hence, it makes more sense to propagate only those accesses to lines that have seen conflicts in the past. We optionally introduce a *conflict signature* which is updated every time a cache sends or receives a *nack* message for an address, and checked to determine if a *txaccess* message needs to be sent. A *txaccess* is sent only for L1 hits to lines whose transactional status is not yet set, whose address also belongs to the conflict signature. Because the fraction of lines that experience contention is usually small in comparison with the size of the transactional set, a small signature should suffice to filter out most of the traffic that propagates meta-data from the caches to the directory. As this optimization does not affect correctness, the signature could be periodically cleared, although this could trade off some performance for a reduction in traffic, when conflicts are forgotten and accesses to contended data are not immediately reported to the directory. In our scheme, the conflict signature is cleared from one phase of the execution to the next, i.e. upon arrival to a barrier.

4.5 Evaluation

In this section, we evaluate the proposed scheme of directory-based conflict detection, comparing it against several pertinent HTM design points.

4.5.1 Experimental Setup

Table 4.1 lists all HTM configurations evaluated in this chapter. LogTM-SE [153] acts as the eager-eager (EE) HTM system of reference, and all other HTM configurations are derived from it. The *EE_base* configuration uses perfect signatures to perform transactional bookkeeping at the private cache level, and it implements a hybrid resolution policy in order to avert the *starving writer* pathology, as discussed in Chapter 3.

Table 4.1: HTM configurations evaluated in Chapter 4.

Configuration	**Description**
EE_base	The LogTM-SE design [153] with *perfect signatures*
EE_pred	Idem. as EE_base, augmented with a write-set predictor [22]
BitSel_2048, H3_1024 H3_2048, H3_4096	Idem. as EE_base, with parallel Bloom signatures [116] of *type* and *size* Types: bit-selection or high-quality hash function. Sizes: 1-4 Kbits
DirCD_Magic	Idem. as EE_base, with magic conflict detection at the directory
DirCD_TxDir64	EE system with a 64-entry TXDIR for directory-based conflict detection

The *EE_pred* augments this baseline with a 256-bit, 256-entry write-set predictor, in order to also target the *duelling upgrades* pathology [22], by selectively requesting exclusive permission for predicted loads. Our configuration is similar to Bobba's EE_{HP} system, except for one detail: In our case, the block is not added to the transaction's write set until it is indeed written, as we observed that doing so consistently results in worse relative performance than if only added to the read set, for the benchmarks here considered.

We also evaluate the EE_base system using parallel Bloom signatures. All four signature configurations considered use four hashes. One of them uses bit-selection and 2Kbit signatures, while the other three have hardwired H3 hash functions and signatures sizes of 1, 2 and 4 Kbits.

As for our design, we consider two systems. On the one hand, *DirCD_Magic* acts as an upper bound of the performance achievable by a directory-based conflict detection scheme. It is a modified version of the ideal EE_base (including perfect signatures) in which the directory has instant access to any read or write signature across the chip –without involving any message– and directly *nacks* conflicting requests, while magically informing the nackers about the conflict –to maintain deadlock detection–.

On the other hand, *DirCD_TxDir64* is a detailed implementation of the scheme described in previous section, which uses a TXDIR of finite size to perform

transactional bookkeeping at the directory, and extends the protocol with *txaccess*, *txend* and *txnacked* messages to propagate meta-data, communicate transaction commit/abort and detect deadlocks, respectively. The specific parameters of this configuration are shown in Table 4.2. In order to provide a fair comparison in terms of network traffic, in the DirCD_TxDir64 system we adapted the amount of cycles that a processor stalls after detecting a conflict –before retrying the request–, so that both EE_base and our design have similar intervals of retry, i.e. result in a similar number of reissued requests on a stall of the same length. While EE_base retries after only 3 cycles, DirCD waits longer (50 cycles) before reissuing the conflicting memory request, because it inherently detects conflicts quicker. This implementation of DirCD extends each L1 cache entry with a single transactional bit, used to detect conflicts on racing requests, as well as to decide if a *txaccess* message needs to be sent to directory. When this *tx* bit is asserted, the coherence state conservatively determines if the line belongs to the read set (S or E states) or write set (M state). As for the ability to tolerate L2 evictions of transactional data, we model a single overflow bit-vector (one bit per core) on each tile. Nonetheless, we have not experienced any such events in our experiments, mainly due to the L2 cache's fairly large size and associativity of the CMP modelled throughout the thesis (512MB per tile, 8-way associative).

Table 4.2: Specific parameters of the DirCD system.

Directory-based Conflict Detection (DirCD)	
Core Settings	
Conflict-Retry interval	50 cycles
Conflict signature	256 bits
Memory Settings	
L1D-cache	1 Tx bit per line
Directory Settings	
TXDIR	64-entry, 8-way associative
TXDIR victim cache	8-entry, fully associative
TXDIR overflow signature	16 x 64-bit H3 parallel
L2 cache overflow	16 x 1 bit

Table 4.3: Inputs to the additional workloads used in Chapter 4.

Benchmark	Input
vacation-vhigh	-n2 -q1 -u1 -r128 -t4096

Workloads. Regarding the benchmarks used in the evaluation carried out in this Chapter, we have changed one of genome's compile-time parameters in order to use a larger chunk size in the first step of the algorithm. We have increased *chunk_step1* from its default value of 12 in the release of STAMP, to 36, so as to stress the bookkeeping mechanism of the evaluated HTM systems –signatures– with transactions of a larger footprint. Yen adopts similar strategies when evaluating signatures [152]. Furthermore, we excluded labyrinth because it requires support for the *early release* construct in order to benefit from parallel execution: Removing addresses from read sets is not possible in systems that use real signatures, including our DirCD scheme, and thus the utility of labyrinth in this evaluation is limited. To make up for the loss benchmark, we modified the input parameters of vacation, as shown in Table 4.3, to create a new configuration that exhibits very high levels of contention on transactions of a fairly large footprint. While intruder also shows highly-contended transactions, they are of a rather small size. Very high contention levels in vacation are achieved by reducing the initial size of the database as well as the percentages of queried relations and *user* (read-only) transactions. With the inclusion of *vacation-vhigh*, we intend to observe how our scheme handles this type of scenario in comparison to the baseline system.

4.5.2 Performance Analysis

4.5.2.1 Idealized Systems

Figure 4.7 shows the potential performance gains of a directory-based scheme to conflict detection (DirCD), relative to the cache-based approach traditionally used by EE systems. In comparison to EE_base, it shows how the "magic" DirCD system greatly reduces the execution time of highly-contended applications such as intruder and vacation-vhigh, in percentages that vary from 22% in vacation-vhigh to roughly 50% in intruder+. This confirms that the directory indeed creates a bottleneck in the conflict management mechanism in situations of high contention affecting a few cache lines. DirCD_Magic anticipates the performance gains that can be expected if detection is decoupled from coherence, i.e. if it can be carried out without the need of request forwarding –thus without transitioning to the line to a busy state–.

Figure 4.8 presents a breakdown of transactional cycles for the EE_base system, compared to the realistic implementation of DirCD analyzed in the next subsection. The total transactional time corresponds the sum of tx-useful and

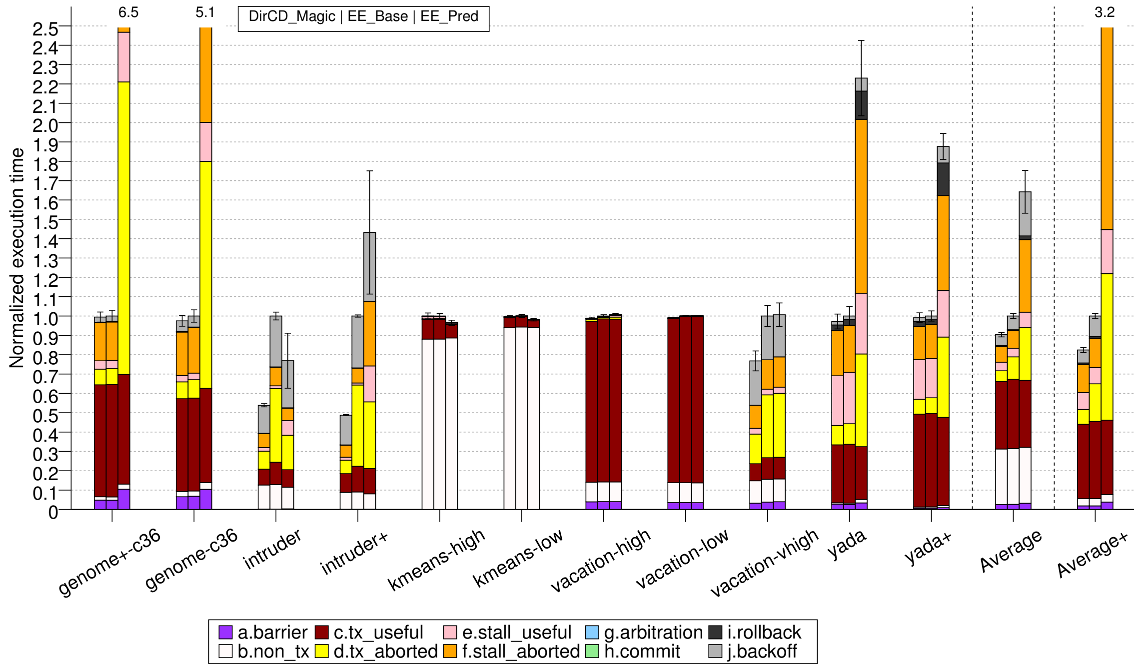


Figure 4.7: Relative performance of "magic" directory-based vs. ideal cache-based conflict detection.

tx-aborted components of Figure 4.7. The figure divides transactional cycles into *tx-hit* (non-memory or L1 hits) and cycles waiting for a memory request to complete –excluding retried requests, which are accounted as stall time in Figure 4.7–. Memory access time is in turn further broken into the time the request was queued at the directory due to busy states (*tx-busy-dir*), and actual miss time (*tx-load-miss* or *tx-store-miss*), which reflects the compulsory time taken by coherence messages to travel across the interconnect, L2 cache access time, etc. The figure demonstrates how the reductions achieved by our proposal in the tx-useful and tx-aborted components of intruder and vacation-vhigh are due to the removal of the bottleneck formed at the directory in the baseline system, which limits its ability of resolving conflicts during contention.

intruder. Both DirCD and EE_Base systems suffer the aforementioned pathology of *duelling upgrades* that leads to many aborts, shown in Table 4.4. The improvement of DirCD with respect to the EE_Base is not so much due to the reduction in the number of aborts –aborts of transaction with TID0 go down by

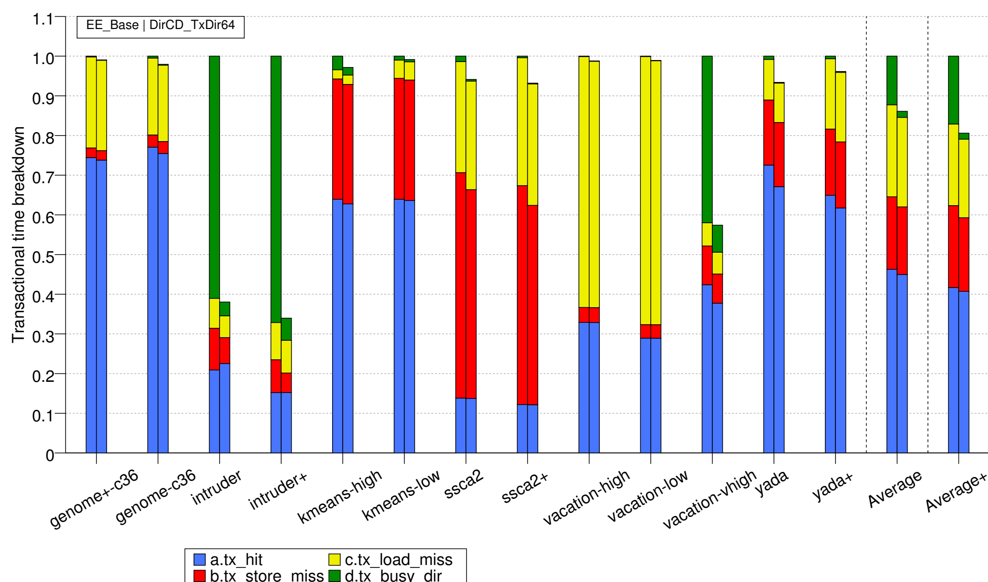


Figure 4.8: Transactional cycle breakdown in baseline vs. directory-based schemes.

Table 4.4: Number of aborted transactions for the three systems in Figure 4.7.

	Total Aborts			TID0 Aborts		
	DirCD	EE_Base	EE_Pred	DirCD	EE_Base	EE_Pred
genome+	1880	1841	31059	1146	1186	30535
genome	1267	1265	14714	759	784	14383
intruder+	103431	133610	121074	71520	109076	150
intruder	22076	29668	20805	6057	14859	249
kmeans-high	679	598	8	473	426	2
kmeans-low	239	215	5	86	79	0
ssca2+	258	190	64	14	13	13
ssca2	330	280	112	47	51	51
vacation-high	159	122	204	150	116	180
vacation-low	22	23	35	22	23	35
vacation-vhigh	8207	8132	7850	193	198	189
yada+	6643	6520	16526	2066	2016	398
yada	3261	3312	8362	893	890	222

40 to 60%, as we can see in Table 4.4– as it is due to the higher throughput of detected conflicts achieved by DirCD. TID0 corresponds to a *queue pop* operation that first reads and then writes a highly-contended line. The detection at the directory allows the highest priority writer to proceed quickly –since its write request is processed immediately when it arrives at the directory, as it demon-

strated Figure 4.8– causing the abort of the lower priority readers (upgraders). The fact that lower priority conflicting transactions are aborted much faster in DirCD is reflected in the radical decrease of the *tx-aborted* component seen in Figure 4.7 when compared to EE_Base; the total number of aborts also decreases and accounts for the shrinkage of the *backoff* component too. In comparison to EE_Pred, the write-set predictor successfully averts the duelling upgrades and almost completely removes all TID0 aborts by directly requesting the line for exclusive access, as shown in Table 4.4. TID0 transactions serialize one after another without incurring in the huge number of aborts seen in the other two systems. However, the predictor does not represent a generalizable solution. For the small input of this benchmark, we see how the total number of aborts is decreased by around 1K, in spite of a reduction in TID0 aborts of almost 6K; for the medium input, despite removing 110K TID0 aborts, the total number of aborts goes up by 18K. This indicates that the write set predictor penalizes transactional execution of other transactions of the program, mainly due to the combination with a hybrid resolution policy: more write requests means more (often unnecessary) aborts of concurrent reader transactions.

vacation-vhigh. This benchmark corroborates our observation that the performance improvements seen in DirCD do not stem exclusively from a reduction in the number of aborts, but rather from a reduction in the amount of cycles that transactions waste while waiting for a conflicting request to be processed at the directory, as shown in Figure 4.8. In both traditional EE systems, the directory spends most of its time in a busy state while messages are forwarded to the possible transactional owners of the line, which are responsible for the conflict check. In DirCD, the directory spends much less time in busy states and thus can attend and respond to messages immediately, which in turn is translated in faster resolution of the conflict.

Other benchmarks. DirCD and EE_Base perform comparably for applications whose transactions are either long running or not heavily contended. In these cases, there is no performance advantage in detecting the conflict sooner, since the resolution consists in stalling the requester the majority of the times. In regards to EE_Pred, we observe that the effect of mispredicted upgrades becomes very acute for benchmarks with large transaction footprints like yada and genome, which experience severe performance drops that vary from around 2X slowdowns in the case of yada, up to 5-6 times slower in the case of genome. We can see in Table 4.4 how TID0 is the transaction responsible for the pathological behaviour of the write-set predictor in genome, which is precisely the transaction with the largest read and write sets.

4.5.2.2 Realistic Systems

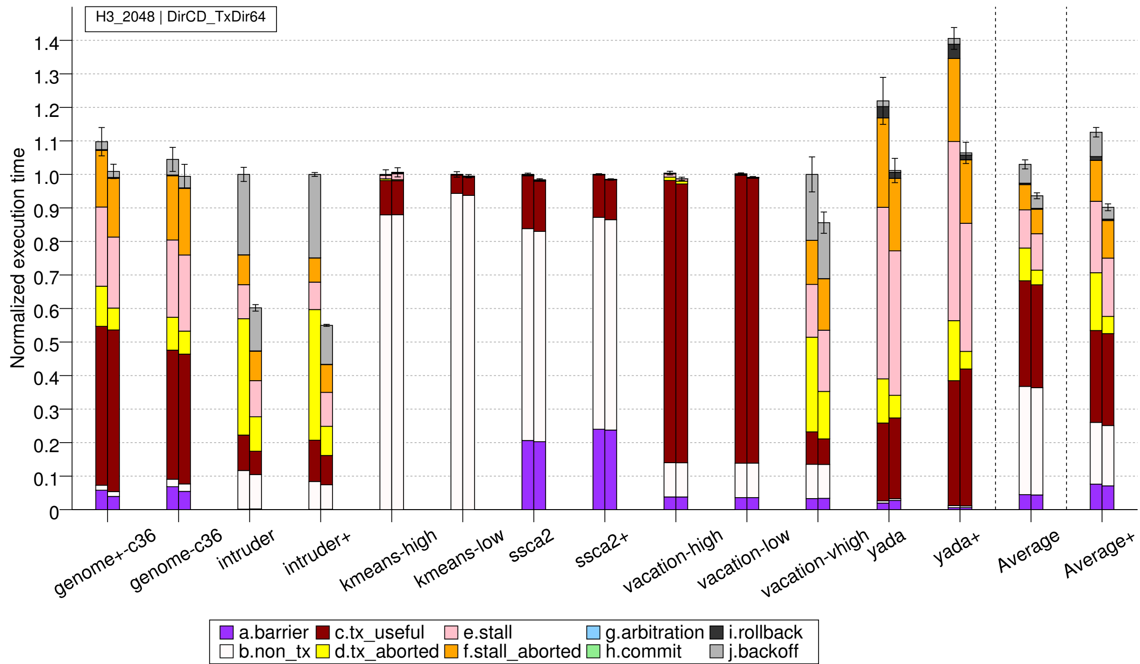


Figure 4.9: Relative performance of realistic directory-based vs. cache-based conflict detection.

Figure 4.9 shows the relative performance of our detailed implementation of DirCD that uses a transactional directory, compared to an EE system that uses Bloom signatures whose total size is comparable to the overhead of the structures introduced by our scheme. Figure 4.10 presents performance numbers for four EE systems that use real Bloom filters to track transactional read and write sets. The results shown in both figures are normalized with respect to the EE_Base configuration. We can see in Figure 4.9 how DirCD excels for applications with high contention like intruder and vacation-vhigh, for the reasons discussed earlier. Furthermore, DirCD closely tracks the performance of the EE system with perfect signatures for workloads with large transactions like genome and yada. This reveals that the number of false conflicts that arise in DirCD is very low, in spite of its finite capacity to precisely track transactional meta-data. This is partly due to its ability to identify and ignore some false positives signalled by the TXDIR

overflow signatures, simply by performing a logical *AND* between the result of the signature check and the corresponding bit in *sharers*.

DirCD consistently performs equally or better than the H3_2048 configuration, which uses a total of 4Kbits to hash-encode both transactional sets (2048 bits each signature). As it can be derived from Table 4.2, our detailed DirCD implementation spends a bit less than 5Kbits: 3Kbits (including tags and "data") in the TXDIR and associated victim cache, an extra 1Kbit for the 16 overflow signatures of 64 bits, the 256-bit conflict signature and an additional 512 bits for the transactional flag kept per L1 cache entry (although these bits are only maintained for dealing with races as well as reducing redundant meta-data propagation). We choose to compare DirCD against the H3 scheme of encoding, as it is more efficient than bit-selection for same-sized signatures [116], as demonstrated by Figure 4.10. For a similar meta-data storage capacity of less than 5Kbits per tile, our scheme keeps the number of false positives to a minimum, demonstrating that storing transactional accessors on a per-address basis at the shared cache level is a more efficient encoding of transactional sets than tracking addresses on a per-core basis using signatures. Using parallel H3 filters, it is necessary to increase their joint size to 8Kbits in order to avoid most false positives that arise in benchmarks with large-sized transactions like genome or yada, though it would still be of no help for highly contended benchmarks like intruder or vacation-vhigh. Considering that TM does not discourage programmers from using coarse grain transactions, the efficient encoding of transactional addresses is an important point in HTM design that is met by our proposed bookkeeping scheme.

4.5.3 Traffic Considerations

Figure 4.11 plots the network traffic results for the baseline EE system and our detailed DirCD model. It shows both network message counts generated by each system –broken down according to message type–, as well as the flit count. Both measures are normalized to the data obtained for EE_base. As we can observe in the average plots, DirCD generates between 25 and 35% less traffic (flits) than the baseline system. The first relevant difference shown by the breakdown is how DirCD completely eliminates *filter check* messages broadcast on every L2 miss, as discussed in the motivation section. On average, filter checks approximately account for 15% of all network messages generated, although they reach over 30% for workloads with large working sets like both original configurations of vacation. The number of acknowledgement messages is also severely reduced in DirCD, since each filter check is responded with an *ack*. Another difference

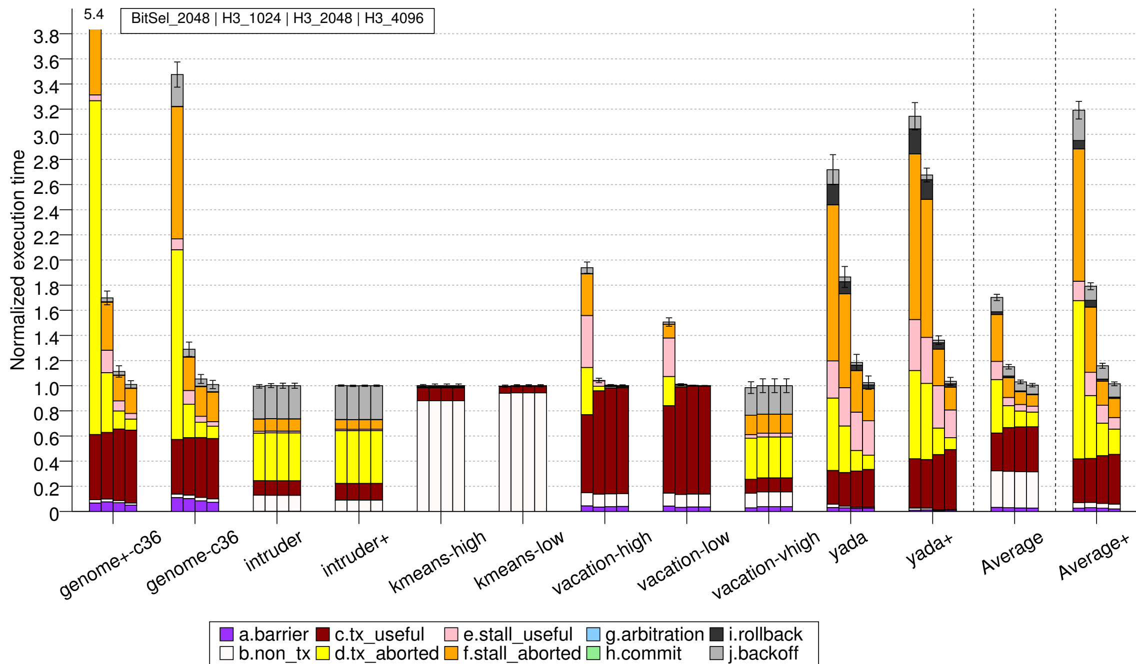


Figure 4.10: Effect of false positives on performance for various signature schemes and sizes.

is the removal of virtually all *unblock-cancel* messages, as the vast majority of the conflicts are detected at the directory, which does not forward requests that are known to be conflicting and thus does not enter a busy state. Formidable reductions in the amount of invalidation messages achieved by DirCD indicate that write-read conflicts are solved with a single *nack* message, avoiding both the *invs* and the corresponding *ack*/*nack* responses, as described in Figure 4.3. For contended workloads with long-running transactions like genome and yada, both *invs* as well as requests forwarded to exclusive owners are significantly reduced, which gives an idea of how DirCD allows the simple *stall-and-retry* resolution policy of the baseline system, at a much lower cost in terms of the network traffic generated by retries. The number of requests, data, unblock and *nack* messages stays more or less constant across all benchmarks, for both systems.

DirCD achieves the above reductions in the network traffic associated to conflict detection at the cost of introducing new messages that do not exist in the baseline system. Their main purpose is to propagate (via *txaccess* messages) or

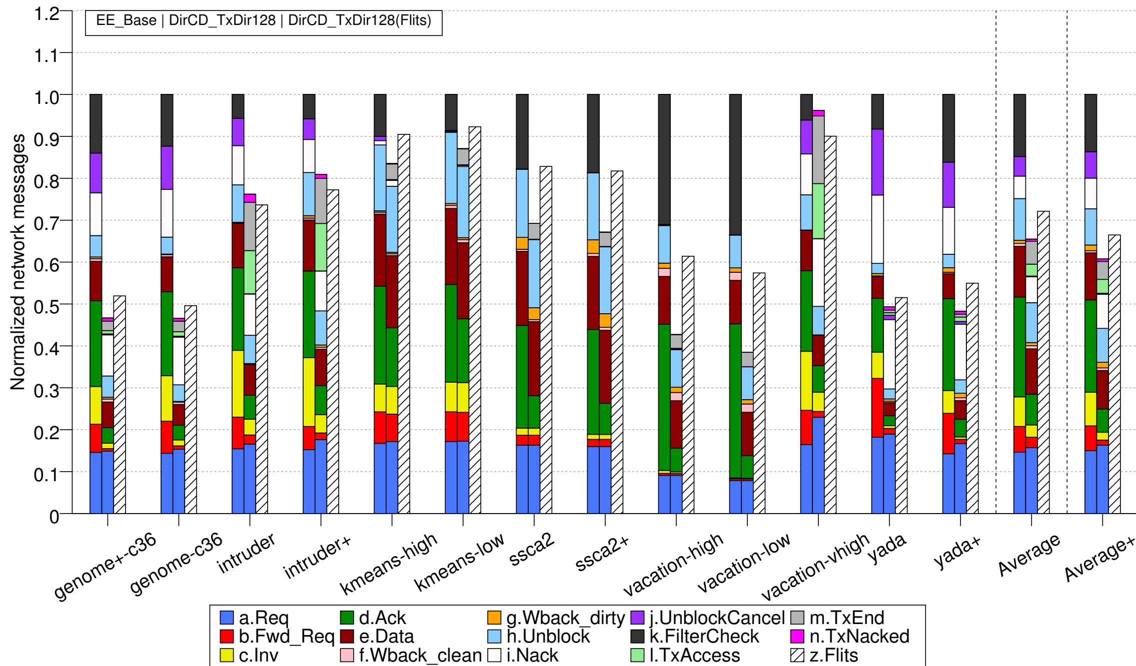


Figure 4.11: Network message breakdown and network flits.

clear (via *txend* messages) the meta-data kept at the directory. The number of *txend* and *txaccess* messages depends on the total number of transactions attempted (both committed and aborted) as well as the transaction footprint –which is in turn proportional to the number of different L2 banks accessed–. Therefore, they grow proportionally to the level of contention, and much faster if contention affects transactions of larger footprints. This explains why it is responsible for around 20% of all messages in intruder (small transactions), while it can reach almost 30% for vacation-vhigh (larger data set). For workloads with few large transactions with moderate levels of contention like yada, the number of *txend* and *txaccess* messages is almost negligible. In all other benchmarks, they account for less than 5% of all messages, mainly due to the filtering effect of the conflict signature used at the private cache level to avoid sending *txaccess* messages for lines that have not seen conflicts recently. The role of conflict signature explains the low number of *txaccess* messages in low contended benchmarks like vacation-low and vacation-high, in spite of their large transaction size. For the same reason, both ssca2 inputs should show much larger counts of *txend* as a result

of the huge number of non-contended transactions executed, but the conflict signature avoids them by filtering out most or all *txaccess* messages that would have been sent to the directory otherwise.

In summary, despite the newly added messages, the significant reductions in other types of messages associated to conflict detection still tips the scale in favour of DirCD across all evaluated workloads, confirming that the extra cost of detecting conflicts at the directory does not only pay off in terms of performance, but it is also more efficient on its use of the interconnect.

4.6 Concluding Remarks

In this chapter, we have presented a new approach to conflict detection targeted to eager TM systems which make use of a distributed directory to maintain coherence over a point-to-point network, as it is the case of a tiled CMP architecture. We have demonstrated that in traditional approaches, the directory becomes a bottleneck in situations of high contention by limiting the throughput of the conflict management mechanism. To this end, we have proposed a design that decouples conflict detection from cache coherence in order to overcome pathological situations that degrade the performance of an eager HTM system, enabling quicker reaction to high-contention scenarios. Our experimental evaluation has shown that our technique deals with contention more efficiently, leading not only to fewer aborted transactions, but most importantly to a lower overall latency of contended memory accesses within transactions. Our experiments have shown average reductions in execution time of 6 to 10% with respect to a LogTM-SE system with ideal signatures, while simultaneously decreasing its use of the network by 30% on average. In particular, by alleviating the bottleneck created at the directory, we have observed performance gains of up to 45% for those workloads that suffer very high contention over a small number of lines. We have also compared our work to systems that use signatures of equivalent hardware cost at the cache level, and found that our scheme reduces the performance degradation caused by false positives as it virtually removes all false conflicts. Our novel bookkeeping scheme leverages the inherent characteristics of the directory to globally encode all transactional sets by associating addresses to transactional accessors, instead of redundantly tracking addresses in each core. This allows for a more efficient global encoding, and enables the elimination of some false transactional accessors –due to signature aliasing– by leveraging the directory information itself. In summary, we claim that augmenting the role of

the directory to include the conflict detection functionality is a natural evolution in its responsibilities within a cache coherent HTM system.

Chapter 5

ZEBRA: A Data-Centric, Hybrid-Policy HTM System

5.1 Introduction

Fast implementations of transactional programming constructs that provide optimistic concurrency control with stringent guarantees of atomicity and isolation are necessary for TM to gain widespread usage. Software TM implementations impose too high an overhead and do not fare well against traditional lock based approaches when performance is important. Hardware TM (HTM) systems show much greater promise. Yet, within the design space of HTM systems, there are tradeoffs to be made among various pertinent metrics like design complexity, speed and scalability. Early work on HTM proposals [56, 153] fixed critical TM policies like versioning (how speculative updates in transactions are dealt with) and conflict resolution (how and when races between concurrent transactions are resolved). These designs choose a point in the HTM design space and analyze utilization of available concurrency in multithreaded applications within that framework.

