Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2520

Advancements towards non-speculative concurrent execution of critical sections

EDUARDO JOSÉ GÓMEZ HERNÁNDEZ

ACTA UNIVERSITATIS
UPSALIENSIS
2025

ISSN 1651-6214
ISBN 978-91-513-2437-1
urn:nbn:se:uu:diva-552947

UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in Salón de Grados, Facultad de Informatica (Building 32), University of Murcia, Murcia (Spain), Tuesday, 3 June 2025 at 16:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Daniel Sorin (Duke University).

Abstract

Gómez Hernández, E. J. 2025. Advancements towards non-speculative concurrent execution of critical sections. (Avances hacia la ejecución concurrente y no especulativa de secciones críticas). *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 2520. 74 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-2437-1.

Parallel programs require, besides the cache orchestration, another mechanism that guarantees synchronization among other threads of the same program.These synchronization mechanisms will induce overheads, by slowing down certain operations and stalling threads, among many others, to comply with the requirements established by the programmer.

The thesis's objective is the efficient execution of critical sections, that is, regions of code that must be executed atomically.The most efficient method is the concurrent and non-speculative executions of these sections.To achieve this, we present the 3 steps we have taken:1) single-atomic instructions can be used to implement non-speculative critical sections, therefore, we develop an updated version of the well-known Splash benchmark suite that uses single-address atomic instructions to implement most of the critical sections (Splash-4);2) a new set of multi-address atomic instructions, and a methodology on how to efficiently implement them, that can be used for small critical sections (MADs);3) without the direct intervention of the programmer, a more generic method that limits the retries required to execute contended critical regions (CLEAR).

For an efficient evaluation of the results, we have used the most up-to-date tools possible in each case, and even, when possible, real machines instead of simulations.For the simulations, we have used the gem5 simulator, at all times performing multiple runs.The simulator has been configured to emulate, as reliably as possible, processors based on the latest intel generations.

In our first step, Splash-4, we have managed to reduce the execution time by using 64-cores by 50%, while maintaining the original structure and algorithms.In the second objective (MADs), the new atomic instructions implemented, reduce execution time by 80% compared to the classical lock mechanism, and by 60% by using a transitional memory technique similar to intel TSX, adding only 68 bytes per core.Finally, CLEAR is able to limit the number of re-executions of critical sections executed under speculative methods, increasing by 35% the number of sections that complete on the first retry, and reducing from 37% to 15% the number of sections that need to reach fallback. All this improving the execution time by 35% against an Intel TSX implementation and 23% against PowerTM.

Keywords: Computer Architecture, microarchitecture, atomic instructions, benchmark suite, non-speculative execution.

Eduardo José Gómez Hernández,

ISSN 1651-6214
ISBN 978-91-513-2437-1
URN urn:nbn:se:uu:diva-552947 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-552947)

To my family and friends

Acknowledgement

Developing a doctoral thesis is never an easy task. This is the conclusion, but also a new beginning, to my learning process. This challenge did not start 5 years ago, it started close to 8 years in the past. I had never anticipated that the number of life changes during this time would be that large. Like it is always said, “We walk on giant’s shoulders”. I would like to thank every person that has been there during these years.

First, to Raúl, my partner, my love. I found you when I was in my worst moments, I just exited several months of depression and some romantic issues. We have been distant friends for a long time ago, but we never connected that much. But, that day, in Barcelona, just for fun, it was the best day of my life. Since then, I trust you, I share all this with you. At the time of writing this thesis, we have been together for almost 3 years. I will save all the experiences we have been living during this time and the travels we have done in my heart forever. I love you. I just want to add, many thanks for teaming up with me in the biggest challenge of all, life.

To my parents (Isabel and Tomás) and my brother (Miguel). Things are hard, I know that my relationship with my brother is not as good as you always expected. I wanted to thank you for allowing me to do what I wanted to do, and to support me in the hard times. But most of all, thank you for accepting me like I am, with my oddities. Miguel, even if we have our differences, you also started a PhD after I started mine, and I hope things go well for you. Even after those fights we always have as brothers, many thanks for being there.

To my grandparents. You are no longer with us, but you have been very important in my life. Lelo, lela, thank you so much for teaching and caring about me all this time. Abuelo, abuela, even without being that close, you have been there, and I will always remember you. All of you have planted a seed in me and now it has bloomed. Many thanks for all of your time and effort. Just let me add, that I am very happy and very proud of being your grandson.

To my partner’s parents and sister. Thank you so much for hosting me on those trips to Barcelona. We have also shared good and bad moments. It was great to see one of your dreams come true (your new house). Julia, I am so glad we were able to exchange books on every trip I made, it was great to have someone close that have similar reading interests. We need to make that reading club a reality.

To the rest of my family. You are a big family, I am not kidding, but even after those dad jokes that instead of breaking the ice, they froze the surrounding air, you have been there. Usually to have a happy time. I remember very clearly those moments when you all met together, and because I was in Sweden, you sent me videos of you having fun, in part to encourage me to join you the next time. Thanks for being who you are.

To my “Morenos” and “TechCraft ESP” friends. This is a bit hard. I am sure that I would not be there if it were not for you. The day that a friend (Javi) told a teacher that I used to play chess, I met Radixan, my best friend, the first one to make a drastic change in my life. “TechCraft ESP” has influenced me so much, and I want to thank my friends Radixan and Jona for helping me there to build a dream. Also, I do not forget most of the players that have been around the server. “Morenos”, the community where I met most of my friends. With special attention to Arekusanda, Sucraris (ArisNyan), and Anveloy. You are the best friends I have ever had. Also, Zoiris, thank you for sharing your time with me in those really bad moments I had. I want to give special thanks to Matu, Micki, Nil, and all the friends in the community that have been part of my life. To all of you, thanks.

To my World of Warcraft Guild friends (Ultimo Try - Tyrande). I am sorry that I left the game, and because of that, interacting with some of you has become a bit hard. But I want to say, thank you. From our mother, Ilmatar, to our breakable tank Akodo. Of course, I will never forget all those moments, all those raids, all those screams. Also, I keep a special space in my mind for Hati, Icelus, Blind, Eretikos, Kya, Kudo, Kysle, Mainal, Pelo, Volthum, and many more that have been coming and leaving on each big update. Thank you for supporting me, but also for all those nights stuck in the raid bosses, and mythic+.

To Jose Manuel García Carrasco, thanks to you, I am here. At the third of my bachelor’s I was not expecting to be doing some research, I even had an offer to work with some friends on a topic I loved at the time. However, that email, after finishing AOC, and my first experience doing some research was the ignition spark of this engine. I know, for sure, that if it was not for that email, I would not be here right now.

To JuanMa, so many hours talking, discussing, and supporting each other. I remember very clearly that day that you went into the laboratory and told me about Alberto looking for me to work on the ECHO project. Not only that but working alongside you was a pleasure. You have taught me a lot of technical things. But technical discussions are not everything, you have taught me how to develop solutions, but also about life itself. Many thanks for all this time.

To my advisors Alberto and Stefanos. First of all, sorry for being so annoying, but many thanks for all the support you gave, and are still giving me. Those moments in Uppsala, the discussions on the whiteboard, that evening working in the HPCA PC meeting, and the messages we exchanged, of it are part of my precious memories that I will never forget. The only thing I regret was not using my time there in Uppsala better to learn much more from you. Alberto, you have introduced me to this lovely field, teaching me why most of my previous knowledge was wrong, teaching how this system works. Each discussion I had, and still have, with you is so fulfilling. Also, thank you so much for all those days we were able to talk outside the university and the academic field. Thank you so much.

To my colleagues at the University of Murcia: David, Paco, Ashkan, Sawan, Nikitin, Sebas, Agus, Nico, Pascual, Victor, Adrian, Ruben, Manolo, Ricardo, Juan Luis, Alexandra, and everyone in the department. Those stressful moments with David, those discussions and happy moments with Paco, those long discussions with Ashkan, those jokes with Sawan, those dad jokes with Agus, those HTM tantrums with Nikitin, there are so many more things that it would take pages to enumerate. I just want to say, thank you, everyone.

To my colleagues at the Uppsala Universitet: CH, Per, Chris, David, Gustaf, Anirban, Pavlos, Hassan, Oskar, Ali, Rashid, Ahmed, Shiming, Marina, Alireza, Johan, Yuan, and many more. Thank you for welcoming me and supporting me during the year I spent there. Working with you, going to midsommar, and living in Uppsala, has been a pleasure.

Paco, you have listened to me in the weak moments and given us a lot of support, many thanks for being there. Ester, you are not only a cleaner lady, but you are also a friend, I love those days talking, they have been very inspirational. The rest of the university staff, your support is invaluable.

To my students, I am not the best teacher, but it was special to see all of you growing up and where you are right now. I would like to mention Nico, Alvaro, and my 2024 SOEAR students (Pascual, Victor, Arturo, Adrian, and Emilio). Some of you are now colleagues, it was impressive to see how much you have learned and improved in these years, you have been part of keeping up this motivation engine we have inside. Now, most of you are teaching me new things.

To all of you, thank you very much for making all of this possible.

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Eduardo José Gómez-Hernández, Juan M. Cebrian, Stefanos Kaxiras, Alberto Ros, ”Splash-4: A Modern Benchmark Suite with Lock-Free Constructs”. 2022 IEEE International Symposium on Workload Characterization (IISWC 2022)

II Eduardo José Gómez-Hernández, Juan M. Cebrian, Rubén Titos-Gil, Stefanos Kaxiras, Alberto Ros, ”Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations”. 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54)

III Eduardo José Gómez-Hernández, Juan M. Cebrian, Stefanos Kaxiras, Alberto Ros, ”Bounding Speculative Execution of Atomic Regions to a Single Retry”. International Conference on Architectural Support for Programming Languages and Operating Systems 2024 (ASPLOS 2024)

Reprints were made with permission from the publishers.

Contents

List of Tables

List of Figures

List of Listings

1. Abstract

Continuing the trend of increasing the performance of individual cores is no longer an easy task. During the last decades, manufacturers have moved the focus to adding multiple cores into the same chip (Chip-Multi-Processor or CMP). This paradigm change has allowed us to continue showing attractive performance improvements on each chip generation. On ideal conditions, an application should be able to increase its performance by a factor of the number of cores available in the chip, but several limitations will prevent the scalability of these applications.

CMPs are based on the Symmetric Multi-Processing model, where cores are identical and share the same unified memory space. However, the breach between memory and computing made each core require its own local high-speed memory, a cache. During the program's execution, each cache will start storing stale data, that is, data that no longer represents the actual expected value of a memory location. Cache coherence protocols eliminate this issue by orchestrating data movement between the memory and the caches. Despite the orchestration done by the coherence protocol, multiple threads may try to read and write into the same memory location (data race), producing an incorrect result.

Parallel programs require, besides the cache orchestration, another mechanism that guarantees synchronization among other threads of the same program. These synchronization mechanisms will induce overheads, by slowing down certain operations and stalling threads, among many others, to comply with the requirements established by the programmer.

A big issue when evaluating new proposals is the lack of comparison lines that are up-to-date and representative. The most used ones are benchmark suites, but most of them were crafted over 20 years ago. While some of them try to keep up with the architectural changes of the processors, many are left behind. One of the most misrepresented capabilities is the synchronization mechanisms.

The code regions that require synchronization are known as critical sections (or atomic regions). Depending on the nature and properties of these sections, different approaches can be used to protect them. Small ones with few addresses, "fine-grain", tend to be more efficient, but much harder to develop and debug; while big ones with several addresses, "coarse-grain", are trivial, even automatized, less prone to errors, but much less performant.

Different methods have been developed over the years to facilitate high-performance coarse-grain critical sections. Hardware Transactional Memory

and Speculative Lock Elision are the most famous ones in this context. The main issue is that they introduce a lot of challenges to hardware designers while still showing doubts if they are a good approach.

The thesis's objective is the efficient execution of critical sections, that is, regions of code that must be executed atomically. The most efficient method is the concurrent and non-speculative executions of these sections. To achieve this, we present the 3 steps we have taken: 1) single-atomic instructions can be used to implement non-speculative critical sections, therefore, we develop an updated version of the well-known Splash benchmark suite that uses single-address atomic instructions to implement most of the critical sections; 2) a new set of multi-address atomic instructions, and a methodology on how to efficiently implement them, that can be used for small critical sections; 3) without the direct intervention of the programmer, a more generic method that limits the retries required to execute contended critical regions.

2. Sammanfattning

Att fortsätta trenden med att öka prestandan hos enskilda kärnor är inte längre en lätt uppgift. Under de senaste decennierna har tillverkarna fokuserat på att lägga till flera kärnor på ett enda chip (Chip-Multi-Processor eller CMP). Detta paradigmskifte har gjort det möjligt att fortsätta att visa upp attraktiva prestandaförbättringar för varje generation av chip. Idealt sett borde en applikation kunna öka sina prestanda med en faktor som är lika med antalet tillgängliga kärnor på chipet, men det finns flera begränsningar som hindrar skalbarheten hos dessa applikationer.

CMP:er bygger på den symmetriska multiprocessmodellen, där kärnorna är identiska och delar samma enhetliga minnesutrymme. Gapet mellan minne och beräkning innebär dock att varje kärna behöver ett eget lokalt höghastighetsminne, en cache. Under exekveringen av ett program kommer varje cache att börja lagra gammal data, dvs data som inte längre representerar det faktiska värde som förväntas från en minnesplats. Protokoll för cachekoherens eliminerar detta problem genom att styra dataförflyttningen mellan minnet och cacheminnet. Trots den styrning som utförs av koherensprotokollet kan flera trådar försöka läsa och skriva till samma minnesplats (datarace), vilket ger ett felaktigt resultat.

Parallella program kräver, förutom cache-orkestrering, en annan mekanism för att säkerställa synkronisering mellan andra trådar i samma program. Dessa synkroniseringsmekanismer kommer att medföra omkostnader genom att bromsa vissa operationer, stoppa trådar och mycket annat för att uppfylla de krav som programmeraren har ställt.

Ett stort problem vid utvärderingen av nya förslag är bristen på aktuella och representativa benchmarks. De mest använda är benchmark-sviter (eller testsviter), men de flesta av dem skapades för mer än 20 år sedan. Även om vissa av dem försöker hålla jämna steg med processorarkitekturens förändringar, släpar många efter. Det finns några testsviter som hänger med i tiden och som till och med är prestandanormerande för dagens kommersiella processorer. Problemet med dessa moderna testsviter är exekveringstiden. Simulatorerna, med optimeringar, kan simulera några sekunders simulering efter flera timmar. Det är inte möjligt att köra de modernaste uppsättningarna på dem, vilket tvingar fram alternativa och oftast föråldrade testsviter. En av de mest felaktigt presenterade funktionerna är dock de tidigare nämnda synkroniseringsmekanismerna.

De delar av koden som kräver synkronisering kallas kritiska avsnitt (eller atomregioner). Beroende på hur dessa sektioner ser ut och vilka egenskaper de har kan olika metoder användas för att skydda dem. Små med få adresser, finkorniga, tenderar att vara effektivare, men mycket svårare att utveckla och felsöka; medan stora med flera adresser, grovkorniga, är triviala, till och med automatiserade, mindre felbenägna, men till priset av lägre prestanda. Det är välkänt att atomiska instruktioner med en enda adress är det mest effektiva sättet att utföra en uppdatering atomiskt i förhållande till resten av applikationstrådarna, eftersom de utför denna synkronisering via hårdvara. Många programmerare har dock valt att använda lås i kritiska avsnitt eftersom de är lätta att programmera.

Under årens lopp har olika metoder utvecklats för att underlätta grova kritiska sektioner med hög genomströmning. Syftet har varit att delegera till hårdvaran att fatta bästa möjliga beslut i varje givet ögonblick. I många fall är tanken att man inte ska behöva vara inlåst, utan fortsätta framåt, och i de fall där atomicitet inte kan garanteras, ångra ändringarna och försöka igen. Transaktionsminne i hårdvara och spekulativ låsning är de mest kända i det här sammanhanget. Huvudproblemet är att de medför många utmaningar för hårdvarukonstruktörer, samtidigt som det fortfarande råder tvivel om huruvida de är en bra metod.