Results in research so far do not show a clear winner or an optimal design point. Lazy HTMs, which confine speculative updates locally and run past data races until a transaction ends, do seem to be more efficient at extracting concurrency [122] but require elaborate schemes [29, 102, 143] to make race free publication of speculative updates (i.e. transaction commit) scalable. Eager HTMs, which version data in place and resolve conflicts as they occur, make

such publication rather trivial at the expense of complicating behavior when speculative execution needs to be undone to resolve data races (i.e. transaction abort). Eager HTMs fit very naturally into existing scalable cache coherent architectures and can tolerate spills of speculative data into the shared memory hierarchy, unlike their lazy counterparts. When comparing the performance of the two such designs, a clear winner cannot be established. With workloads that demand high commit throughput, eager systems perform substantially better, while with high contention workloads lazy designs come out on top.

This reasoning suggests that a new HTM design that selects the best performing policy (eager or lazy) depending on workload characteristics would be close to the most suitable HTM design for the scalable architectures under consideration. A key factor would then be the complexity involved in realizing such a design in hardware. Some solutions have been proposed that attempt to provide a hybrid-policy HTM design. UTCP [81] is a cache coherence protocol that allows transactions in a multithreaded application run either eagerly or lazily based on some heuristics like prior behavior of transactions. Although it lays down an interesting approach, We feel that the protocol is a significant departure from existing cache coherence designs and the additional complexity involved for just supporting TM represents too high a design cost. FlexTM [122] allows flexibility in policy but it does so by implementing critical policy managers in software. It provides a significant improvement in speed over software TM implementations by proposing the use of *alert-on-update* hardware, but the considerable cost of software intervention renders a comparison with pure HTMs moot. LV* [96], a proposal that utilizes snoopy coherence, allows programmer control over policy in hardware but with the constraint that all transactions in an application must use the same policy at any given time. A scalable alternative has not yet been proposed. The requirement of programmer-assisted policy change is a drawback too since the same phase of an application can exhibit different behavior with varying datasets.

In this chapter we propose a solution that is simple and yet powerful and flexible. We recognize the fact that assuming all data accessed in a transaction possesses the same characteristics can lead to sub-optimal solutions. Based on our study of conventional HTM design points we infer that only a relatively small fraction of data accessed inside transactions is actively contended. The rest is either thread-private (stack or thread-local memory) or not actively contended. Treating these two categories of data the same inside transactions leads to inefficiencies –a prolonged publication phase at commit when using a lazy design or increased contention leading to expensive aborts when using an eager

approach–. Our work attempts to break this restriction by choosing a granularity for data at which minimal changes are required in existing scalable architectures –that of the cache line–. Efficient scalable cache coherence implementations exist and have been extensively studied for a long time. Our design leverages these by annotating cache lines as being either contended or not. Contended lines are managed lazily thereby permitting greatest concurrency among transactions. It should be noted that eager systems disallow reader-writer concurrency while in lazy systems it can occur quite naturally if the reader commits before the writer. All non-contended lines are versioned eagerly and thus, on transaction commit, only contended lines need to be published. When contention is discovered (e.g. when aborting or stalling) the offending cache line(s) is (are) marked as contended. Over the course of execution of a workload, versioning of lines that are contended transitions from eager to lazy. In the steady state we can expect only the contended subset of the working dataset to be managed lazily. As we shall show in the analysis presented here, substantial gains over existing fixed policy HTM designs can be seen. The incremental cost of implementing this approach is minimal since only very modest behavioral changes are required in the cache coherence protocol. We call this hybrid-policy HTM protocol *ZEBRA*[1].

Figure 5.1 depicts an interleaving of three concurrent transactions and highlights some important behavioural aspects of our proposal. In the eager case (Figure 5.1-a), we see that although transactions *T1* and *T3* are independent, *T3* is stalled because of a chain of dependencies created via transaction *T2*. This does not occur in the hybrid-policy ZEBRA design (Figure 5.1-b) or the purely lazy case (Figure 5.1-c) and in the example shown all three transactions commit without conflicts. It should be noted here that in ZEBRA writes to A and B by T2 and T3 are managed lazily, since the lines were annotated as contended at an earlier stage of the execution. On the other hand, in the lazy case *T2*'s commit is delayed because *T1*, having a relatively large write-set, has locked resources that *T2* needs to publish its updates. This in turn delays *T3*'s commit. With ZEBRA *T1* is able to perform an instant commit since none of the lines in its write-set are contended and, hence, are managed eagerly, allowing *T2* and *T3* to proceed with their commit operations without any delay on account of *T1*.

There are certain other benefits that stem from using such an approach. Deadlock avoidance mechanisms are not required since contended lines are eventually managed lazily, thereby guaranteeing forward progress. Significant

[1] An African folktale speaks of how the white zebra fell into a fire and burning sticks scorched black stripes on its flawless coat. Here, transactions manage data purely eagerly (white) to begin with but acquire lazy lines (black stripes) when they conflict (fall into a fire)

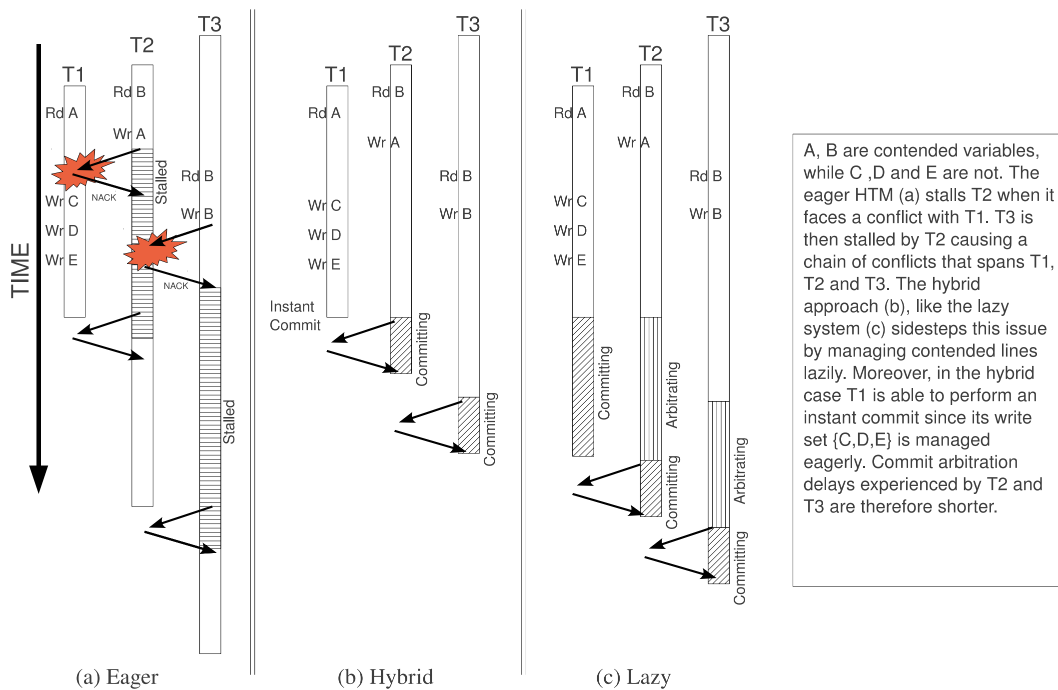


Figure 5.1: Behavioral differences between different HTM design points.

reductions in transaction commit delays result in a major contraction of the window of contention for concurrent transactions. The burden on lazy versioning mechanisms is considerably reduced enabling much larger transactions to run without resorting to safety nets (like serialization via a single global lock). This effect combines synergistically with a coherence-decoupled lazy version buffer – write-write conflicts, downgrade and abort misses (defined later) can be largely eliminated, amplifying gains achieved from the central idea. Since the design does not lock policy it can adapt to changing workload conditions and is resistant to pathologies that fixed policy HTMs suffer from. Therefore, this proposal touches upon a sweet spot in the HTM design space that offers both simplicity of design and robust performance.

5.2 Background

Most HTM systems proposed to date implement fixed policies for version and conflict management mechanisms. Fixed-policy HTM systems are faced with

challenges that limit the concurrency of transactional workloads. Eager HTM systems like LogTM [92] perform poorly when the frequency of conflicts is high, yet they introduce minimal synchronization overheads when contention is low. By hiding speculative updates from the rest of the system during transactional execution, lazy HTMs [56] are more efficient at extracting concurrency in high contention, but this in turn places significant pressure on the commit phase, whose performance then becomes critical to the scalability of the system. Parallelism at commit is important when running applications with low contention but a large number of transactions: Transactions that do not conflict should ideally be able to commit simultaneously. The very nature of lazy conflict resolution protocols makes it difficult since only actions taken at commit time permit discovery of data races among transactions. Simple lazy schemes like ones employing a global commit token do not permit such parallelism. Hence most lazy protocols employ more complex approaches like finer-grained locks on shared memory [29], optimizing certain safe interleavings [102] and early discovery of conflicts [143]. Eager schemes [92] do not suffer from this problem and allow truly parallel commits thanks to their in-place updates and early detection of conflicts. Thus, complicated protocol extensions to support higher commit parallelism are not critical to improve common case performance for such workloads.

Sanyal *et al.* [117] proposed filtering of thread-private data with support from the cores and the operating system. While this reduces pressure on versioning mechanisms in HTMs, it does not separate contended data from non-contended data. This separation is not as distinct as that between thread-private and shared data and can only be known by runtime adaptability, as we propose in this work, or by fine-grained profiling of application behaviour and access patterns. The latter is not always feasible because of large variations due to different datasets and thread interleavings.

Mixed-policy HTM designs like DynTM (UTCP) [81] and LV* [96] have been introduced earlier. DynTM deserves further discussion since it chooses a different dimension and granularity of data when compared to our work. It works at the granularity of a transaction and then develops a cache coherence protocol around it that supports multiple ways to version the same shared memory block. This choice of granularity does not match that of the underlying coherence infrastructure which works at the granularity of cache lines. The result is thus increased complexity of design, which will be a significant criterion in any decision to incorporate TM in silicon.

Both ZEBRA and DynTM are adaptive HTM designs. DynTM adapts based on a history of past transactions, trying to figure out the best policy for each

transaction. This is an inherently slow process. Switching entire transactions from eager to lazy is cumbersome as well. ZEBRA, on the other hand, adapts seamlessly. If a transactional reader exists for an eager line when a conflicting write is issued, policy switch to lazy occurs without need for either stall or abort. A stall can occur only if an eager writer exists and lasts until the writer commits or aborts. Moreover, unlike DynTM, after a policy switch the behavior of a transaction does not change drastically. As ZEBRA discovers contention for shared data a gradual shift in behavior occurs permitting fine-grained adaptability.

Work by Shriraman *et al.* (the FlexTM design [122]) allows flexibility in policy by implementing managers in software, using simple hardware mechanisms (like alert-on-update) to detect conflicting scenarios. The commit mechanism relies on compare-and-swap hardware primitives and does not provide strict forward progress guarantees. Although pathological scenarios might be rather rare, they pose significant verification challenges to confirm that this is indeed the case. While the use of hardware techniques to provide TM support speeds up the FlexTM design making it substantially faster than software TM systems, the need for frequent invocation of software handlers is indicative of a significant performance shortfall when compared to HTM systems.

5.3 Design and Operation

5.3.1 Conceptual Overview

The ZEBRA system is built on top of the baseline tiled CMP architecture described in Chapter 3. Figure 5.2 shows the salient features of the architectural framework. Each tile comprises a processing core, a slice of a shared inclusive L2 cache and corresponding directory entries. The tiles are interconnected by a mesh-based routing network. Each processing core has private Level 1 instruction and data caches. The directory keeps private caches coherent using a MESI protocol. In this case, two single-bit speculative access annotations are maintained at the private caches for each cache line - SR (for speculatively read lines) and SM (for speculatively modified lines). Such annotations have been used by several prior HTM proposals [56, 92] to track transactional reads and writes. Read set signatures [27] are employed to permit speculatively read lines to be evicted from private caches. Like other lazy HTM designs that leverage private caches for lazy versioning, our design is capable of performing gang-invalidation of those lines whose SM bit is set.

In order to track contention in the ZEBRA HTM design, we extend per-cache line metadata at the directory and at the private caches with just one additional bit – *"contended bit"* – hereafter referred to as the *C-bit*. The C-bit is transported with all coherence requests and data responses. A C-bit value of "1" indicates that the line has experienced contention in the past. The bit is reset if a line is flushed from the on-chip cache hierarchy or when a non-transactional update is seen by the directory.

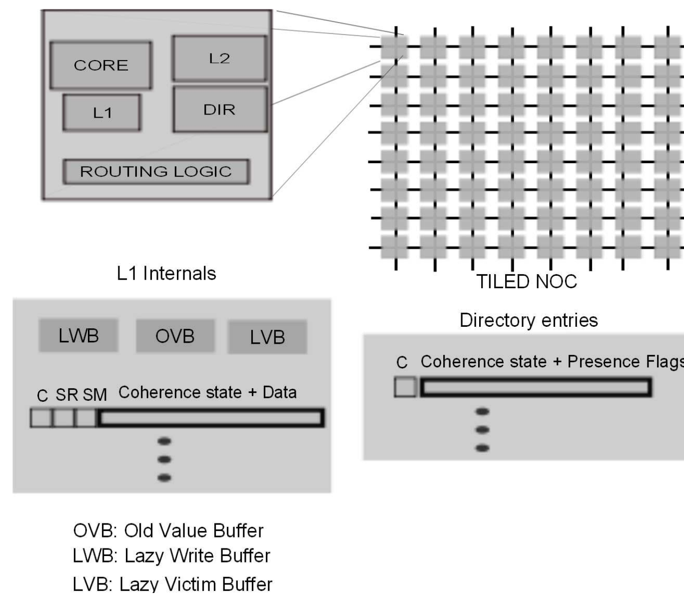


Figure 5.2: ZEBRA – Salient architectural features.

The number of contended lines accessed by a transaction is usually quite small in the workloads we have experimented with. As will be shown in Chapter 7, keeping such writes away from the cache improves performance by reducing the number of *contamination misses* [149] – misses due to invalidation of speculatively updated lines on aborts – and redundant permission downgrades from exclusive or dirty state to shared state (which we term *downgrade misses*) that allow detection of conflicts. Moreover, this also mitigates the effect of false writer-writer conflicts. Therefore, we deemed it prudent to introduce a *Lazy Write Buffer (LWB)* to contain speculative updates to contended lines. This buffer is sized to be large enough to accommodate the contended fraction of the write set of a transaction in the common case. We have found that a 32-word buffer is sufficient to handle most commonly occurring cases. This buffer is drained when committing a transaction and discarded when aborting. Writes buffered in this structure do not participate

in coherence until the transaction starts commit. Occasional situations when the buffer is completely filled up are handled by buffering subsequent contended-line updates in the cache. Prior to such a cache line update, non-exclusive (shared) access is acquired to the line (line-fill if not present; or downgrade to shared with write-back if dirty) in order to preserve its old value. Figure 5.3 shows how writes from the processor are dealt with by the private cache controller. To minimize the possibility of spilling lazy speculative state from the L1 cache we prioritize retention of such lines in the private cache and add a small (8 cache lines) *Lazy Victim Buffer (LVB)* to contain rare spills due to limited associativity. This approach works well for the workloads considered here. In the rare case of spills of contended lines, we enforce serialization.

Updates to lines that are either non-contended or have unknown C-bit status bypass this buffer (see Figure 5.3) . This can cause coherence requests to be issued to the directory if L1 line-fill or write permissions are required. If the result of a coherence operation indicates that the line is contended, the write is buffered in the LWB. In either case, the line is allocated in the cache (if not already present) and its C-bit state is updated. If the C-bit is not set, the update happens in place and the old contents of the line are recorded in an *Old Value Buffer (OVB)* or written to a thread-private log in virtual memory in case OVB capacity is exceeded. This aspect of eager behavior is similar to that of LogTM [153].

A transaction with no updates to contended lines can commit without delay, permitting true commit parallelism in such a case. If there are some lazy updates, they must be validated and made globally visible. We adopt the simplest possible approach to do so by having the committer acquire a global commit token. Our results show that in workloads where lazy conflict resolution yields best results we compete very well against or better the performance of the more sophisticated scalable commit approach adopted by STCC [29]. While a more scalable lazy commit scheme would further enhance our proposal, the design choice is orthogonal to the key ideas described in this work. All writes in the LWB are made globally visible at commit. All lines in the shared (S) state in cache with SM indicator set are upgraded to modified (M) state after the directory grants exclusive permissions to the line.

All coherence messages generated in response to speculative accesses by the core are distinguished from ordinary ones by setting a special flag in such messages. An abort occurs when any non-speculative coherence message hits a line speculatively accessed by a transaction. It should be noted that invalidations that result when lazily managed lines are committed are non-transactional. For eagerly managed lines a requester-retry policy similar to the one adopted by

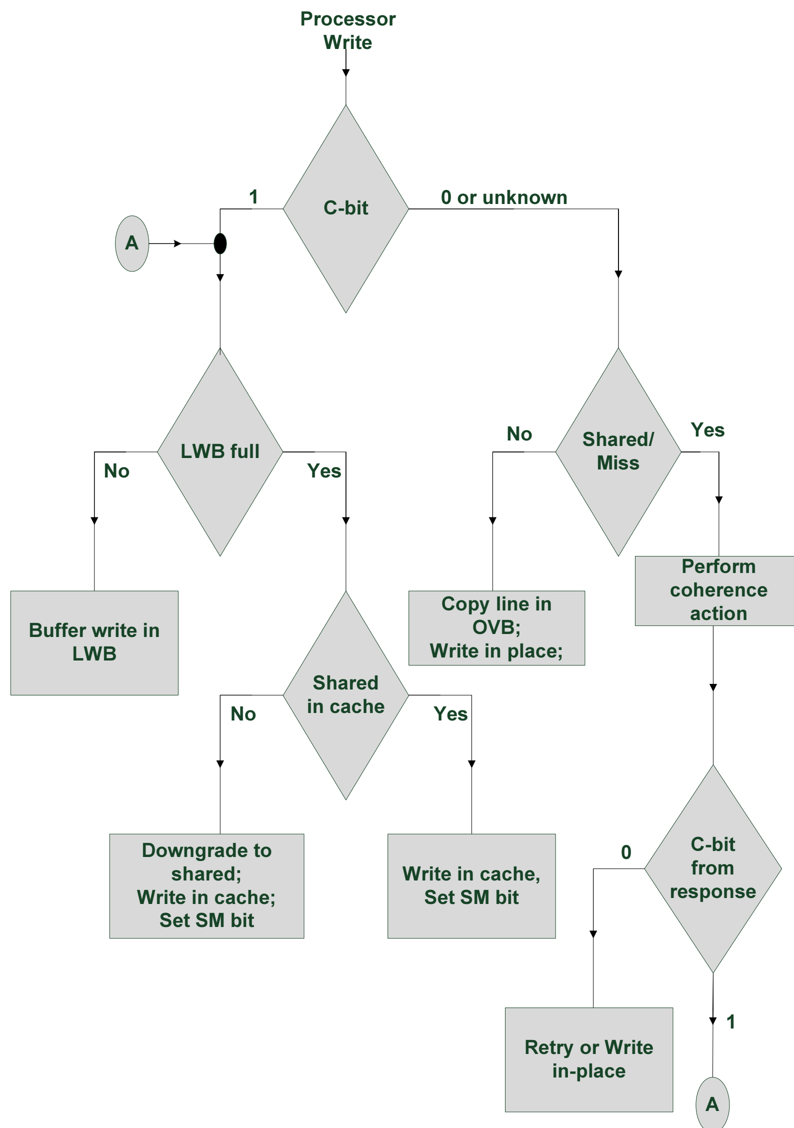


Figure 5.3: Write handling at the private cache.

LogTM [153] is used. If a cyclic dependency on eager lines is detected (refer usage of possible cycle flag in [92]) at one or more transactions in the dependency chain, they abort to break the deadlock. No software intervention is required. A unique aspect of our design is that offending cache lines will henceforth be treated lazily during re-execution and, thus, will no longer have the potential to

cause deadlock. This effect renders LogTM's usage of TLR-like timestamps [106] unnecessary for guaranteeing forward progress.

C-bits at the directory are set when unblock messages sent by cores to indicate completion of in-flight coherence operations indicate contention. Contention may be reported if a requester discovers a conflict with another transaction or when a committer publishes its contended lines. The directory reports this status in all subsequent coherence messages. The C-bit is cleared when a non-transactional update to the line is completed allowing memory to be recycled without the old C-bit value affecting behavior in the new usage context. In most applications it is highly unusual to find non-transactional updates to a cache line interleaved with transactional accesses. The C-bit is also cleared if a line must be evicted from the directory.

5.3.2 Protocol behavior

Standard directory-based MESI cache coherence is employed for detecting and managing conflicts. Coherence messages now contain two new flags - *transactional status* and *contended status*. An additional flag, *commit status* is added to UNBLOCK requests indicating whether they correspond to commit-time updates. Figure 5.4 depicts key protocol actions that occur when contended lines are accessed. All cache lines are managed eagerly by default.

Figure 5.4-b shows steps taken when a switch to lazy management occurs on encountering contention on a cache line for the first time. The transaction interleaving considered here is the one between transactions *T1* and *T2* shown in Figure 5.4-a. Core 1 (running *T2*) initiates a write to line A (address *0x204*, step 1). The store misses in the private cache structures (step 2) and results in a *TGETX* (GETX with transactional flag set) request to the directory (step 3). This coherence request results in a *TINV* (transactional invalidation) being sent to the reader, Core 0, and data being sent from the L2 to Core 1 (step 4). Core 0, running transaction *T1*, on receiving *TINV* checks if it is currently managing the line eagerly. It finds that it has only read the line transactionally (SR is set) (step 5). Hence it is in a position to forward data to Core 1 for lazy management (otherwise the requester would be stalled till T1 commits or aborts). It marks the line as contended in its private cache (step 6) causing any future write from Core 0 to be managed lazily. A acknowledgment with *contended status* is sent to Core 1 (step 7). Core 1 on receiving such a response places the line in shared state in its cache and sets the local C bit. The write, instead of updating the cache line, is now buffered in the LWB (step 8). It then indicates completion of the

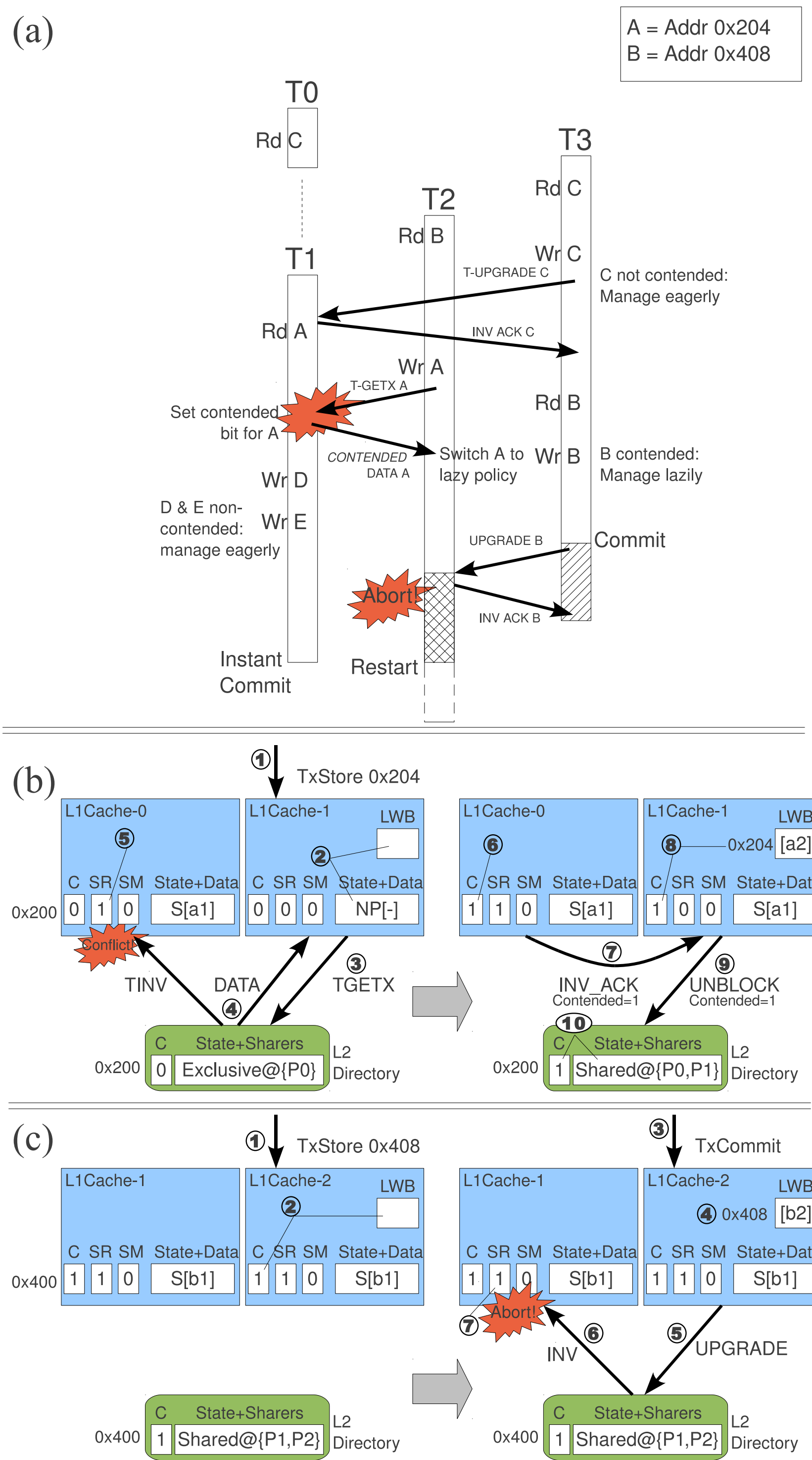


Figure 5.4: ZEBRA – Key protocol actions.

coherence operation by sending an *UNBLOCK* message with contended status to the directory (step 9). On finding a contended status in the *UNBLOCK* message the corresponding C-bit is set at the directory (step 10). The line will now be managed lazily by all accessors until a non-transactional access causes a C-bit reset.

Figure 5.4-c shows protocol actions that occur when lazily managed lines are published upon commit. The details correspond to interactions between transactions *T2* and *T3* in Figure 5.4-a. Core 2 (running *T3*) initiates a write to line B (address 0x408, step 1). The line is found in cache with C-bit set. Hence, the write is buffered in the LWB (step 2). When *T3* commits (step 3), it first acquires a global commit token. It then drains the LWB (step 4) acquiring exclusive ownership over line B by sending a non-transactional *UPGRADE* request to the directory (step 5).The directory responds by sending a *INV* (non-transactional invalidation) set to Core 1 (step 6). *T2* on Core 1 aborts when a non-transactional invalidation conflicts with a speculatively accessed line (SR is set, step 7). Since *T2* had the lone lazy write to A in its write set, no old value restoration is required. LWB is reset and re-execution of *T2* can start immediately or when deemed right by a back-off algorithm. It should also be noticed that line C, also part of *T3*'s write set, does not need to be published since it was managed eagerly. Core 1 completes its upgrade operation by sending an *UNBLOCK* message to the directory. This message has the *commit flag* set, causing the directory to maintain a value of 1 for the C-bit. Ordinary requests for exclusive ownership generated from non-transactional code result in *UNBLOCK* messages without the *commit-flag* set and cause the directory to reset the bit.

Cache controllers at both L1 and the L2-directory now support a few new transitions summarized in Figure 5.5. New transitions are represented by black dotted lines in the figure, while transitions that already exist in the baseline MESI protocol are shown in light grey. For clarity, only states and baseline transitions that aid in illustrating the changes are shown. At the directory level, the behavior of *TGETX/TUPGRADE* requests is similar to that of their non-transactional counterparts, but the transactional variants can eventually result in the reception of *contended UNBLOCK* messages that cause a transition to shared state (SS). For example, a *TGETX* from Core 3 could cause a directory transition from *SS@{1,2}* (shared by Cores 1 and 2) to *SS@{1,2,3}* if contention is detected. This permits lazy versioning of contended data at the new requester while still allowing conflict detection to happen. Similarly, the transition from *MT@{2}* (exclusive/dirty at Core 2) to *SS@{1,2}* is supported when handling *TGETX* requests. Such a situation might arise if Core 2 forwards a contended line to Core 1, behaving as if the

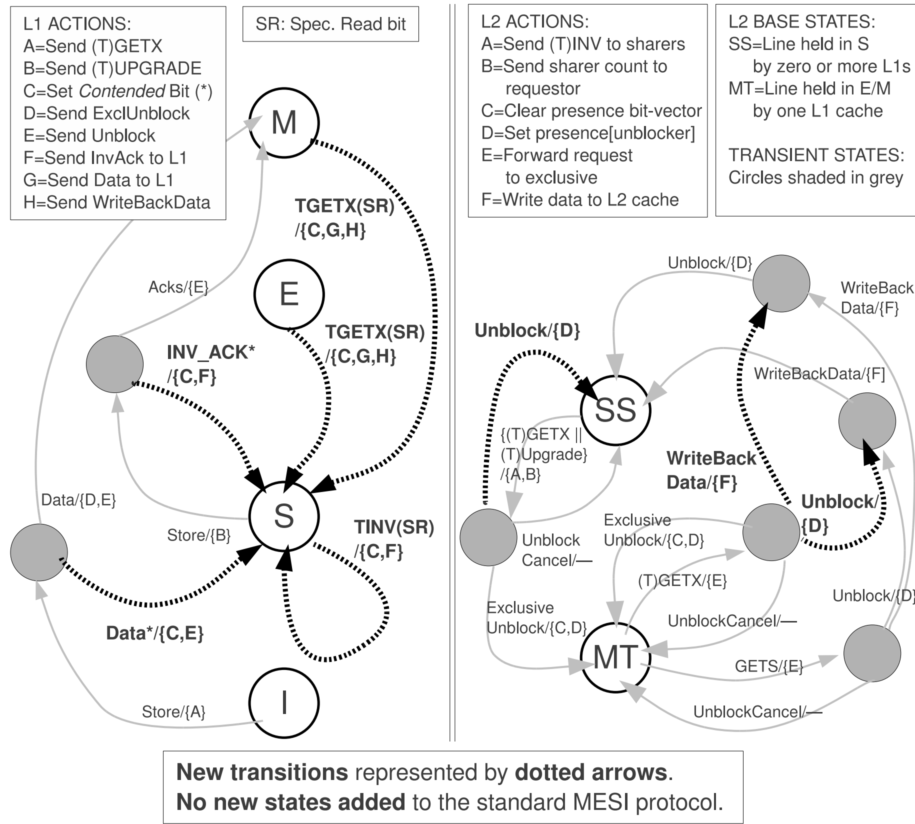


Figure 5.5: Support for new transitions at L1 (left) and L2-directory (right) controllers.

request had been a GETS (the line is also written back to L2). At the L1, we support transitions to shared state on local write misses and upon receiving *TGETX* or *TINV* requests from the directory. This allows coherence mechanisms to be used to detect conflicts on such data after this event has occurred.

The examples above highlight key behavioral aspects of the ZEBRA protocol. Other cases are handled in a similar fashion. If a transaction is managing a line eagerly, it is given a chance to reach commit by permitting it to stall other requesters. When such events occurred the C-bit for the line is set. This causes the line to be managed lazily once the transaction commits or aborts. The risk of deadlocks is avoided by using a LogTM-like *possible-cycle* bit, but in a different way. The bit is set if the transaction has stalled a requester attempting to access eagerly managed data. When a transaction is stalled by another, it checks if *possible-cycle* flag is set. An abort is triggered if so. The eagerly managed line will henceforth

be managed lazily and will no longer be able to cause deadlocks. Transaction timestamps, employed by LogTM for conservative deadlock avoidance, are no longer transported in coherence messages. However, care must be taken when contended lines are evicted from the L2.

Coherence requests issued by non-transactional codes result in an abort if they hit a transactionally accessed line. The flexibility to ask the requester to retry such requests can be incorporated by transporting the *commit status* flag with invalidation messages. This flag would be set for non-transactional invalidations sent out when the lazy portion of a transaction's write-set is being committed. The receiver, if transactional, would then know that it cannot ask for a retry and must abort if such an invalidation hits a speculatively accessed line.

5.3.3 Resetting C bits

The ZEBRA design also incorporates a mechanism to revert lazy lines back to eager. This is useful for applications which show instances where a high contention phase precedes a low contention phase, while operating on the same data. If a lazy line is found to be present in L1 at commit with the requisite permissions consistently, it is reasonable to assume that it is no longer actively contended. The ZEBRA design tracks a limited number of such lazy lines, recording their recent commit histories in a fully associative 16-entry structure called STET (Switch-To-Eager Table). Each entry in the table consists of a cache-line address, a 4-bit counter and replacement policy metadata. Entries are made to STET when write-hits are encountered for lazy writes at commit time. Every subsequent commit-time write hit increments an associated counter. When the counter reaches a threshold value, the C-bit is reset both locally and at the directory. In case STET becomes full replacements occur based on a least recently committed strategy (which is implemented using an age-bits based pseudo-LRU scheme). Entries are also removed when conflicts are detected on those lines. Algorithm 2 summarizes key actions taken by the prediction logic.

5.4 Evaluation

5.4.1 Experimental Setup

Table 5.2 lists all HTM configurations evaluated in this work. GEMS simulation infrastructure provides support for LogTM-SE [153] as the eager-eager (EE) HTM system of reference, as well as an implementation of a global commit

Algorithm 2 C-bit Reset Prediction Logic

```
if LazyWriteHit@Commit(A) then
    if PresentInSTET(A) then
        STET(A) ← STET(A) + 1
    else
        InsertSTET(A)
        STET(A) ← 1
    end if
    if STET(A) > STETTHold then
        ResetCbit(A)
        RemoveSTET(A)
    end if
else
    if PresentInSTET(A) then
        STET(A) ← 0
    end if
else
    if Conflict(A)andPresentInSTET(A) then
        RemoveSTET(A)
    end if
end if
```

token-based, lazy-lazy (LL) HTM system described by Bobba *et al.* [22]. We extended this infrastructure with detailed implementations of STCC [29] and ZEBRA, our hybrid-policy HTM protocol, allowing fair comparison of several major HTM design points within the same architectural framework. While the original implementation of the global commit token (LL-GCT) system models a private, per processor infinite write buffer, we extended the simulator to precisely model finite buffering for transactional writes. Furthermore, we have also modelled an LL design with an idealized commit scheme (LL-ideal), in which the commit arbiter magically detects conflicts against currently committing transactions, allowing non-conflicting transactions to commit in a truly parallel fashion. For the implementation of ZEBRA, special status bits are added to coherence messages as described earlier, and the behavior of cache and directory controllers is suitably modified. For all simulations we use an ideal bookkeeping scheme to track read sets (*perfect signatures*) even when some speculatively read lines have been evicted, in an attempt to isolate our study from the effects of false conflicts arising from non-ideal signature schemes like bloom filters.

We simulate both small and medium datasets for workloads that exhibit differences in transactional behavior across various design points. This not

Table 5.1: Specific parameters of the ZEBRA system.

ZEBRA System	
Memory Settings	
Lazy Write Buffer	Non-coherent, private, 128 bytes
Old Value Buffer	8 cache lines
Lazy Victim Buffer	8 cache lines

Table 5.2: HTM configurations evaluated in Chapter 5.

Configuration	Description
EE	The LogTM-SE design [153]
LL-GCT	Global commit token design with finite buffering [22]
LL-ideal	Lazy with idealized truly parallel commits
LL-STCC	The Scalable TCC design [29]
ZEBRA	The ZEBRA design

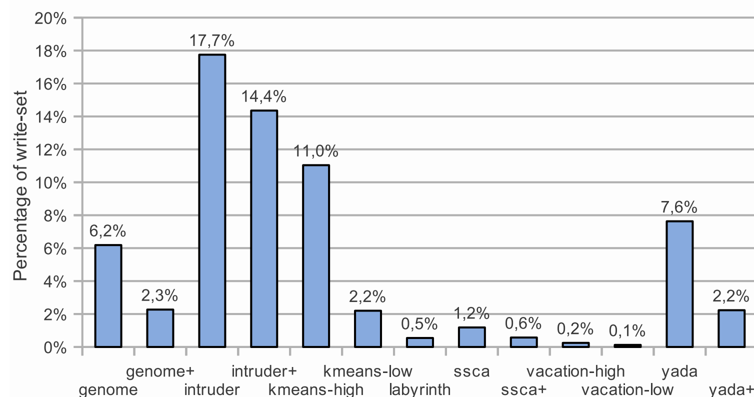


Figure 5.6: Relative sizes of conflict sets for STAMP applications.

only yields more credible statistics but also allows the hybrid design to achieve steady-state performance.

5.4.2 Workload Characteristics

Conflict set sizes. To measure the proportion of contended data in a typical transaction's write set we define the *conflict set* of a transaction as the set of lines that were written and managed lazily over the duration of execution of a

transaction. Figure 5.6 shows the cardinality of the conflict set as a percentage of the corresponding write set size averaged over an entire run. We see that even in applications with moderate to high contention, like yada and intruder, the conflict set is far smaller than the write set. Workloads like ssca2, that have both high concurrency and a high commit rate, experience contention on less than 1% of the write-set. Moreover, for longer running workloads (small to medium in Figure 5.6), the ratio of the two set sizes drops even further. A common case size of less than 20% of the write-set bears out our choice to the separately manage the large non-contended fraction of the write set.

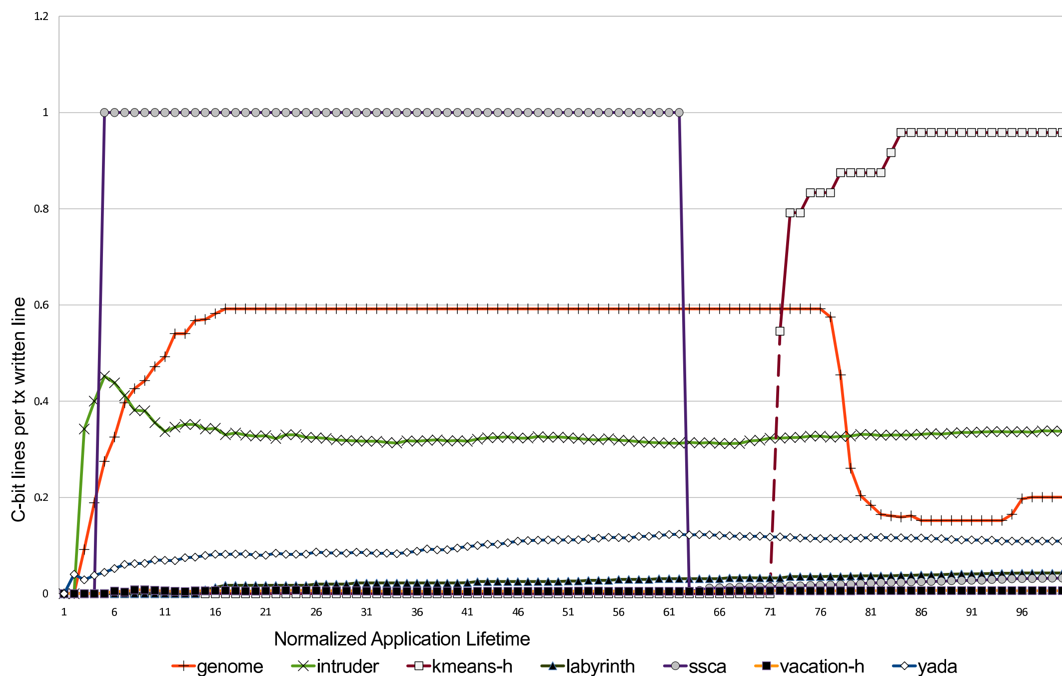


Figure 5.7: Proportion of C-bit lines over time.

Contention Characteristics. Figure 5.7 shows the proportion of contended data as a fraction of the total number of cache lines written by an application at various points during workload execution (only the parallel section is considered). The data has been collected by running each workload on the ZEBRA configuration. Each application shows a characteristic shape based on contention seen during various application phases. Periods of rise indicate spreading contention. Periods when the curve drops are indicative of transactional execution with relatively low contention. Flat periods represent non-transactional execution or periods

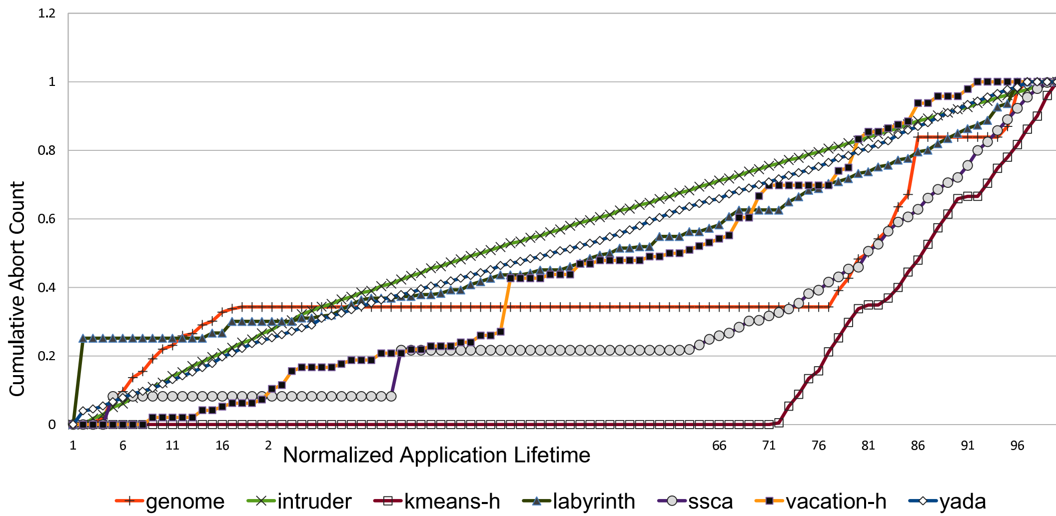


Figure 5.8: Contention over time.

when the level of contention is stable. More insight is gained when Figure 5.7 is interpreted in conjunction with Figure 5.8. Figure 5.8 shows the cumulative number of aborts at various points during workload execution as a fraction of the total number of aborts seen when the workload eventually completes. Rising periods on this plot denote high contention phases.

For a high contention application like intruder we see that most contended lines can be discovered early. The cumulative abort curve is close to linear indicating uniform contention throughout the execution of the application. Yada and labyrinth show uniform contention (with varying intensities), but, as can be seen from Figure 5.7, the proportion of contended data shows a slow but steady increase over most of the workload duration. Vacation shows low but uniform contention. Ssca2, a low contention workload with tiny transactions, shows two phases – a long low contention phase with few transactions that only touch contended lines followed by a second low contention phase where a substantial amount of non-contended transactional data is written. This second phase accounts for 80% of the aborts seen. Genome passes through a distinct high contention phase when it begins. This accounts for about 35% of the total aborts seen. It then enters a long low contention phase which is followed by a moderately contended phase (contributing 65% of aborts) where a lot of non-contended data is also written. Kmeans (high contention Kmeans) begins with a long (about 70% of execution time) no-contention phase. The last 30% of the

application shows a sharp rise in contention, which occurs over a limited data set. This is indicated by the contended fraction that reaches saturation rather quickly around a value close to 1.

5.4.3 Performance Analysis

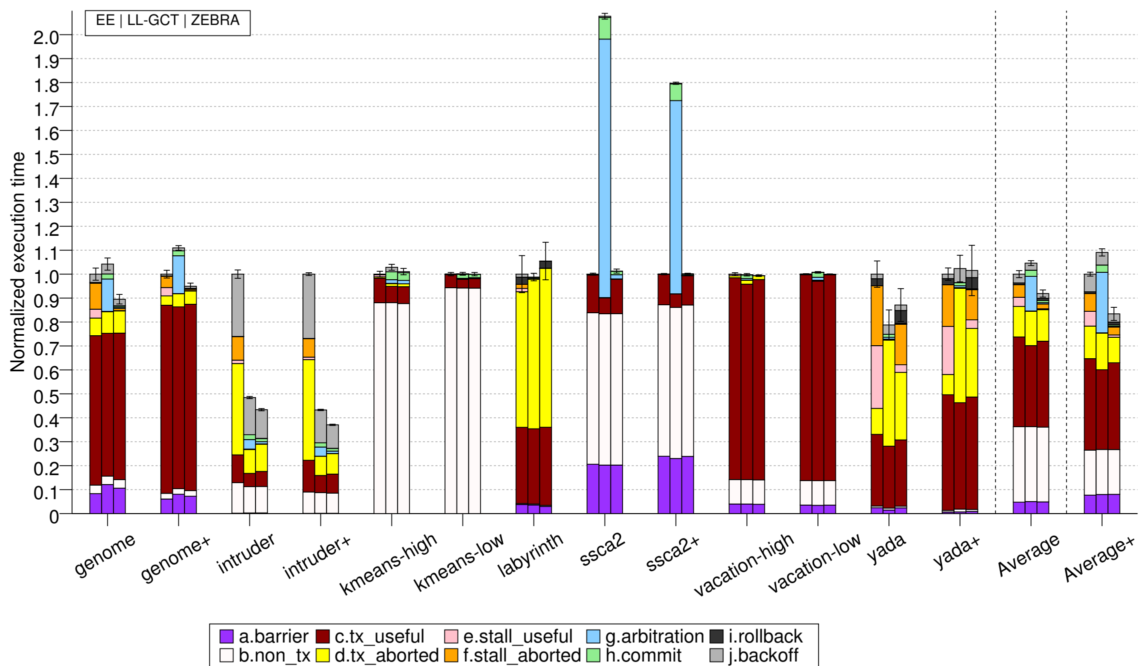


Figure 5.9: ZEBRA vs. fixed-policy HTM designs.

Figure 5.9 shows the relative performance of ZEBRA and two basic HTM designs – EE and LL-GCT (see Table 5.2). Figure 5.10 then compares the performance of ZEBRA to that of three purely lazy designs – the basic LL-GCT configuration, LL-ideal and LL-STCC. The average for long running workloads (marked with the suffix +) has been calculated separately (appears as Average+). Figure 5.9 shows that the ZEBRA design provides marked gain in overall performance (18% over EE and 30% over LL-GCT). It closely tracks the performance of the best policy for each workload and excels when applications show mixed transactional behavior – having both contended and non-contended phases of execution. Figure 5.10 highlights how ZEBRA can achieve performance at par

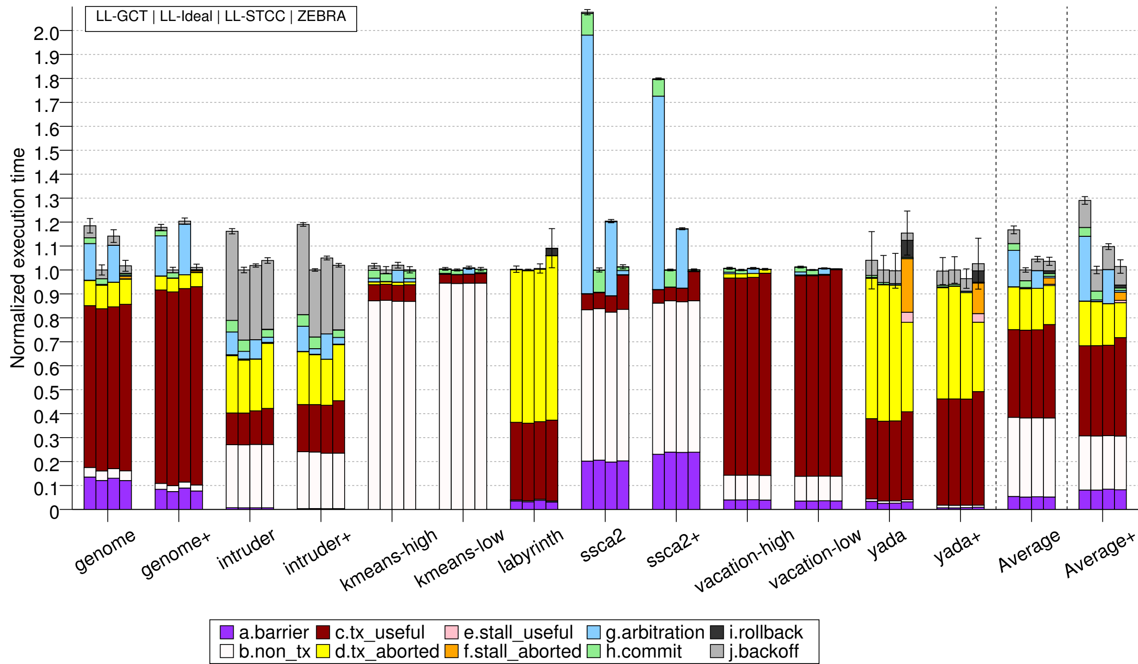


Figure 5.10: ZEBRA vs. idealized lazy designs.

with that of the idealized LL design (LL-ideal) and is noticeably faster (8% overall) than LL-STCC for long running workloads.

Figure 5.11 shows two measures – average deviation from the best observed performance over all workloads and the standard deviation of performance

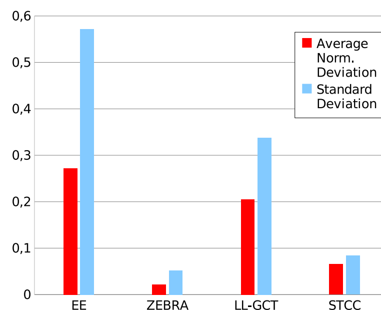


Figure 5.11: Deviation from best observed performance.

normalized to the best across all workloads. The hybrid approach achieves by far the lowest swings, implying consistent performance and robustness.

We have further investigated the behavior of the hybrid approach. Figure 5.12 shows the distribution of purely eager, partly-eager-partly-lazy (hybrid) and purely lazy commits in each application. Table 5.3 shows utilization of LWB and OVB by each transaction (identified by *TID*) in various STAMP benchmarks. Write-set sizes (WS in the table) and OVB occupancy have been shown in cache lines. LWB occupancy represents the number of bytes that were managed lazily in the structure.

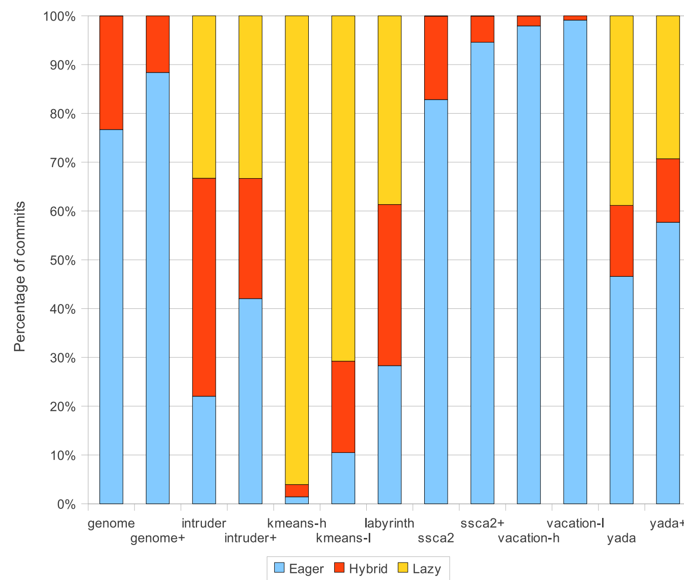


Figure 5.12: ZEBRA – Policy distribution at commit.

The discussion below highlights important observations and presents insights gained from detailed study of interactions between HTM policies and the behavior of individual workloads.

Genome. This workload exhibits a high contention phase early in its execution where lazy designs outperform the EE system. This phase involves removal of duplicates (hash-table insertions). Reader - writer conflicts dominate at the beginning of the phase and lazy approaches inherently allow greater concurrency in such a situation. ZEBRA quickly switches the management of contended cache lines to lazy and completes the phase faster than EE, but a bit slower than LL-GCT or LL-STCC. The second phase is dominated by transactions with

moderate write-sets (3.4 cache lines on average) with accesses to predominantly non-contended data and is the determinant of overall performance. The eager approach proves to be the quickest here. ZEBRA runs most transactions in a completely eager way and does not suffer from commit overheads seen in the lazy designs. Occupation measurements of the OVB and LWB structures shown in Table 5.3 confirm how the hybrid system adapts to the transactions of this second phase: TID1 is always eagerly managed (OVB usage equals write-set size), while TID2 and TID3 are purely eager in 70% and 80% of their commits, respectively. No transaction in genome ever commits in a purely lazy fashion (contended lines always comprise only a small fraction of the Wset), demonstrating the benefits of the proposed data-centric approach for policy selection on a per-cache line granularity. The third phase again exhibits low to moderate contention. Since the application shows mixed behavior, our hybrid approach outperforms all others. Overall, we find that this result demonstrates the efficacy of quick adaptability to changing workload conditions in ZEBRA, achieving performance close to that of the idealized lazy (LL-Ideal) configuration. Genome+ (genome with a medium sized dataset) shows much less contention, thereby widening the gap between the purely lazy and EE or ZEBRA configurations. As we show in Figure 5.12, in ZEBRA almost 90% of commits happen eagerly for genome+.

Table 5.3: ZEBRA – LWB and OVB utilization.

	TID0			TID1			TID2			TID3			TID4		
Workload	**WS**	**OVB**	**LWB**	**WS**	**OVB**	**LWB**	**WS**	**OVB**	**LWB**	**WS**	**OVB**	**LWB**	**WS**	**OVB**	**LWB**
genome+	1.3	1.2	1.4	1.0	1.0	0.0	**3.4**	**3.3**	**0.7**	3.4	3.4	0.4	2.2	1.8	2.8
genome	1.3	1.1	1.6	1.0	1.0	0.0	**3.5**	**3.1**	**1.9**	3.5	3.3	0.6	2.5	1.4	6.1
intruder+	1.0	0.0	4.0	**5.7**	**4.5**	**7.1**	1.2	1.0	0.9	-	-	-	-	-	-
intruder	1.0	0.0	4.0	**6.1**	**3.8**	**13**	1.5	1.0	2.2	-	-	-	-	-	-
kmeans-h	**2.0**	**0.1**	**65**	1.0	0.0	3.9	1.0	0.0	4.0	-	-	-	-	-	-
kmeans-l	**2.0**	**0.5**	**47**	1.0	0.0	4.0	1.0	0.0	4.0	-	-	-	-	-	-
labyrinth	0.9	0.1	3.1	**217**	**8.0**	**41**	3.8	2.9	4.1	-	-	-	-	-	-
ssca2+	1.0	0.1	3.6	1.0	0.0	4.0	**2.0**	**1.9**	**0.2**	-	-	-	-	-	-
ssca2	1.0	0.1	3.8	1.0	0.0	4.0	**2.0**	**1.8**	**0.7**	-	-	-	-	-	-
vacation-h	**6.8**	**6.8**	**0.2**	5.7	5.7	0.1	4.0	4.0	0.1	-	-	-	-	-	-
vacation-l	**6.1**	**6.1**	**0.1**	5.3	5.3	0.1	2.5	2.5	0.0	-	-	-	-	-	-
yada+	2.5	0.0	11	0.0	0.0	0.0	**70**	**8.0**	**18**	1.0	0.8	0.7	1.3	0.3	5.1
yada	2.0	0.0	8.1	0.0	0.0	0.0	**60**	**8.0**	**40**	1.0	0.5	2.1	1.4	0.2	7.0

Intruder. This workload shows high contention, even with large input sizes. Eager transactions acquire exclusive ownership to data before they are guaranteed to commit. This, in conjunction with a high probability of conflicts, leads

to prolonged stalls and pathological cases where transactions form chains of dependencies causing aborts. In this contended scenario, lazy systems are able to exploit concurrency much better resulting in far fewer aborts. This workload has 3 transactions. TID0 extracts elements from a highly contended queue of packets, causing the EE system to experience 15K aborts (out of total of 29K aborts overall). Lazy designs reduce this number to 4K (13K aborts overall). ZEBRA quickly discovers contended lines, and the conflicting location (pointer to the head of the queue) becomes lazy (as indicated by an LWB occupancy of 4 bytes in this transaction) decreasing the number of aborts when preforming transactional dequeue operations to 3K (total of 11K). With ZEBRA, the largest transaction (TID1) can commit eagerly 25% of the time on average, even though it accesses relatively large amounts of contended data (see Figure 5.6). A large fraction of TID1's write set (6.1 lines) is still non-contended and thus an average of 3.8 lines are managed eagerly, as revealed by the OVB occupancy in Table 5.3. TID2 also exhibits a predominantly eager behavior, committing eagerly about 50% of the time, with one eagerly managed write on average (out of 1.5 written lines). Hence, it outperforms lazy approaches achieving close to ideal performance (see Figure 5.10). Commit durations for TID1 and TID2 are significantly shorter as a large number of the transactionally modified lines are not contended and, therefore, committed instantly. We can see in Figure 5.10 how the overhead due to the arbitration and commit is substantially lower in the hybrid system, in comparison to both LL-GCT and LL-STCC systems.

SSCA2. It has a large number of tiny transactions that demand high commit bandwidth. Inherently parallel commits in eager approaches serve this requirement very well. Lazy approaches suffer, even STCC, which has a degree of scalability. This is clearly evident in Figure 5.9 where commit delays represent the primary overhead in lazy designs (except LL-ideal where transaction commit is not a bottleneck). ZEBRA is able to manage almost the entire write-set eagerly for most transactions. This can be clearly seen in the high proportion of eager commits (see Figure 5.12) and the low LWB utilization (see Table 5.3). Hence, ZEBRA is able to match the performance of the EE design.

Yada. It has a rather large working set and exhibits high contention. The workload traverses the dataset in a manner which makes it longer for ZEBRA to complete the discovery of contended lines. A longer time to achieve steady state means that in short duration runs we do not perform as well as the lazy systems. TID2, the dominant transaction (see Table 5.3), exceeds OVB capacity. This coupled with a relatively high fraction of contended data in the write-set (see Figure 5.6) results in expensive software rollback operations that degrade performance of

EE and ZEBRA. With longer runs we notice that the differences tend to become less marked. The contention in this case is less severe (as is evident the higher proportion of eager commits for yada+ in Figure 5.12) and the eager approach regains lost performance.

Labyrinth. The results generated by this workload depend significantly on the interleaving of threads resulting in marked variability in execution times (note the error bars in Figure 5.9). The data presented thus should be viewed keeping this fact in mind. The dominant transaction is TID1 as can be seen in Table 5.3. A significant fraction of data is managed eagerly (see Figure 5.6) but OVB capacity is exceeded since write-set sizes are large. Thus, relatively high contention results in expensive rollback operations on abort (see Figures 5.9 and 5.10). Consequently, lazy designs perform slightly better than the EE or ZEBRA.

Kmeans /Vacation. These applications are highly concurrent and do not show major differences in execution times with changes in policies. Nevertheless, protocol efficiencies at commit in the EE and ZEBRA result in minor improvements in performance.

5.4.4 Analysis of C-bit reversion mechanism

Data accessed in certain workloads exhibits a behavioral shift over time. For example, an initial high contention phase might be followed by a long low or no contention phase with a final moderately contended phase. In this case, relying upon non-transactional updates or evictions from the on-chip hierarchy to reset C-bits might result in lost performance. ZEBRA includes a mechanism that allows reversion of policy from lazy back to eager as described in Section 5.3.3. However, STAMP benchmarks do not include workloads that clearly highlight such behavior and hence, do not stress this potentially important feature. We believe that such behavior may not be that uncommon either. A typical example could be a workload could spawn threads that do substantial work initially creating global data structures (e.g. hashtables) in a highly contended manner. When the data structure is large enough contention drop, as a result of increased structure size or division of labor among threads.

To model such behavior and highlight the potential performance gain to be had, we created a microbenchmark, key behavioural aspects of which are described in Figure 5.13 (left). The benchmark operates on a large array of shared data. Each phase is separated from the next by a barrier. The results gathered have been averaged over several benchmark runs (with randomization enabled within the benchmark). Four HTM configurations have been evaluated: EE, LL-

Microbenchmark Outline:

```
procedure worker_thread {

     /* High contention */
     1. update global data structure
with large txns

     barrier

     /* No contention */
     2. partition global data
between threads and access each with
high update rate

     barrier

     /* No contention */
     3. transactionally read global
data from all threads.

     barrier

     /* Low contention */
     4. Update global data randomly
with tiny transactions

}
```

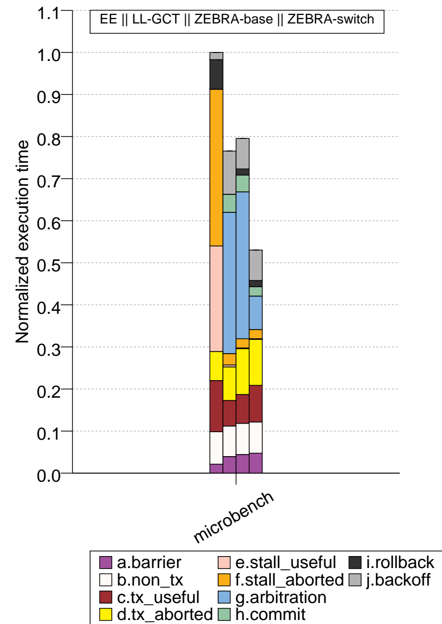


Figure 5.13: Microbenchmark analysis.