Syftet med avhandlingen är att på ett effektivt sätt exekvera kritiska avsnitt, dvs. områden i koden som måste exekveras atomiskt. Den mest effektiva metoden är samtidig och icke-spekulativ exekvering av dessa avsnitt. För att uppnå detta presenterar vi de tre steg vi har tagit: 1) enkla atomära instruktioner kan användas för att implementera icke-spekulativa kritiska avsnitt, så vi utvecklade en uppdaterad version av den välkända Splash-testsviten som använder atomära instruktioner med en adress för att implementera de flesta av de kritiska avsnitten; 2) en ny uppsättning atomära instruktioner med flera riktningar, och en metod för hur man implementerar dem effektivt, som kan användas för små kritiska sektioner; 3) utan direkt inblandning av programmeraren, en mer generisk metod som begränsar antalet försök som krävs för att exekvera begränsade kritiska områden.

För en effektiv utvärdering av resultaten har vi använt de mest aktuella verktygen i varje enskilt fall och även, när det varit möjligt, riktiga maskiner i stället för simuleringar. För simuleringarna har vi använt gem5-simulatorn och hela tiden utfört flera körningar och validerat de erhållna resultaten. Simulatorn har konfigurerats för att så tillförlitligt som möjligt emulera processorer baserade på de senaste intel-generationerna.

Splash-4, den nya versionen av Splash testsvit som vi har utvecklat under detta examensarbete, ersätter en betydande del av de kritiska sektioner som implementerats med hjälp av olika typer av konstruktioner. Dessa konstruktioner, som använder instruktioner med en adress, kan uttrycka högnivåoperationer som: ”Om x är mindre än det värde som lagras på adress y, skriv x på adress y”. Dessutom har vi inkluderat konstruktioner som implementerar befintliga operationer i andra atomics, men för datatyper som inte stöds, till exempel “atomically add x and what is stored at address y, and write the result to y” men för flyttal. Vi har stött på två exceptionella situationer: i den ena har vi delat upp en kritisk sektion i flera enkelriktade sektioner; i den andra har vi med hjälp av en konstruktion inom en konstruktion lyckats implementera en dubbelriktad kritisk sektion med hjälp av enkelriktade atomics. Kritiska sektioner är dock inte allt, det finns andra synkroniseringsprimitiver, t.ex. barriärer, som vi också har ersatt med alternativ som använder atominstruktioner med en adress för att minska overhead för en sådan primitiv. Denna overhead var särskilt viktig, eftersom det arbete som ska utföras mellan barriär och barriär är så litet på moderna processorer att exekveringstiden i vissa applikationer helt dominerades av väntan på andra trådar.

På grund av de stora begränsningarna hos atomära instruktioner med en adress har utvecklare i många år velat ha atomära instruktioner med flera adresser. Denna typ av instruktioner skulle göra det möjligt att implementera mer komplexa kritiska sektioner med samma effektivitet som atomiska instruktioner med en adress, med tanke på att atomiska instruktioner är det mest effektiva sättet att utföra en uppdatering i en riktning atomiskt i förhållande till resten av applikationens trådar. Detta är dock inte en enkel uppgift. När vi vill låsa flera minnesadresser måste vi vara extremt försiktiga så att vi inte hamnar i deadlocks där exekveringen inte kan fortsätta, till exempel tråd 1 har låst “x” och vill låsa “y”, medan tråd 2 har låst “y” och vill låsa “x”. För att lösa detta är den enklaste lösningen att använda en global ordning som alla följer: För att låsa “x” och “y”, lås först “x” och sedan “y”. Den här lösningen fungerar utmärkt i programvara, där resurserna antas vara tillräckliga (om än inte obegränsade). Atomiska instruktioner implementerar dock vanligtvis sina lås på cacheminnet, en fysisk struktur med ändliga och välbegränsade resurser.

Hur kan vi implementera atomära instruktioner med flera adresser om vi har begränsade resurser? Svaret vi föreslår är att vi tar hänsyn till dessa begränsningar när vi ordnar adresserna som ska blockeras. Genom att följa en serie mycket specifika steg när vi utför en låsning, och använda den lexikografiska ordningen på adresserna med hänsyn till storleken på den privata cachen, har vi lyckats låsa upp till 4 adresser samtidigt. Med hjälp av den här metoden har vi implementerat flera olika typer av atomära adressinstruktioner, inklusive den välkända compare-and-swap.

Trots genombrottet för atomära instruktioner med flera adresser är det fortfarande en stor utmaning att realisera komplexa algoritmer med dem. En av de största begränsningarna, förutom antalet adresser, är att de adresser som används är föränderliga. Atomiska instruktioner med en eller flera adresser kräver att de minnesadresser som ska låsas är kända i förväg och inte kan ändras. En kritisk sektion är inte föränderlig om de adresser som används, både vid läsning och skrivning, alltid är desamma när den exekveras flera gånger med samma processorregister. Med andra ord är minnesadresserna inte beroende av andra minnesadresser.

Det skulle vara mycket enklare om maskinvaran själv kunde hitta dessa adresser, bestämma mutabiliteten i den kritiska sektionen och utföra låsningen i rätt ordning utan att programmeraren behöver planera den kritiska sektionen och därmed bara markera början och slutet på den kritiska sektionen.

De två mekanismer som befriar programmeraren från att definiera vilken typ av låsning som ska utföras och lämnar den uppgiften till maskinvaran själv är transaktionsminne i maskinvaran och undvikande av spekulativa deadlocks. Båda förslagen uppnår vad de föresatt sig att göra - programmeraren bestämmer bara kodregionerna och hårdvaran tar hand om att försöka exekvera dem så effektivt som möjligt. Ingen av metoderna är dock garanterad att göra framsteg, vilket innebär att de kan misslyckas och måste prövas på nytt om atomiciteten bryter samman. I båda situationerna är den vanligaste lösningen en alternativ exekveringsväg som förvärvar ett block (eller en latch) som tvingar fram atomicitet, på bekostnad av prestanda.

Men tänk om vi kunde dra nytta av dessa omförsök och låsa de nödvändiga minnesadresserna för att säkerställa att exekveringen slutförs framgångsrikt utan risk för konflikter? cleAR använder den första exekveringen för att analysera det kritiska avsnittet, få alla minnesadresser och även analysera beroendena för att välja den optimala exekveringsmetoden för att begränsa antalet omförsök som behövs till 1. När det kritiska avsnittet väl är i körning släpper cleAR på konfliktförebyggande mekanismen för att få så mycket information som möjligt från den första körningen. Om exekveringen avslutas med en konflikt väljer cleAR mellan följande tre alternativ: 1) blockera alla adresser och utföra en icke-spekulativ exekvering; 2) blockera en del av adresserna och utföra en spekulativ exekvering; och 3) blockera ingenting och fortsätta med basmekanismen. Om alternativ 1 väljs begränsas antalet omförsök till 1 och det finns ingen ytterligare risk för konflikter. Om den kritiska sektionen inte uppfyller alla krav för alternativ 1 kan vi ändå öka chanserna till framsteg genom att blockera de adresser som är benägna att orsaka konflikter (alternativ 2). Som ett sista alternativ kommer cleAR att delegera till den spekulativa mekanism som används för att hantera de kritiska avsnitt där den inte kan tillämpa någon av ovanstående mekanismer.

I vårt första steg, Splash-4, har vi lyckats minska exekveringstiden genom att använda 64 kärnor med 50%, samtidigt som vi behållit den ursprungliga strukturen och algoritmerna. I det andra målet (MADs) minskar de nya atomära instruktionerna exekveringstiden med 80% jämfört med den klassiska låsmekanismen och med 60% när man använder en teknik för övergångsminne som liknar intel TSX och bara lägger till 68 byte per kärna. Slutligen kan cleAR begränsa antalet omkörningar av kritiska avsnitt som exekveras med spekulativa metoder, öka antalet avsnitt som slutförs vid första omkörningen med 35% och minska antalet avsnitt som måste nå fallback från 37% till 15%. Allt detta förbättrade exekveringstiden med 35% jämfört med en Intel TSX-liknande implementation och 23% jämfört med PowerTM.

3. Resumen

Continuar la tendencia de aumentar el rendimiento de los núcleos individuales ya no es tarea fácil. En las últimas décadas, los fabricantes se han centrado en añadir varios núcleos a un mismo chip (Chip-Multi-Processor o CMP). Este cambio de paradigma ha permitido seguir mostrando atractivas mejoras de rendimiento en cada generación de chips. En condiciones ideales, una aplicación debería poder aumentar su rendimiento en un factor equivalente al número de núcleos disponibles en el chip, pero existen varias limitaciones que impedirán la escalabilidad de estas aplicaciones.

Los CMP se basan en el modelo de multiprocesamiento simétrico, en el que los núcleos son idénticos y comparten el mismo espacio de memoria unificado. Sin embargo, la brecha entre memoria y computación hace que cada núcleo necesite su propia memoria local de alta velocidad, una caché. Durante la ejecución de un programa, cada caché empezará a almacenar datos obsoletos, es decir, datos que ya no representan el valor real esperado de una ubicación de memoria. Los protocolos de coherencia de caché eliminan este problema orquestando el movimiento de datos entre la memoria y las cachés. A pesar de la orquestación realizada por el protocolo de coherencia, varios hilos pueden intentar leer y escribir en la misma posición de memoria (carrera de datos), produciendo un resultado incorrecto.

Los programas paralelos requieren, además de la orquestación de la caché, otro mecanismo que garantice la sincronización entre otros hilos del mismo programa. Estos mecanismos de sincronización inducirán sobrecargas, al ralentizar ciertas operaciones, detener hilos, entre muchos otros, para cumplir con los requisitos establecidos por el programador.

Un gran problema a la hora de evaluar nuevas propuestas es la falta de líneas de comparación actualizadas y representativas. La más utilizada son las suites de benchmarks (o conjunto de pruebas), pero la mayoría de ellas fueron creadas hace más de 20 años. Aunque algunas de ellas intentan mantenerse al día con los cambios arquitectónicos de los procesadores, muchas se quedan atrás. Existen alunos conjuntos de pruebas que se mantienen al día, e incluso son los referentes del rendimiento en los procesadores comerciales actuales. El problema de estos conjuntos modernos es el tiempo de ejecución. Los simuladores, con optimizaciones puedes llegar a simular unos pocos segundos de simulación tras varias horas. Resulta inviable ejecutar los conjuntos más modernos en ellos, forzando al uso de conjuntos de pruebas alternativos, y mayormente desactualizados. Aún, así, una de las capacidades más tergiversadas son, los ya mencionados, mecanismos de sincronización.

Las regiones de código que requieren sincronización se conocen como secciones críticas (o regiones atómicas). Dependiendo de la naturaleza y propiedades de estas secciones, se pueden utilizar diferentes enfoques para protegerlas. Las pequeñas con pocas direcciones, grano fino, tienden a ser más eficientes, pero mucho más difíciles de desarrollar y depurar; mientras que las grandes con varias direcciones, grano grueso, son triviales, incluso automatizadas, menos propensas a errores, pero a costa de un menor rendimiento. Es bien conocido que las instrucciones atómicas de dirección única son las manera más eficiente de realizar una actualización atómicamente respecto al resto de hilos de la aplicación, ya que realizan esta sincronización por hardware. Sin embargo, muchos programadores han optado por la secciones críticas usando bloqueos por su sencillez a la hora de programar.

A lo largo de los años se han desarrollado diferentes métodos para facilitar las secciones críticas de grano grueso de alto rendimiento. El objetivo era delegar en el hardware para que tomase la decisión más acertada posible en cada momento. En muchos casos la idea es no tomar ningún bloqueo, continuar hacia delante, y en el caso de que no se pueda garantizar la atomicidad, deshacer los cambios y volver a intentar. La memoria transaccional por hardware y la elisión especulativa de bloqueos son los más conocidos en este contexto. El principal problema es que introducen muchos retos para los diseñadores de hardware, al tiempo que siguen mostrando dudas sobre si son un buen enfoque.

El objetivo de la tesis es la ejecución eficiente de secciones críticas, es decir, regiones de código que deben ejecutarse atómicamente. El método más eficiente es la ejecución concurrente y no especulativa de estas secciones. Para lograrlo, presentamos los 3 pasos que hemos dado: 1) se pueden utilizar instrucciones atómicas simples para implementar secciones críticas no especulativas, por lo que desarrollamos una versión actualizada del conocido conjunto de pruebas Splash que utiliza instrucciones atómicas de dirección única para implementar la mayoría de las secciones críticas; 2) un nuevo conjunto de instrucciones atómicas multiple dirección, y una metodología sobre cómo implementarlas eficientemente, que pueden utilizarse para secciones críticas pequeñas; 3) sin intervención directa del programador, un método más genérico que limita los reintentos necesarios para ejecutar regiones críticas contendidas.

Para una eficiente evaluación de los resultados, hemos utilizado las herramientas mas actualizadas que nos han sido posibles en cada caso, e incluso, cuando ha sido posible, máquinas reales en lugar de simulaciones. Para las simulaciones, hemos usado el simulador gem5, en todo momento realizando multiples ejecuciones, y validando los resultados obtenidos. El simulador ha sido configurado para emular, lo más fiablemente posible, procesadores basados en las últimas generaciones de intel.

Splash-4, la nueva versión del conjunto de pruebas Splash que hemos desarrollado durante esta tesis, reemplaza una parte significativa secciones criticas implementadas usando distintos tipos de constructos. Estos constructos, usando instrucciones de dirección única, son capaces de expresar operaciones de alto nivel como: “atómicamente si x es menor que el valor almacenado en la dirección y, escribe x en la dirección y”. Adicionalmente, hemos incluido constructos que implementan operaciones existentes en otros atómicos, pero para tipos de datos no soportados como “atómicamente suma x and lo almacenado en la dirección y, y escribe el resultado en y” pero para punto flotante. Nos hemos econtrado con dos situaciones excepcionales donde: en una de ellas hemos dividido una sección crítica es multiples secciones de una única dirección; en la otra hemos conseguido, usando un constructo dentro de otro constructo, implementar una sección crítica de dos direcciones usando atómicos de dirección única. Sin embargo, las secciones críticas no lo son todo, existen otras primitivas de sincronización, como las barreras, las cuales también hemos reemplazado con alternativas que usan instrucciones atómicas de dirección única para reducir la sobrecarga de dicha primitiva. Esta sobrecarga era especialmente importante, debido a que en procesadores modernos, la cantidad de trabajo a realizar entre barrera y barrera es tan pequeño, que en algunas aplicaciones el tiempo de ejecución estaba completamente dominado por esperar a otros hilos.

Debido a las grandes limitaciones de las instrucciones atómicas de unica dirección, los atómicos de multiple dirección han sido el deseo de los desarrolladores durante muchos años. Este tipo de instrucciones permitiría implementar secciones críticas más complejas con los misma eficiencia que las instrucciones atómicas de dirección única, recordemos que las instrucciones atómicas son la manera más eficiente de realizar una actualización en una dirección de manera atómica respecto al resto the hilos de la aplicación. Sin embargo, realizar esto no es tarea sencilla. Siempre que queramos bloquear multiples direcciones de memoria hay que tener un cuidado extremo con no caer en puntos muertos donde la ejecución no puede continuar, por ejemplo el hilo 1 tiene bloqueado “x”, y quiere bloquear “y”, mientras el hilo 2 tiene bloqueado “y” y quiere bloquear “x”. Para solucionar esto, la solución más sencilla es usar un orden global que todos sigan: Para bloquar “x” e “y”, primero hay que bloquear “x” y luego “y”. Esta solución funciona perfectamente en software, donde los recursos se asumen que son suficientes (sino ilimitados). Sin embargo, las instrucciones atómicas suelen implementar sus bloqueos a nivel de caché, una estructura física con unos recursos finitos y bien restringidos.