GCT, ZEBRA-base (with reversion disabled) and ZEBRA-switch (with reversion). As seen in Figure 5.14, the sharp drop in C-bit proportion (between 80% and 90% of application lifetime) results in dramatically improved performance for the ZEBRA-switch configuration reflected in Figure 5.13 (right). The reversion scheme makes ZEBRA almost 50% better than EE and about 20% better than LL-GCT or ZEBRA-base. Phase 1 dominates application execution time (since it includes large, highly contended transactions). ZEBRA configurations quickly discover contention and are able to do better than lazy designs since they are able to commit a certain number of lines eagerly. Phase 2 corresponds to the sharp decline in C-bit proportion in Figure 5.14). The policy reversion mechanism detects change in behavior of most lines, with C-bit fraction going down to almost zero. Phase 3 and Phase 4 are responsible for improvements in execution time over LL-GCT and ZEBRA-base, shrinking to 10% of total execution time in ZEBRA-switch (which manages to commit most such transactions eagerly).

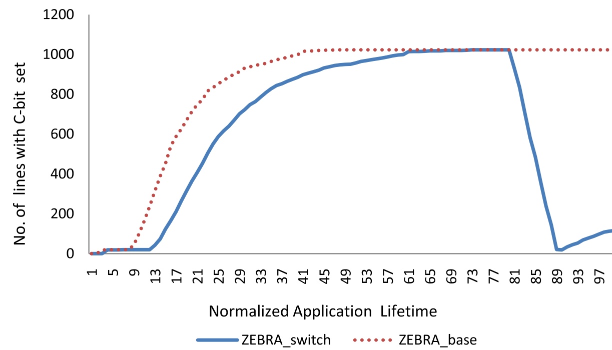


Figure 5.14: C-bit microbenchmark analysis.

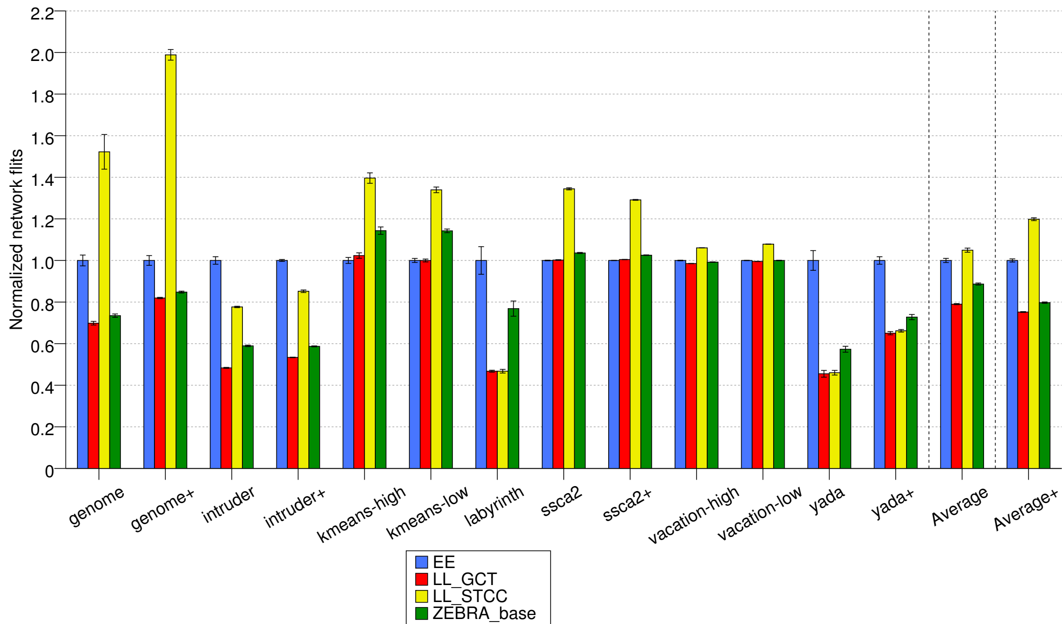


Figure 5.15: Network traffic.

5.4.5 Traffic Considerations

Traffic generated by each HTM design when running STAMP applications is shown in Figure 5.15, which shows traffic volumes in flits normalized to the EE design. Figure 5.16 plots the distribution of various protocol messages types transported through the network. In terms of traffic the hybrid approach

performs well across all workloads and puts signficantly lower demands on network bandwidth than LL-STCC.

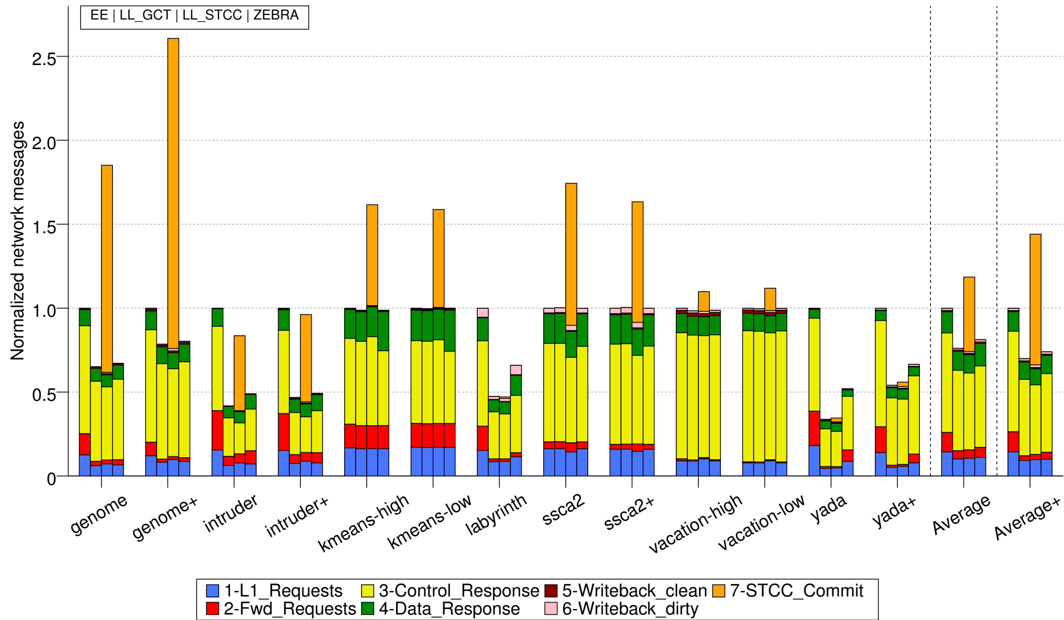


Figure 5.16: Normalized network message count.

LL-STCC shows remarkably high flit counts for several applications, which, as can be seen in the traffic distribution plot, arises due to messaging for scalable commits. In the experimental setup used for this study, large intra-chip communication bandwidth is available as only 16 in-order cores run. The parallel commit algorithm employed by the design is thus able to hide most of messaging latency. In architectures that have a low peak bandwidth or run workloads that impose high communication demands, this latency may not remain hidden and LL-STCC protocol efficiency could suffer.

5.4.6 Key inferences

We would like to highlight the following key observations during the design and analysis of ZEBRA:

- Freezing policy in hardware precludes several possible sources of performance gains as the design is then biased towards or against certain workload characteristics.

- Working at the granularity of cache lines allows existing coherence protocols to be leveraged easily with minimal changes in their behavior.
- Associating contention with data rather than code allows ZEBRA to achieve more finely balanced operating points (partly eager - partly lazy transaction commits), utilize resources better (reduced pressure on speculative data storage) and permit hardware optimizations for speeding up common case scenarios (LWB and OVB).
- Contention characteristics of data may change over time. This represents an opportunity for speedup that would remain inaccessible to systems that cannot adapt.

5.5 Concluding Remarks

In this chapter we have outlined a fresh approach to hybrid-policy HTM design. Instead of viewing contention as a characteristic of an atomic section of code, we view it as a characteristic of the data accessed therein. Our observation that contended data forms a relatively small fraction of data written inside transactions reinforces our decision to incorporate mechanisms that support efficient management of such data in the common case. In the process, our proposal – the ZEBRA HTM system – manages to bring together the good aspects of both eager and lazy designs with very modest changes in architecture and protocol. ZEBRA supports parallel commits for transactions that do not access contended data and allows reader-writer concurrency when contention is seen. We have shown, both qualitatively and quantitatively, that it can utilize concurrency better and consistently track or outperform the best performing scalable single-policy design – performing as well as the eager design when high commit rates limit performance of lazy designs and, on average, substantially better than both eager and lazy systems when contention dominates. On average, it places lower demands on intra-chip communication bandwidth. It also achieves the lowest deviation from the best measured performance over a diverse set of workloads corroborating our claim that the design is robust and less susceptible to pathological conditions. We hope this work would spur further efforts in the area of low complexity hybrid-policy HTM systems. More research can be done to develop designs that adapt to workload needs quicker and are still cost-effective enough to attract the attention of computer architects.

Chapter 6

π-TM: Pessimistic Invalidation for Scalable Lazy HTM

6.1 Introduction

Committing a hardware transaction requires making all transactional updates visible to other processors instantaneously. Depending on the choice of policy for the conflict management mechanism, the complexity of the commit process can range from a trivial local step in eager HTMs [92], to distributed algorithms in lazy HTMs [29]. By leveraging coherence traffic to check for conflicts on each individual load and store instruction, transactions in eager HTMs carry out the detection of data races with other concurrent transactions as execution progresses. As a result of this pessimistic approach, when an eager transaction reaches the end of an atomic block and executes the "transaction commit" instruction, all conflicts with other concurrent transactions have been detected and already resolved, if any aroused. This makes the implementation of commits a straightfoward operation in eager HTMs, since no further communication is required to validate the consistency of the transaction and publish the speculative updates. Commit then simply consists of releasing isolation over the read and write sets by clearing the read/write bits or signatures used for transactional book-keeping, so that remote requests can fetch those cache lines that belonged in the transactional sets of the committed transaction. For this reason, eager HTM systems impose close to zero synchronization overheads when running workloads that exhibit little

contention and demand plenty of commit bandwidth, allowing transactions to commit in a fully parallel fashion.

As discussed in Chapter 5, eager HTMs expose inherent problems when extracting parallelism in situations of moderate to high contention, limiting available concurrency and deteriorating performance. Resolving conflicts as soon as they are detected leads to sub-optimal decisions about which transaction is more likely to commit, because such judgement is made *a priori* when little information about the involved transactions is available. Eager resolution of conflicts leaves two alternatives: Either aborting one of the two transactions –based on some priority scheme– or stalling the conflicting request with the hope that the offended transaction will commit and then allow the offending one to proceed. If directly aborting one of the transactions is a rather draconian solution that can be susceptible to livelocks, stalling the requester brings in the possibility of deadlocks when circular dependencies exist, and thus requires schemes to detect and stop such cycles from forming.

Moreover, the conservative resource acquisition of eager HTMs allows a transaction to *hold state* before it is known if it will successfully commit: When conflicts are eagerly resolved by stalling the requester, each transaction *blocks* those cache lines accessed during the course of its execution, impeding any remote access to such lines if they can result in a potential race. In fact, even when conflicts are successfully resolved through stalls (no cycles are formed), eager HTMs create chains of conflicting transactions that, while stalling for the access to a cache line, continue blocking data that may be necessitated by other transactions to make progress. The result is that transactions that have no data dependencies and could execute in parallel may not be able to do so because of conflicts with other transactions that *connect* them through a chain of conflicts.

In contrast to the eager approach, lazy conflict resolution allows concurrency in more scenarios. For instance, when a race exists between a reader transaction and a writer transaction, execution of both transactions runs past the conflict in a lazy HTM, thereby permitting the possibility of a safe interleaving when the reader transaction commits before the writer transaction. On the contrary, eager HTMs must adopt a conservative approach here, resolving the conflict in favor of one of the transactions. Previous research has illustrated how lazy conflict detection can allow more parallelism [22, 121]. Delaying the resolution of conflicts to commit time avoids making the difficult decision of which is the best transaction to abort.

While lazy HTMs are theoretically more efficient than eager designs (we will discuss about this in the next chapter), the requirement of en-masse publication

of speculative updates to shared memory poses challenges at commit time, limiting the scalability of the lazy approach. Quite often several concurrent non-conflicting transactions attempt to commit simultaneously. Simple schemes, like the acquisition of a global commit token, take a conservative approach and preclude simultaneous commits of such transactions. This severely limits performance in workloads with light contention and a fairly large number of transactions that demand a high commit bandwidth. More elaborate schemes attempt to provide a degree of commit parallelism when transactions target different directory banks [29,102]. Although they leverage the coherence protocol, information in transactional coherence messages that may indicate contention is not utilized, and only commit-time invalidations are relied upon to detect races and abort conflicting transactions.

Early conflict detection can be employed in lazy HTM designs to allow non-conflicting transactions to commit in parallel. Though this has the potential to improve performance, it has not been utilized effectively so far [143]. Enabling parallel lazy commits in hardware requires correct handling of all possible interleavings of transactions. Figure 6.1-(a) shows what might happen if a buggy early conflict detection protocol does not enforce atomicity correctly. A transaction T1 that has updated cache lines A and B initiates commit operations, aborting transaction T2 in the validation phase. T2 now restarts and reads the new value of line A since T1 has acquired it exclusively. However, it is possible for T2 to read the old value of line B and potentially commit this unsafe execution, resulting in an atomicity violation. A trivial solution to overcome this issue requires every transaction to send an indication to the directory whenever the first read or the first write happens to a cache line, even though the line may be present in the private cache. In the example above, as shown in Figure 6.1-(b), T2 will send a message to the directory which will then be forwarded to the committer, T1. T1 will then ask T2 to retry the access till it has acquired exclusive ownership over line B. Thus every first read and first write to any cache line accessed by a transaction is effectively a miss in the private cache hierarchy. This turns out to be wasteful, particularly when contention is low or when large coarse-grained transactions abound.

Some careful thought reveals that the trivial approach described above, followed by EazyHTM [143], does not go far enough when utilizing the information available from coherence messages regarding potential data races. Records of conflicts are maintained only at the granularity of cores. Doing so leaves only three ways of dealing with a transaction's read set when aborting - first, wait for lines to be invalidated; second, invalidate the entire read-set; third, as described

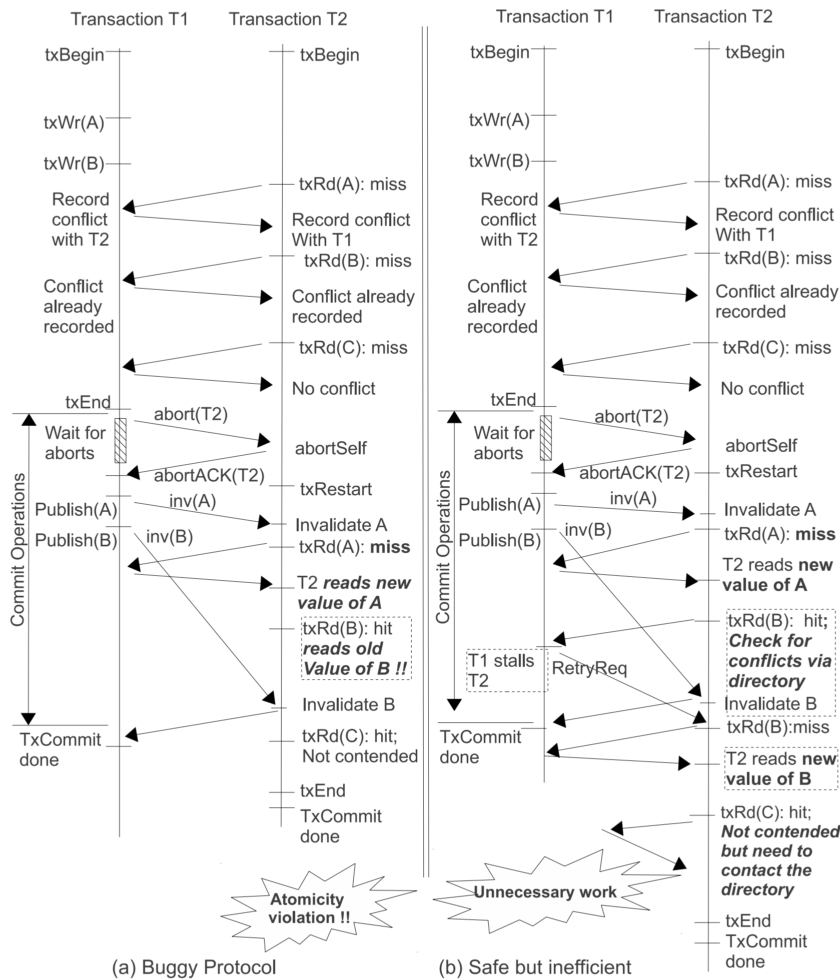


Figure 6.1: Problems with detecting conflicts early in lazy designs.

above, requires protocol support for delaying the re-executing transaction so that it is able to read the correct version of a conflicting cache line. The first option demands communication between cores and extensive protocol support. The second and third options are too conservative and represent equivalently severe penalties in common-case scenarios, limiting scalability due to excessive communication to the directory.

In this chapter we present π-TM, a novel solution that achieves scalable lazy commits by detecting conflicts early and keeping track of contended lines to perform self-invalidation at the end of the transaction. It consists of a few simple extensions to a directory-based coherence protocol and uses mild pessimism in

the uncommon case to keep common case transactional execution unencumbered. As described later in Section 6.3, our proposal works by recording conflicts at the granularity of cache-lines, simply by maintaining an additional single-bit annotation, called the π-bit, at the private cache. On an abort, only those cache-lines in a transaction's read-set that have seen a conflict during the course of its execution are invalidated along with speculatively updated lines. The transaction can now immediately restart and can use private cache data without any need to contact the directory. The committing transaction can now update shared system state in a much simpler fashion since the protocol ensures that no other private caches contain cache lines in its write-set. For safety, the committing transaction also invalidates any contended lines in its read set. We shall show later the number of lines that actually face contention in large, coarse-grained transactions is *far smaller* than the total number of lines accessed in such transactions. Moreover, the proposed method works equally well for small, fine-grained transactions. We believe that this solution overcomes a major performance bottleneck in scalable, lazy designs that make use of coherence protocols to detect and resolve conflicts.

6.2 Background

Cache coherence protocols allow HTM implementations to detect conflicts among concurrently running transactions at the granularity of cache lines. For most transactional workloads this represents a good trade-off between design cost and performance. Hence, it is no surprise that most scalable HTM design proposals choose to leverage coherence mechanisms for conflict resolution. In particular, eager conflict resolution protocols like LogTM [92, 153] fit very naturally onto a cache-coherent CMP substrate, needing only modest design extensions to support TM semantics.

Lazy conflict resolution protocols, however, require protocol extensions to permit the existence of multiple speculative copies of a cache line while retaining the ability to use coherence messages to detect conflicts. This can typically be done in two ways. The *first method* involves containment of writes within the private cache until commit. At commit time the writes are completed in shared memory causing conflicts to be detected at all concurrently running transactions that have accessed the line. The simplest way to do this is to allow only one transaction to commit at a time through the acquisition of a global commit token by the committing thread [56]. Scalable TCC [29] attempts to do better by letting

commits targeting different directory banks proceed in parallel. This improves performance to an extent, as we shall see later, but does not provide true commit parallelism seen in designs like LogTM. This is rather pessimistic resulting in marked performance degradation in scenarios when a large number of non-conflicting concurrent transactions exist and compete for commit permission.

The reason for limited parallelism at commit time is that the committing transaction has no knowledge of which other concurrently running transactions must abort to preserve atomicity. Thus, either commits must proceed one by one or complex commit protocols, like the one described by Pugsley et al. [102], must be employed to provide a degree of parallelism. In a cache coherent design this turns out to be inefficient. Information pertaining to potential conflicts is readily available from coherence messages during the lifetime of any transaction suggesting a *second method* of performing lazy conflict resolution. This information, if retained over the course of execution of a transaction, can at commit time allow true commit parallelism, since all potentially conflicting transactions that must be aborted would be known. All committers that have no races among themselves can then be sure that they can safely commit in parallel.

Designs like FlexTM [122] and EazyHTM [143] provide lazy conflict resolution by recording conflicts as they happen, using this information to enable distributed commits. FlexTM chooses to do so in software and sacrifices progress guarantees to gain greater parallelism. Performance costs associated with software intervention and software verification challenges without watertight forward progress guarantees could limit the value of this approach. EazyHTM, on the other hand, provides parallel lazy commits in hardware and ensures forward progress, but trades off common-case performance to achieve it. EazyHTM uses a special messaging protocol and hardware to indicate transactional accesses to cache lines to all potential sharers. Every first read to a cache line in a transaction requires all sharers be notified and any potential conflicts indicated before the transaction can safely proceed. A similar action is required for every first write to a cache line. It is quite clear that this action, although ensuring safety in the infrequent case described in Figure 6.1, represents a heavy burden on common-case transactional execution.

6.3 π-TM: An Adaptable Lazy HTM Design with Parallel Commits

Coherence messages carry information that can not only be used to detect which core/transaction might be conflicting but also to figure out the cache line address of the contended line. This information can be recorded using a single bit cache line annotation in each private cache, which can later be used to resolve conflicts in a far more efficient manner. To develop a framework for investigating this idea further, in this section we describe in detail the implementation of a baseline lazy HTM design that performs eager conflict detection, employing concepts developed in [143]. We then extend this design with simple mechanisms to record contention at cache line granularity and resolve conflicts safely without burdening common-case transactional execution.

6.3.1 Baseline eager detection-lazy resolution

For detecting conflicts while a transaction runs, π-TM inherits most of the conflict information flow between concurrent transactions from EazyHTM [143]. It maintains a bitmap of *racers* that tracks which transactions must be aborted on commit to preserve atomicity. This list over-approximates the set of transactions that need to be aborted, and it is sufficient to maintain correctness. False aborts can happen if a conflicting transaction on another processor aborts, and a different non-conflicting transaction starts in its place and then receives an abort request that was intended for the previously aborted transaction. A second list of *killers* is used so as to avoid these false-aborts, so that abort messages from cores not indicated as killers in the *killers-list* are ignored. Every time a transaction observes a remote access to a line that belongs to its read and/or write sets, it enables the appropriate bits in its *racers* and *killers* corresponding to the remote accessor.

All concurrent transactions that might have written a line must be notified when the line is read by a transaction. Similarly, when a line is written by a transaction all sharers must be notified that a new potential conflict might exist. The requester must also be made aware of which concurrently running transactions have conflicts with it. If the new accessor is a writer, it is added to the *killers* list of each existing transactional reader of the line. If it is a reader, each existing transactional writer adds the new reader to its *racers* list.

Unlike EazyHTM, we choose to piggy-back the information about transactional reader/writer status on usual coherence messages (TGET, TINV and various ACKs) using two single-bit flags – *txReader* for transactional read, *txWriter* for

transactional write. In our opinion, this is simpler than having a host of new messages exclusively to manage conflicts, as used in [143]. Our design thus leverages the network-on-chip communication fabric for all information exchange, rather than dedicating a special purpose core-to-core interconnect used only for conflict management [143]. In our opinion, most TM use-cases do not justify investment of hardware resources into ad-hoc communication mechanisms solely dedicated to conflict detection. Moreover, because π-TM leverages the standard messages of the coherence protocol to carry out conflict detection, such task involves the usual 3-way communication: i) TGET (with txWriter/txReader) $\rightarrow$ directory; ii) TINV $\rightarrow$ sharers; and iii) acknowledgments from sharers $\rightarrow$ requestor. This is different from EazyHTM, which needs 4-way communication for the same task: i) TxMark $\rightarrow$ directory; ii) TxAccess $\rightarrow$ sharers; iii) TxReader/TxWriter/NonTxnal status from sharers $\rightarrow$ requestor; and iv) TxReader/TxWriter acknowledgments from requestor $\rightarrow$ sharers.

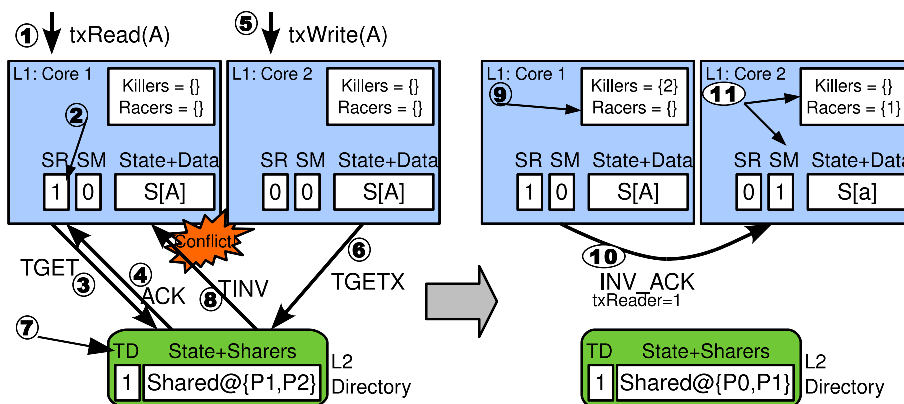


Figure 6.2: Key protocol action: Baseline eager conflict detection in lazy HTMs.

Figure 6.2 depicts how the baseline protocol operates. Transaction *T1* on core 1 wishes to read cache line *A* (step 1) which is in "S" state in its private cache. Core 1 sets the SR flag for the line and sends a *TGET(A)*, with *txReader* high, request to the directory (steps 2,3). The directory finds the *TD-bit* not set and responds with an *ACK* allowing *T1* to proceed (step 4). Subsequently, transaction *T2* on core 2 attempts to write line A and sends a *TGET(A)* request, with *txWriter* set, to the directory (steps 5,6). The directory sets the TD-bit for the line and sends a TINV(A), with *txWriter* set, to core 1 (steps 7,8). Core 1 notices the conflict, adds core 2 to the killer list and sends an *INV_ACK* acknowledgment to core 2 with the *txReader* flag set (steps 9,10). Core 2, on receiving the acknowledgment

knows that *T1* is a racer and adds core 1 to its racers list, completing the conflict detection protocol. Before it publishes any of its updates on commit, *T2* will abort *T1* by sending a *TABORT* message to core 1. The rare case of racing commits is handled by using a unique core identifier as priority.

Next we describe our proposal to extend this design with simple mechanisms to record contention at cache line granularity and resolve conflicts safely without burdening common-case transactional execution.

6.3.2 π-TM: Pessimistic invalidation of contended lines

π-TM builds on the baseline eager detection-lazy resolution HTM system described earlier, by adding a new *π-bit* to existing private cache lines. When a transaction that has only read a line finds the line contended, it sets the corresponding π-bit in its private cache. Lines with the π-bit set are invalidated when the transaction commits or aborts. We term this as *pessimistic invalidation* (hence the name *π-TM*) since all *possibly conflicting* lines are invalidated rather than invalidating only those lines that are being written by the committing transaction. The rationale behind this approach is that the number of such lines (speculatively read and contended) is typically small and the cost of this minor pessimism is far outweighed by performance advantages provided by unencumbered common-case transactional execution. Invalidation of such lines at commit is required to preserve atomicity. This prevents a subsequent transaction from reading an old value from the contended line and newly committed value from a different line, when both lines belong to the write-set of another committing transaction.

Invalidation of lines with π-bit set guarantees that all valid lines in the private cache can be safely read by transactional code without contacting the directory. Since lines that have not been accessed transactionally are invalidated by incoming *TINV* messages, no unsafe accesses can be handled locally. Lines in the read set could also be invalidated at the point when contention is noticed, as it is simple to track the read set with a signature [27]. This invalidation is inefficient as such lines could be accessed again in the transaction. More importantly, it just tackles one part of the problem by ensuring safety only in the case where a line is speculatively read before contention is noticed.

Figure 6.3 depicts key protocol actions when pessimistic invalidation is used. Core 1 running transaction *T1* reads a line, *A*, in its cache, setting the corresponding *SR-bit* (steps 1,2). Note that no communication with the directory occurs. The line is subsequently written by core 2 running transaction *T2* which attempts to acquire exclusive permission over the line by sending a *TGET(A)*, with txWriter

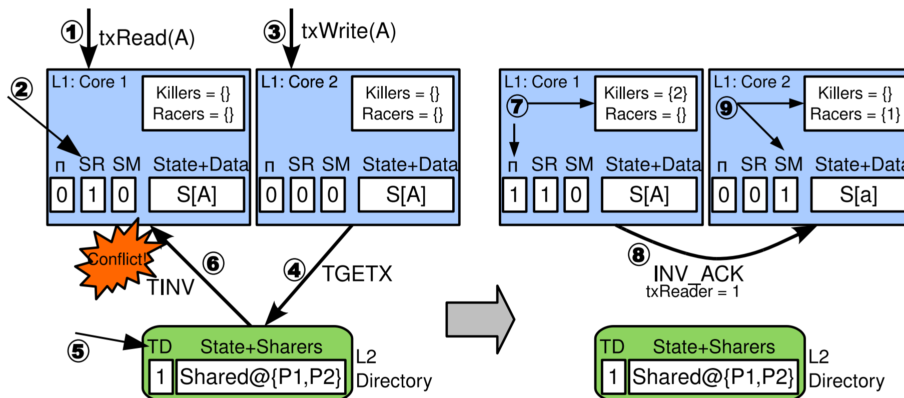


Figure 6.3: Key protocol actions: Conflict detection in π-TM.

set, to the directory (steps 3,4). The directory now sets the *TD-bit* and sends a *TINV(A)*, with *txWriter* set, to core 1 (steps 5,6). Core 1 recognizes the conflict and adds core 2 as a killer setting the π-bit for the line (step 7). It then sends an ACK to core 1 with *txReader* flag set, allowing core 2 to add core 1 to its racers list (steps 8,9). This completes the π-TM conflict detection protocol in this example.

Now, two scenarios (shown in Figure 6.4) can arise based on whether or not *T1* commits before *T2* aborts it. If *T1* commits before *T2* can abort it, the line with the π-bit set is invalidated and the thread can continue further execution. If *T2* commits before *T1*, core 2 sends a *TABORT* message to core 1. Since core 2 marked as a killer, *T1* aborts and invalidates the contended line. In any scenario, the possibility that core 1 later accesses line *A*, a contended line, without communicating with the directory (thereby preventing conflict detection) is eliminated. A key point to be noted here is that a speculatively written line (with SM-bit set) may be read by the transaction without any communication to potential killers. Thus, lines in the write-set are always reported as *both read and written* when conflict detection acknowledgments are sent.

6.3.3 Adaptable π-TM

It is difficult to come across a one-size-fits-all HTM design. This is true both for eager resolution designs that perform poorly when contention is high and for lazy resolution designs that suffer when contention is low. Early conflict detection tries to bring together the best of both worlds, but in doing so acquires a weakness not found in other designs. This situation arises when small to moderate sized

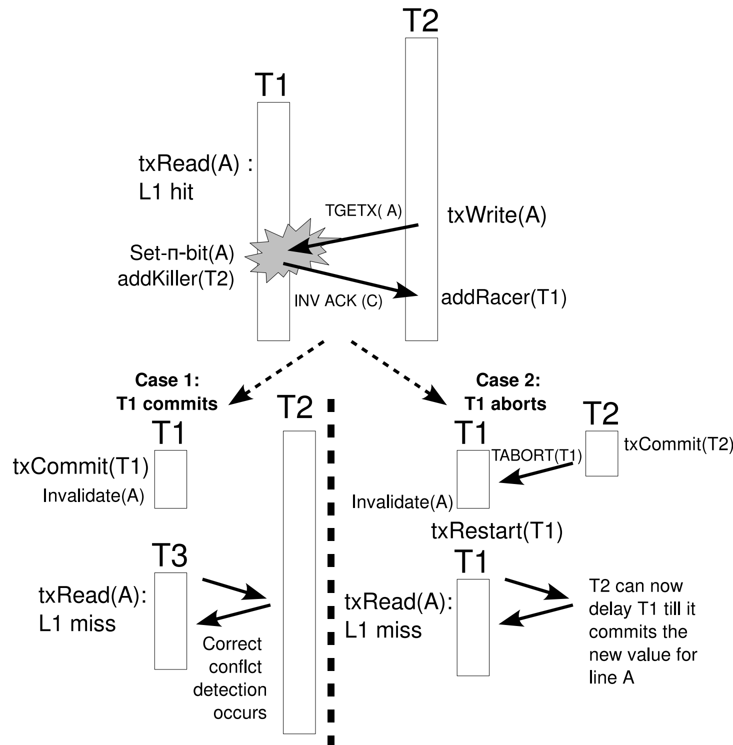


Figure 6.4: High level behavior of π-TM.

transactions see high contention on late accesses that occur shortly before commit. In such cases the requirement to detect all conflicts before the transaction can commit brings the latency for doing so in the critical path. We also note that lazy conflict detection works well in this case since it can combine conflict detection with write-set publication. This suggests a way to build in adaptability into the system by detecting the condition and switching policy to achieve higher performance. We choose the simplest lazy conflict detection protocol – the global commit token (*GCT*) approach laid out in [22] – to evaluate the idea.

Each atomic block in a thread can operate in one of two modes – π-*mode* or *GCT-mode* – independently of the rest. A simple algorithm, shown in Algorithm 1, determines how transactional execution switches between operation modes. Since both modes confine speculative updates within private caches the two protocols can interact safely. A running GCT-transaction aborts when it receives a *TABORT* message from a π-transaction. A committing GCT-transaction aborts a validating π-transaction by responding to the *TABORT* message with *NACK*. Moreover, a

GCT-transaction does not release its write-set until the entire commit is done, thereby preventing unsafe interleavings from occurring. When threads operate in mixed-mode, all π-transactions have accurate information regarding racers (transactions that must be aborted before commit). However, a π-transaction may not know all its killers, since killer GCT-transactions do not indicate conflicts over their write-set. This is not a safety issue since such GCT-transactions will cause any racing π-transactions to be aborted by non-transactional invalidations.

The algorithm for switching relies upon two pairs of thresholds to commute between the two modes of opertation. In π-mode, *stallCtr* represents the time between arrival at the end of a transaction and an abort before the initiation of validation operations. Consistently high stall counts and subsequent aborts indicate high contention and late conflicts – conditions which degrade early conflict detection performance. In GCT-mode, *stallCtr* represents the time spent in arbitrating for a global commit token before a successful commit. Consistently high arbitration stalls preceding successful commits indicate low contention and demand for commit bandwidth, i.e. conditions where π-mode works best. Since latencies depend strongly on network topology and available on-chip bandwidth, the two pairs of thresholds can be determined reasonably well at design time and need not be learned online.

Figure 6.5 shows structures needed to support this adaptability. These requirements are not very large. Transaction identifiers (e.g. instruction pointers or sequence numbers assigned to "*start-transaction*" instructions) are mapped to one of several predictors (we use 16 predictors). Hardware overheads are minor, as can be inferred from the figure. It should be noted that the decision to switch modes is taken entirely based on information locally available at each core.

Algorithm 3 Mode switch prediction logic.

```
if π-abort OR GCT-commit then
    if stallCtr > stallThold(curMode[xid]) then
        modeSwitchCtr[xid] ← modeSwitchCtr[xid] + 1
        if modeSwitchCtr[xid] > modeSwitchThold(curMode[xid]) then
            flipCurMode(xid)
            modeSwitchCtr[xid] ← 0
        end if
    else
        modeSwitchCtr[xid] ← max(0, modeSwitchCtr[xid] − 1)
    end if
else
    if π-commit OR GCT-abort then
        modeSwitchCtr[xid] ← 0
    end if
end if
```

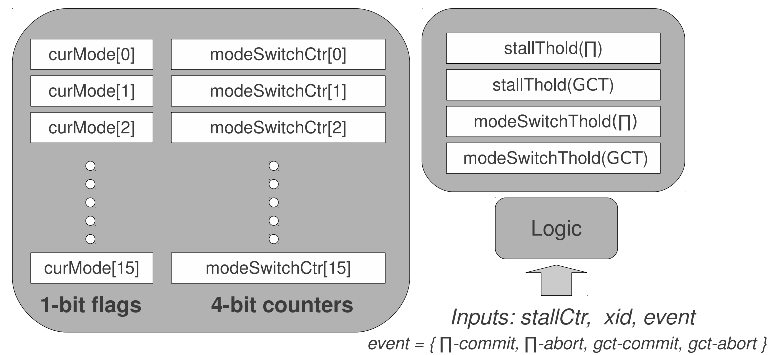


Figure 6.5: Hardware structures that enable mode switches.

6.3.4 Protocol support for π-TM

Figure 6.6 summarizes new protocol transitions required at the private cache controller. The diagram does not show the transitions for transactional loads since they require no special treatment and are satisfied just like regular loads. The grey transitions are part of the standard MESI coherence protocol. Grey circles represent transient states for in-flight operations. No new coherence states are needed. Those transitions depicted using dashed black lines allow pessimistic invalidation of cache lines using the π bit. In the figure, those messages that have at least one transactional status flag set –either TxReader or both TxReader and TxWriter– are denoted with the T prefix.

Forwarded exclusive (GETX) requests to E/M lines, which are usually satisfied by the invalidation of the line plus a cache-to-cache transfer of exclusive data from the current owner to the requestor, behave in a slightly different fashion in the transactional (TGETX) case, if the line has been accessed inside a transaction. TINV or TGETX messages to lines that have not been accessed in a transaction (SR bit not set) cause those to be invalidated, as usual. However, if the targeted E/M line has the speculatively read (SR) bit set, the line is not immediately invalidated by the current owner, but rather downgraded to S state, and marked with the π bit. The new transactional writer then obtains a shared copy of the data that it can use for lazy versioning in its private cache. Similarly, a transactional reader/writer responds to transactional invalidation (TINV) requests with an acknowledgement message –reporting its own status of transactional accessor– but it does not invalidate the line in this moment, but simply sets the π bit. When the line is shared amongst several transactional accessors, a transactional read request TGETS obtains data from L2 but still needs to check the status of the

current sharers, so the directory also forwards TINVs to the accessors as in a TGETX request. The new transactional accessor always sets the π bit for the requested line if any of the responses (i.e. at least one TINV acknowledgement, or the TDATA message) has the TxWriter bit set.

Transitions in bold black lines in Figure 6.6 indicate the support for lazy versioning in the standard MESI protocol, so that speculative writes can be buffered in private-copy cache lines that appear as shared copies to the coherence protocol. Unlike their non-transactional counterparts, transactional store misses do not fetch exclusive data but instead can only complete if a shared copy (S) of the line is present. Once the speculatively modified (SM) bit is set after the first transactional write, the remaining write accesses are cache hits. If the targeted line is in E or M state, a downgrade to S state (not shown in Figure 6.6) with a possible write-back of dirty data is necessitated in order to write the last committed value to the shared level, thus preserving memory in a consistent state and retaining the ability to discard speculative updates on abort via silent

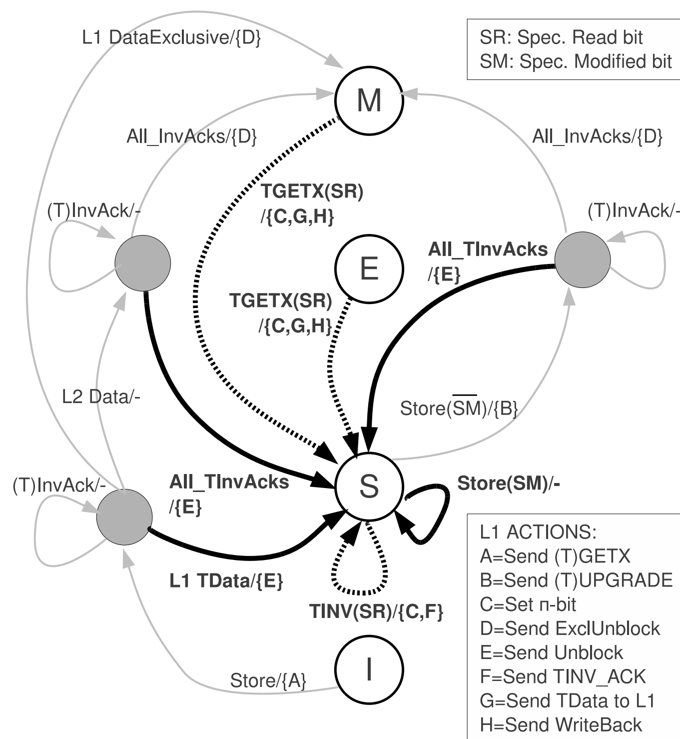


Figure 6.6: Supporting transitions at L1.

invalidations of shared SM lines. If the line is already present in S state but the SM bit is not set (first transactional store), the store cannot be immediately performed in cache, because it still needs to check with the remaining sharers for correct conflict detection. This is handled in a similar fashion to non-transactional writes to S lines: an UPGRADE request is sent to the directory, which forwards invalidation messages to all the sharers. These, in turn, respond with invalidation acknowledgments. The main difference is that, in the transactional case, the UPGRADE and INV requests have the TxWriter status bit set, as well as the INV_ACK responses, leveraged by the sharers to report their transactional status via the TxReader/TxWriter flags. In the case of a new transactional writer, an UNBLOCK message is sent to the directory to indicate that the in-flight memory operation has terminated, and the requestor has become a sharer. In the non-transactional case, however, the requestor always becomes the exclusive owner and the directory only expects EXCLUSIVE_UNBLOCK messages.

In order to guarantee strong atomicity [17], a π-TM transaction aborts when it observes a non-transactional INV or forwarded GETX request that targets a SR/SM cache line. Therefore, the protocol does not need any special behaviour to correctly handle conflict detection between GCT- and π-mode transactions: commit-time invalidations generated by GCT transactions appear as non transactional and thus cause the abort of any other transactional accesor of the committed data.

The coherence protocol also includes support for handling negative acknowledgements (NACKs), which can be sent by transactions in certain situations. For instance, when a transaction has entered the validation phase (has sent abort requests to its racers, and it is collecting the responses), it responds with a NACK message to any TINV or TGETX request that targets its read/write set lines. This is because once validation has begun, the transaction does not accept new racers nor killers. Similarly, requests to write set lines are nacked when a transaction has entered the committing phase, since π-TM does not implement the *critical-cacheline-first* optimization proposed in [143].

Handling of accesses at the directory is summarized in Figure 6.7. The directory is capable of resolving the source for forwarded data in case of transactional reads. This could be the L2 if the line is not modified at some core or the private cache containing the consistent copy. At the Level 2 cache, a TD bit (*transactionally dirty*) in maintained as proposed in [143], which when set indicates the presence of one or more modified speculative versions of the line. It is used to avoid unnecessary communication with sharers when no potential conflicts exist (i.e. reader-reader scenarios). Obviously, the value of the TD bit is only meaningful

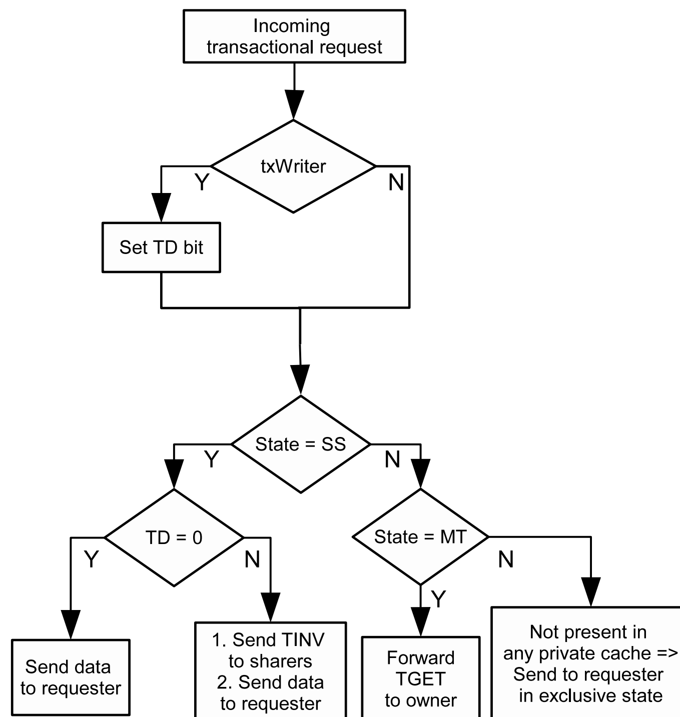


Figure 6.7: Access handling at directory.

when the directory state for the line is SS (shared by zero or more caches), and it is automatically reset to 0 when the state changes to MT (exclusively held by one cache). The bit is set by TGET requests that have the TxWriter flag set, before the message is considered by the coherence controller. The changes in the directory controller to support π-TM are minimal, since no new states are needed. The directory always copies the TxReader/TxWriter bits of incoming requests to all outgoing forwarded requests, so that these status flags are piggy-backed in regular coherence requests for correct conflict detection. Transactional requests (TGET) are treated just like common GETX requests when the TD bit is set (i.e. invalidations are sent to the sharers). When TD not set, the directory directly provides the data to the requestor. The only relevant changes to the directory controller are not specific to π-TM, but common to any directory-based MESI protocol that supports lazy versioning of data in private caches: New transitions are added so that a TGETX request can end up leaving the directory in shared (SS) state, whereas a non-transactional GETX request always brings the line to exclusive state (MT).

6.4 Evaluation

The performance of several HTM design points is now evaluated and compared. Four lazy HTM design points with early conflict detection are considered:

1. A baseline system with eager detection and lazy resolution, similar to [143], described in Section 6.3.1.

2. π-TM, which builds on top of the previous baseline system, and augments L1 caches with π bits so as to perform pessimistic self-invalidation of contended lines. This system was presented in Section 6.3.2.

3. Adaptable π-TM, as described in Section 6.3.3, which combines eager (π-style) and lazy (GCT-style) conflict detection policies in the same system, and uses a predictor to select the appropriate operation mode on a per-transaction basis.

4. Ideal π-TM. This is an idealized implementation of the π-TM system, in which transactions magically check for conflicts with remote transactions without any message exchange. On each memory access, a transaction accesses the read and write sets of every other running transaction to determine if a conflict exists, and accordingly updates both local and remote lists of racers and killers. This system attempts to estimate an upper performance bound for lazy HTM systems with eager conflict detection, by reducing the latency of the conflict detection mechanism to zero.

Table 6.1: HTM configurations evaluated in Chapter 6.

Configuration	Description
Base	Baseline Early Conflict Detection [143]
PI-TM	Basic π-TM scheme
aPI-TM	Adaptable; can choose between π-mode or GCT mode
idealPI-TM	Idealized π-TM scheme
LL-GCT	Lazy conflict resolution using a global commit token [22]
LL-STCC	Scalable TCC [29]
EE	Eager conflict resolution based on LogTM [153]

These eager-lazy systems are compared against the three design points presented in Chapter 3: LogTM-SE [153], the global commit token (GCT) system [22], and a detailed implementation of the Scalable TCC (STCC) design [29]. Similar

to the GCT and STCC systems, the three lazy HTMs capable of early detection rely on a dedicated write buffer of unlimited capacity for buffering of speculative updates. Exclusive permissions over the associated written cache lines are acquired on commit, before publishing updates to shared memory. All the design points considered in this chapter are summarized in Table 6.1.

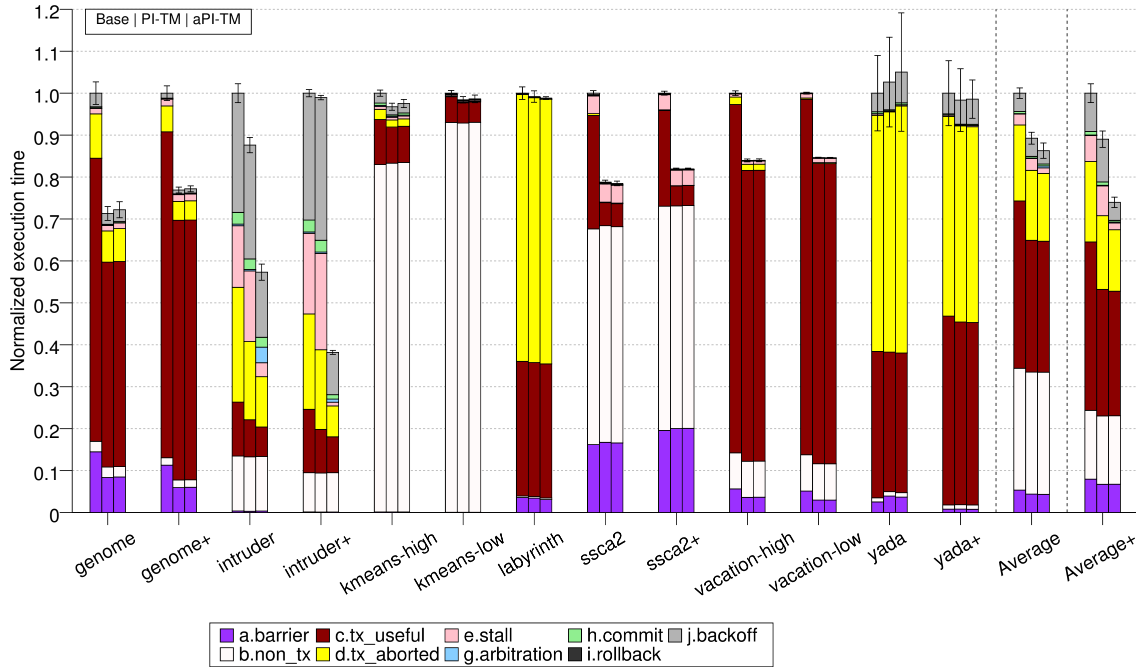


Figure 6.8: A comparison of early conflict detection schemes.

6.4.1 Early Conflict Detection Performance

Figure 6.8 shows the relative performance of the three early conflict detection designs – the baseline design, π-TM and adaptable π-TM. We can see that π-TM is substantially (more than 10%) better than the baseline. The adaptable variant (*aPI-TM*) performs the best, about 16% better than Base and about 25% better for long running workloads.

In Figures 6.9 and 6.10 we see that π-TM achieves major reductions in network traffic, both in terms of flit count (about 15% less) and number of messages released into the network by the workload (about 20% less). In the CMP fabric

used for simulations large on-chip communication bandwidth is available to each in-order processor. In systems with limited bandwidth, it can be expected that the lower demands of π-TM would translate into improved performance.

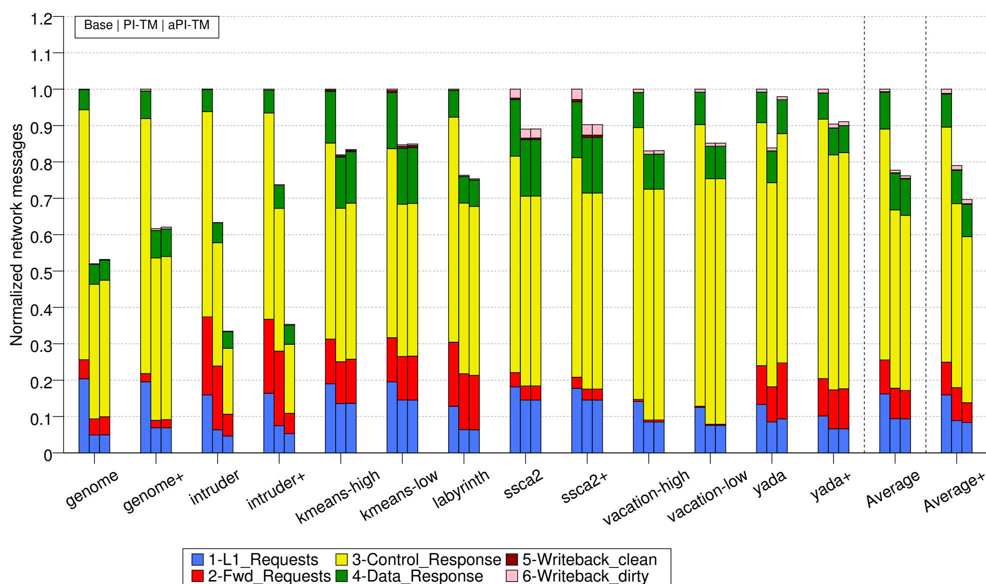


Figure 6.9: Network messages.

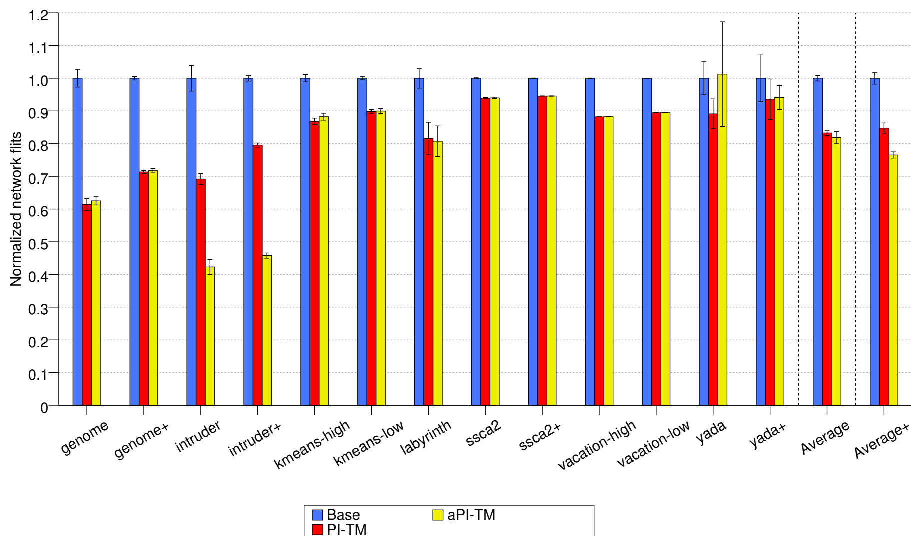


Figure 6.10: Network flits.

Figure 6.11 compares the miss-rates at the private caches over the duration of execution of each benchmark. As we can observe, π-TM achieves important reductions in the miss rate compared to Base, clearly showing how our approach of pessimistic invalidation allows transactions to use privately cached data without the need to contact the directory on every fresh access.

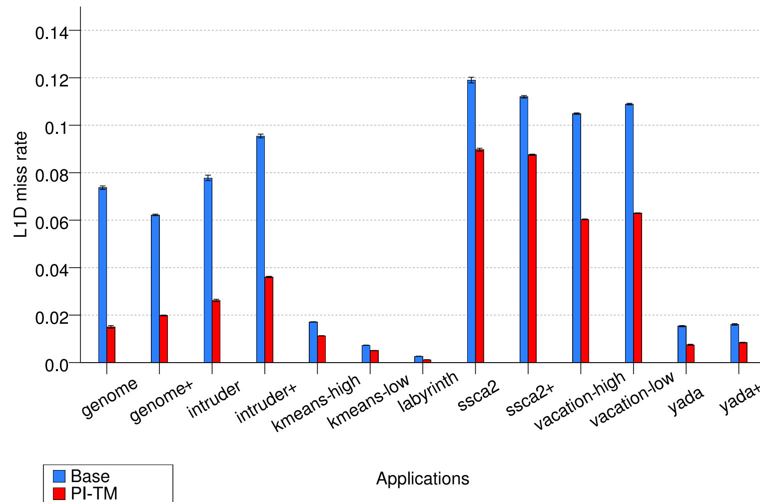


Figure 6.11: Miss rates.

Figure 6.12 shows the distribution of transaction handling modes (π or gct) in different workloads for the adaptable configuration (aPI-TM). In this plot we see how the adaptable version of our design switches to a different execution mode for workloads with small, highly contended transactions. This mode change is responsible for the significant performance improvement (up to 60% reduction in execution time) achieved by aPI-TM in such kind of applications, in comparison to our non-adaptable scheme, seen in Figure 6.8.

Table 6.2 presents numbers regarding contention in various STAMP workloads. The number of lines that are invalidated pessimistically constitutes a very small fraction of the read-set size for most applications, corroborating choices made in the π-TM design. When transactions commit almost the entire read-set is non-contended. When transactions abort the number of invalidations typically amounts to less than 10% of the read-set size at commit. Only in the case of yada about 48% of the read-set is invalidated on aborts. This still represents savings in access time over the remaining 52% of the read-set.

Table 6.2: Contention statistics.

	π-lines @abort	π-lines @commit	Rset (#lines) @commit	π-lines (% Rset) @abort	Killers @commit	Killers @abort	Racers @commit	Racers @abort
genome+	1.91	0.01	22.8	8%	0.01	1.91	0.07	1.17
genome	2.70	0.04	25.7	11%	0.03	2.87	0.18	2.23
intruder+	0.32	0.03	11.2	3%	0.05	3.11	1.53	4.01
intruder	0.54	0.07	9.1	6%	0.09	3.18	1.39	3.98
kmeans-high	0.63	0	5.4	12%	0	1.12	0.22	0.98
kmeans-low	0.65	0	5.4	12%	0	0.94	0.04	0.79
labyrinth	1.31	0.08	71.1	2%	0.11	1.33	0.63	1.02
ssca2+	0.07	0	3	2%	0	1.66	0	1.39
ssca2	0.11	0	3	4%	0	1.81	0.01	1.61
vacation-high	0.65	0.01	66.8	1%	0.01	0.97	0.03	0.03
vacation-low	0.60	0	53.9	1%	0	0.99	0.01	0.03
yada+	16.27	0.23	33.8	48%	0.12	4.46	0.37	4.4
yada	13.31	0.48	27.5	48%	0.25	3.89	0.63	3.7

We now analyze individual workloads in detail highlighting aspects that influence performance.

Genome, ssca2 and vacation. Communication overheads induced by baseline early conflict detection mechanisms result in considerable degradation of private cache performance as can be seen in Figure 6.11. This translates into slower execution of transactions, a fact which is clearly highlighted by the significantly (almost 40% for genome) higher useful transactional execution time (*tx_useful*) component in the execution time breakdown shown in Figure 6.8. The performance between the basic π-TM design and its adaptable variant is little. This is because contention is relatively low for most of the application.

Table 6.3: Mode-switch threshold values.

Adaptable π-TM Thresholds	
$\pi \rightarrow GCT$	2 consecutive 512-cycle stalls
$GCT \rightarrow \pi$	4 consecutive 256-cycle stalls

Intruder. Intruder exhibits high levels of contention. Small transactions with late updates to shared data compete. Though the basic π-TM design shows some improvement over the baseline on account of better cache performance, it suffers from the pathology described in Section 6.3.3. The requirement to complete conflict detection before commit operations can be initiated results in substantial increases in aborted execution and backoff time. The adaptable variant is able

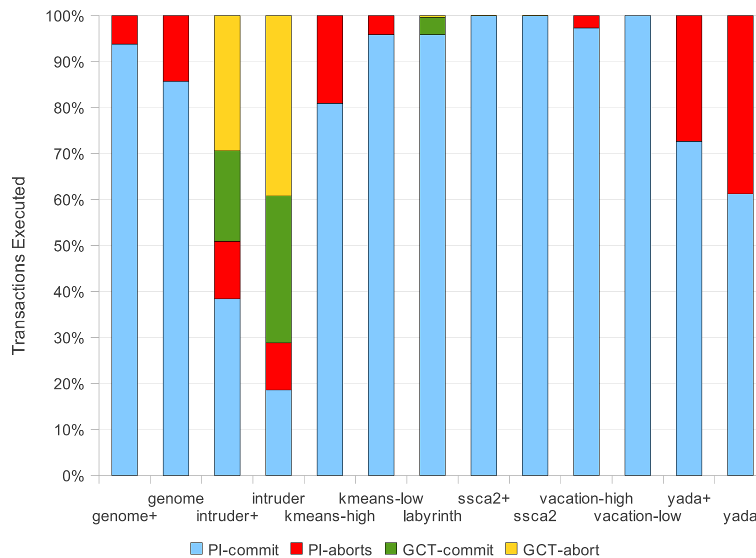


Figure 6.12: Adaptable π-TM: Mode distribution for transactions

to switch policies quickly for offending transactions and avoids performance degradation, showing a performance improvement of 40-60%. Figure 6.12 shows how the switch to GCT-mode occurs with about 50-70% of the transactions committing in the GCT-mode.

Kmeans. This application spends only a small fraction of its execution time (<10%) executing transactions. Moreover, these transactions are tiny. Therefore, we see no major deviations in performance although the two π-commit variants perform approximately 2% better than the baseline.

Labyrinth. This application has very large transactions where write sets run into hundreds of cache lines. Most such writes are also first accesses to lines, targeting the local grid, and hence, cache performance is very good, even in the baseline scenario. No major performance deviations are seen across all four configurations.

Yada. This mesh refinement algorithm exhibits moderate to high contention spread over a relatively large dataset. Although cache performance improves (by 50%, see Figure 6.11), this does not directly translate into improved performance when using the small input set. Execution times vary by 30% across different runs. When using the larger dataset, we see that the π-commit variants perform marginally better than the baseline (by approximately 2%).

6.4.2 Comparison with other designs

Figure 6.13 shows the relative performance of the adaptable π-TM design and other pertinent design points. These include LogTM (shown as EE), a lazy global commit token approach (LL-GCT) and the scalable TCC design (LL-STCC). The execution times have been normalized to those achieved by the baseline configuration.