¿Como podemos implementar instrucciones atómicas de multiple dirección si tenemos recursos limitados? La respuesta que proponemos es: teniendo en cuenta esas limitaciones a la hora de ordenar las direcciones a bloquear. Siguiendo una serie de pasos muy especificos a la hora de realizar un bloqueo, y usando el orden lexicografico de las direcciones teniendo en cuenta el tamaño de la caché privada, hemos conseguido bloquear hasta 4 direcciones simultaneamente. Usando este método, hemos implementado instrucciones atómicas de multiple dirección de distinto tipo, entre ellos el conocido compare-and-swap (comparar e intercambiar).

A pesar del gran paso de las instrucciones atómicas de multiple dirección, realizar algoritmos complejos con ellos sigue siendo un reto importante. Uno de los mayores limitantes, además del número de direcciones, es la mutabilidad de las direcciones usadas. Las instrucciones atómicas, de unica o multiple dirección, requieren que las direcciones de memoria a bloquear se conozcan de ante mano, no sean mutables. Una sección crítica no es mutable si al ejecutarla multiples veces con los mismos registros del procesador, las direcciones accedidas, tanto en lectura como en escritura, son siempre las mismas. O dicho de otra manera, las direcciones de memoria no dependen de otras direcciones de memoria.

Sería mucho más sencillo si el propio hardware fuera capaz de encontrar esas direcciones, determinar la mutabilidad de la sección crítica, y realizar el bloqueo en el orden correcto sin requerir que el programador planifique la sección crítica, y por tanto solo marcando el inicio y fin de la misma.

Los dos mecanismos que alivian a los programadores de definir el tipo de bloqueo a realizar, y dejando esa tarea al propio hardware son memoria transaccional por hardware y la elisión especulativa de bloqueos. Ambas propuestas consiguen lo que se proponen, el programador solo determina las regiones de código y el hardware se encarga de intentar ejecutarlas de la manera más eficiente que sea posible. Sin embargo, ninguna de las propuestas tiene garantía de progreso, eso quiere decir que pueden fallar y tener que reintentar si se rompe la atomicidad. En ambas situaciones, la solución más empleada es un camino de ejecución altenativo que adquiere un bloque (o cerrojo) que fuerza la atomicidad, a costa del rendimiento.

Pero, ¿y si pudiesemos aprovechar esos reintentos y bloquear las direcciones de memoria necesarias para garantizar que se complete la ejecución de manera satisfactoria sin posibilidad de conflictos? cleAR usa esa primera ejecución para analizar la sección crítica, obtener todas las direcciones de memoria, e incluso analizar las dependencias, para seleccionar el método de ejecución óptimo para limitar el número de reintentos necesarios a 1. Una vez la sección crítica está en ejecución, cleAR relaja el mecanismo de prevención de conflictos para poder obtener toda la información que sea posible de la primera ejecución. Si una vez terminada, la ejecución termina con un conflicto, cleAR elige entre las tres siguientes opciones: 1) bloquear todas las direcciones y realizar una ejecución no especulativa; 2) bloquear parte de las direcciones y realizar una ejecución especulativa; y 3) no bloquear nada y continuar con el mecanismo base. En el caso de seleccionar la opción 1, el número de reintentos está limitado a 1, y no hay más posibilidad de conflictos. Sin embargo, si la sección crítica no cumple con todos los requerimientos para la opción 1, aún así podemos aumentar las posibilidades de progreso bloqueando aquellas direcciones que sean propensas a conflictos (opción 2). Como última opción, cleAR delegará en el mecanismo especulativo que se esté usando para manejar aquellas secciones críticas a las que no sea capaz de aplicar ninguno de los mecanismos anteriores.

En nuestro primer paso, Splash-4, hemos conseguido reducir el tiempo de ejecución al usar 64-cores en un 50%, manteniendo en todo momento la estructura y algoritmos originales. En el segundo objectivo (MADs), las nuevas instrucciones atomicas implementadas, reducen un 80% el tiempo de ejecución al compararse con el mecanismo de locks clásico, y un 60% al usar una tecnica de memoria transacional similar a intel TSX, añadiendo solo 68 bytes por core. Por último, cleAR, es capaz de limitar la cantidad de rejecuciones de las secciones criticas ejecutadas bajo métodos especulativos, aumentando en un 35% la cantidad de secciones que se completan en el primer reintento, y reduciendo del 37% al 15% la cantidad de secciones que requieren llegar al fallback. Todo esto mejorando el tiempo de ejecución en un 35% contra una implementación tipo Intel TSX y un 23% contra PowerTM.

4. Introduction

Processors' manufacturers found that increasing the performance of a core was no longer an easy task. While single-core performance is still on manufacturers' focus, most of them spend a lot of resources adding multiple cores on a processor chip (CMP or Chip-Multi-Processor). Compared to the small performance gains by improving a core, adding more cores allows continuing to show attractive performance improvements on each chip generation. CMPs, because of their huge performance, have established Symmetric Multi-Processing (SMP) as today's standard for high-performance computing. SMP exploits the concept of having multiple identical cores connected to the same main memory and devices. Ideally, an application that uses all the available cores would be able to increase the performance by a factor of the number of cores used. However, most applications require arbitration where the threads running on each core require synchronization, that is, sharing data or waiting for others to catch up.

With the rapid performance increase of the cores, a huge latency breach was established with the memory. This potential bottleneck forced manufacturers to integrate local caches on each core to reduce the cost of accessing memory by keeping local copies of data blocks that will be reused by the core without having to access memory again. As the execution progresses, potential stale copies may appear, leading to inconsistencies of the same address accessed by different cores returning different values. The cache coherence protocol oversees orchestrating permissions, by giving or revoking them on each cache, to keep the data coherent between cores.

One simple solution, to keep the coherence of the data, is to notify all the caches on each operation performed (snooping). Snooping protocols, while easier to implement, induce a huge penalty that prevents the inclusion of more cores per chip, leading to their replacement with more complex, directory-based cache coherence models. The directory is a structure shared by all cores that map on which cache is each cache block, allowing to revoke or grant permissions with limited communication to the nodes that are needed. Despite the directory taking care of the cacheline orchestration, data races are an issue. Multiple threads may try to read the same data to later do an update (Shared Memory Model). Depending on the specific application and the runtime order of the threads, the result will vary.

For example, in a system with two threads that try to perform an addition on a shared memory location (`x`): `a = load x; a = a + 1; write a in x`; `x` initially holds the value 125. Thread 1 reads `x` (125) and saves the value in its local memory a, then thread 2 reads x (125) and also saves the values in its local memory b[1]. The execution continues, thread 1 and thread 2 perform the additions (1: a = 126 = 125 + 1, 2: b = 126 = 125 + 1). When storing the result, thread 1 will invalidate the cacheline in thread 2, and then write (overwriting 125 with 126). Thread 2 will follow by doing the invalidation and storing the data (overwriting 126 with 126). At the start, we were expecting that with two threads increasing the value to 125, the result would be 127, but instead, in this execution, it is 126.

This scenario shows the need for a mechanism that elides threads from reading or writing when synchronization is required. Atomicity is the property that is not conservated here. In the previous example, one of the threads should wait until the other one has finished reading, adding, and writing the result. Waiting means that the program could run slower as some of the threads are waiting for others to reach a certain point, so programmers should minimize the amount of synchronization points to increase the concurrency and potentially the performance of the application.

To fix the scenario, the section of the code that reads, modifies, and writes back the data needs to be encapsulated into a critical section. Critical sections guarantee that the code is executed, or appears to be executed, isolated from the rest of the system. That is, no other core can read the intermediate state of the data nor write to it. To encapsulate the code, a lock can be set up to prevent others from executing that part of the code at the same time. The previous example can be modified as: `lock; a = load x; a = a + 1; write a in x; unlock`. However, while this solution works, for simple scenarios with only one variable, there is a much better approach, atomic instructions. Again, the previous example can be modified as: `atomic_fetch_and_increment x;`. Being hardware instructions, they are much faster than locks and can be implemented in clever ways. For example, a common atomic instruction implementation is by locking the cacheline that the data belongs, preventing other cores from reading or writing to it without/minor modifications to the core or the coherence protocol.

4.1 Opportunities

During the research and development of a new system, a comparison line needs to be drawn to evaluate how the proposed changes affect performance, energy consumption, or any of the other metrics of interest that are being looked to be improved. The standard current solution is the use of benchmark suites. Benchmark suites are a collection of applications and inputs that try to evaluate how a specific system would perform in a real scenario.

[1] Note that to read the value thread 2 does not need to invalidate or send any message to thread 1.

The main issue is how old these benchmark suites are. From the most used ones (SPLASH [46, 52, 44], SPEC [47], PARSEC [53], STAMP [38], ...) are crafted over 20 years ago. Some of them try to keep up and continue receiving updates (like SPEC). Systems, compilers, programmers; all of them change over time, and quite quickly something that was thought to be the best solution to a problem, now becomes the worst. An example of this was the algorithm to exchange values between two variables. An algorithm called xor_swap was used in the past, but with more clever compilers and with new instructions (xchg) the xor_swap is even worse than using a third variable to perform the swap. Benchmark suites cannot keep up with these trends and always target the most recent solutions. Despite the issue that a solution can be better on one platform and worse on another.

One of the most misrepresented capabilities in benchmarks is the synchronization of parallel programs. Not all parallel programs are "embarrassingly parallel" [24], a lot of applications require some kind of orchestration between their work units. These points of synchronization are commonly known as *critical sections/critical regions/atomic regions*. Depending on the specific characteristics of the required kind of synchronization, different solutions may be applied. 1) Small critical regions that modify a limited set of addresses are "fine-grain". They tend to be quite efficient, due to their small size, but they become a real challenge to be developed. 2) Big critical regions with large sets of addresses are "coarse-grain". By default, they are quite easy to use, even automated in some cases, but with the cost of performance.

Over 10 years ago, the two main developing languages in the world, C and C++, introduced atomic instructions into their standard [28, 29]. Atomic instructions have been demonstrated to be the most efficient way of updating a variable atomically, and they started to be available in any platform, with some minor differences [27, 26, 5, 51, 23]. From their introduction in the programming standard, they were no longer limited to be used only in libraries or in operating system operations (syscall) to manage atomicity, they can be used by mainstream developers.

"Fine-grain" critical sections require a deep understanding of the data dependency between the different threads in the system. By carefully thinking about the problem that is being solved, and with the use of the blazing new standardized atomic instructions, "fine-grain" critical sections can be implemented. However, they can operate on a single-address. For simple scenarios, atomic instructions directly replace critical sections completely. But, for the complex ones, some programmers have developed tricks to implement them with these restrictions, but most developers just prefer to avoid that hustle and just use mutex lock instead.

Due to its easiness of development and being less error-prone, "coarse-grain" critical regions are used more often. Research and industry have tried to develop solutions that make "coarse-grain" critical regions to improve performance. Speculative Lock Elision (SLE) and Hardware Transactional Memory

(HTM) are the main two approaches to this problem. However, they introduce a lot more challenges and hardware complexity which shows doubts if they are a good solution or not. Both approaches execute speculatively until a conflict (violation of the atomicity) is found, introducing the concept of retries. First, the waste of time and energy executing speculatively and then reverting the changes hoping that in the next retry, there is no conflict. In many cases, the amount of time required, retrying, to complete the section goes beyond the time spent if the section was serialized with traditional mutexes. Second, the pollution of structures and predictors. Depending on the implementation of the speculative execution and the recovery mechanism, after a retry, certain structures may not be restored, such as the branch predictor, the return-address-stack, and even low cache levels. In certain cases, this “pollution” helps the section to be executed correctly in the next retry [45], but if the execution path changes or the section is retried multiple times, the accuracy of these, performance-critical, structures may decline.

4.2 Contributions

Contribution 1 (Splash-4): We propose an updated version of the Splash benchmark suite that manages to exploit the hardware synchronization capabilities using current up-to-date synchronization instructions. This new version introduces: 1) a different barrier synchronization primitive optimized for short waits, and 2) lock-free alternative versions of the majority of the critical sections using atomic instructions and lock-free constructs. Splash-4, executed about 50% faster in real hardware, allows the hardware designers to unveil the real causes that prevent the applications from scaling even more in contemporary.

Paper Ⓘ: Eduardo José Gómez-Hernández, Juan M. Cebrian, Stefanos Kaxiras, Alberto Ros, ”Splash-4: A Modern Benchmark Suite with Lock-Free Constructs”. 2022 IEEE International Symposium on Workload Characterization (IISWC 2022)

Contribution 2 (Multi-Address Atomics): we propose a new set of operations, and a deadlock-free locking methodology, that can perform atomic updates on multiple addresses relying solely on the coherence protocol and a predefined locking order. In contrast to speculative methodologies, this new set of operations follows the idea of atomic instructions, being non-speculative, therefore not requiring undoing work. Each cacheline required is locked with no extra communication with any other core, just the normal coherence protocol messages. Then the operation is executed atomically, and when completed, the cachelines are unlocked allowing other cores to read and write to/from them. This methodology allows us to implement Multi-Compare_And_Swap and Multi-Atomic_Fetch_And_Add, among many others, simplifying the development of these critical sections and increasing their performance in high-contended scenarios.

Paper (II): Eduardo José Gómez-Hernández, Juan M. Cebrian, Rubén Titos-Gil, Stefanos Kaxiras, Alberto Ros, ”Efficient, Distributed, and Non-Speculative Multi-Address Atomic Operations”. 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-54)

Contribution 3 (Cache Locking of Atomic Regions): We propose a technique that bounds the number of retries required to complete a coarse-grain critical region while keeping the concurrency of the execution. Instead of just executing speculatively, the structure, addresses, and dependencies of the critical section can be gathered during the retries. Therefore, using this information, a non-deadlocking order can be established to concurrently execute the section without speculation by locking the data at the cache level. We noticed that most of the time, in a critical section that is retried with the speculative solutions, the address set where it performs its operations is the same as during the first retry. Therefore, by locking that set of addresses, and preventing external access, we can guarantee that the section will be complete without having to retry more times. We go one step further, by completely disabling the speculative method and executing non-speculative if the address set is immutable, that is, the address set cannot be changed between retries. This can be seen as converting the critical section into an atomic after the first failed retry.

Paper (III): Eduardo José Gómez-Hernández, Juan M. Cebrian, Stefanos Kaxiras, Alberto Ros, ”Bounding Speculative Execution of Atomic Regions to a Single Retry”. International Conference on Architectural Support for Programming Languages and Operating Systems 2024 (ASPLOS 2024)

Figure 4.1 shows a summary of the contributions of the thesis. Contribution 1 solves 1-address immutable critical sections in a non-speculative way by using the atomic instructions available in current architectures. Contribution 2 solves up to 4-addresses immutable critical sections with the use of our new multi-address atomic instructions. Contribution 3 solves any size immutable and likely-immutable critical sections by analyzing the section in the first speculative execution and converting it into its non-speculative version. Future research is open to tackle mutable critical sections.