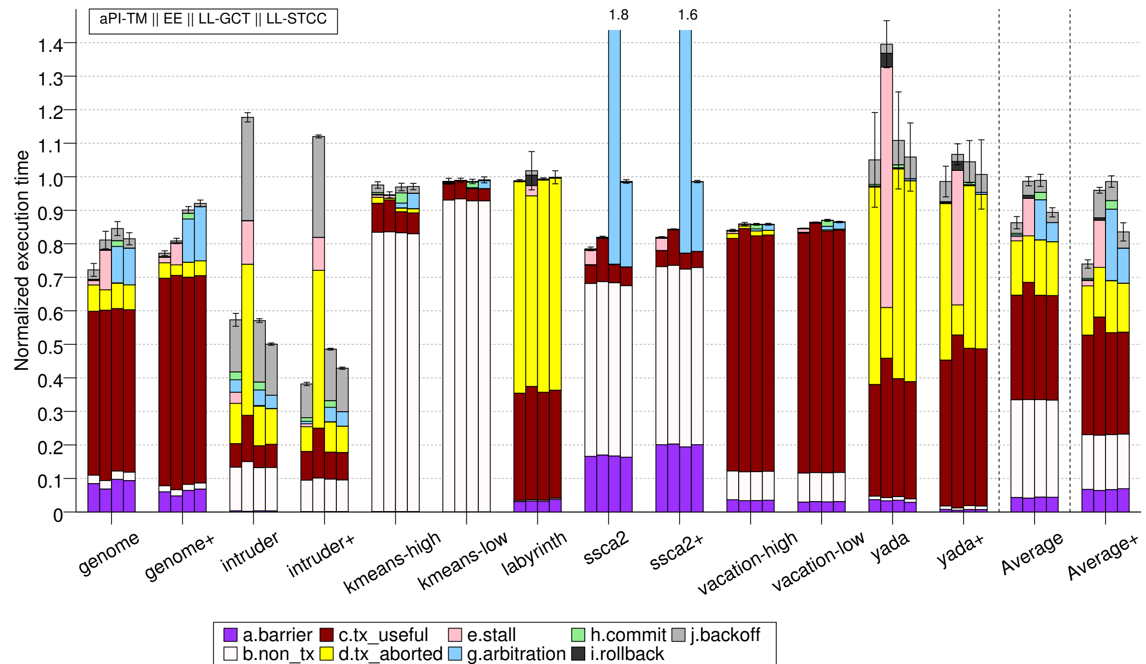


Figure 6.13: π-TM: Comparison with other designs.

All four designs perform better than the early conflict detection baseline design. This is primarily because of the poor private cache performance of the baseline. Furthermore, we note that adaptable π-TM achieves the best performance overall. This design performs consistently well for all workloads, indicating robustness and the ability to avoid pathological scenarios. The difference between the lazy conflict detection designs and the adaptable π-TM can be seen in the reduced arbitration component. This clearly shows how parallel commits can be effective in improving overall system performance.

We now present some important workload-specific observations gathered from the data.

Genome. This workload is interesting because π-TM performs significantly better than other designs. As we have explained earlier, this is primarily due to benefits of pessimistic invalidation under moderate contention and large transactions. As a result of contention for commit permissions, LL-GCT shows a large *arbitration* component in Figure 6.13 which is absent in the π-TM case. LL-STCC allows a degree of commit parallelism, but commit-time communication to the directory banks results in longer commit durations. This clearly highlights the efficacy of parallel commits in π-TM. The EE design detects conflicts on memory stores as they occur before proceeding with further execution. π-TM, on the other hand, is able to run-ahead past potentially conflicting stores overlapping the line-fetch with useful work.

Intruder. In this high contention application lazy designs come out on top. The ability to commit in parallel does no bring much advantage here as most transactions conflict. Early conflict detection mechanisms suffer from the pathology mentioned in Section 6.3.3. However, the adaptable π-TM variant is able to sidestep it by quickly switching pathological *queue pop* transactions to GCT-mode. The design is about 5% faster than LL-STCC in long running simulations (intruder+).

Ssca2. This application has very low contention but the very large number of tiny transactions stresses commit bandwidth in each design. Both variants of π-TM perform the best here, marginally (3-5%) better than the EE design. The reason, as seen in the case of genome, is the lack of latency hiding capabilities for stores in the EE design. Limited commit bandwidth when using a global commit token results in pathological behavior. Scalable TCC mitigates this effect but is around 25% slower than LogTM or π-TM due to the commit-time communication with the directory.

Yada. Yada exhibits moderate contention spread over a relatively large dataset. The adaptable π-TM design performs as well as or marginally better than LL-STCC, taking full advantange of both commit parallelism (Figure 6.12 shows no GCT-mode transactions) and lazy execution. The EE design does not perform well and turns out to be the slowest (40% slower for yada and about 10% slower for yada+ when compared to the adaptable π-TM design). As we have seen before, the EE design suffers due to exposed latency for stores.

Kmeans, labyrinth, vacation. These applications show no major deviation in performance as transactions either contribute little to the overall execution cycles (e.g. kmeans) or the contention is low (e.g. vacation) or the cache performance is good (labyrinth).

6.5 Concluding Remarks

The design space of scalable HTM systems has been investigated heavily in recent research. Yet, the quest for true optimism has been thwarted by the inherent *pessimisms* in different TM policies. Traditional lazy designs are pessimistic when they attempt to commit. Eager designs are pessimistic when they detect conflicts. Information regarding conflicts is readily available in the coherence traffic generated during transaction execution, and it can be used to detect conflicts prior to commit. If properly used, this information can enable a simple design which supports parallel commits of non-conflicting transactions, while maintaining the optimistic behaviour that allows lazy transactions to run-ahead, past conflicts. Unfortunately, early work in this direction has reverted to pessimism in another form and in a rather more critical scenario – namely, every fresh transactional access to a cache line might be contended. As our evaluation has demonstrated this degrades performance very substantially. The π-TM design described in this chapter drops this pessimism in most scenarios, where it turns out to be prudent to do so, resorting to a far milder form of pessimism that leaves the common-case unburdened. The chapter presents strong evidence which supports this claim. We also underline the importance of incorporating adaptability in design to changing workload characteristics as a way to not only achieve higher performance but also have a more robust system.

Chapter 7

Implications of Store Buffering on the Performance of HTM Systems

7.1 Introduction

HTM systems must ensure that speculative updates made in a transaction are not visible to the rest of the system until it commits. The two approaches –eager or lazy– to satisfy this property are reflected in the policies that control the version and conflict management mechanisms, and this choice largely shapes the design. On the one hand, lazy HTM systems often rely upon thread-private structures like private caches to contain transactional writes, impeding speculatively modified lines from being written back to shared levels of the cache hierarchy. The possible future values are thus confined in local structures and cannot be observed outside the scope of the transaction, while the current, consistent values stay accessible to others in shared memory. On the other hand, eager HTM designs make their speculative updates directly to shared memory ("in-place") and then rely on the coherence protocol to ensure their isolation. Prior to the speculative update, a copy of the cache line is added to a private log so that memory can be restored to a consistent state if data-races cause the abort of the transaction. The choice of version management policy largely determines that of conflict resolution. Eager versioning of writes necessitates eager resolution of conflicts, as races must be detected and resolved on individual memory references, particularly before an in-place shared memory update is attempted. Lazy versioning, however, allows

conflict resolution to be either eagerly performed or deferred until a transaction tries to commit.

Regardless of the choice of version management policy, modelling the effects of store buffers is particularly important in HTM research. Modern processors implement a variety of techniques to hide or tolerate the latency of memory accesses, and structures such as store buffers are typically employed to improve performance by allowing the processor to continue executing instructions without having to wait until a write is complete. Data forwarding mechanisms are implemented so that a load instruction can obtain its data from a store instruction located in the store buffer, this is, before the written data is presented to the memory hierarchy. Because store buffers lay at the interface between processor and memory subsystem, they naturally appear as a possible candidate for buffering of speculative updates when it comes to supporting TM in silicon. After all, the role of store buffers in out-of-order microarchitectures is to contain values until they are no longer speculative, keeping them away from memory until they are written to L1 cache on retirement of the store instructions.

Considering the presence of store buffers is indispensable when designing and evaluating HTM systems that opt for eager policies of conflict resolution and version management. Typically, eager HTMs detect conflicts when a processor write is received by the cache hierarchy. The core waits for the result and takes corrective action if a conflict is signalled. With store-buffering, the processor can continue execution past a potentially conflicting update, a feature found typically in lazy HTM designs. In fact, as private structures to the processor, using the store buffers to confine transactionally modified data introduces a degree of laziness in the system. Figure 7.1 depicts the store-buffering model considered in this chapter. In spite of the significant impact of these latency-hiding effects, most of the HTM literature does not quantify how store buffers affect the overall peformance of eager systems [80, 81, 92, 93, 100, 137–139, 153]. While the influence of policy is important, the implications of store-buffering need to be considered when evaluating the performance of eager HTM designs. In this context, this chapter presents some insights related to the interplay between buffering mechanisms, system policies and workload characteristics.

For lazy HTMs, coherent storing of speculative updates in private L1 caches leads to inefficiencies due to cache pollution. Transactional workloads that exhibit relatively high contention suffer frequent aborts that cause repeated invalidations of speculatively modified lines in the private cache. This situation is aggravated in coarse grained transactions, precisely the style of synchronization expected in TM applications developed under the principles of programming ease claimed by the

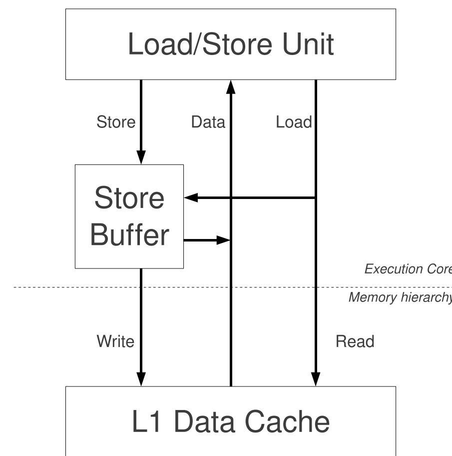


Figure 7.1: Store buffer model from this chapter.

TM paradigm. In such transactions with larger footprints, only a small fraction of the write set exhibits contention, in general. Although a large portion of the updated lines may not be shared or concurrently accessed by several threads, the entire write set gets discarded by gang-invalidating all the speculatively modified lines in cache. Consequently, a restarted transaction encounters costly private cache misses for every first access to such self-invalidated lines. Buffering of transactional updates before they reach the private cache can largely eliminate these misses, preserving the privately cached copies in their consistent state. In this way, only lines that are actually contended are invalidated upon the reception of coherence messages initiated by a committing transaction.

Many lazy HTM designs also employ a *write-back commit* scheme that defers the write-back of committed data to shared levels until the line is requested by another cache, or a new transactional store targets the dirty line. This is required for the shared levels of the cache hierarchy to be updated with the last committed value for consistency, as well as to resume the discovery of data races with new accessors through standard coherence invalidations. In low-contended workloads with high probability of commit, such coherence downgrade and subsequent upgrade actions are redundant and inherently pessimistic, as they penalize the common, non-contended case in order to correctly handle the unlikely case of abort. This penalty can be circumvented by buffering transactional updates emitted by the processor before they reach the coherent private cache, much like out-of-order processors only release values to memory on retirement of store instructions. Using store buffers, speculative updates would be released to the

coherent cache hierarchy only when a transaction attempts to commit, or when the buffer overflows. If conveniently sized, such structure can contain most transactional updates in the common case, significantly reducing redundant protocol actions. Moreover, the idea of using store buffers for containing transactional updates can also be extended to multilevel private cache hierarchies, in which first-level caches do not need to maintain coherence on transactionally updated lines.

In this chapter we demonstrate that in both eager and lazy conflict resolution schemes, store-buffering can achieve noticeable reductions in transactional execution times due to the interplay of conditions described above. When suitably modified for containment of transactional updates, store buffers can provide significant reduction in HTM protocol overheads. This, in turn, results in a contraction of the window of contention for concurrent transactions, possibly amplifying performance gains even further. While the utility and ubiquity of store buffers in standard microprocessors is well recognized, their use and implications in the context of HTM have not been studied in depth in prior work. This chapter shows how store buffers impact transactional behaviour of a diverse set of workloads. We quantify the performance gains and reductions in redundant work and stall times achieved by store-buffering for several pertinent design points. Our study of both eager and lazy HTM flavours notes remarkable convergence of performance of different TM design policies when such structures are introduced and used well to support the common case. Eager designs can achieve good performance even under high contention scenarios. The performance of lazy designs is also improved due to mitigation of unnecessary actions at transaction commit or abort. With well-balanced store buffers of modest sizes, eager systems are as good as or better than lazy designs, in contrast to previous studies that show how lazy systems are able to utilize concurrency more efficiently and hence perform better. Additionally, this chapter underlines the importance of modeling standard processor structures accurately after applying straightforward optimizations in order to support transactional memory more effectively. Not doing so can result in measurements that are substantially different from what might be seen in real-world implementations and, more importantly, can lead to erroneous biases in favor of or against certain design options.

7.2 Background

Store buffers have been implemented in commercial processors for many years now. To cite only a few cases, the Alpha 21264 microprocessor had a 32-entry buffer that held stores until their data was sent to the L1 cache on retirement [89]. The Sun UltraSPARC-IIi had a similar buffer as part of its load/store unit [98]. More recent architectures like the Intel Core i7, i5, i3 processors have buffers of 32 entries, while Intel Core 2 Duo processors have 20 entries [37]. In-order execution cores like those in the Sun UltraSPARC T2 (Niagara 2) [71, 130] also include an 8-entry store buffer per thread to hide the latency of write misses and bypass data to subsequent loads directly from the buffer [7].

HTM design point studies have so far been primarily concerned with the impact of TM policies on performance. While the influence of policy is important, the effects of common structural optimizations that are present at the interface between the processing core and the memory system must also be taken into account. In this context, the implications of store-buffering on HTM performance and its interactions with the conflict detection mechanism have not been analyzed in the literature so far. Therefore, this study on the impact of store-buffering in HTM systems encompasses insights that apply to a large body of work done on the topic. One common characteristic in most studies is that they do not investigate different speculative buffering mechanisms which, as we show in this chapter, can cause significant variation in key performance metrics. Indeed, these buffers are absent or not clearly described in the vast majority of the HTM literature, leading to an over-simplified interface of the memory subsystem with the in-order processor model commonly assumed. Considering the sensitivity of performance to the structural optimizations highlighted in this chapter, ascribing improvements in performance metrics to changes in policy or high level protocol design is fraught with the risk of imprecision and oversimplification. This is of particular importance in scalable network-on-chip hierarchies where communication delays can be the major determinant of performance. This work hence emphasizes the importance of accurate modeling when considering the complex interaction of multithreaded code with synchronization mechanisms in hardware.

The most natural approach for systems that manage speculative data lazily is to use on chip private caches for buffering speculative data [9, 29, 33, 56, 96, 102, 107, 143]. This allows speculative and non-speculative data to dynamically share the storage capacity available in each processor and provides support for fast associative searches [53, 73, 127]. On abort, cache lines that have been

speculatively written need to be invalidated. Each cache line generally augmented with a *speculatively modified* (SM) bit that is set by transactional stores and cleared on commit/abort. When a violation is detected, a gang-invalidation operation over all SM lines can be done in a handful of cycles using inexpensive custom circuitry [87], making the actions required to discard speculative state a trivial operation for lazy HTMs.

The TCC proposal [55, 56] implements coherent buffering in a private cache, but it proposes a write-through commit scheme that greatly simplifies its coherence protocol. A committing transaction writes back all its speculative updates to the shared memory hierarchy using bus-based global commit arbitration. While commit data packets place important demands on bus bandwidth, no upgrades or downgrades of dirty data occur since the shared levels always have the up-to-date version of the last-committed data. Arguing that programmers or compilers could give hints about local-only data, Hammond *et al.* explicitely removed stack references from the generated traces that were fed to their simulator, with the intent of reducing commit bandwidth since thread-local stores that do not need to be broadcast nor snooped by other processors. However, this study does not quantify the reduction in the amount of committed lines achieved by filtering thread-local data. Moreover, speculative updates to local data must still be buffered and discarded on abort, and they cause invalidations whose effects in cache performance were not considered in such analysis.

A later refinement of TCC [29] provides a scalable commit algorithm which allows for considerable parallelism in directory-based DSM systems. It works by dividing the directory into several banks. Transactions can commit in parallel if they do not observe directory bank conflicts. Commit sequence numbers are assigned to prioritize transactions when such conflicts occur. In EazyHTM [143], Tomic *et al.* describe an eager conflict detection design that commits transactions lazily, utilizing directory coherence in MESI based systems. Both Scalable-TCC and EazyHTM incorporate multilevel private caches that make overflows due to the limited speculative buffering hardware quite an unlikely situation [34]. While several mechanisms have been proposed to handle transactions of any size [33, 107], the use of large private L2 caches for tracking transactional state makes these overflows a rather uncommon case. In these designs that support write-back commit [29, 143], standard invalidation-based coherence protocols are leveraged so that dirty lines are allowed to be privately cached, thus reducing the bandwidth requirements of commit. Negi *et al.* developed a broadcast-based lazy commit protocol in [96] that eliminates the need for write-backs or cache-line invalidation messaging at commit. However, none of these proposals quantifies

the effects of misses on restarted transactions due to cache pollution, nor the increased commit latencies and traffic caused by redundant downgrade/upgrade coherence actions for thread-local or non-actively shared data written during a transaction.

LogTM [153] describes a protocol where transactional stores update memory in-place and store old values on the side, using a per-thread log in virtual memory that is unrolled by a software abort handler in case of abort. The design leverages coherence to implement conflict resolution and isolation. Later refinements of this protocol include support for nesting [93] and Bloom filters that ease virtualization of transactions [153]. One common characteristic in these proposals is that they do not investigate the interplay of in-place speculative writes and store buffers, and are thus oblivious to the performance implications of their latency-hiding effects, which as we show in this chapter, can cause significant variation.

Bobba *et al.* [22] identify and analyze several pathologies that cause performance differences between three HTM systems that explore different points of the design space. One of the modelled systems, referred to as LL, uses both lazy version management and lazy conflict resolution, and relies upon an independent store buffer of unlimited size that contains all speculative updates until the transaction commits. Therefore, LL does not suffer from cache contamination, nor redundant coherence actions at commit time due to write-backs of dirty lines targeted by transactional stores. The paper compares the performance of LL against that of a purely eager system (EE), similar LogTM-SE [153]. However, the modelled EE system does not include a store buffer, so that write misses stall the execution until the line is fetched with the appropriate permissions. In contrast to EE, stores in their LL system are always single cycle instructions. While the authors do not attempt to determine which of these systems is best, but rather seek to identify pathological execution behaviours, the inexistence of a store buffer penalizes the eager design and results in an biased performance comparison of design policies.

In [149], Waliullah et. al coin the term *contamination miss* as those cache misses encountered by restarted transactions after a speculatively modified line was gang-invalidated on the preceding abort. This study evalutes the frequency of contamination misses as well as the loss in performance caused by this new class of cache misses, yet it does not explore the opportunities presented by standard store buffers to filter the amount of transactional updates that are received by the private cache. In fact, systems that manage speculative data lazily using private caches must have a mechanism for collecting all of its modified cache lines that need publication. This can implemented as a FIFO address buffer that maintains

a list of the line tags whose SM bit is set [29], though it is also possible to reuse an existing structure such as the store buffer to maintain both data and addresses completely separated from cache for those transactions whose footprint does not exceed its capacity.

Shriraman *et al.* [121] performed a comparative study of contention management policies in a hybrid FlexTM [122] based design. This study claims that systems lazy contention management achieves higher performance better than those with eager management. We would like to emphasize that conflict resolution policy is a factor that contributes towards overall HTM system performance but it is not the sole one. Effective management of updates in store buffers can tip the scales.

Sanyal *et al.* [117] proposed schemes, involving both paging hardware and the operating system, to manage thread-local data separately to ease the burden on speculative versioning mechanisms. In this work, store buffers can achieve similar effects when they are large enough to capture most updates. Caches are not contaminated by speculative updates and commits and aborts do not penalize accesses to thread local data. Dahlgren *et al.* [40] analyzed the efficacy of write caches in parallel architectures supporting relaxed consistency models and demonstrated major improvements in miss penalties associated with coherence misses. While the study is not directly related to TM, the results therein suggest that transactional semantics permit flexibility in handling updates issued within atomic code blocks.

7.3 Buffering speculative updates in L1 caches

This section discusses how speculative updates are buffered in coherent caches. We look at both lazy and eager designs. Lazy designs need discussion since they require modest deviations from the way coherence protocols typically work. Eager designs do not require behavioural changes in the protocols to support transactional updates but such updates have performance implications, nevertheless.

7.3.1 Lazy HTMs

For lazy HTMs, the utilization of first-level caches as speculative store buffers is a popular strategy, as private caches make a good place to maintain the transactional bookkeeping information necessitated for conflict detection, and

they generally have enough capacity and associativity to contain speculative updates, in the common case. Furthermore, using private caches for transactional buffering allows cache coherence protocols to be leveraged for the detection of data races at the granularity of cache lines. To do so, however, requires certain protocol actions be taken, actions that incur a cost in terms of latency when they are performed, as well as the possibility of having to revert their effect in the future. Since lazy HTMs must allow for the existence of multiple uncommitted versions of the same data written by different transactions, utilizing the private cache for buffering introduces a degree of incoherency in the memory hierarchy. While multiple speculatively modified lines can coexist in different private caches, the coherence protocol treats all such copies as sharers of the last consistent state of the cache line, which is kept in the shared levels. When a transaction attempts to commit and makes its updates globally visible, any other transaction that has read such data must violate and abort. Committing generally entails the acquisition of exclusive ownership over all lines in the write set of a transaction. The coherence protocol ensures that all other shared copies of the line are invalidated, and all other transactional accessors leverage these invalidations to detect races and abort.

***Downgrade* misses.** Rather than using a write-through approach to the publication of values [56] at commit time, many lazy HTM designs employ a *write-back commit* scheme that allows committed values to remain in private cache after the transaction finished [22,27,29,102,143]. Although new values become globally accessible once commit is complete, their write-back to shared levels is deferred until one of two events occur: i) when the line is requested by another cache, in which case a copy of the dirty line is sent to the directory besides the cache-to-cache transfer –usual behaviour of the coherence protocol–; or ii) when a new transactional store targets the dirty (or exclusively owned) line. In the latter case, the coherence protocol must be explicitely adapted to support lazy versioning, as a downgrade to shared state with a possible write-back of dirty data is needed, for two important reasons. First, the shared levels of the cache hierarchy must be updated with the last committed value for consistency, and second, the discovery of data races with new accessors through standard coherence invalidations must be resumed. In scenarios of low contention, where the probability that a transaction will commit without aborting is high, such coherence actions to enable conflict detection are redundant. In fact, it is quite likely that, due to locality of reference, transactional updates hit lines that are already dirty or exclusively owned by the private cache. Therefore, downgrading these lines to shared state to correctly handle the unlikely case when a conflict

might be seen on the lines is inherently pessimistic and penalizes the common, non-contended case.

To buffer speculative data in private caches, per-cache line meta-data is augmented with two bits, SR and SM, which indicate whether a line has been speculatively read or speculatively modified, respectively. During the course of execution of a transaction, writes appear as reads to the coherence protocol. In order to preserve its last globally consistent value, a dirty (non-speculative) line is written back to the shared memory hierarchy prior to the first speculative update to it in a transaction, resulting in a downgrade of its coherence state from M to S. Commits commonly imply acquisition of ownership over all lines with SM set, while aborts imply invalidation of all such lines.

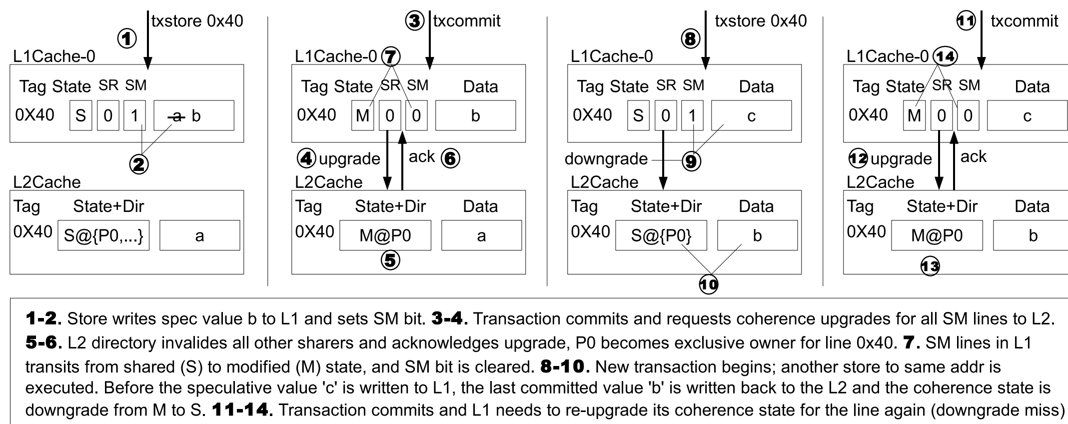


Figure 7.2: *Downgrade miss*: Redundant cache-state changes when a transaction eventually commits.

Let us now consider the case presented in Figure 7.2. A line that is only written by one transaction (it is either thread-private or not actively shared in the current phase of the workload) might, in the steady state, be found with high probability in M state in the private cache. To preserve the old content of the line in the absence of non-coherent store-buffering, it must be written back prior to the write, resulting in a transition to S, as shown in Figure 7.2 (step 9). On commit we need to reacquire exclusive ownership to the line. Since such a line will not have any sharers, this work (M⟶S and S⟶M) to ensure no races exist is largely redundant. We refer to such events where unnecessary work prolongs the commit phase of a transaction as a *downgrade miss*. Coarse

grained transactions that have a relatively large fraction of the write set as private data (stack, thread-local storage) are expected to show the most degradation in performance. In Section 7.5 we examine in detail the impact of downgrade misses and see that for some applications their elimination results in a significant contraction of commit delays.

Downgrade misses could be avoided if speculative writes to E lines (clean) were allowed, as the shared cache levels still retain the last committed value. However, the changes in the coherence protocol to support speculative writes to exclusively owned cache lines are rather complex: First, the exclusive owner (speculative writer) would not be able to respond with a copy of the consistent data upon a forwarded shared request from a new reader; the usual cache-to-cache transfer in this case would need to be replaced with a more artificial behaviour in which the requested data is supplied by the shared level and the line is downgraded to shared by the exclusive owner. A second drawback of this approach is that gang-invalidation of speculatively modified lines would no longer be possible, as silent invalidations of E lines would leave the directory in inconsistent state.

***Contamination* misses.** In highly-contended scenarios, frequent aborts cause repeated invalidations of speculatively modified lines in the private cache. Figure 7.3 depicts this case: A transaction speculatively updates a non-contended line present in its cache and then aborts. The line would not be found in the private cache on re-execution as all lines in the write-set have now been invalidated. Such misses are referred to as *contamination* misses [149]. Workloads with large write sets and high contention over small amounts of shared data would experience the greatest drop in private cache hit rates. As Section 7.5 will show, elimination of contamination misses using store buffers results in a marked overall improvement in private cache hit rates.

Write-write conflicts. Lazy HTM systems possess the ability to overcome write-write conflicts when speculative updates are performed in non-coherent buffers. Multiple speculatively modified versions of the same data element can exist in different transactions, and if no other races exist, transactions can commit one after another, freely overwriting previously modified data in a clearly sequenced manner, thus handling write-after-write dependencies in a legal way. Unfortunately, this inherent property of lazy versioning is lost when coherent caches are used for buffering speculative updates in invalidation-based coherence protocols. Despite the fact that write-write conflicts are not true data dependencies, transactions must violate and invalidate the conflicting cache lines.

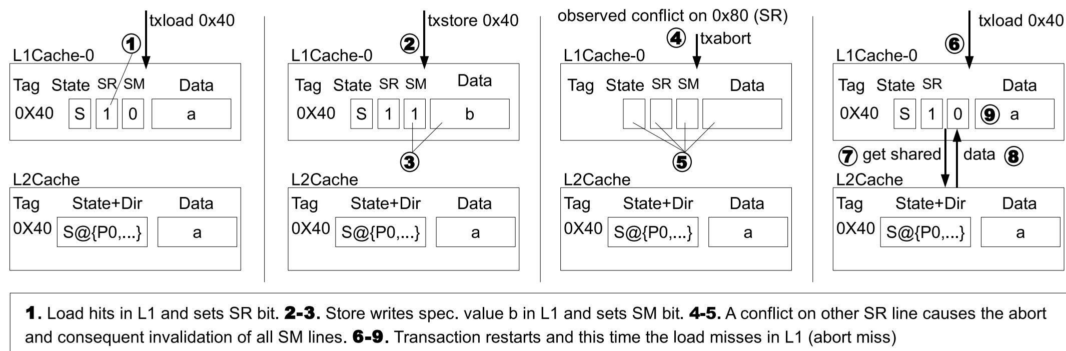


Figure 7.3: *Contamination miss*: Invalidations that could be avoided using a store buffer.

7.3.2 Eager HTMs

Eager HTMs do not require special coherence actions to be taken when updating lines. Yet, they require the old value of the line to be preserved in a special undo-log, a part of which may exist as a hardware structure in the private cache and a part implemented as a data structure in virtual memory. In the architectural setup for this study, the cache has a small 8-cache-line *old-value-buffer (OVB)*. Overflows from the OVB are accommodated in the software log.

Before a transactional write can be performed, exclusive ownership permission on the target cache line is required. Thus, for shared or absent cache lines the write must be buffered in the standard miss state handling mechanism, MSHR. If the old-value logging operation misses in the OVB an update to the software log must also be initiated which can result in a second in-flight access being added to the MSHR. The old value of the line must be preserved till the software-log cache line is allocated in the cache. If a race with another transaction causes a coherence request to be retried, the request can sit in the MSHR being sent out periodically into the network.

Thus, it can be observed that a number of actions that operate on entire cache lines may be invoked when a transactional store hits the coherent cache hierarchy resulting in increased cache controller usage, traffic and even stalls when stores are bursty.

7.4 Use of store buffers

Store buffers are widely used in modern processors in order to hide latency for completing updates to memory, effectively avoiding stalls when write misses are encountered. If conveniently sized, the store buffer can contain most transactional updates in the common case, significantly reducing redundant protocol actions. The concept of store buffers can be extended to multilevel private cache hierarchies, in which first-level caches need not maintain coherence on transactionally updated lines.

Private caches are present primarily to keep frequently used data close to the processor core. Their use in buffering uncommitted data should be made conservatively. Write-buffers between the core and the coherent first-level cache can be used to prevent transactional updates from polluting the coherent cache hierarchy. The idea can also be extended to inclusive two-level private caching schemes, wherein the first-level private cache can be made non-coherent when handling transactional updates.

This can be incorporated very simply into the design by capturing writes issued by the processor and then releasing those to the private cache in a controlled manner. In a lazy HTM, writes would be captured throughout the execution of a transaction or as long as buffer capacity is not exceeded. On commit, the buffer would be flushed causing all writes to enter the memory hierarchy as quickly as possible. On abort, the buffer contents would simply be discarded. In an eager HTM, such a buffer would obviously also participate in any store forwarding mechanism. In fact, existing store-buffering schemes could be suitably modified to enable such functionality.

It can be observed quite easily that unnecessary switches in coherence state and invalidations of aborts can be completely eliminated if write-sets are fully contained in store buffers. Since speculative data can be recorded in the store buffer, we can eliminate write-backs and downgrades of M lines to shared (S) state. On commit, since the line would likely be present in the cache in the M state, it can simply be written into the private cache without any coherence action. An abort results in the speculative contents inside the store buffer to be discarded. No cache lines need invalidation and, thus, the transaction on re-execution would still find such lines in the private cache. Writes contained in the store buffer can be probed for cache-hits and if they are not found in the cache a non-exclusive prefetch can be made. This is done in the expectation that a long latency write-miss at commit can now be converted to a potentially less

costly upgrade-to-exclusive request. Later in this study, we find that doing so improves performance in most cases.

Eager HTMs also benefit from the reduction in the number of contamination misses. Buffered writes can then be released in a controlled manner into the private cache depending on the drainage policy. An in-place update in shared memory is attempted in a non-blocking manner while the processor continues to run ahead. Logging of old values in the undo-log as necessitated by eager versioning is also taken out of the critical path. In case a conflict exists, the request can be retried in the background while execution of the transaction continues. Thus, transactional writes never stall execution on the core. They also benefit from the largely hidden latencies for eager conflict detection on stores. When executing transactions, there are several choices for store-buffer behaviour. This relate to how buffered writes are drained into the coherent memory hierarchy. We define *immediate draining* as issuance of the update into the coherent hierarchy as soon as possible after the write has been buffered. *Deferred draining* delays the issue till a later point in the execution of the transaction. This could be based on criteria like the fraction of used entries in the store buffer being greater than a certain threshold.

Another benefit of having store buffers is reduction in the number of write-write conflicts, which are purely an artifact of using coherence messages to detect modifications to (possibly different parts of) the same cache line. Confinement of writes until commit avoids contamination of cache lines and hence result in no ambiguities when the final cache line update occurs. Since store buffers maintain the association between values and their exact addresses at word or byte level-granularity, it is possible for two concurrent writer transactions to commit one after another, subsequently merging their respective updates onto the same cache line. In fact, store-buffering makes transactions appear as non-transactional accessors for their write-only lines, so that coherence invalidations to such lines are acknowledged without causing violations. The cache, when it does not buffer any speculative updates, only records the read set of a transaction. Hence any invalidations resulting from transaction commits result in aborts only when there is a possible true data race –the write set of the committer conflicts with the read set of the other– though the detection at the granularity of cache lines can still provoke unnecessary aborts due to reader-writer false sharing.

Table 7.1: Store buffer configurations.

Lazy	
infWB	infinite store buffer
noWB	no store buffer
realWB	finite store buffer
realWB_pf	finite store buffer with prefetch
realWB_pf_pc	finite store buffer with prefetch and parallel commit-time write-set acquisition
Eager	
EE_base	GEMS baseline
EE_lw	GEMS baseline with lingering write optimization
EE_infWB	infinite store buffer
EE_realWB_DD	finite store buffer with deferred draining
EE_realWB_ID	finite store buffer with immediate draining

7.5 Evaluation

In this section, we evaluate performance implications of both buffering in coherent caches and store-buffering in several pertinent HTM design points.

Lazy designs. The lazy HTM system modeled in this evaluation is an extension of the lazy system considered by Bobba *et al.* [22], available in the GEMS v2.1 release. While Bobba's LL system models a private, per processor infinite store buffer, for this study we extended the simulator to precisely model finite buffering for transactional writes. We limited the capacity of the non-coherent store buffer, so that once it fills up, transactional stores happen in the private data cache. Unlike writes to the L1 cache, which need the line present in cache to be able to complete, writes to the store buffer allow the core to execute ahead. A non-blocking *prefetch-read* for the line is sent to the L2 cache if the line is absent in the L1 cache. We modified the replacement policy of the L1 data cache by giving the highest priority to speculatively written lines, in order to minimize the number of transactional overflows when executing large transactions. Nonetheless, we incorporate a speculative victim buffer to avoid serialization penalty due to limited buffering capacity, similarly to [22]. In our simulations only yada experiences a few such evictions and a small victim buffer with 8 entries proves sufficient. We use an ideal bookkeeping scheme to track read sets (*perfect signatures*) even when some speculatively read lines have been evicted, in an attempt to isolate our study from the effects of false conflicts arising from non-ideal signature schemes like

bloom filters. A simple commit token algorithm is used to serialize transaction commits: Transactions arbitrate for the token using a zero-latency broadcast bus. Once the token is acquired, a transaction enters the commit phase and issues coherence requests to gain exclusive ownership over all lines in its write set. We later present results for the five lazy HTM configurations listed in Table 7.1. Finite store buffers are 128 bytes in size and have word (32bit) granularity. The last two configurations attempt non-exclusive prefetches on cache lines targeted by writes if not already present in the cache. These prefetches do not block execution on the core and are done in the expectation that it will help reduce the quantity of data transferred at commit time. To perform a fair comparison with eager HTM designs (Section 7.5.3), *realWB_pf_pc* attempts to speed up commits by parallelizing issuance of writes into the coherent hierarchy at commit time.
Eager designs. The eager design is based on Log-TM [153]. We have introduced a simple *lingering-write* optimization (referred to as *EE_lw*) in the basic eager implementation. If writes are pending when a transaction aborts they are silenced and completed in background, perhaps even after the transaction has restarted. In fact, the in-flight access (i.e. the lingering write) often proves useful bringing in data required during re-execution. This has two benefits: first, the silenced write acts like an *exclusive prefetch*, and second, the transaction can now restart earlier and has a better chance to use the cache line before it is invalidated due to contention. This provides a significant boost in performance when contention is high and transactions are small. This simple optimization results in a fairer comparison of policies and effects of other structures. We felt it would be useful for readers to see how this compares to the basic protocol available with GEMS. Hence, results for eager HTMs also include the GEMS reference implementation (*EE_base*).

The baseline system is then augmented with a finite store buffer. Transactional execution on a core is now not stalled when an update requires coherence actions. It is simply buffered in the store buffer. In contrast to lazy designs, where we attempt to keep writes in the buffer for as long as possible, the eager design drains buffered writes into the coherent hierarchy at some point before commit. We model two configurations: *EE_realWB_ID* which implements immediate draining, and *EE_realWB_DD* which implements deferred draining. Immediate draining involves issuing writes into the buffer as soon as possible. Deferred draining attempts to keep writes in the buffer until 80% of the buffer fills up. To implement a limit study, we also model a configuration with an infinite store buffer (*EE_infWB*) that does not drain until a transaction attempts to commit.

7.5.1 Lazy HTM Results

In this section, we analyze the impact of the four buffering schemes with lazy conflict resolution described earlier and quantify the effectiveness of store buffers in improving cache performance and reducing the number of coherence actions required on commit. Figure 7.4 shows the average miss rate of L1 data caches for each STAMP benchmark. A dedicated store buffer reduces miss rate for almost every benchmark. In Figure 7.5, we present the number of contamination misses suffered on average by a transaction that restarted at least once. The same plot also shows the average number of downgrade misses per committed transaction. We see reductions in both metrics when buffer sizes are large enough to contain most of the writes.

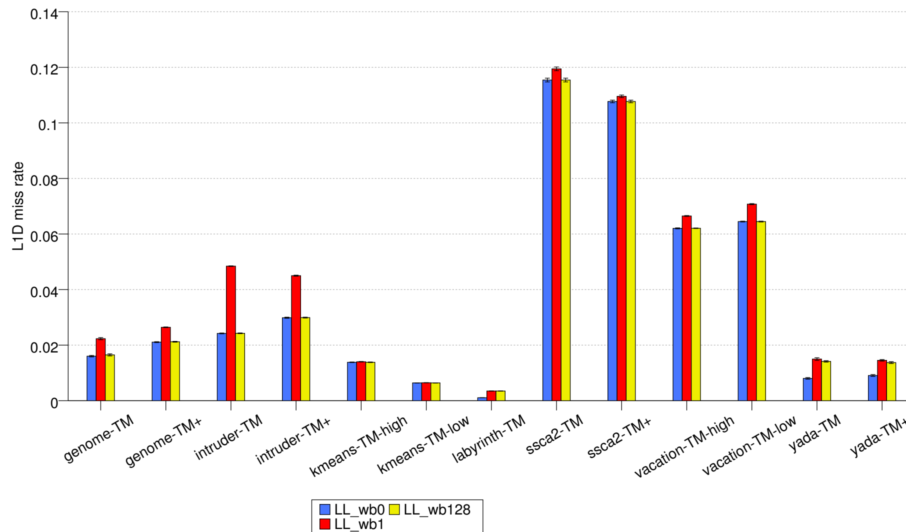


Figure 7.4: Lazy design points: L1 data cache miss rates.

Figure 7.6 shows the execution time breakdown of all applications, normalized to the execution time of the configuration with no store buffer, running 16 threads. The execution time is broken down into eight components – *barrier* is a measure of the time spent waiting at barriers; *non-txnal* corresponds to the number of cycles spent executing non-transactional code; *tx-useful and tx-aborted* represent cycles spent in transactional execution, split into useful and aborted cycles, respectively; *stall* is the time a transaction spent stalled in a data access, because such data was in the write set of a committing transaction; *backoff* represents the wait before an aborted transaction restarts, determined using a linear-backoff algorithm;

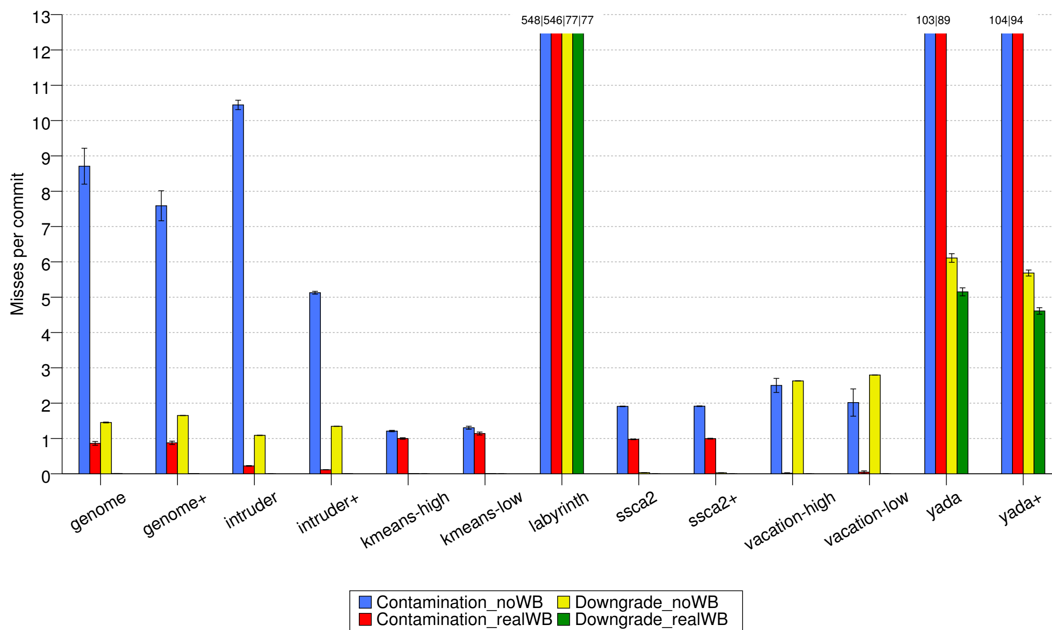


Figure 7.5: Lazy design points: Downgrade and contamination misses.

arbitration and *acquisition* represent the overheads experienced at commit time, due to arbitration for the commit token and acquisition of exclusive ownership over modified lines, respectively. The figure shows a consistent improvement in performance when store buffers are present. Figure 7.7 zooms in on the commit overheads, represented by the sum of cycles spent arbitrating for commit and acquisition of the write set, imposed by various configurations.

Genome. This workload runs high contention transactions in its first phase. Store buffers prove sufficient and are able to substantially mitigate contamination misses, resulting in substantial improvements in L1 performance (seen in Figure 7.4). In the latter phases, contention is relatively low and reductions in downgrade misses (see Figure 7.5) provide further performance boost. Prefetching lines targeted by buffered speculative writes (*realWB_pf*) also yields substantial benefits (8-10%) by overlapping latency for data transfer with useful execution.

Intruder. The improvement in L1 cache performance is the most significant in intruder – an application with high contention, a large number of transactions and a medium-sized write set (about 50 bytes spread across 6 cache lines on average for its main transaction). Here, the impact of contamination in the private cache due to speculative writes is considerable. As described in Section

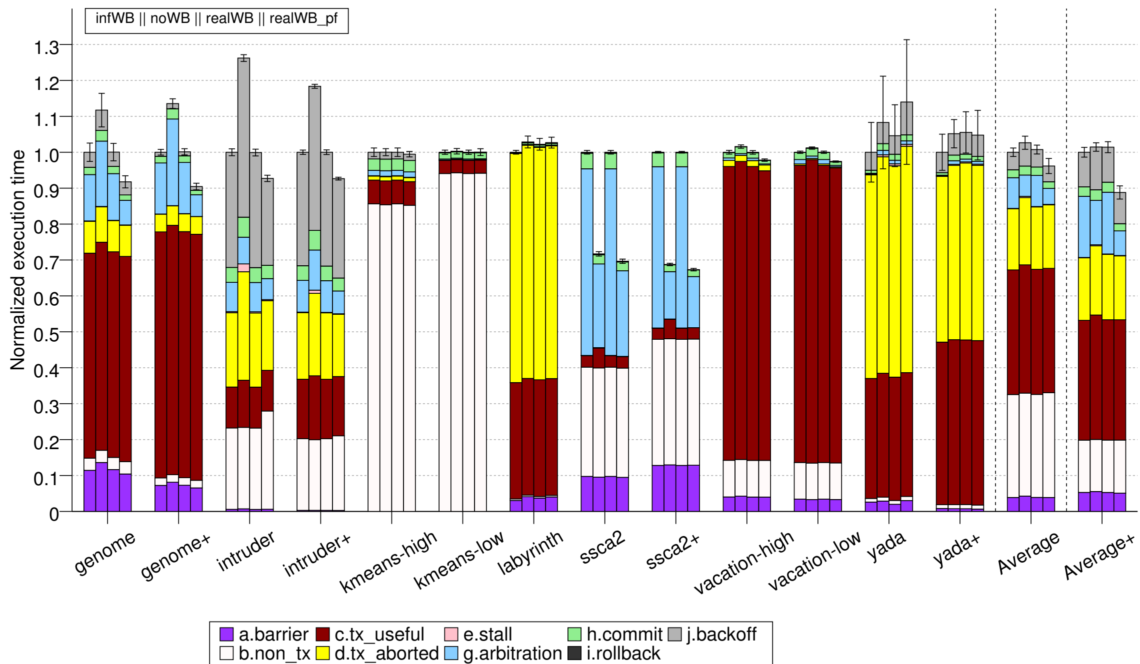


Figure 7.6: Lazy design points: execution time breakdown.

7.4, repeated aborts cause invalidations of speculatively dirty data, which then result in misses when the transaction re-executes. Figure 7.5 clearly shows the number of contamination misses suffered by restarted transactions is significant in intruder, with an average of 10 such misses until a (perhaps repeatedly) restarted transaction eventually commits. This causes severe degradation in the L1 cache miss rate. The use of a store buffer completely eliminates contamination misses for this application, and effectively reduces its cache miss rate by 40%, as shown in Figure 7.4, for both configurations with speculative store-buffering enabled. The improvement in L1 cache performance shortens the duration of the transaction and thus reduces its probability of conflicting with other concurrent transactions. The nett effect is a substantial decrease of the number of aborted transactions (from almost 14000 in *noWB* to around 12200 in *realWB/idealWB*). This explains reductions in both *tx-aborted* and *backoff* components of the total execution time.

Kmeans. This application spends little time executing transactions. Moreover these transactions are tiny and behavioural variations between different config-

urations do not substantially impact performance. Nevertheless, store buffers achieve minor reductions in the number of contamination misses (see Figure7.5).
Labyrinth. In this benchmark, each thread replicates the global grid into its thread-local memory, and then applies Lee's routing algorithm on a local grid. Every time a thread creates a copy of the global grid, the cache lines that contain the local grid are likely to be still in modified state since the last commit, and thus must be written back to the L2 as well as downgraded to shared state before being speculatively modified again. At commit time, the writes to the local grid are indistinguishable from those to the global structure, and hence result in a large number of redundant coherence requests. Transactions here have extremely large write sets running to several hundred cache lines. Finite buffers prove insufficient and execution times remain largely independent of buffering configurations. The same trend is seen in L1 cache performance and the number of abort and downgrade misses. The idealized infinite store buffer configuration coupled with extremely high L1 hit rate significantly reduces commit and arbitration delays as seen in Figure 7.7.
SSCA2. This workload has a very large number of predominantly non-conflicting tiny transactions that stress commit bandwidth. Thus, configurations that are able to reduce communication at commit time perform far better (30%) . Since contention is low, prefetching lines that would eventually be updated at commit improves performance significantly (realWB_pf in Figure 7.6). Here, *noWB*

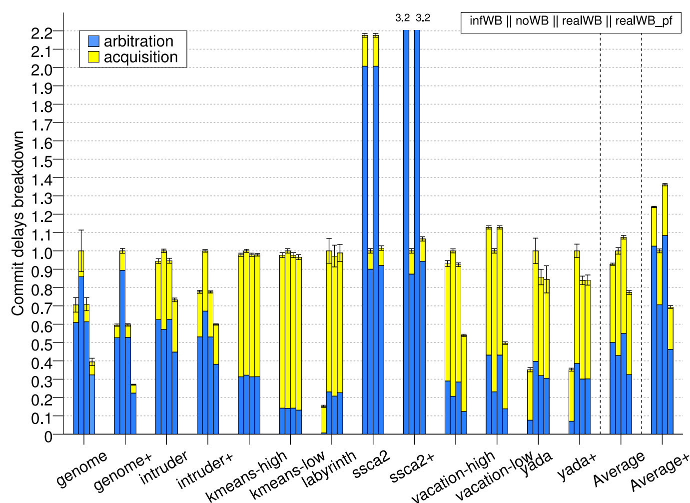


Figure 7.7: Lazy design points: Commit and arbitration delays.

configuration is interesting too since it effectively behaves like the prefetch configuration, reading in lines that would be eventually written.
Vacation. This workload shows little contention. Small improvements in performance can be noticed due to reductions in abort and downgrade misses. Prefetching also provides marginal improvements in performance.
Yada. Yada exhibits a high degree of cache contamination in the configuration with coherent buffering, with 92 contamination misses per each commit of a restarted transaction. However, its very large write set (60 cache lines on average for its main transaction, with 2124 bytes written) makes it impossible for the 128-byte store buffer configuration to contain any substantial number of writes. Yet, it can be seen that *infWB* and *realWB* configurations perform better than others. This is because they are able to avoid, to a certain extent, interference when write-write conflicts exist.

7.5.2 Eager HTM Results

Figure 7.8 compares execution times of the five eager configurations listed in Table 7.1. The first four components are as described earlier; *stall_useful* represents stalls in cases when the transaction commits without aborting after the stall is released; *stall_aborted* represents stalls in cases when the stalled transaction eventually aborted; *rollback* represents cycles spent in restoring old values upon abort. The figure shows store-buffering, and immediate draining in particular, improves performance significantly. Figure 7.9 presents a breakdown of transactional cycles spent waiting for memory accesses to complete. We discuss below important insights for each benchmark gathered from the data.
Genome. The early high contention phase in genome benefits from store-buffering. Transactions in this phase are relatively small so the size of the buffer does not matter much. Infinite buffering and deferred draining show marginal improvements over immediate draining (see Figure 7.8). This is indicative of useful run-ahead execution past conflicts allowing readers to complete without stalling. An overall speedup of about 7% over the baseline is seen.
Intruder. This application benefits the most (more than 50% for intruder+) from introduction of store-buffering with immediate draining (Figure 7.8). This is because conflicts are detected early resulting in stalls and consequently less demand for otherwise highly contended shared memory resources. Furthermore, immediate draining of writes behaves like prefetch of non-contended data. Deferred draining (and the infinite buffering case) causes injection of a lot of traffic into the network when transactions commit. This chokes directory resources and

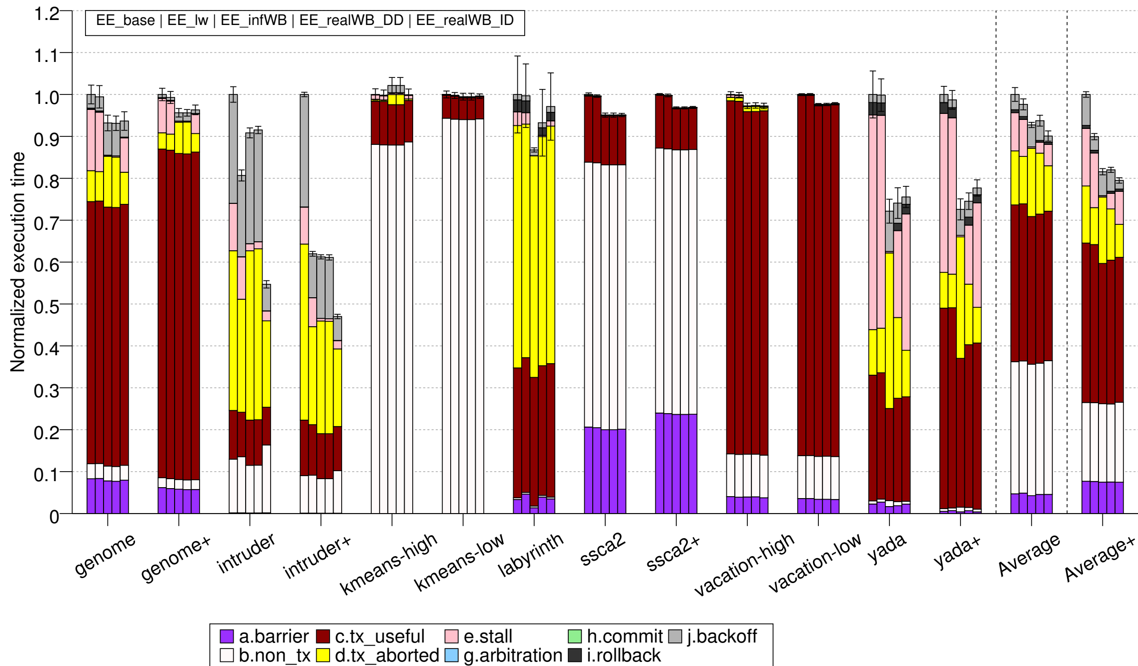


Figure 7.8: Eager design points: execution time breakdown

is unproductive since a large number of concurrently running transactions are conflicting. The overall impact of reduction in transactional execution time with immediate draining is magnified as it results in a contraction of the window of contention leading to greater concurrency. Cache performance is significantly better too (see Figure 7.9). The lingering write optimization (EE_lw) also shows significant improvements (20% for intruder and 38% for intruder+) over the GEMS baseline implementation. Lingering writes are often re-activated when the restared transaction performs a new write to the same location, and in some cases effectively obtains lines in exclusive mode for transactional read-modify-write operations to complete successfully.

Kmeans. The application (both high and low contention variants) shows no major differences in execution time across different configurations. Transactions are tiny and most of the application is non-transactional. Minor (3%) degradation is seen with an infinite buffer or a finite deferred draining buffer when contention is high. This is because of the lighter commit time operations due to prefetch effects of immediate draining and reduced demands on directory banks at commit time.

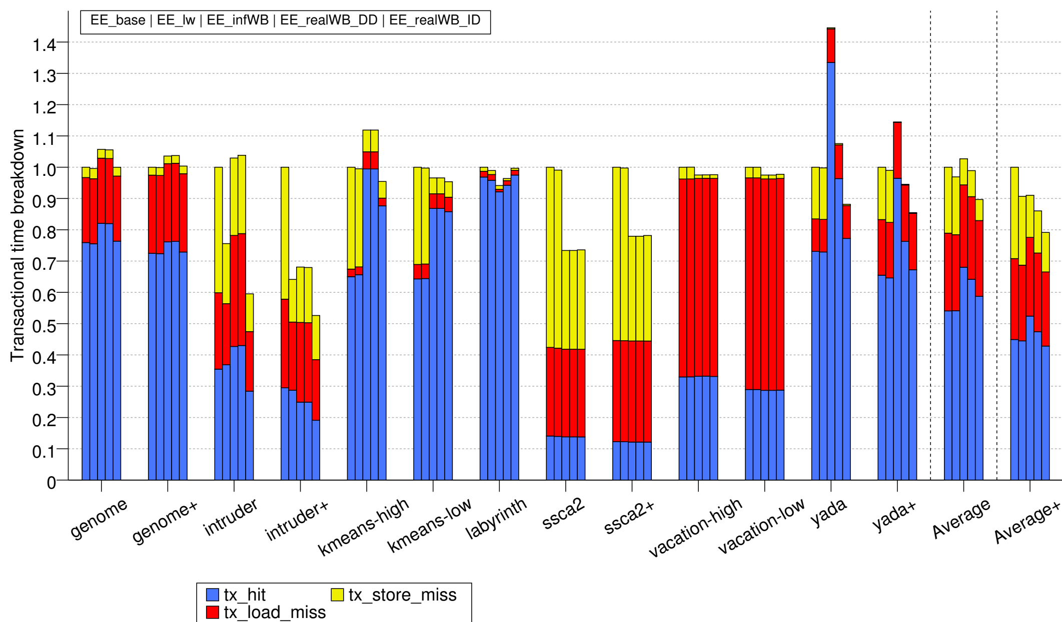


Figure 7.9: Eager design points: transactional time breakdown

Labyrinth. Finite buffering proves inadequate in this case. Deferred draining and infinite buffering manage to avoid unnecessary interference during execution time by mitigating write-write conflicts, achieving speedups of about 8-12%, partly because commits are typically fast due to the good cache performance over its write set (see Figure 7.9).

SSCA2. Eager designs in general perform well for SSCA2 because of extremely low contention and high demands on commit bandwidth. Nevertheless, store-buffering manages to hide store latencies (Figure 7.9 shows 50% reduction in store miss times) and provides a 5% improvement that is consistent across all three buffering configurations. Prefetch effects of immediate draining are not visible because of tiny transactions.

Vacation. Its behaviour is similar to that of SSCA2, with the exception that transactions are larger and fewer, and do not stress the system much. Write-buffering obtains marginal improvements (2-3%) in execution times, shown in Figure 7.8.

Yada. Yada, like intruder, also shows significant improvements (30%) when store buffers are introduced. However unlike intruder, yada exhibits a significant number of write-write conflicts. These are hidden entirely in the infinite buffering

configuration and to a smaller extent in the deferred draining configuration. Thus, these configurations perform 4-8% better than the immediate draining configuration.

7.5.3 Eager vs. Lazy: Relative performance

Figure 7.10 compares the performance of three finite buffering configurations – EE_realWB_ID, LL_realWB_pf and LL_realWB_pf_pc –. Overall we notice a leveling out of performance. These configurations are the best performing ones for each policy. Results in several previous studies consistently favor lazy designs. Here, we show that with the right buffering mechanisms, which are not related to TM policies at all, there is an equalization of performance. Lazy systems are still better in a highly contended application like intruder (by about 35%), but this is offset by significantly better performance by the eager design for SSCA2 (20-35% better). The eager design also shows substantial speedups over lazy for yada (12-22%). Overall, we see that the eager design performs as well as the lazy ones. This key insight highlights the importance of store-buffering in HTM systems, and shows that it is as important a consideration in HTM design as conflict resolution policy.

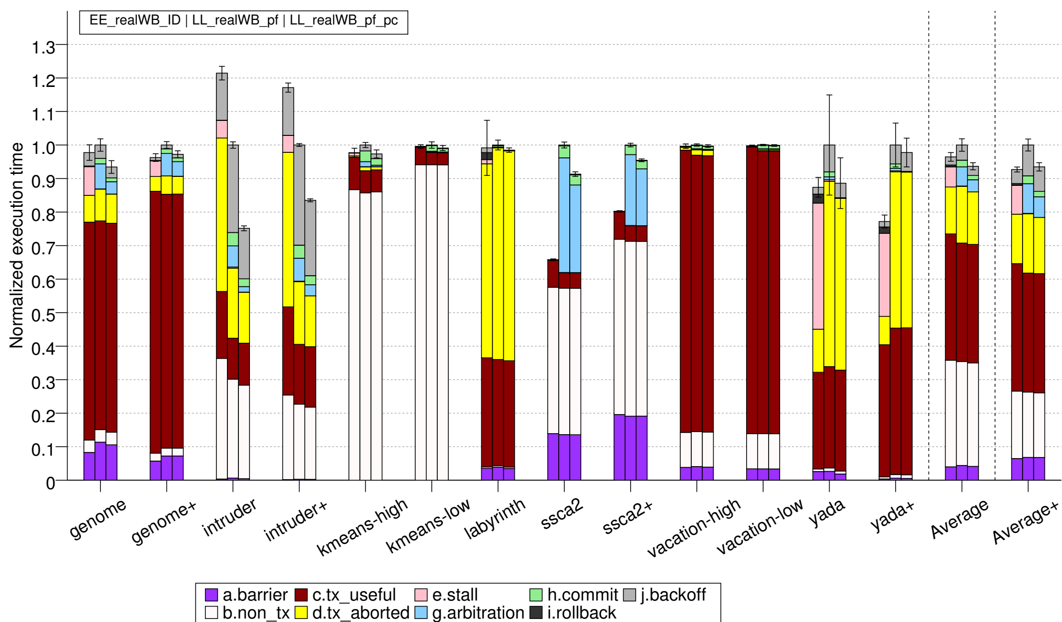


Figure 7.10: Policy performance comparison: eager vs. lazy

7.6 Concluding Remarks

In this chapter we have described and analyzed the inefficiencies that can be caused by buffering of speculative writes in coherent structures like private caches. While we do not recommend exclusive use of store buffers for managing transactional data as area and power restrictions may severely limit its utility, the importance of having such buffering to support the common case efficiently has been underlined. The performance impact of store-buffering has been quantified and shown to yield significant improvements in the set of benchmarks analyzed here. The expectation is that TM programming constructs would eventually enable workloads with coarse grained transactions, where non-contended data could be written along with actively contended data. Without appropriate store buffer support, in high contention scenarios contamination misses would result in significant degradation of cache performance in both eager and lazy designs. Under low contention but high commit throughput scenarios downgrade misses might result in substantial slowdowns in lazy designs due to prolonged arbitration. Moreover, when store buffers are present eager designs benefit both from the capability to hide store latencies and reduced logging actions. Shortened transactional execution times reduce the window of contention and improve overall performance. Another interesting insight is that store buffers bridge the performance gap between eager and lazy designs when contention is high, thereby indicating possible routes for the development of a general purpose, low complexity, high performance HTM architecture. We hope this will serve as a useful guide to architects planning to integrate hardware support for transactional memory in their designs.