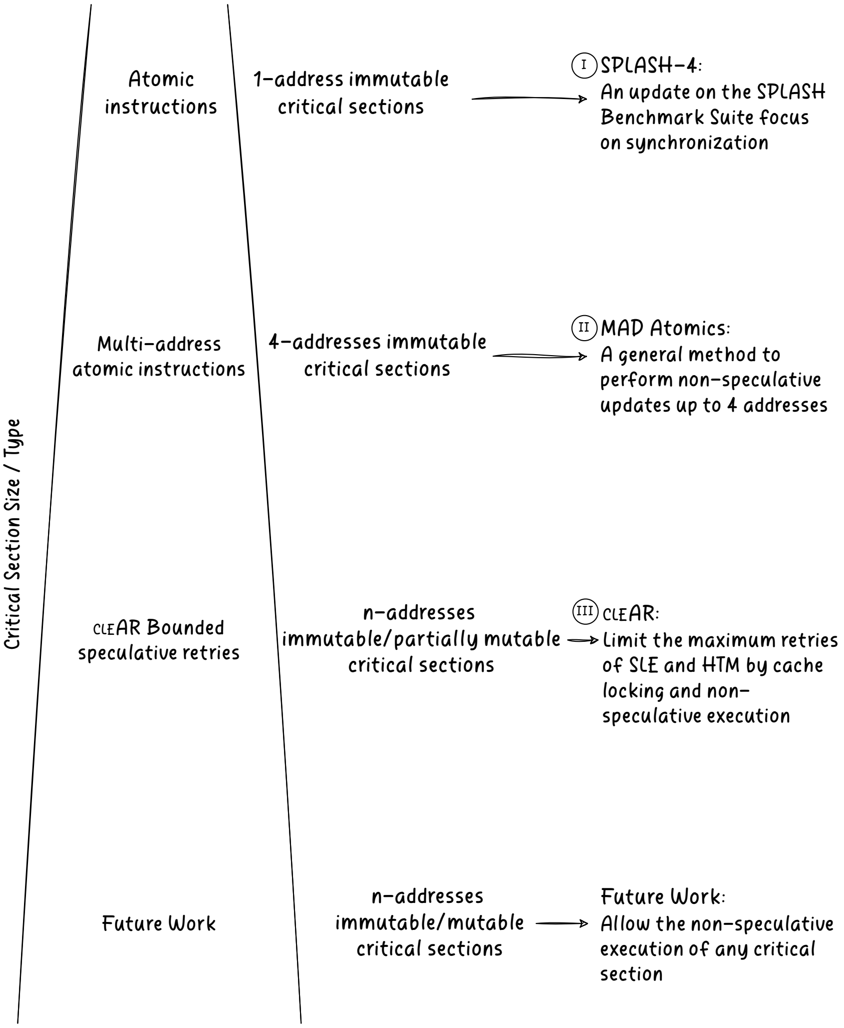

Critical Section Size / Type
Atomic instructions
1-address immutable critical sections
I SPLASH-4: An update on the SPLASH Benchmark Suite focus on synchronization
Multi-address atomic instructions
4-addresses immutable critical sections
II MAD Atomics: A general method to perform non-speculative updates up to 4 addresses
clEAR Bounded speculative retries
n-addresses immutable/partially mutable critical sections
III clEAR: Limit the maximum retries of SLE and HTM by cache locking and non-speculative execution
Future Work
n-addresses immutable/mutable critical sections
Future Work: Allow the non-speculative execution of any critical section

Figure 4.1. A high-level graph of each contribution organized by papers.

5. Background

In this chapter, we introduce the main concepts needed to fully comprehend the rest of this thesis fully. Section 5.1 introduces the cache coherence protocols, this would be the main concept that we will use to build the rest of the thesis. Then Sections 5.2, 5.3, and 5.4, exploring the synchronization mechanisms and benchmark suites, are the main bases for our Paper I. While Sections 5.4 and 5.5 show the current state the atomic instructions, which is the base of our Paper II. Lastly, Sections 5.6 and 5.7 reach the current state of speculative and non-speculative approaches, our main target of Paper III.

5.1 Cache coherence

Current CMPs consist of multiple cores with each code including its own private cache, usually followed by a shared structure. Before reading and writing any data to the memory, the block that holds the requested data is commonly placed into the cache hierarchy. To keep the data coherent between the different private caches, the cache coherence protocol establishes a set of rules, messages, and procedures to follow on each operation to guarantee that every core sees the same address with the same value, despite having multiple copies in separate places.

There are multiple different cache configurations and different protocols depending on the power and performance target, physical size, and capabilities desired, among many other variables. The two main variables to consider are 1) which unit gets each message and 2) which states and messages are used to implement the protocol.

In current high-performance processors, distributed approaches minimize broadcasting and generate point-to-point messages, and directory-based cache coherence protocols tend to dominate the market. Other solutions like snooping protocols can be used in systems where it is not viable to maintain a directory-based system, or where it is not critical, as they are much simpler to implement. Other proposals exist, and some of them have even been implemented in real hardware, but at least in HPC, the directory is the king of them.

There are many more intermediate states in the protocol, but here we mention the main ones that define the permissions that the core has over a specific cacheline. The simplest coherence protocol is `MI`. It only has 2 states, modified (**M**) and invalid (**I**). While **I** represents that the cacheline is not present in the cache, the state **M** allows the cacheline to be read and written.

The main issue of this simple protocol is that no cacheline can be available for reading by two different cores at the same time, which is pretty common. To make the protocol handle this scenario, the state shared (**S**) can be added. In the MSI protocol, a cacheline that is only read goes to state **S** instead of **M**, saving the latter one for writing (and reading after writing).

An issue arises in MSI when a cacheline in the state **M** is downgraded to **S** because another core wants to read it, it has to be written back to the previous cache-level (or main memory) before completing the transaction. To mitigate this issue, the state owner (**O**) appears. **O** allows a cacheline that is still dirty in the cache, to be forwarded to another core that is transitioning to the state **S**. Therefore, MOSI was born with a lot of new opportunities.

But, another concern is reached when a core that is the only one that has read a cacheline wants to update it. It is the unique core that has the cacheline, and still needs to check for other copies in the system. The state exclusive (**E**) solves this issue by keeping track of unique copies in the cache hierarchy. Because MSI was very used because of its simplicity, both MESI and MOESI became very popular in modern hardware implementations.

Lastly, another state can be introduced to MESI to mitigate a similar issue that **O** but for multiple clean shared copies. If a cacheline is shared by multiple cores, and no one has the ownership, all of them are in state **S**. Therefore, when another core wants to read the data, all the caches in the state **S** will answer the request, generating a lot of unnecessary traffic in the network. The state forward (**F**) solves this issue by taking responsibility for answering the request for read requests. The MESIF protocol was introduced, and used by some manufacturers for a long period of time [25, 49].

Both states **O** and **F** reduce the accesses to the Lower Level Cache (LLC) by obtaining the requested data from a remote cache instead of the previous cache level.

Changes from one state to another are triggered by messages initiated by the core. Current protocols include a lot of internal messages for managing intermediate states, prefetching, and many other features, but the main ones are the ones that alternate between the main states of the coherence protocol. A core will, typically, read or write into a cacheline, then, depending on the state, the cache will answer or forward the message to another structure. GETS and GETM will bring the data to the cache prepared to be read or written, respectively. A cacheline might be already present in the state **S** when performing a write, then a UPGRADE will be triggered. In the case that the cacheline is present in another cache, and its state is not compatible with the current request (i.e. performing a write and the cacheline is shared in another cache), an INVALIDATION is sent to the corresponding cache. If the cacheline that receives an INVALIDATION is dirty, that is it has been written, it will trigger a WRITEBACK to write the data back to the lower cache levels. At any moment, when an operation is completed, an ACK is sent back to inform that it can continue (if the message also contains data it sends a DATA instead).

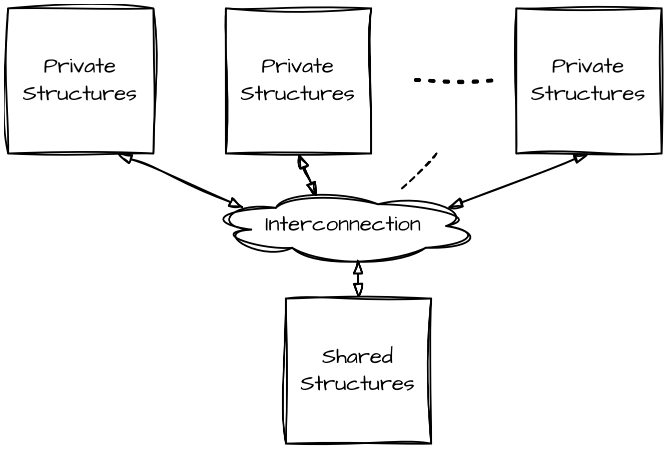


Figure 5.1. High-level vision of a high-performance architecture with private and shared structures.

As a high-level overview (Figure 5.1), each set of private structures (including caches) is connected through an interconnection network, but it is also connected to the shared structures. Please note that shared structures can be, and usually they are, distributed among the system in slices.

In directory-based protocols, the directory is considered a shared structure that holds the mapping of where is each cacheline (typically as a compressed list of sharers). Directories are commonly implemented in a cache-like structure, sometimes embedded inside the LLC. On each request that requires communication with other caches, first, the directory is checked to decide the destination of the message, and then it is sent to each destination.

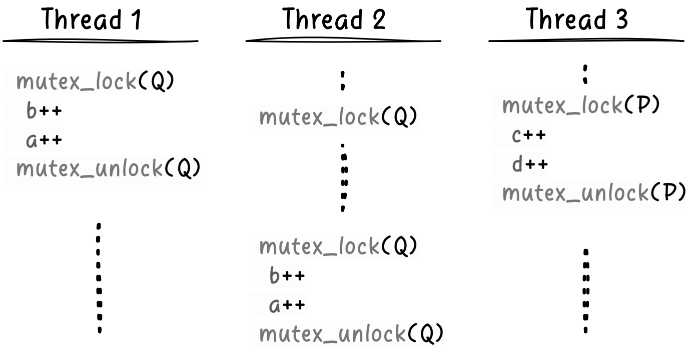
5.2 Parallel benchmark suites

SPLASH (or Standford ParalleL Applications for SHared memory), while still in use, was developed over 20 years ago [46]. Splash-2 was the first major benchmark suite with the purpose of demonstrating shared-memory scalability [52]. Splash-2 has been demonstrated to be an essential instrument in the development of today's shared memory multiprocessors. Subsequent updates (Splash-2X) fixed several coding errors, and performance bugs, and tried to update the benchmarks to the standards of the time [53]. A particularly important addition to extending the life of the benchmarks was to support and include bigger input sizes, allowing to mitigate the cost of the synchronization and stress newer hardware while being the same benchmarks. The next version, Splash-3 [44], was the biggest update on the benchmarks, exposing data races and updating the code to comply with the C standard of memory access (Data Race Free). After both updates, the Splash Benchmark suite has a tough time when compiled under the current C compilers and standards, introducing even more logic and performance bugs. While still relevant, the state of these benchmarks no longer represents the current situation of processors.

PARSEC (Princeton Application Repository for Shared mEmory Computers) was released with the same focus as SPLASH, evaluate chip-multiprocessors with shared memory [9, 8, 53]. However, it has stopped being developed over 10 years ago.

5.3 Synchronization primitives

Coherence protocols avoid having stale data copies in caches; however, they do not introduce any synchronization properties for the data read in the processor. The main reason is that the data inside the registers or the instructions are beyond the scope of the coherence protocol.



```
mutex_lock(Q)
 b++
 a++
mutex_unlock(Q)
```

```
mutex_lock(Q)

mutex_lock(Q)
 b++
 a++
mutex_unlock(Q)
```

```
mutex_lock(P)
 c++
 d++
mutex_unlock(P)
```

Figure 5.2. Mutex example

Synchronizing between different threads of the same application is quite expensive, since while synchronizing (waiting for another core to finish) no useful work can be performed. The cost for synchronization increases with the amount of threads as the contention increases more cores will be forced to wait. Therefore, using the correct synchronization primitives in the right way is essential for performance. There is a vast set of different primitives that can be used, but each of them has its own issues and oddities. The two most common synchronization mechanisms used are `mutexes` (or locks) and `barriers`.

Listing 5.1. Locking two variables

```
mutex_lock   (mutexes[i])    // Acquire Lock i
mutex_lock   (mutexes[j])    // Acquire Lock j
do_something (i,j)           // Operate with i and j
mutex_unlock (mutexes[j])    // Release Lock j
mutex_unlock (mutexes[i])    // Release Lock i
```

Mutexes are the most used synchronization primitive. They allow one thread to execute the contents of the atomic region and make others wait (Figure 5.2). While fast, when there is little contention, they become an issue under high contention scenarios, showing a large overhead. The protected atomic region can be, virtually, as big as the programmer wants (coarse-grain), but large atomic regions are prone to be more contended, as the amount of time the lock is taken is bigger. Protecting coarse-grain critical regions is trivial, but threads will be serialized more than needed. Fine-grain critical regions typically require more than one mutex per region. The main issue of multiple mutexes, per critical section, is the risk of deadlocks due to cycles between threads. However, synchronizing the order is impractical for a huge number of cores. A global order that guarantees no cycles while not requiring any communication allow each independent core to make decisions on which addresses to lock in which order without waiting other cores to acknowledge in the order.

Listing 5.2. Djistra locking

```
// Copy values to sort locks
li = i
lj = j
// Check if j is smaller than i
if (lj < li) {
  t = lj
  lj = li
  li = t
}
// Perform the critical section
mutex_lock   (mutexes[li])  // Acquire lock li (the smallest ↩
    one)
mutex_lock   (mutexes[lj])  // Acquire lock lj (the biggest ↩
    one)
do_something (i,j)          // Operate with the original ↩
    values
mutex_unlock (mutexes[lj])  // Release lock lj
mutex_unlock (mutexes[li])  // Release lock li
```

Barriers stop the execution of any thread that reaches this primitive until a specific amount (or a set) of them have reached the primitive (Figure 5.3). A barrier allows slower threads to catch up with faster ones to continue executing together. One of the main uses of a barrier is to wait for a specific phase (or computation) of an application to finish.

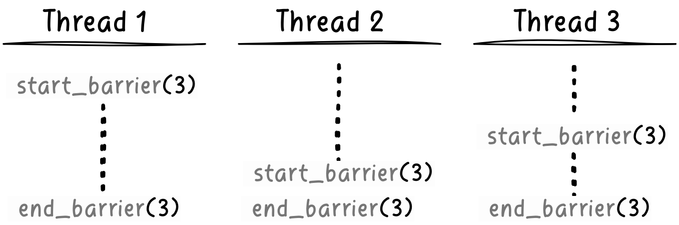


Figure 5.3. Barrier example

5.4 (Single-Address) atomic instructions

When simple operations atomically access only one memory location, instead of mutexes, atomic read-modify-write (atomic RMW) operations can be used. For example, atomic_fetch_and_add, atomic_fetch_and_sub, atomic_fetch_and_inc, and atomic_fetch_and_xor are some atomic read-modify-write instructions that perform atomic addition, subtraction, increment, and xor respectively. Atomic RMW operations are hardware operations that update a memory location while preventing other threads from seeing or changing the intermediate value. Because being implemented completely on hardware, they are the most efficient way of performing those updates.

Despite the coherence protocol does not control the synchronization points with the processor, it controls when a cache block can leave a cache and enter another different one. Atomic instructions are implemented, in modern systems, with the idea of delegating the locking of the data from a synchronization primitive in software to the coherence protocol. With this premise, an atomic instruction will notify the coherence protocol that it wants to load some data, that later will write, and no other core should be able to access it until the write is performed, informing that the data should be unlocked.

Listing 5.3. While&CAS structure

```
/* CAS */
var readValue = LOAD(ptr);
var oldValue;
var newValue;
do {
  oldValue = readValue;
  newValue = new;
} while ((readValue = CAS(ptr, oldValue, newValue)) != ↩
    oldValue);
```

Among the possible atomic operations that can be available in an ISA, the CAS (compare and swap) is the most used one. Its first benefit is being data agnostic (if it fits into a register). This property allows clever programmers to create a “While&CAS construct” (Listing 5.3). Its main issue is that it requires retries. The “Compare and Swap” part is atomic, but generating the value to be written is outside the instruction scope. This small window allows other cores to interfere. But that scenario means that another thread has been written, so, at least one thread makes progress.

Non-blocking (lock-free) algorithms heavily rely on atomic operations to do most of the synchronization. Designing algorithms and data structures with solely atomic operations is a notoriously challenging task, deadlock-prone, and extremely hard to debug and verify.

5.5 Multi-address atomic instructions

Implementing the data structures used by applications (or benchmarks) in a highly efficient way is a big challenge. One of the options is to use atomic operations but guaranteeing both efficiency and correctness is not an easy task. Many researchers claimed that if atomic operations were able to access and update at least two addresses, their work would be much better and simpler [20, 37].

The Motorola 68000 series (1979) was the first processor to implement the DCAS (Double Compare And Swap) instruction [39]. However, due to its limitations and problems, the instruction was commonly avoided in production. While DCAS is often seen as the solution to lock-free programming, it is not perfect. Due to its validation complexity and its usefulness compared with CAS, the community should move to stronger alternatives [15].

More recently, in 2017, Patel et. al presented an implementation of arbitrarily large CAS operation on hardware [40]. The proposal introduces a new structure that reorders locks, the MCAS table, on each entry stores the address, the old value, and the new value. Entries are inserted in the table with an MTS instruction, followed by an MCAS instruction when all entries are set up. The first issue with this approach is that a context change, interruption, or any other important event can interfere with the setup of the table. In this case, there is not a clear mechanism to recover the table contents after the intermission. The second main issue is not considering the limited resources of the hardware. They propose two different versions: 1) MCAS-BASE, a simple approach that can end up deadlocking by resource limitations; 2) MCAS-OPT, which deploys a back-off alternative when an invalidation is received, working as a speculative approach with retries.

Like the software lock situation, locking multiple addresses requires a predetermined non-deadlocking order. While in software locks, due to the view of unlimited resources, a simple order (address order [14]) solves the issue, in hardware locks there is a limited set of resources that can be occupied without reaching a deadlock scenario.

Ros and Kaxiras [43] developed a methodology to order the set of addresses in a non-deadlocking order. They establish a subset of the address location as the lexicographical order. This value can be used to reorder the addresses in a non-deadlocking manner when there are no conflicts. In the case of a conflict (two addresses creating a cycle), the set of addresses to be locked (group), is split into two different sets, avoiding the possible deadlock. While in that work, this is a valid solution, when dealing with addresses set by the programmer, no such split is possible; all addresses must be locked together in the same group to guarantee the correctness of the program.

5.6 Speculative execution of critical sections

While the previous techniques are quite interesting, a part of the community was focused on giving the hardware the capability of managing the isolation of any kind of critical region. These regions are not only easier to program but much less error-prone. The two main approaches, that have been hardware implementations, are Speculative Lock Elision (SLE) [41, 42] and Transactional Memory (TM) [22]. Despite being similar, SLE focuses on eliding the lock and in case of conflicts, acquiring it; while TM changes the paradigm by defining regions of codes called transactions that will be executed atomically, as well as possible.

SLE, and its hardware variant Hardware Lock Elision (HLE), elides executing the lock function of the critical section and keeping the changes in speculation until the unlock function is found. When an intermission from another thread is found, the speculative state is discarded, and the lock is taken.

Following a similar idea, Transactional Memory, in its hardware version (Hardware Transactional Memory or HTM), includes new instructions to delimit when a critical section starts and ends (transaction). When the transaction starts, the speculative state is saved in the case of an abort is required. A return value is generated to determine the current state of the transaction after the start, to determine what to do in the case of an abort. In contrast with SLE, after an abort, the resolution policy is software-made. This software approach allows us to establish: the number of retries before taking a lock, which kind of lock, and heuristics to reduce future conflicts, among many other features to resolve the conflict, hopefully, in the most efficient way.

Many different HTM proposals have been presented to reduce the deficient performance obtained in simple best-effort implementations. The one that has obtained most attention is Power-TM [13], driven by its simplicity. When a transaction aborts, Power-TM sets one transaction into power mode. In power mode, the transaction has more priority over non-power ones. To implement this behavior, an extra bit is sent to the memory packages to indicate that the transaction is running in power mode. Additionally, no transaction can abort a power mode transaction due to memory conflicts.

5.7 Non-speculative and concurrent execution

Speculative approaches have the benefit of being, traditionally, agnostic to the number of addresses. Nevertheless, its main drawback of requiring retries without any certainty of even completing, that is, there is no guarantee of progressing in a finite number of steps, makes them require an alternative execution path that guarantees progress even if it hurts performance and wastes energy. While atomics and lock-free programming seem non-speculative, some constructs like “While&CAS” require retries. However, these retries are guaranteed to be limited, as for any construct that fails, there is at least one that succeeds. These constructs are in the middle of both approaches because they do not speculate the read and write of the data like HTM and SLE, but they require retries. In this thesis, we consider that this approach is non-speculative as the data accessed by the critical section and its modification is not done under speculation, instead, they are protected by the atomic constraints of the atomic instructions.

Several approaches [7, 50, 6, 2, 48] tried to do innovative things to improve the execution of critical section, and without speculation, but they fail to allow multiple threads to make progress at the same time.

To the best of our knowledge, only the combination of lock-free constructs with atomic instructions (either 1-address or multiple addresses) are the only non-speculative and concurrent methods for non-speculative and concurrent execution of critical sections. However, fine-grain locking can also be considered non-speculative and concurrent, if well implemented, and can be applied to certain critical sections.

As mentioned our goal is to achieve non-speculative and concurrent execution of critical sections.

6. Methodology

6.1 Simulator

During the development of this thesis, different methodologies have been used to develop and evalute our proposals. Our main development platform is the microarchitectural full-system simulator gem5 [10, 35].

Gem5 allows simulating custom CPU architectures with a quite large community supporting and contributing to it. In this thesis, different versions of the gem5 simulation have been used, trying to keep up to date with the updates in order to simulate as faithfully as possible the existing actual systems. In all our works, several modifications have been used to improve the simulator and mimic as much as possible the actual hardware. Execution and issue latencies are modeled as measured on real hardware by Fog [18]. Inside the simulator, that we run in full-system, we run Ubuntu 16.04 with the Linux kernel 4.9.4. The cache hierarchy is implemented in detail using Ruby. The interconnection network is modeled using Garnet [3]. The rest of the parameters are shown in Table 6.1.

6.2 Real Machine

The simulator is a great tool to evaluate proposals that cannot be tested in actual hardware. However, our first contribution can, and should, be tested in actual machines. Therefore, we use an AMD EPYC 7702P CPU with 64 cores @ 2GHz, 32KB L1-D and L1-I, 512 KB L2, and 16 MB L3 caches. The system is running Ubuntu 18.04 with the Linux Kernel 5.4.0.

6.3 Benchmarks

The selection of benchmarks plays a crucial role in the research study. It is tentative to use the same benchmark as everyone uses, but no benchmark can stress all parts of the system while matching the computational patterns of all kinds of workloads. In our case, we have used a combination of Splash benchmarks [46, 52, 53, 44] for our first contribution, while mainly using mcas-benchmarks [30] for our second contribution and third contribution, but the last contribution also includes the STAMP benchmarks [38].

The Splash benchmark suite contains 14 benchmarks made from 11 different applications and methods. Barnes and FMM are three-dimensional and

Gem5 Version	Paper I: Gem5-20, Paper II: Gem5-19, Paper II: Gem5-21
Core	32-core out-of-order Icelake-like (Skylake-like in Paper II). Fetch/Decode/Rename width: 5 instructions per cycle; Issue/-Commit width: 10 instructions per cycle; ROB: 352 uops (224 in Paper II); LQ: 128 entries (72 in Paper II); SQ: 72 entries (56 in Paper II); RAS: 64 entries (16 in Paper I and II); Branch predictor: LTAGE (TAGE_SC_L_64K in Paper I and II)
L1 Cache	Instructions: 32KiB, 8-way, 1-cycle access latency; Data: 48KiB (32KiB in Paper II), 12-way (8-way in Paper II), 1-cycle access latency.
L2 Cache	512KiB (256KiB in Paper II), 8-way, 10-cycle access latency.
L3 Cache	4MiB (2MiB in Paper II), 16-way, 45-cycle access latency.
Memory	80-cycle access latency.
Coherence	Three-level MESI protocol interconnected with a crossbar. Directory has 800% coverage.
HTM	Intel TSX-like requester wins, and Power-TM. Best of 1 to 10 retries before taking the fallback lock.

Table 6.1. *gem5 configuration*

two-dimensional n-body simulations. Cholesky, Radix and LU are matrix factorization algorithms. FFT is a fast fourier transformation. Ocean is a large ocean simulation. Water is a force molecular simulation of water molecules. Volrend is a 3D rotating volume renderer. Radiosity perform a light distribution equilibrium. Raytrace is a 3D raytrace renderer. We use the default inputs in our evaluation, also commonly known as simsmall.

The mcas-benchmarks contain multiple data structure algorithms implemented using different synchronization methods (mutex lock, lock-free, and MCAS). Arrayswap [19] exchanges values between two positions in a big array. Binary Search Tree (BST) [40, 23], Deque [15, 32, 11, 23, 33], Hashmap [21, 12], Queue [40, 23], Stack [23], and Sorted List [23], implement the corresponding data structure. MWObject [16, 17] is a synthetic application that contains the maximum contention possible by implementing 4 increments to 4 different variables in the same cacheline. We perform 10^5 operations within the data structures.

STAMP is a collection of transactional applications. Bayes is a bayesian network structure learning benchmark. Genome simulates a gene sequencing benchmark. Intruder simulates a network intrusion detection. Kmeans performs the k-means clustering with a set of data. Labyrinth solves the maze routing problem. SSCA2 is a graph kernel. Vacation is a travel agency reservation system. Yada executes a delaunay mesh refinement. We run STAMP with the recommended simmedium inputs.

Additionally, we include Bitcoin, an application that performs bitcoin credit computation on wallets extracted from the blockchain [31]. In parallel, it executes the movement of money from one wallet to another, so each owner can see their current balance. Bitcoin runs 10^4 wallet transactions per thread.

6.4 Methods

The simulator is bootstrapped once, keeping the rest of the system deterministic. This is because, in full system mode, gem5 booting Linux is able to obtain different outcomes on each boot, therefore to keep our results coherent between reruns, the system is bootstrapped only once.

We run all the applications completely from start to end. However, upon reaching the start of the region of interest (ROI), defined by the original developers of the benchmarks, the stats are reset. When the ROI is completed, the simulation dumps all the statistics.

To include some variability in the executions, a sleep timer is added just before running the benchmark. This sleep timer is different for each consecutive run. Note that running the same sleep timer multiple times obtains the exact same results.

6.5 Metrics

To measure the impact of the proposals, we include a performance metric, an energy metric, and some impact metrics that depend on the specific proposal target.

For performance metrics, it is already well known that IPC (Instruction Per Cycle) does not work well for multithreaded applications [4]. The main issue for avoiding IPC is its lack of capturing the real progress of the application. Synchronization points tend to have spin-loops or library calls that end up in the operating system to check if it should wake up to make progress. These instructions add to the IPC statistic, but the application does not make any progress. And, because the amount of time spent in these synchronization points depends on other threads, the statistic becomes unusable. Our preferred metric for performance is normalized execution time. It gives actual information about how much time the proposals are saving over the original, baseline, version.

Energy metrics are very hard to capture, first, the tool used (McPAT [34, 1]) only supports up to 22nm of integration, which is quite far from the 10nm, 7nm, 5nm, 3nm, and 2nm currently being used in industry. In our work, we use two different energy metrics depending on which one is better for each context. One of them is normalized committed instructions. The idea was that if the introduced changes are small enough that their energy can be ignored, and the rest of the statistics are very similar, the energy saved should be somewhat proportional to the committed instructions reduced. Our second energy metric is actual dynamic and static energy. The catch here is that we use the energy metric from an Intel processor, then manually map each simulator stat with each energy value, and then apply the same formulas that McPAT. With this approach, the numbers obtained seem to be more realistic than the raw McPAT. However, while this approach is enough to show some energy savings, energy should still be normalized as the raw numbers do not represent real energy consumption.

Each proposal requires its own metric numbers to show its improvement in the different fields it targets. We summarize the rest of them:

- A scalability/speedup measure compared to 1 core to see how much improvement is obtained by increasing the number of cores.
- An Intel Top-Down view to show the reason for the stalls in the pipeline.
- A synchronization time overhead breakdown.
- The number of aborts per committed transaction.
- A breakdown based on the type of aborts in transactions.
- A breakdown based on the type of commit in transactions.

7. Splash-4

7.1 Introduction

Traditional benchmarks used in computer architecture research tend to be outdated and not being updated as the hardware evolves. Splash-2, while still in use, its code goes back to the early 90s, the computation and the hardware have drastically changed since then. While an updated version (Splash-3) exists, it still does not exploit newer hardware capabilities introduced in the last decades.

The first step towards the objective of this thesis is the single-address atomic instructions and their integration into the current benchmarks. We need to understand and check if they are truly better performant and why they are not being used.

In Section 7.2 we introduce the main issue on these old benchmark suites. Then, in Section 7.3 we explore deeper by proposing changes to the current synchronization methods exploiting single-address atomic instructions. Lastly, in Section 7.4 we present the results obtained both in real hardware and in the simulator infrastructure.

7.2 Synchronization Issues

Splash-3 represents a solid step towards adhering to the C standard, while also fixing data races and several performance issues, but it is not enough to keep the benchmarks relevant. In Splash-3, big critical sections are protected with one mutex, they are coarse grain sections. While coarse grain critical sections are not an issue by themselves, when they are contended, they introduce a noticeable overhead due to threads waiting for mutexes to be freed, even when there are no data conflicts. This issue is partially already mitigated by using multiple locks per critical section in some cases.

Current processors are so blazing fast that the execution time between barriers is negligible. As the Splash benchmark suite is developed with an intensive use of barriers, most of the time is spent waiting for other threads to reach the barrier.

7.3 Efficient synchronization

Our first approach to the problem is to do a characterization of the applications. The analysis is done using a binary instrumentation tool implemented using Intel PIN [36]. The tool was able to accurately detect the synchronization points and report information like the number of times executed (Table 7.1).

To better understand the dependencies between critical sections, the interaction of the threads, and the data movement, we developed the concept of barrier groups. A barrier group is a region of code and all the critical sections and synchronization points inside the region. With this concept, we can evaluate the dependencies and the requirements of atomicity per synchronization point.

Application	**Barriers**		**Critical Sections**					
			Mutex		C11		CAExch	
	St	Dyn	St	Dyn	St	Dyn	St	Dyn
Splash-3								
Barnes	6	19	10	2140090	0	0	0	0
Cholesky	4	6	8	95182	0	0	0	0
Fft	7	9	1	64	0	0	0	0
Fmm	13	36	38	488126	0	0	0	0
Lu	5	69	1	64	0	0	0	0
Lu-NonContiguous	5	69	1	64	0	0	0	0
Ocean	20	902	4	13312	0	0	0	0
Ocean-NonContiguous	19	872	4	13312	0	0	0	0
Radiosity	5	12	48	3861123	0	0	0	0
Radix	7	17	1	64	0	0	0	0
Raytrace	3	3	8	355184	0	0	0	0
Volrend	15	146	12	311164	0	0	0	0
Water-Nsquared	9	22	8	68672	0	0	0	0
Water-Spatial	9	22	6	1217	0	0	0	0

Table 7.1. *Splash-3 Synchronization: 64 cores, default entries, each execution may vary the numbers a bit, numbers obtained with our pin tool. St(atic) is the number of instances present in the code, while Dyn(amic) is the number of instances executed in runtime.*

During the characterization, we noticed that the main cause of stalling the threads is waiting on barriers. Stalling on barriers can be caused by thread unbalance. However, in most of the applications, each thread is doing the same work, so it should not be that predominant. But default pthread barriers can put threads to sleep, and the wakeup time for each thread can be drastically different, generating an unbalanced execution.

```
local_sense = !local_sense;
if (atomic_fetch_sub(&(count), 1) == 1) {
    count = threads;
  STORE(sense, local_sense);
} else {
  do {} while (LOAD(sense) != local_sense);
}
```

Listing 7.1. Sense-reversing barrier

For this reason, we propose changing the current barrier implementation with the centralized sense-reversing barrier (Figure 7.1). By using atomic instructions and a spin-loop that traps each thread, the centralized sense-reversing barrier is able to wake up the threads quicker, it is optimized for short waits.