Chapter 8

Conclusions and Future Ways

8.1 Conclusions

Transactional Memory is a novel programming paradigm that addresses some of the key challenges associated with concurrent programming. The performance of sequential applications can no longer be expected to improve as we have grown used to in the past decades, and programmers must consider parallel algorithms as the most viable alternative to speed up their codes and solve ever more complex tasks in reasonable execution times. In this context, transactions are an intuitive abstraction that promises to simplify the way multiple threads synchronize their accesses to data in a shared-memory environment. Software implementations of TM impose significant performance overheads, thereby making the design of high-performance HTM implementations necessary. In this thesis we have investigated hardware techniques to incorporate TM support onto a tiled CMP substrate. We have identified several inefficiencies that affect state-of-the-art HTM designs, considering both eager and lazy approaches. We have found important sources of overhead that are inherent to each policy, and we have proposed new solutions that overcome such limitations.

The first conclusion inferred from this thesis is that no fixed-policy design performs the best across the spectrum of transactional benchmarks that are available. In response, this thesis presented a case for an adaptable, hybrid-policy hardware transactional memory system, arguing that if transactional memory is going to become a mainstream model, then it would require hardware solutions that are capable of sidestepping pathological conditions and provide

robust performance over a variety of transactional workloads. The ZEBRA HTM design embodies the main contribution of this thesis. ZEBRA adapts its behaviour depending on the characteristics of the data accessed by transactions, providing significant gains in performance over existing designs, by combining the advantages of both eager and lazy approaches in a low complexity design. In this work we have outlined a fresh approach to hybrid-policy HTM design that views contention as a characteristic of the data accessed within a transaction, rather than of the transaction itself. From our observation that contended data forms a relatively small fraction of data written inside transactions, we conclude that purely lazy designs do not efficiently handle commit if they treat all the data as having the same characteristics. Similarly, no purely eager design can perform well if highly contended data is treated just like non-contended or private data, due to the inherent concurrency limitations of *a-priori* resource acquisition. This thesis proposes the techniques that manage to bring together the good aspects of both eager and lazy designs with very modest changes in architecture and protocol: ZEBRA supports parallel commits for transactions that do not access contended data and allows reader-writer concurrency when contention is seen. We have shown that ZEBRA utilizes concurrency better and consistently tracks or outperforms the best performing scalable single-policy design.

In spite of the advantages of a hybrid-policy solution, the hardware requirements of ZEBRA are considerable, as it combines in the same design both logging and in-cache versioning support. For that reason, this thesis has also focused on addressing performance limiting factors that affect fixed policy designs, whose implementation demands are lower than those of a hybrid scheme. In this thesis, we have presented a new approach to conflict detection targeted to eager TM systems which make use of a distributed directory. We have demonstrated that the directory becomes a bottleneck in situations of high contention by limiting the throughput of the conflict management mechanism. To this end, we have proposed a design that decouples such basic TM mechanism from cache coherence, enabling quicker reaction and lower penalties in highly-contended workloads. Our experimental evaluation has shown that our technique deals with contention more efficiently, leading to fewer aborts, lower overall latency of cache misses and better use of the interconnect. We have found that our scheme reduces the performance degradation caused by false positives, when compared against systems that use signatures of equivalent hardware cost at the cache level. While this technique does not completely bridge the gap between eager and lazy systems for contended workloads, it does represent a considerable palliation of

the inefficiencies caused by the eager approach in the context of a tiled CMP with distributed directory coherence.

In this thesis we have also investigated the design space of lazy scalable HTM systems. We have come to acknowledge that the lazy approach is most efficient at extracting concurrency during contention, but their scalability in low contended scenarios heavily depends on the ability to commit non-conflicting transactions in parallel. When implemented on a coherent CMP substrate, information regarding conflicts is readily available and, if used well, can lead to a design which comes close to achieving the best of both worlds –parallel commits with optimistic lazy run-ahead past conflicts–. We have discovered that this idea, while it has the potential to improve the scalability of commits, it has not been utilized effectively so far. Enabling parallel lazy commits in hardware requires correct handling of all possible interleavings of transactions, and we observed that early work in this direction reverted to pessimism in another form so as to provide correct semantics in the event of some races. Our work in this topic manages to drop this pessimism in most scenarios, resorting to a far milder form that leaves the common-case unburdened. The conclusion that we draw from this work is that a lazy HTM implementation should ideally perform some kind of eager conflict detection –in order to record data races while the transaction progresses–, but it must leverage the information contained in coherence requests properly to successfully alleviate the pressure on the commit phase without encumbering transactional execution in the common case.

Another relevant conclusion that we arrive at with this thesis is that the performance implications of store-buffering have been underestimated in prior HTM research. With our work, we have demonstrated that store-buffering can achieve noticeable performance improvements in transactional execution times by reducing HTM protocol overheads. While the utility and ubiquity of store buffers in standard microprocessors is well recognized, we believe their use and implications in the context of HTM have not been carefully considered in prior work. We find particularly interesting the remarkable performance convergence of eager and lazy TM design policies when such buffers are introduced. With store buffers, eager designs can achieve good performance even under high contention scenarios, while the performance of lazy designs is also improved due to mitigation of unnecessary actions at transaction commit or abort. Our work clearly reveals that obviating this common structural optimization from a simulated HTM model can result in measurements that are substantially different from what might be seen in real-world implementations and, more importantly, can lead to erroneous biases in favour of or against certain design options. In fact,

our in-depth study of the conflict detection mechanism in tiled CMPs further demonstrated that the poor performance exhibited by eager HTM systems is in partly due to the bottleneck formed at the directory. We conclude that coupling coherence and conflict detection mechanisms in a directory protocol, while it constitutes a natural and straightforward solution to detect data races, it is susceptible to pathological situations that badly hurt eager HTM performance during contention.

8.2 Future Ways

The results presented in this thesis open a number of interesting new research paths. Amongst them, we have identified the following:

In regards to the hardware requirements of a hybrid-policy HTM design, we find that they could be minimized by relying exclusively on the inherent versioning capabilities of private caches, eliminating the logging circuitry. This could enable a best-effort system that results more appealing to architects, yet it is able to combine both eager and lazy approaches at a cache-line granularity to simultaneously achieve parallel lazy commits and instantaneous eager aborts [80], without a departure in terms of complexity.

We want to explore the implementation of the conflict detection mechanisms using other novel cache coherence protocols for tiled CMP [113]. Better coherence mechanisms may be able to mitigate the impact of high contention on eager HTM performance. Direct coherence is a promising technique that effectively avoids the indirection problems of a directory protocol. The idea of adding an L1 coherence cache could be extrapolated to TM systems in which a private cache did not only contain the read and write set meta-data associated to its local transaction, but also tracked the remote transactional accessors for its cached lines.

We find interesting to investigate how HTM hardware can be leveraged for applications beyond synchronization, such as fault-tolerance [115]. Log-based TM hardware has already been considered to this end, but the benefits of lazy versioning provided by caches have not been considered in the context of a fault tolerant system based on redundant execution. Using a lazy approach, the memory could remain unmodified until a successful verification, avoiding the use of other mechanisms to bypass memory values between redundant threads.

Similarly, the hardware support for TM discussed in this thesis could be of use towards the facilitation of debugging and the detection of data races in both transactional and non-transactional parallel programs.

Bibliography

[1] Advanced Micro Devices. AMD demonstrates world's first x86 dual-core processor, 2004. http://www.amd.com/us/press-releases/Pages/Press_Release_89872.aspx. 1.1

[2] Advanced Micro Devices. AMD Phenom II processor product data sheet, 2010. http://support.amd.com/la/Processor_TechDocs/46878.pdf. 1.1

[3] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1996. 2.2

[4] Niket Agarwal, Li-Shiuan Peh, and Niraj Jha. Garnet: A detailed interconnection network model inside a full-system simulation framework. Technical Report CE-P08-001, Princeton University, 2008. 3.1

[5] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th Symposium on High-Performance Computer Architecture*, pages 7–19, 2003. 3.2

[6] Alaa R. Alameldeen and David A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4):8–17, 2006. 3.2

[7] Waleed Alkohlani, Jeanine Cook, and Ram Srinivasan. Extending the monte carlo processor modeling technique: Statistical performance models of the niagara 2 processor. In *Proceedings of the 39th International Conference on Parallel Processing*, pages 363–374, 2010. 7.2

[8] Saman Amarasinghe. The looming software crisis due to the multicore menace, 2007. http://groups.csail.mit.edu/commit/papers/06/MulticoreMenace.pdf. 1.2

[9] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th Symposium on High-Performance Computer Architecture*, pages 316–327, 2005. 1.3.1, 2.4, 4.1, 7.2

[10] ARM. The ARM Cortex-A9 Processors, 2011. http://www.arm.com/products/processors/cortex-a/cortex-a9.php. 1.1

[11] Adria Armejach, Azam Seydi, Rubén Titos-Gil, Ibrahim Hur, Adrián Cristal, Osman 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*, 2011. 2.4

[12] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. 1.1

[13] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Making nested parallel transactions practical using lightweight hardware support. In *Proceedings of the 24th International Conference of Supercomputing*, pages 61–71, 2010. 2.4

[14] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 115–126. 2008. 1.3.1

[15] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970. 2.4

[16] Colin Blundell, Joe Devietti, E Christopher Lewis, and Milo Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 24–34, 2007. 2.3.2, 2.4, 4.3.1

[17] Colin Blundell, E Christopher Lewis, and Milo Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), 2006. 1.3, 6.3.4

[18] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, 2006. 2.4, 3.6.5

[19] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 233–244, 2009. (document), 2.6

[20] Colin Blundell, Arun Raghavan, and Milo M.K. Martin. RETCON: transactional repair without replay. In *Proceedings of the 37th International Symposium on Computer Architecture*, pages 258–269, 2010. 2.4, 3.6

[21] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 81–91, 2008. 2.4

[22] 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 International Symposium on Computer Architecture*, pages 81–91, 2007. 2.3.5, 2.4, 3.5, 3.6.3, 3.6.3, 4.2.2, 4.4.3, 4.4.5, 4.1, 4.5.1, 5.4.1, 5.2, 6.1, 6.3.3, 6.1, 6.4, 7.2, 7.3.1, 7.5

[23] Justin Boggs. AMD CPU roadmap, 2007. http://developer.amd.com/assets/Develop_Brighton_Justin_Boggs-1.pdf. 1.1

[24] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54:67–77, 2011. 1.1

[25] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE Intl. Symposium on Workload Characterization*, pages 35–46, 2008. 3.3, 3.6

[26] 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 International Symposium on Computer Architecture*, pages 69–80, 2007. 1.3.1

[27] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 227–238, 2006. 1.3.1, 2.3.2, 2.4, 4.2.3, 4.3.1, 5.3.1, 6.3.2, 7.3.1

[28] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 278–289, 2007. 2.4

[29] 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 Symposium on High-Performance Computer Architecture*, pages 97–108, 2007. 1.3.1, 2.3.3, 2.4, 3.5.2, 3.6.7, 3.6.8, 5.1, 5.2, 5.3.1, 5.4.1, 5.2, 6.1, 6.2, 6.1, 6.4, 7.2, 7.3.1

[30] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, 2009. 1.3.1, 1.4, 2.3.1, 2.3.2

[31] Ben Chelf. Ensuring code quality in multi-threaded applications. http://www.coverity.com/library/pdf/coverity_multi-threaded_whitepaper.pdf. 1.1.1

[32] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 347–358, 2006. 2.4

[33] 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 Symposium on Architectural Support for Programming Language and Operating Systems, pages 371–381, 2006. 2.4, 7.2

[34] 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 Symposium on High-Performance Computer Architecture*, pages 266–277, 2006. 7.2

[35] Jaewoong Chung, David Christie, Martin Pohlack, Stephan Diestelhorst, Michael Hohmuth, and Luke Yen. Compilation of thoughts about AMD advanced synchronization facility and first-generation hardware transactional memory support. In *TRANSACT '10: 5th ACM SIGPLAN Workshop on Transactional Computing*, 2010. 2.3.1

[36] Cliff Click. Azul's experiences with hardware transactional memory, 2009. http://sss.cs.purdue.edu/projects/tm/tmw2010/talks/Click-2010_TMW.pdf. 1.3.1, 1.4

[37] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, vol. 3. Technical report, 2010. http://www.intel.com/Assets/PDF/manual/325384.pdf. 7.2

[38] Standard Performance Evaluation Corporation. SPECjbb2000 benchmark, 2000. 3.3

[39] David E. Culler, Jaswinder P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999. (document), 2.2, 2.4, 2.2, 4.2.1

[40] Fredrik Dahlgren and Per Stenstrom. Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 26:193–210, 1995. 7.2

[41] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 336–346, 2006. 1.3.1

[42] Koen De Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, André

Seznec, Per Stenstrom, and Olivier Temam. High-performance embedded architecture and compilation roadmap. In *Transactions on HiPEAC I*, pages 5–29, 2007. 1.1, 1.4

[43] Rajagopalan Desikan, Simha Sethumadhavan, Doug Burger, and Stephen W. Keckler. Scalable selective re-execution for edge architectures. In *Proceedings of the 11th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 120–132, 2004. 2.4

[44] David Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 19th Intl. Symposium on Distributed Computing*, 2006. 1.3.1

[45] Aleksandar Dragojevic and Rachid Guerraoui. Predicting the scalability of an STM. In *TRANSACT '10: 5th ACM SIGPLAN Workshop on Transactional Computing*, 2010. 3.3

[46] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th International Symposium on Principles of Distributed Computing*, pages 99–108, 2011. 2.4

[47] Michael Feldman. IBM specs out Blue Gene/Q chip, 2011. http://www.hpcwire.com/hpcwire/2011-08-22/ibm_specs_out_blue_gene_q_chip.html. 1.1, 1.4

[48] International Technology Roadmap for Semiconductors, 2009. http://www.itrs.net/Links/2009ITRS/Home2009.htm. 1.1

[49] Epifanio Gaona, Rubén Titos-Gil, Juan Fernandez, and Manuel E. Acacio. Characterizing energy consumption in hardware transactional memory systems. In *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 9–16, 2010. 2.4

[50] David Geer. Industry trends: Chip makers turn to multicore processors. *Computer*, 38:11–13, 2005. 1.1

[51] Robert Golla. Niagara2: A highly threaded server-on-a-chip, 2006. http://www.opensparc.net/pubs/preszo/06/04-Sun-Golla.pdf. 1.1, 3.4

[52] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, 1983. 2.2

[53] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the 4th Symposium on High-Performance Computer Architecture*, pages 195–206, 1998. 7.2

[54] Tom Groenfeldt. Software programmers lag behind hardware developments, 2011. http://blogs.forbes.com/tomgroenfeldt/2011/04/21/software-programmers-lag-behind-hardware-developments/. 1.1.1

[55] 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), 2004. 1.3.1, 2.3.1, 2.3.2, 2.4, 7.2

[56] 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 International Symposium on Computer Architecture*, pages 102–113, 2004. 2.3.2, 2.3.3, 2.3.5, 2.4, 3.5.2, 4.1, 4.3, 4.3.1, 5.1, 5.2, 5.3.1, 6.2, 7.2, 7.3.1

[57] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan & Claypool, 2010. 1.3, 1.3.1, 2.1, 2.2, 2.3

[58] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 4th edition, 2006. 1.2

[59] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Symposium on Principles of Distributed Computing*, pages 92–101, 2003. 1.3.1

[60] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993. 1.3, 1.3.1, 2.1, 2.3.1, 2.3.2, 2.4, 4.1, 4.3

[61] Arik Hesseldahl. Moore's law is alive and well, and intel will prove it today. allthingsd.com, 2011. http://allthingsd.com/20110504/moores-law-is-alive-and-well-and-intel-will-prove-it-today. 1.1

[62] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Desktop Platforms Group, and Intel Corp. The microarchitecture of the pentium-4 processor. *Intel Technology Journal*, 1:1–13, 2001. 1.1

[63] Owen S. Hofmann, Donald E. Porter, Christopher J. Rossbach, Hany E. Ramadan, and Emmett Witchel. Solving difficult HTM problems without difficult hardware. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, 2007. 1.3.1

[64] Owen S. Hofmann, Christopher J. Rossbach, and Emmett Witchel. Maximum benefit from a minimal HTM. In *Proceedings of the 14th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 145–156, 2009. 2.4

[65] Jaehyuk Huh, Jichuan Chang, Doug Burger, and Gurindar S. Sohi. Coherence decoupling: making use of incoherence. In *Proceedings of the 11th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 97–106, 2004. 2.4

[66] Intel. Intel multi-core briefing, 2005. http://download.intel.com/pressroom/kits/pentiumee/ 20050418presentation.pdf. 1.1

[67] Intel. Intel unveils new product plans for high-performance computing, 2010. http://www.intel.com/pressroom/archive/releases/2010/ 20100531comp.htm. 3.4

[68] Intel. The SCC platform overview, 2010. http://techresearch.intel.com/spaw2/uploads/files/ SCC_Platform_Overview.pdf. 3.4

[69] Syed Ali Raza Jafri, Mithuna Thottethodi, and T. N. Vijaykumar. LiteTM: Reducing transactional state overhead. In *Proceedings of the 16th Symposium on High-Performance Computer Architecture*, pages 1–12, 2010. 2.4

[70] Behram Khan, Matthew Horsnell, Ian Rogers, Mikel Luján, Andrew Dinn, and Ian Watson. An object-aware hardware transactional memory. In *Proceedings of the 10th International Conference on High Performance Computing and Communications*, pages 93–102, 2008. 2.4

[71] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 211–222, 2002. 3.6.7, 7.2

[72] Leonidas I. Kontothanassis and Michael L. Scott. Efficient shared memory with minimal hardware support. *SIGARCH Computer Architecture News*, 23:29–35, 1995. 1.2

[73] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48:866–880, 1999. 7.2

[74] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 Conference on Programming Language Design and Implementation*, pages 211–222, 2007. 3.3

[75] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220, 2006. 1.3.1

[76] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. Cell broadband engine architecture and its first implementation: a performance view. *IBM Journal Research and Development*, 2007. 1.1

[77] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal Research and Development*, 51:639–662, 2007. 3.6.7

[78] Sean Lie. Hardware support for unbounded transactional memory. Master's thesis, 2004. Massachusetts Institute of Technology. 2.4

[79] Yi Liu, Xin Zhang, He Li, Mingxiu Li, and Depei Qian. Hardware transactional memory supporting I/O operations within transactions. In *Proceedings of the 10th International Conference on High Performance Computing and Communications*, pages 85–92, 2008. 2.4

[80] Marc Lupon, Grigorios Magklis, and Antonio González. 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, 2009. 2.4, 4.1, 7.1, 8.2

[81] Marc Lupon, Grigorios Magklis, and Antonio González. A dynamically adaptable hardware transactional memory. In *Proceedings of the 43rd International Symposium on Microarchitecture*, pages 27–38, 2010. 2.4, 5.1, 5.2, 7.1

[82] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002. 3.1

[83] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th Intl. Symposium on Distributed Computing*, 2005. 1.3.1

[84] Ami Marowka. Back to thin-core massively parallel processors. *Computer*, 99(PrePrints), 2011. 1.1, 3.1

[85] Milo M.K. Martin. *Token Coherence*. PhD thesis, CS Dept., Univ. of Wisconsin-Madison, 2003. 2.4

[86] 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. *Computer Architecture News*, pages 92–99, 2005. 3.1, 3.4

[87] José F. Martínez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 3–14, 2002. 7.2

[88] Austen McDonald, JaeWoong Chung, D. Carlstrom Brian, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 53–65, 2006. 1.3.1

[89] E. J. McLellan and D. A. Webb. The alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, pages 90–96, 1998. 7.2

[90] Nebojsa Miletic, Vesna Smiljkovic, Cristian Perfumo, Tim Harris, Adrian Cristal, Ibrahim Hur, Osman Unsal, and Mateo Valero. Transactification of a real-world system library. In *TRANSACT '10: 5th ACM SIGPLAN Workshop on Transactional Computing*, 2010. 3.6.5

[91] Louis Monier and Pradeep S. Sindhu. The architecture of the dragon. In *30th IEEE Computer Society Int'l Conference*, pages 118–121, 1985. 2.2

[92] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, pages 254–265, 2006. 1.3.1, 2.3.2, 2.3.3, 2.3.4, 2.3.5, 2.4, 3.5.1, 3.6.8, 4.1, 4.2.2, 4.2.3, 4.2.4, 4.3.1, 4.3.2, 4.4.5, 5.2, 5.3.1, 5.3.1, 6.1, 6.2, 7.1

[93] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 359–370, 2006. 2.4, 7.1, 7.2

[94] Anurag Negi, Rubén Titos-Gil, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Eager meets lazy: The impact of write-buffering on hardware transactional memory. In *Proceedings of the 40th International Conference on Parallel Processing*, 2011. 1.4

[95] Anurag Negi, Rubén Titos-Gil, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. π-TM: Pessimistic invalidation for scalable lazy hardware transactional memory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (Poster)*, 2011. 1.4

[96] Anurag Negi, M.M. Waliullah, and Per Stenstrom. LV*: A low complexity lazy versioning HTM infrastructure. In *Proceedings of the Intl. Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS 2010)*, pages 231–240, 2010. 2.3.2, 2.4, 3.3, 4.2.3, 4.3.1, 5.1, 5.2, 7.2

[97] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman.

Open nesting in software transactional memory. In *Proceedings 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 68–78, 2007. 2.4

[98] Kevin B. Normoyle, Michael A. Csoppenszky, Allan Tzeng, Timothy P. Johnson, Christopher D. Furman, and Jamshid Mostoufi. UltraSPARC-IIi: Expanding the boundaries of a system on a chip. *IEEE Micro*, 18:14–24, 1998. 7.2

[99] Salil Pant and Gregory Byrd. Extending concurrency of transactional memory programs by using value prediction. In *Proceedings of the 6th ACM conference on Computing Frontiers*, pages 11–20, 2009. 2.4

[100] Salil Pant and Gregory Byrd. Limited early value communication to improve performance of transactional memory. In *Proceedings of the 23rd International Conference of Supercomputing*, pages 421–429, 2009. 2.4, 7.1

[101] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *Proceedings of the 2010 International Symposium on Performance Analysis of Software Systems*, pages 97–108, 2010. 3.6.5

[102] 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, 2008. 2.4, 5.1, 5.2, 6.1, 6.2, 7.2, 7.3.1

[103] Ricardo Quislant, Eladio Gutierrez, and Oscar Plata. Improving signatures by locality exploitation for transactional memory. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 303–312, 2009. 2.4, 4.2.3

[104] Ricardo Quislant, Eladio Gutierrez, and Oscar. Plata. Multiset signatures for transactional memory. In *Proceedings of the 25th International Conference of Supercomputing*, pages 43–52, 2011. 2.4, 4.2.3

[105] 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, 2001. 2.4

[106] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Symposium on Architectural Support for Programming Language and Operating Systems*, pages 5–17, 2002. 2.4, 5.3.1

[107] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 494–505, 2005. 1.3.1, 2.4, 4.1, 7.2

[108] Hany E. Ramadan, Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Dependence-aware transactional memory. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 246–257, 2008. 2.4

[109] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. MetaTM/TxLinux: transactional memory for an operating system. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 92–103, 2007. 2.4

[110] Nicholas Riley and Craig Zilles. Hardware transactional memory support for lightweight dynamic language evolution. In *Dynamic Language Symposium*, 2006. 2.4

[111] Ben Rooney. ARM CEO: Moore's law "alive and well", 2011. http://blogs.wsj.com/tech-europe/2011/03/15/arm-ceo-moores-law-alive-and-well. 1.1

[112] Alberto Ros. *Efficient and Scalable Cache Coherence for Many-Core Chip Multiprocessors*. PhD thesis, Depto. Ingeniería y Tecnología de Computadores, Universidad de Murcia, 2009. (document), 2.2, 2.2, 2.5, 3.6.7

[113] Alberto Ros, Manuel E. Acacio, and José M. García. A direct coherence protocol for many-core chip multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1779–1792, 2010. 8.2

[114] Christopher J. Rossbach, Hany E. Ramadan, Owen S. Hofmann, Donald E. Porter, Aditya Bhandari, and Emmett Witchel. TxLinux and MetaTM: transactional memory and the operating system. *Communications of the ACM*, 51:83–91, 2008. 2.4

[115] Daniel Sanchez, Juan L. Aragon, and Jose M. Garcia. A log-based redundant architecture for reliable parallel computation. In *Proceedings of the 17th International Conference on High-Performance Computing*, 2010. 4.1, 8.2

[116] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 123–133, 2007. 2.4, 4.2.3, 4.3.1, 4.1, 4.5.2.2

[117] Sutirtha Sanyal, Adrián Cristal, Osman S. Unsal, Mateo Valero, and Sourav Roy. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *Proceedings of the 11th International Conference on High Performance Computing and Communications*, pages 171–179, 2009. 2.4, 5.2, 7.2

[118] Sutirtha Sanyal, Sourav Roy, Adrián Cristal, Osman S. Unsal, and Mateo Valero. Clock gate on abort: Towards energy-efficient hardware transactional memory. In *Proceedings of the 5th Workshop on High-Performance, Power-Aware Computing (HPPAC)*, pages 1–8, 2009. 2.4

[119] Wang Shaogang, Dan Wu, Zhengbin Pang, and Xiaodong Yang. DTM: Decoupled hardware transactional memory to support unbounded transaction and operating system. In *Proceedings of the 38th International Conference on Parallel Processing*, pages 228–236, 2009. 2.4

[120] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995. 1.3.1

[121] Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In *Proceedings of the 23rd International Conference of Supercomputing*, pages 136–146, 2009. 2.4, 6.1, 7.2

[122] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 139–150, 2008. 2.4, 5.1, 5.2, 6.2, 7.2

[123] Arrvindh Shriraman, Virendra J. Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer III, and Michael F. Spear. Hardware acceleration of software transactional memory.

In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006. 1.3.1

[124] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *Workshop on TM Workloads*. 2006. 3.1, 3.3

[125] Stephen L. Smith. Intel roadmap overview, 2008. http://download.intel.com/pressroom/kits/events/idffall_2008/SSmith_briefing_roadmap.pdf. 1.1

[126] Sthephen L. Smith. 32nm Westmere family of processors, 2009. http://download.intel.com/pressroom/kits/32nm/westmere/32nm WSM Press.pdf. 1.1

[127] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th Symposium on High-Performance Computer Architecture*, pages 2–14, 1998. 7.2

[128] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23:12–24, 1990. 2.2

[129] John Stokes. AMD aims to stay in the race with Magny-Cours 12-core CPU, 2010. http://arstechnica.com/hardware/news/2009/09/amd-makes-tradeoffs-in-upcoming-12-core-server-cpu.ars. 1.1

[130] Inc. Sun Microsystems. In *OpenSPARC T2 Core Microarchitecture Specification*. 2007. 7.2

[131] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. 30(3), 2005. 1.1, 1.1.1

[132] Fuad Tabba. *A Study of Hybrid Transactional Memory*. PhD thesis, CS Dept., Univ. of Auckland, 2011. 1.4

[133] Fuad Tabba, Andrew W. Hay, and James R. Goodman. Transactional conflict decoupling and value prediction. In *Proceedings of the 25th International Conference of Supercomputing*, pages 33–42, 2011. 2.4

[134] Fuad Tabba, Cong Wang, James R. Goodman, and Mark Moir. NZTM: Nonblocking, zero-indirection transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007. 1.3.1

[135] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal Research and Development*, 46:5–25, 2002. 1.1

[136] Rubén Titos-Gil, Manuel E. Acacio, and Jose M. Garcia. A directory-based scheme for eager conflict management in hardware transactional memory. *IEEE Transactions on Parallel and Distributed Systems (Submitted for review)*. 1.4

[137] Rubén Titos-Gil, Manuel E. Acacio, and José M. García. Characterization of conflicts in log-based transactional memory (logTM). In *16th Euromicro PDP*, pages 30–37, 2008. 1.4, 7.1

[138] Rubén Titos-Gil, Manuel E. Acacio, and José M. García. Directory-based conflict detection in hardware transactional memory. In *Proceedings of the 15th International Conference on High-Performance Computing*, pages 541–554, 2008. 1.4, 7.1

[139] Rubén Titos-Gil, Manuel E. Acacio, and José M. García. Speculation-based conflict resolution in hardware transactional memory. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*, 2009. 1.4, 7.1

[140] Rubén Titos-Gil, Anurag Negi, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Adaptive, data-centric contention management in hardware transactional memory. *IEEE Transactions on Parallel and Distributed Systems (Submitted for review)*. 1.4

[141] Rubén Titos-Gil, Anurag Negi, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Zebra : A data-centric, hybrid-policy hardware transactional memory design. In *Proceedings of the 25th International Conference of Supercomputing*, 2011. 1.4

[142] Sasa Tomic, Adrian Cristal, Osman Unsal, and Mateo Valero. Hardware transactional memory with operating system support, HTMOS. In *Proceedings of the 13th European Conference on Parallel Processing (Euro-Par)*, pages 8–17, 2007. 2.4

[143] Sasa Tomic, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd International*

Symposium on Microarchitecture, pages 145–155, 2009. 2.3.1, 2.3.2, 2.3.3, 2.4, 3.6, 5.1, 5.2, 6.1, 6.1, 6.2, 6.3, 6.3.1, 6.3.4, 6.3.4, 1, 6.1, 7.2, 7.3.1

[144] Hans Vandierendonck and Tom Mens. Averting the next software crisis. *IEEE Computer*, 44:88–90, 2011. 1.1.1

[145] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, 2008. 2.4

[146] M. M. Waliullah. Efficient partial roll-backing mechanism for transactional memory systems. *Transactions on high-performance embedded architectures and compilers*, 3:256–274, 2011. 2.4

[147] M. M. Waliullah and Per Stenstrom. Schemes for avoiding starvation in transactional memory systems. *Concurrency and Computation: Practice and Experience*, 21:859–873, 2009. 2.4

[148] M.M. Waliullah and Per Stenstrom. Reducing roll-back overhead in transactional memory systems by checkpointing conflicting accesses. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium*. 2008. 2.4

[149] M.M. Waliullah and Per Stenstrom. Classification and elimination of conflicts in transactional memory systems. Technical report, Chalmers University of Technology, TR 2010:09, Dept. of Computer Science, 2010. 5.3.1, 7.2, 7.3.1

[150] Ian Watson, Chris Kirkham, and Mikel Lujan. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 388–398, 2007. 3.3

[151] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995. 3.6.8

[152] Luke Yen. *Signatures in Transactional Memory Systems*. PhD thesis, CS Dept., Univ. of Wisconsin-Madison, 2009. 3.6.2, 4.5.1

[153] Luke Yen, Jayaram Bobba, Michael R. 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 Symposium on High-Performance Computer Architecture*, pages 261–272, 2007. 1.3.1, 2.3.1, 2.3.2, 2.3.3, 2.4, 3.5, 4.1, 4.2.2, 4.2.3, 4.2.4, 4.3.1, 4.4.5, 4.5.1, 4.1, 5.1, 5.3.1, 5.4.1, 5.2, 6.2, 6.1, 6.4, 7.1, 7.2, 7.5

[154] Luke Yen, Stark C. Draper, and Mark D. Hill. Notary: Hardware techniques to enhance signatures. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 234–245, 2008. 2.4, 4.2.3