Replacing a critical section with an atomic instruction (or an atomic construct) is not a trivial task. All the sections that interact with the data of the replaced section, directly or indirectly, must also be replaced, as there is no atomicity protection guarantee between sections protected with different methodologies.

Listing 7.2. taskman.c.in 134

```
// Original
LOCK(global->pbar_lock);
global->pbar_count--;
UNLOCK(global->pbar_lock);

// Pre-Computer memory location
unsigned* count = &(global->pbar_count);
LOCK(global->pbar_lock);
*count = *count - 1;
UNLOCK(global->pbar_lock);

// LockFree
FETCH_AND_SUB(global->pbar_count, 1);
```

Some critical sections have direct analogous C11 atomics (Listing 7.2). Therefore, if the data is not used in other critical sections in the same barrier group, or the dependent ones can be also replaced with C11 atomics, the critical section can be replaced with its analogous C11 atomic. In the Listing 7.2 example, the critical section performs a subtraction on one memory location, that can be calculated in advance. This operation is known as `atomic_fetch_and_sub`, but for portability, we have created a macro that hides all these operations, so they can be replaced with alternatives in the case of not having the required operation in the target platform.

Listing 7.3. multi.c.in 90

```
// Original
LOCK(locks->error_lock)
if (local_err > multi->err_multi) {
  multi->err_multi = local_err;
}
UNLOCK(locks->error_lock)

// LockFree
double expected = LOAD(multi->err_multi);
do {
  if (local_err <= expected) break;
} while (!CAExch(multi->err_multi, expected, local_err));
```

A different scenario is those critical sections that only read or modify a shared location, but they do not have an analogous C11 atomic, even if the atomic instruction exists on some platforms. Listing 7.3 shows this scenario, in Ocean, that in some architectures it can be replaced with an atomic min, however, this atomic min needs to be a double floating point atomic operation. No primitive implements this behavior, so it must be emulated using a Compare-And-Swap or a Compare-Exchange construct. This construct, also known as the While&CAS construct, iterates in a loop all the operations that need to be done atomically, and stores the data in the memory only if the data already present at the start of the iteration has not changed. This approach suffers from the ABA problem but considering all the possible outcomes of any interception, mainly because of the nature of the ‘min’ operation, this problem does not affect the correctness of the execution.

Listing 7.4. interf.c.in 156 & interf.c.in 167 & interf.c.in 179

```
// Original
ALOCK(gl->MolLock, mol % MAXLCKS);
for ( dir = XDIR; dir  <= ZDIR; dir++) {
  temp_p = VAR[mol].F[DEST][dir];
  temp_p[H1] += PFORCES[ProcID][mol][dir][H1];
  temp_p[O]  += PFORCES[ProcID][mol][dir][O];
  temp_p[H2] += PFORCES[ProcID][mol][dir][H2];
}
AULOCK(gl->MolLock, mol % MAXLCKS);

// LockFree
for ( dir = XDIR; dir  <= ZDIR; dir++) {
  FETCH_AND_ADD_DOUBLE(&(VAR[mol].F[DEST][dir][H1]), PFORCES[↩
      ProcID][mol][dir][H1]);
  FETCH_AND_ADD_DOUBLE(&(VAR[mol].F[DEST][dir][O]), PFORCES[↩
      ProcID][mol][dir][O]);
  FETCH_AND_ADD_DOUBLE(&(VAR[mol].F[DEST][dir][H2]), PFORCES[↩
      ProcID][mol][dir][H2]);
}
```

Some critical sections are grouped together without any apparent reason, besides simplicity or reducing the cost of locking and unlocking. That is, the data is not cross-referenced in any other critical section. This is the case of Listing 7.4. In this section, after an incredibly careful and intense study, we have determined that each iteration of the loop is independent (partially because of the used operation ’addition’). But even more, in the barrier group that covers this (and others) critical section, there is no cross-use of the different data. After a thorough test, we split the fig section with a loop into a loop of 3 different lock-free constructs that are independent.

7.4 Results

After all the proposed changes, we run the characterization again to see the evolution of the benchmarks (Table 7.2). There is no change in the number of barriers, as we have changed only the implementation, but not the amount. However, the critical sections are in a completely different situation, as at least 1 critical section has been replaced in all benchmarks, the critical section that sets the thread ids.

			Critical Sections					
Application	**Barriers**		Mutex		C11		CAExch	
	St	Dyn	St	Dyn	St	Dyn	St	Dyn
Splash-4								
Barnes	6	19	9	2140056	1	64	0	0
Cholesky	4	6	6	68979	1	64	1	26238
Fft	7	9	0	0	1	64	0	0
Fmm	13	36	26	442838	1	64	1	5
Lu	5	69	0	0	1	64	0	0
Lu-NonContiguous	5	69	0	0	1	64	0	0
Ocean	20	902	0	0	1	64	3	13248
Ocean-NonContiguous	19	872	0	0	1	64	3	13248
Radiosity	5	12	36	3478298	3	50497	3	6394618
Radix	7	17	0	0	1	64	0	0
Raytrace	3	3	2	252498	5	92455	1	8816
Volrend	15	146	1	1536	8	245519	0	0
Water-Nsquared	9	22	0	0	1	64	15	608384
Water-Spatial	9	22	0	0	1	64	6	1280

Table 7.2. *Splash-4 Synchronization: 64 cores, default entries, each execution may vary the numbers a bit, numbers obtained with our pin tool. St(atic) is the number of instances present in the code, while Dyn(amic) is the number of instances executed in runtime.*

In general, Figure 7.1 shows the execution time reduction from the proposed changes when executed in a real machine. The main benefit comes from changing the implementation of the barriers, showing a reduction of ∼40%. The next improvement is the replacement of many critical sections with their lock-free alternatives, introducing a ∼11% improvement. When both techniques are applied together, the total reduction goes to ∼52%.

With the proposed changes we run a scalability study on the same machine to better evaluate how the changes affect the performance when running with different numbers of cores with respect to their 1-core serial execution. In Figure 7.2 we clearly see how some applications change drastically. Ocean-Cont and Ocean-NonContiguous increase the scalability from 4 to 32 cores. But also, raytrace exhibits linear scalability compared with the original one that stopped at 4 cores. Most of the applications see an increase in scalability,

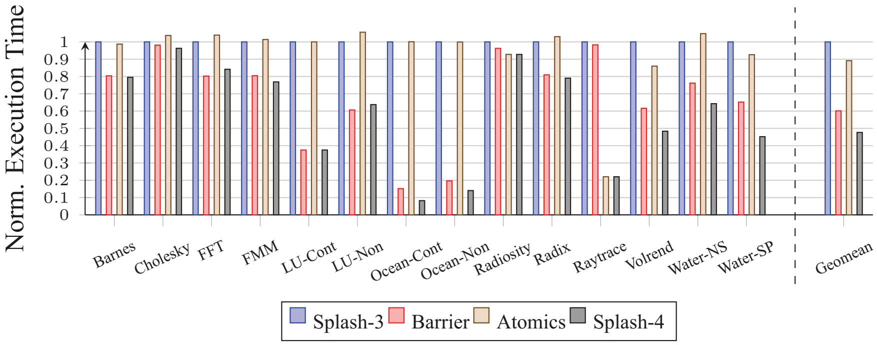


Figure 7.1. Execution time when upgrading Splash-3 *barriers*, *atomics* and both - *Splash-4*- (64-threads on AMD Epyc 7702P)

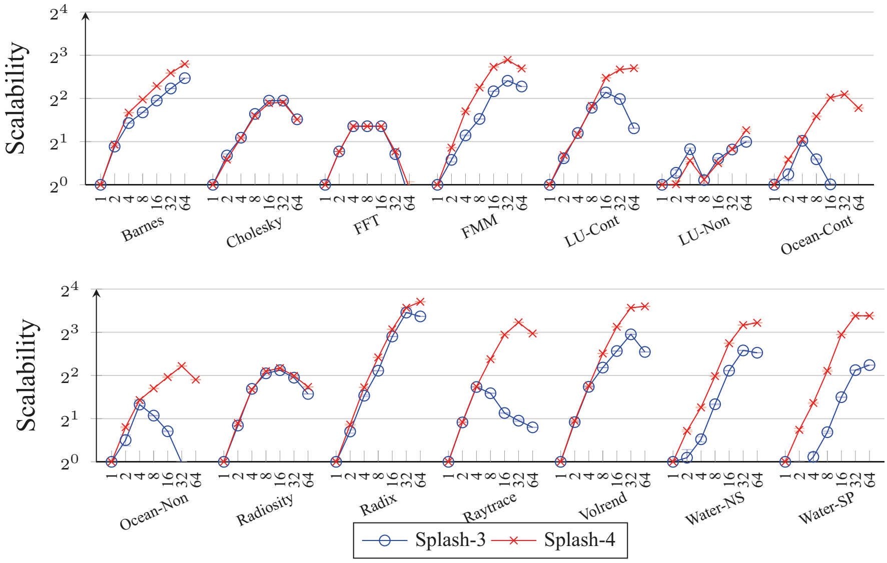


Figure 7.2. Splash-3 vs Splash-4 Scalability on AMD Epyc 7702P

with the exception of Cholesky, FFT, and Radiosity which seem to be stuck in their original performance without any significant regression.

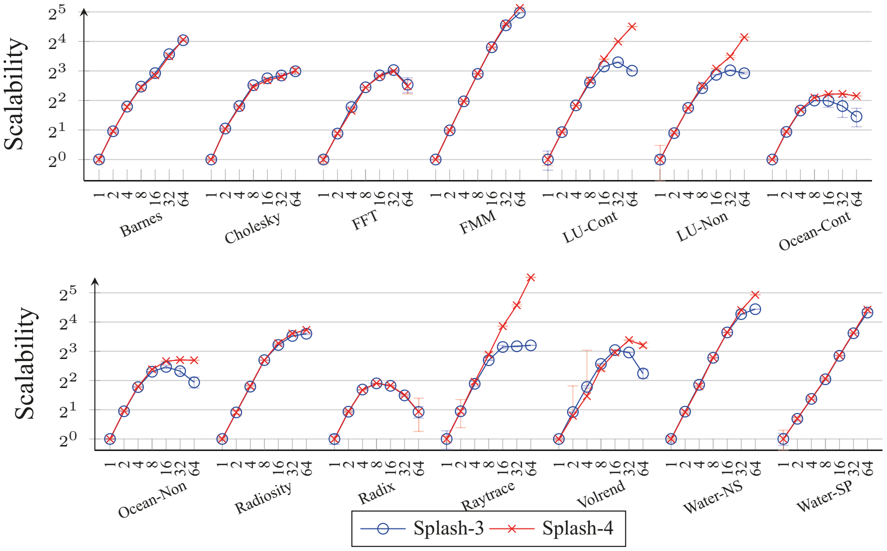


Figure 7.3. Splash-3 vs Splash-4 Scalability in simulated Intel Ice Lake (gem5-20)

Later, we decide to run the same comparison in the simulator, to see how the scalability changes in this context. In Figure 7.3, we observe only some small improvements in both versions of LU and Ocean, but also raytrace and volrend, without any perceptible improvement in others. This behavior may be driven by inaccuracy in the simulator making the applications behave better than they really should.

8. Hardware Multi-Address Atomics

8.1 Introduction

Atomic operations are the most efficient way of updating a memory location in isolation from the rest of the system. However, most of the platforms, are limited to a single memory location, restricting their usage to implement other more complex synchronization primitives.

After understanding the limits of single-address atomic instructions, and without any deadlock-free high-performance alternative, in our second step towards non-speculative advancements, we develop a new set of atomic instructions that can operate on multiple addresses at the same time.

In Section 8.2 we propose a deadlock-free locking order that considers the hardware capabilities of the system. Then, in Section 8.3 we present an example of how to use the locking order with atomic instructions to develop a set of multi-address atomic instructions. Lastly, in Section 8.4 we compare our new atomic instructions against mutex locks, lock-free, and intel TSX versions of the applications.

8.2 Locking order

Dijkstra address locking order can be expressed as an iterative process using Equation 8.1. '`L`' is a set of locked addresses and a new address '`u`' needs to be locked. If the new address is bigger than any of the addresses in the already locked set, then the new address can be locked safely. While Dijkstra's methodology seems to work, it does not take into account any other restriction from the system.

$$\{L\} + u = \{L + u\} \Longleftarrow \forall l \in \{L\} : l < u \tag{8.1}$$

To illustrate the issue, we are going to go through different scenarios adding more details on each step. We start with 4 different addresses `66`, `15`, `4`, and `192`. In address order, locking is done in the following sequence: `4`, `15`, `66` and `192`. Now, let us add a restriction, locked addresses must fit in a structure. In this example, the structure has 4 slots, and they are directly mapped from the address, that is, a subset of the bits determines which slot it takes.

In Figure 8.1, we see that address `192` conflicts with address `4` in the structure. Despite having space in the structure, the way addresses are mapped into

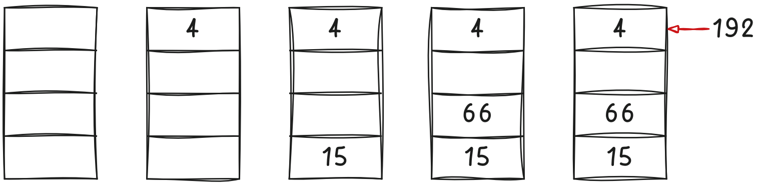


Figure 8.1. Directly mapped structure causing a conflict that prevents locking all addresses despite having enough space.

it generates a conflict that prevents the set of addresses from being locked together. This structure mimics having a private cache in the memory hierarchy that needs to contain all the memory addresses to maintain the atomicity of the full set of addresses by preventing other cores from reading or writing into them.

This "mapping" is what we define as `lexicographical order` (`lex order`). In Ros and Kaxiras' research [43], they defined `lex order` a bit differently, but this definition is more powerful when adding more restrictions. The main point of difference is that the original definition was intended to maximize the number of addresses, but with the possibility stop adding addresses into the protected set at any moment.

However, when dealing with locks defined by the program, they cannot be split into different groups, all of them need to be protected as a group.

Two different points must be considered when ordering the locks: 1) private structures and 2) shared structures. In private structures, the main issue is to guarantee that there is enough space to have all the addresses available at the same time. In most architectures, private caches have at least 4-ways associativity. Later, we define why 4 is enough, but for now, we can guarantee that if the number of addresses is 4 or less, the addresses can be locked, even if in the worst case, they all end up in the same cache set. Therefore, we limit the number of simultaneous addresses to four.

If for some reason, any shared structure (inclusive-LLC, directory, ...) has less associativity, this number should be reduced to match it. However, it is safe to assume that in any modern high-performance system, this is not the case.

This restriction is not enough, shared structures resources must be required by multiple cores to perform an atomic operation, therefore locking the system.

Defining the `lex order` to match the scenario in Figure 8.2, the risk of deadlock can be detected before encountering it. In this configuration, addressees `5`, `53`, `81`, `9`, `57` and `113` have the same `lex order` (this scenario is known as a `lex conflict`). In the case depicted, both cores want to lock two more addresses to complete their atomic group, but neither of them can allocate space for the missing conflicting entry. This definition of the `lex`

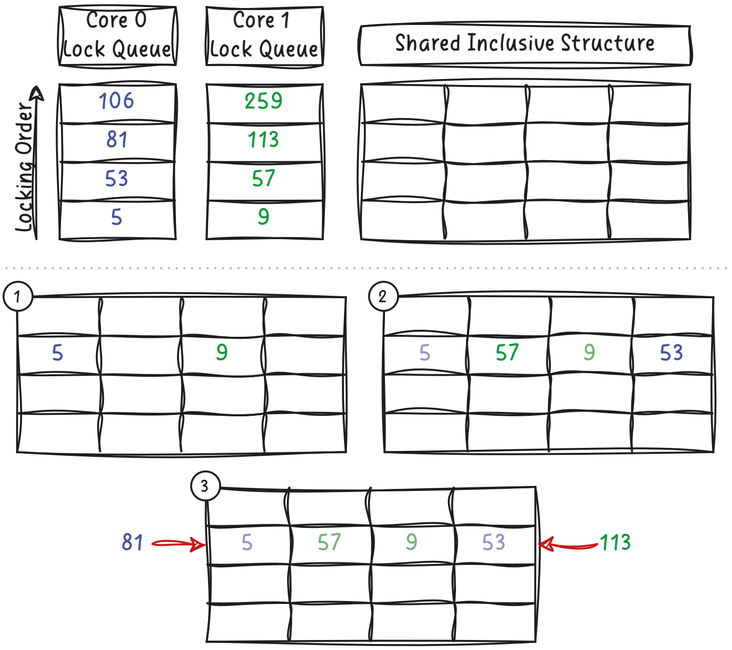


Figure 8.2. Shared structure conflict because multiple cores lock addresses in the same set of a shared inclusive structure.

`order` is similar to the address order but uses the mapped set in the shared cache instead.

To prevent this deadlock, multiple solutions are available, however, for its simplicity, and because of how improbable this situation is, on `lex conflicts`, a set lock is requested for the shared structure. The set lock prevents other cores from using the set for entries with `lex conflicts`. This effectively prevents deadlocks due to the shared cache set but allows non-conflicting entries to concurrently allocate entries in the set, as they are guaranteed to be completed at some point.

Another scenario can arise as a deadlock due to the specified rules. When removing an entry from a structure in the cache hierarchy, to allow the quick allocation of the new entry, evicted entries are moved to a victim cache, where they wait until they can be flushed to the lower levels (if needed) or wait for confirmation from other caches. In Figure 8.3, core 1 is performing normal reads and writes into the cache, but because core 0 is locking entries, they can fill the victim cache, not allowing more entries to get replaced, and therefore not allowing any new replacement to happen in the shared structure. Upon

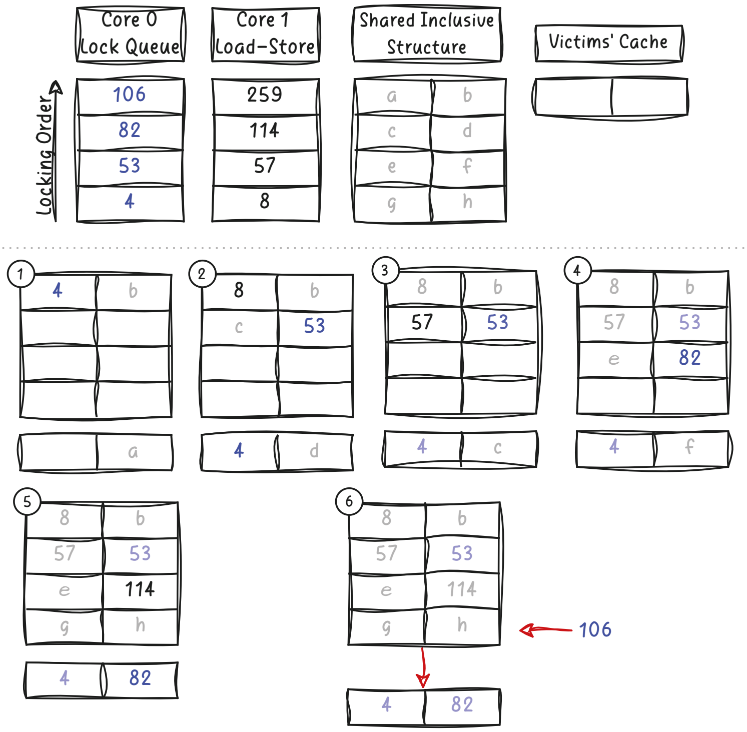


Figure 8.3. Deadlock scenario because the size limit of the MSHRs, caused by a core that is not locking addresses

reaching this point, non-locking accesses are done non-cacheable, and locking accesses must wait until an entry is freed by doing `in-situ` replacements.

$$\{L\} + u = \{L + u\} \Longleftarrow \left| \begin{array}{l} sizeof\{L + u\} \leq min(assoc) \\ \forall l \in \{L\} : LexOrder(u) \geq LexOrder(l)* \end{array} \right. \tag{8.2}$$

*If an entry has the same lex order, additional steps are required.

Following these rules, summarized in Equation 8.2), multiple addresses can be locked without any risk of deadlock.

8.3 Multi-Address atomic instructions

Atomic operations, and our definition of `lex order`, can be combined to create non-speculative multi-address atomics. We developed two kinds of new instructions: 1) Atomic fetch-and instructions and 2) Multi-Compare And Swap.

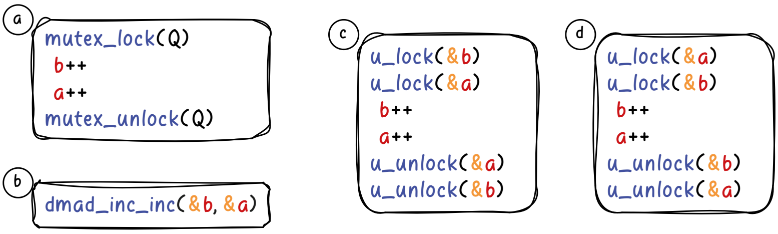
To prevent extra buffers or reading extra data from memory, and following x86 ISA, all the data for the CAS operation must be in registers. A CAS operation requires 3 values: address, old value, and new value. Therefore, an n-CAS operation requires $3 \times n$ registers to be performed. On x86, the number of registers available to the programmer is 16, therefore a maximum of 5-CAS can be implemented.

Listing 8.1. Two-addresses fetch-and-add atomic operation in gem5-like micro-code

```
mfence
// Load-Locking block
  load_lock t1, rax
  load_lock.exec t2, rcx
// Computing block
  add t1, t1, reg
  add t2, t2, regm
// Storing-Unlocking block
  store_unlock t1, rax
  store_unlock t2, rcx
mfence
```

8.4 Results

With this new instruction set for multi-address atomics, we modified several data structure benchmarks, alongside some benchmarks from Splash and STAMP to see the effectiveness of the proposed solution. We show 4 different versions



```
mutex_lock(Q)
 b++
 a++
mutex_unlock(Q)
```

```
dmad_inc_inc(&b, &a)
```

```
u_lock(&b)
u_lock(&a)
 b++
 a++
u_unlock(&a)
u_unlock(&b)
```

```
u_lock(&a)
u_lock(&b)
 b++
 a++
u_unlock(&b)
u_unlock(&a)
```

Figure 8.4. a) Critical section with mutex locks; b) MAD atomic instruction; c) micro-ops generated by the MAD atomic; d) run-time order of instructions.

of the applications: Lock-based, Lock-Free, Intel TSX-like and MAD Atomics. In the case of some applications (Dequeue, MWObject, Water-NS, Water-SP, Intruder), there is no lock-free alternative, so this bar is omitted in those applications.

Figure 8.5 shows the execution time normalized to the lock-based version of the application with the same core count. Stack is the worst-case scenario for the lock-free versions. Stack requires that all the operations performed require modifying the same memory location (the top of the stack), forcing an unbounded set of retries. HashMap shows a lot of overhead with the lock-free implementation, but from 8 cores, this overhead gets reduced. This issue is caused by the increase in complexity of the lock-free implementation of HashMap with 1-address atomics. Due to the unbounded limit of the TSX implementation, even after limiting the number of retries to 6, most of the applications tend to perform much worse than the alternatives. This behavior is exhibited due to the high contention these applications have. In general MAD atomics shows a general improvement over any other methodology, but MWObject. MWObject is the benchmark with the maximum contention possible, a small critical section targeting the exact same addresses with a simple and quick operation. In this unrealistic scenario, the data is forwarded to another core instead of using the locality and executing multiple times the critical section before relinquishing the cacheline. As a summary, MAD atomics improves the execution time by about 80% over mutex locks, while reducing 60% over Intel TSX and slightly improving over lock-free implementations but being much simpler to program and improving significantly over HashMap and Stack.

As the hardware and modifications introduced are small, we chose to show the number of committed instructions (Figure 8.6) as a measure of energy. In this scenario, the buffer required is small, the logic is simple enough that it should not incur significant power costs. In general, committing fewer instructions in a non-speculative approach means sending fewer requests, and flushing the pipeline less times, therefore the energy consumed should be reduced in a similar proportion. As expected, the graphs about time and com-

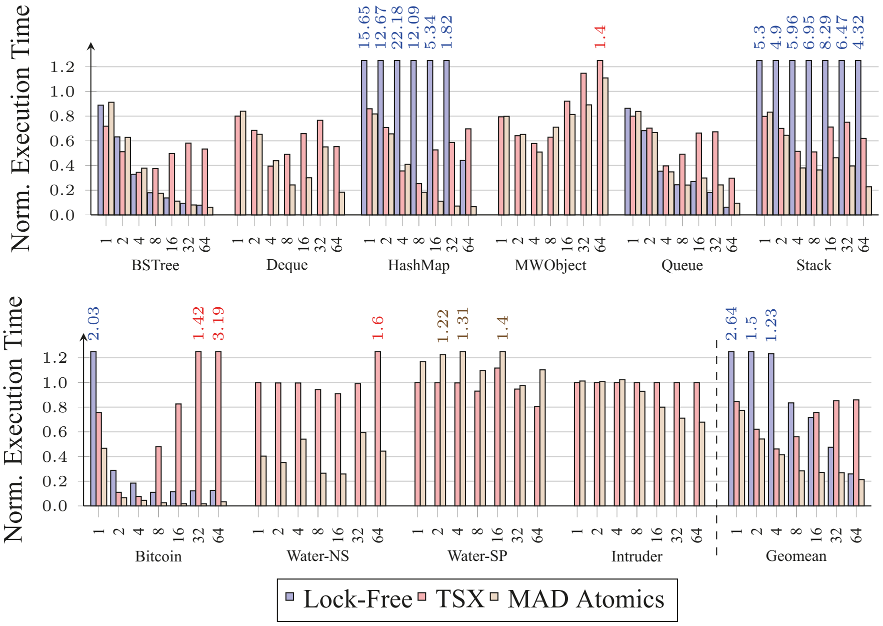


Figure 8.5. Execution time (1 to 64 cores). Data is normalized to the lock version with the same core count. Deque, MWObject, Bitcoin, Water-NS, and Water-SP do not have lock-free version. Intruder does not have a lock or lock-free version, it is normalized against TSX.

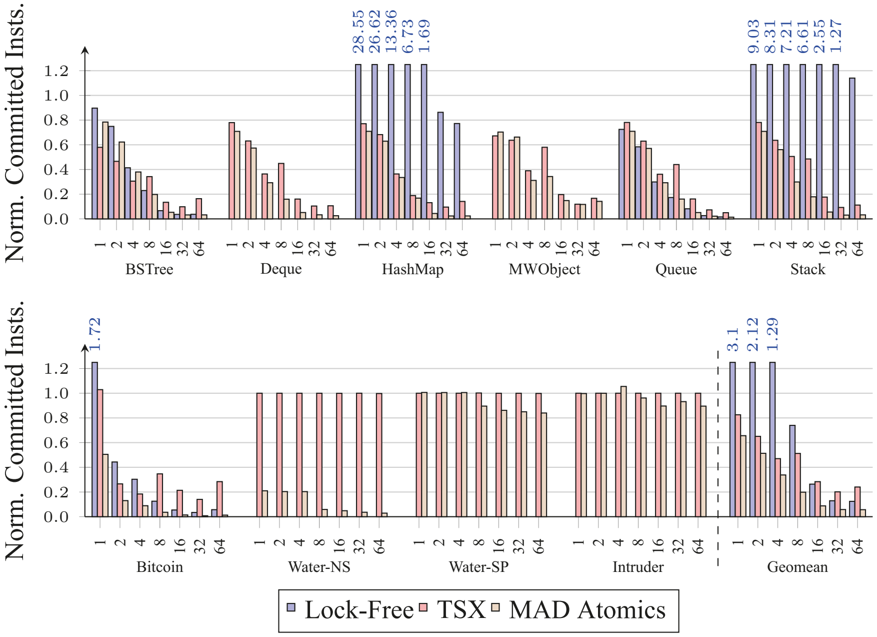


Figure 8.6. Normalized committed instructions (1 to 64 cores). Data is normalized against the lock version with the same core count. Deque, MWObject, Bitcoin, Water-NS, and Water-SP do not have a lock-free version. Intruder does not have a lock or lock-free version, it is normalized against TSX.

mitted instructions are quite similar, but with some exceptions. The overheads introduced in Water-SP and MWObject are at the cost of reducing the amount of committed instructions. In general, the trend is that any methodology over mutex lock seems to work better, also in energy (with the exception of Stack), by a significant margin.

9. cleAR: Bounding TM to a single retry

9.1 Introduction

Hardware transactional memory delegates all the synchronization and conflict resolution to the hardware, in the hope that it will protect the memory in the best way possible. While it is trivial to use by developers, as they only must mark the limits of the critical section, the performance was not as good as expected. The main factor that reduces the performance of transactional memory is the number of retries required to complete a critical section under contention.

This is the third, and last, step in this thesis, we propose the runtime conversion of transactions into their non-speculative version. With this approach, we open the way to enable the non-speculative execution of coarse grain critical sections.

In Section 9.2 we show our observations with high-contended transactions. Then, in Section 9.3 we present a methodology that exploits multi-address locking to bound the number of retries of transactions. Lastly, in Section 9.4 we compare the proposal with both Intel TSX and Power-TM.

9.2 High-contended immutable transactions

High-contented transactions are prone to be aborted multiple times, limiting the concurrency of the transaction. The most trivial examples of these kinds of transactions are the ones that emulate an RMW operation. All the threads will try to read and write in the same memory location. These are an example of immutable transactions, as once they are started, every retry will perform the same memory accesses.

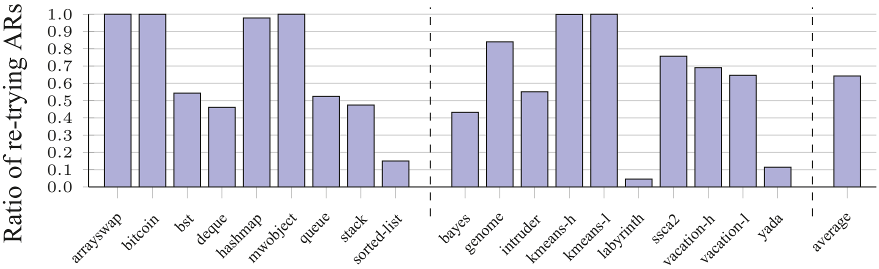


Figure 9.1. ARs that do not change their accessed cachelines on the first retry

Listing 9.1. Inmutable AR. From arrayswap.

```
register uint64_t* a = array[posa];
register uint64_t* b = array[posb];
atomic {
  uint64_t elem_a = *a;
  uint64_t elem_b = *b;
  *a = elem_b;
  *b = elem_a;
}
```

In general, a relevant amount (over 60% in our evaluation Figure 9.1) of transactions tend to operate on the same memory locations when retrying. These likely-immutable transactions can change memory locations after being interrupted by another core, but this is not the common case. In Table 9.1, we show a breakdown of the critical sections in code by their mutability status. Applications like arrayswap and mwobject are composed solely of immutable critical sections, and this is because of the algorithm used. For example, arrayswap swaps two elements predefined (Listing 9.1), therefore the critical section only needs to permute those two locations, and because the elements are predefined, the locations also are. In STAMP, around half of the critical sections are likely immutable, that is, most of the time the memory locations do not change, but sometimes they do.

In most cases, mutable critical sections require having any kind of memory indirection in the code, that is, the memory location to modify is pointed by another memory location. The simplest example is a sorted list (Listing 9.2). The next node, the address to load, and possibly modify, is accessed by a pointer indirection from the current node. However, indirections do not directly indicate that the code is mutable, which is why we include likely immutable critical sections. In Listing 9.3, the value read and written (`.bitcoins`) is obtained from an indirection of a global table `users`. In this case, the addresses will not change between re-executions, unless the pointer to `users` changes. Most of the time, this address will be a constant, so if the compiler detects it, it could move the indirection load outside the critical section, but most of the time, it will go safely, and keep the indirection in the critical section, creating an immutable critical section with an indirection inside.

9.3 Bounding Retries

Our observation leads to: if the memory set is not changed, cache locking can be used to guarantee the completion of the transaction. However, as the addresses are not known in advance, in contrast to atomic operations, a first run is made (discovery), calculating all the addresses and gathering as much

Listing 9.2. Mutable AR. From sorted-list.

```
atomic {
  auto curr = head->next;
  while (curr != tail) {
    if (curr->data == val) n_val++;
    curr = curr->next;
  }
}
```

Listing 9.3. Conditionally Inmutable AR with indirections. From bitcoin.

```
User *users;
atomic {
  users[from].bitcoins -= amount;
  users[to].bitcoins += amount;
}
```

Benchmark	# of ARs	Immutable	Likely immutable	Mutable
arrayswap	2	2	0	0
bitcoin	1	0	1	0
bst	3	0	0	3
deque	2	0	1	1
hashmap	3	0	0	3
mwobject	1	1	0	0
queue	2	0	1	1
stack	2	0	1	1
sorted-list	3	1	0	2
bayes	14	0	5	9
genome	5	0	0	5
intruder	3	0	2	1
kmeans-h	3	1	2	0
kmeans-l	3	1	2	0
labyrinth	3	0	0	3
ssca2	3	2	1	0
vacation-h	3	0	1	2
vacation-l	3	0	1	2
yada	6	1	0	5

Table 9.1. *Characterization of ARs*

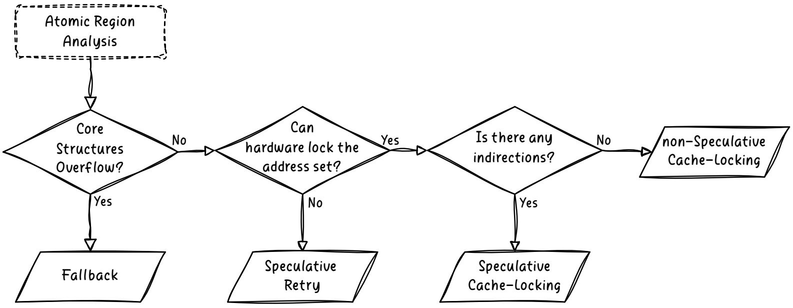

Atomic Region Analysis
Core Structures Overflow?
No
Yes
Fallback
Can hardware lock the address set?
Yes
No
Speculative Retry
Is there any indirections?
No
Yes
Speculative Cache-Locking
non-Speculative Cache-Locking

Figure 9.2. Decision tree of the execution modes of cleAR

information as possible about the critical section. In this first execution, in the case of aborting by memory access of another core, the abort is elided, and the section continues to gather as much information as possible. Upon finding the end of the transaction, the information gathered is used to determine if the section is immutable, likely immutable, or mutable, and therefore, selecting the correct re-execution method (Figure 9.2).

The available execution methods are:

- **Discovery**: First execution of a critical section, eliding the abort if it is caused by memory access. It gathers information about the critical section to select the right re-execution method.
- **non-Speculative Cache-Locking (nSCL)**: The critical section was detected as immutable; therefore, all addresses are locked in the cache, and the critical section is re-executed without running any conflict detection or checkpoint.
- **Speculative Cache-Locking (SCL)**: The critical section is likely immutable, but there is no guarantee. In this case, cache locking is used on some memory locations (written and conflicting addresses) and the critical section re-executes using the conflict detection and checkpoint.
- **Speculative Retry**: The critical section is likely mutable; therefore, no locking is performed, and the section re-executes using the speculation mechanism already mentioned.
- **Fallback**: The section reached a limit where speculation is nearly guaranteed to never succeed, in this case, a fallback global lock is taken, and the section is run without any speculation protection mechanism.

This idea is orthogonal to the speculation method used to execute the critical section, either SLE or TM. The fallback and speculative retry re-executions are provided by the speculative method used, while nSCL, SCL, and discovery are added by cleAR.

9.4 Results

We implemented 4 different versions to evaluate cleAR. The baseline (B) is running an Intel TSX-like implementation. Then, as an up-to-date implementation of HTM, we include PowerTM (P). Finally, we added cleAR to both implementations TSX-like + cleAR (C) and PowerTM + cleAR (W).

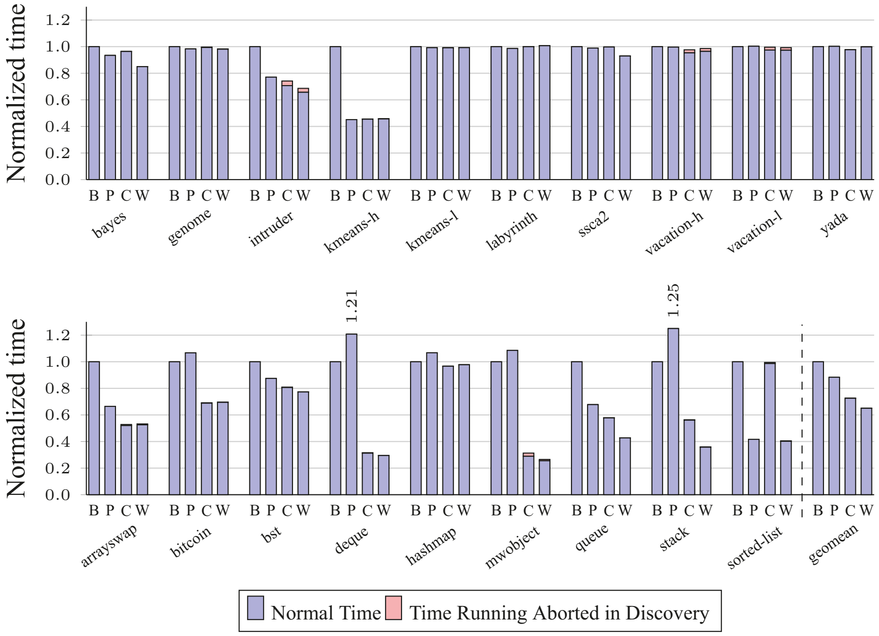


Figure 9.3. Normalized execution time

Our first result is to observe the execution time, alongside the overhead time due to executing during aborted discovery. As can be seen in Figure 9.3, most of the STAMP benchmarks do not exhibit much change. The main reason is that the critical sections in STAMP are quite large, and their address sets are quite big. Two scenarios happen here: 1) the threads do not conflict at all, which happens for example in ssca2, genome, vacation, ... and 2) the threads conflict so much (or they have too many addresses) and the transaction reaches the fallback path, this is happening to bayes, labyrithn, yada, ... However, we are able to obtain significant improvement on bayes, intruder, and kmeans-h, alongside power-tm which also gets some benefit here. In the data-structure benchmarks, with the exception of hashmap, a huge benefit is obtained in most of the benchmarks, and due to its high contention, this is the expected result.

Execution time by itself is not enough to understand how cleAR really improves the execution of the benchmarks. In Figure 9.4, we show the number of aborts per committed transaction. First, please note that even if the number of retries is limited to between 1 and 10, due to the heuristics used in HTM, some kinds of aborts do not count to the retry counter (i.e., being aborted because

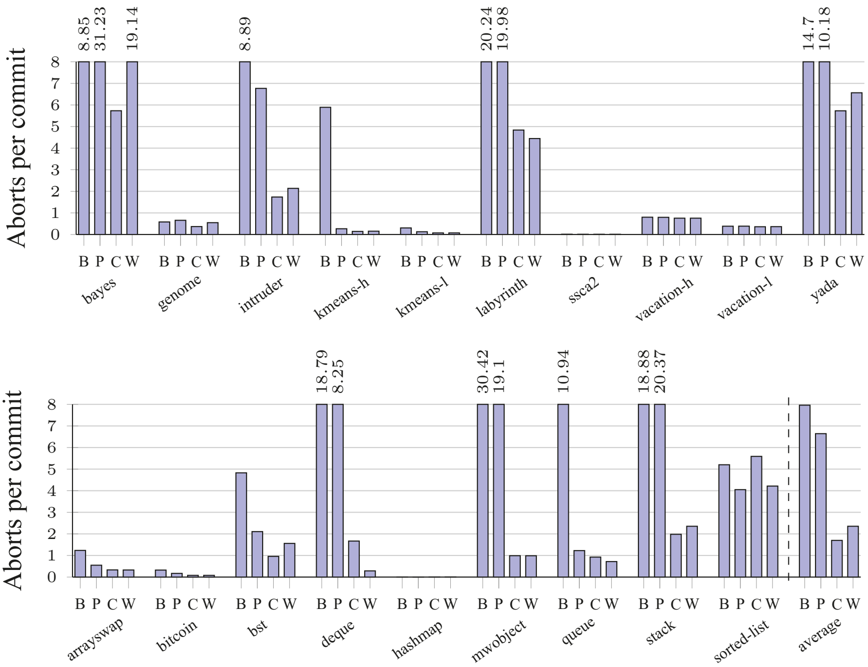


Figure 9.4. Aborts per Committed Transaction

another transaction started in fallback mode). The big picture here can be seen at the average, where the number of retries is reduced from 8 or 7 (baseline and Power-TM respectively) to around 2 (cleAR with and without Power-TM). The rest of the aborts are prevented by executing the highly contended sections non-speculatively, but also by reducing the possibility of conflicts using the locking mechanism in conjunction with the speculative approach (sCL mode).

10. Conclusion and Future Lines

10.1 Conclusion

Non-speculative and concurrent execution of critical sections is a problem far from being solved. This thesis explores the complex world of thread synchronization and atomicity and introduces some improvements over existing solutions.

The very first issue is the outdated state of the benchmarks used in research (Paper I). Most of the suites were crafted over 20 years ago. In Splash-4, we try to show why this issue matters and why the community needs to continue updating benchmarks to prevent misrepresenting the underlying hardware as much as possible. We have also shown how we do this update, so other research can follow a similar method to update other applications. However, this requires a very deep understanding of each application, the data flow, and the locking mechanisms used. It is a time-consuming task.

While atomic instructions are extremely useful, efficient, and high-performant, they are rarely used outside small programs or to create other protection primitives (like mutexes). In our Paper II, we have developed a methodology that allows non-speculative, efficient, and deadlock-free, to lock multiple cachelines at the same time to perform a multiple address atomic operation. These new atomic operations are not only simpler to use because they protect more than one address, but also, they are much more flexible.

Still, atomic operations are hard to introduce in big programs, even if multiple addresses atomic simplify the problem, therefore, they will be relegated to end up in libraries and data structures. Developers tend to still use mutexes, or order coarse-grain locking mechanisms because they do not need to worry about data dependency and deadlocks. One of the proposed coarse-grain protection methods is Transactional Memory, it introduces the concept of transactions, where the developers do not even need to worry about which lock to use on each critical section, the transactional memory system will manage any conflict. Despite how good it sounds, the performance in hardware implementations is not as good as correctly using previous solutions. In our Paper III, we introduce new hardware that, using the locking properties of multi-address atomic operations, can determine the data used by the section and perform a non-speculative re-execution of the section to guarantee its success in just one retry.

Overall, this thesis allows the non-speculative execution of critical sections by incrementally adding more complexity to the critical sections while making it easier for programmers to protect the code regions they require.

In this thesis, we propose our own approach to continue improving the synchronization mechanism between threads, with a big focus on efficiency and deadlock-freeness. I hope this work we have done in the last 5 years serves as an inspiration point for more researchers in the field to continue developing their ideas and expanding their knowledge with new insights in the years to come.

10.2 Future Research Lines

In both MADs and cleAR works, the locking is assumed to be executed in order, which may introduce a bottleneck in certain situations. We have investigated performing this locking out of order, and it is feasible if unordered requests can relinquish the locking permissions upon another core request. At first glance, it seems that it should work identically and not be able to deadlock, but more research needs to be done to keep this guarantee.

Also, in certain situations, one `lex order` may not be enough, this happens with two or more shared structures that have enough capacity but with different indexing policies (i.e., an inclusive LLC and a directory decoupled from the LLC). In this case, it can be solved by having two `lex orders`, in the example, one for the LLC and another for the directory. Then, first, it is needed to guarantee the slots and permissions in one structure (ideally the biggest one first), and when the space is reserved, then allocate the space in the second one. However, this methodology may introduce a lot of complexity and needs more research.

We think it would be interesting to explore is the combination of locking and SIMD instructions. Adding locking SIMD loads, stores, gathers and scatters may open the door to a new paradigm of SIMD critical sections that some applications may exploit.

The Splash benchmark suite still has some issues, like a deadlock in FMM since splash-2, and is missing a lot of optimizations that can boost the performance. For example, FFT has 7 barriers in the code, but only 3-4 are required.

Also, we wanted to explore how the compiler could help with the conversion of critical sections and transactions. The compiler could mark, even alter how the addresses are accessed, and even perform some transformations that could help to perform the non-speculative execution at runtime.

Bibliography

[1] McPAT 1.3. `https://github.com/HewlettPackard/mcpat`, September 2015.

[2] S. Afshar, M. Behnam, R.J. Bril, and T. Nolte. Per processor spin-based protocols for multiprocessor real-time systems. *Leibniz Transactions on Embedded Systems*, 2017.

[3] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, April 2009.

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

[5] ARM. Arm synchronization primitives development article, 2022.

[6] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270, 1993.

[7] Naama Ben-David, Guy E Blelloch, and Yuanhao Wei. Lock-free locks revisited. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 278–293, 2022.

[8] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, October 2008.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.

[11] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, page 21–28, New York, USA, 2005. Association for Computing Machinery.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[13] Dave Dice, Maurice Herlihy, and Alex Kogan. Improving parallelism in hardware transactional memory. *ACM Trans. Archit. Code Optim.*, 15(1), mar 2018.

[14] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *EDW-310, E.W. Dijkstra Archive, Center for American History, University of Texas at Austin*.

[15] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele. Dcas is not a silver bullet for nonblocking algorithm design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, page 216–224, 2004.

[16] Steven Feldman, Pierre Laborde, and Damian Dechev. A practical wait-free multi-word compare-and-swap operation. 2013.

[17] Steven Feldman, Pierre Laborde, and Damian Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, August 2014.

[18] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. `https://www.agner.org/optimize/microarchitecture.pdf`, November 2022.

[19] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *39th Conf. on Programming Language Design and Implementation (PLDI)*, pages 46–61, June 2018.

[20] Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, PODC ’02, page 260–269, New York, NY, USA, 2002. Association for Computing Machinery.

[21] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC ’01, pages 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.

[22] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20st Int’l Symp. on Computer Architecture (ISCA)*, pages 289–300, May 1993.

[23] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[24] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Morgan Kaufmann Publishers (imprint of Elsevier), 2 edition, 2021.

[25] Herbert HJ Hum and James R Goodman. Forward state for use in cache coherency in a multiprocessor system, July 26 2005. US Patent 6,922,756.

[26] IBM Corporation. Power ISA Version 3.1, May 2020.

[27] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Number 325462-072US. Intel, May 2020.

[28] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[29] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.

[30] Nodari Kankava. Exploring the efficiency of multi-word compare-and-swap. 2020.

[31] Daniel Kondor. Bitcoin network dataset. Available (archived) at: `https://web.archive.org/web/20200502094144/https://senseable2015-6.mit.edu/bitcoin/` (accessed on 30 Nov. 2023).

[32] Shubham Lagwankar. A lock-free work-stealing deque. https://github.com/ssbl/concurrent-deque.

[33] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *ACM SIGPLAN Notices*, 48(8):69–80, 2013.

[34] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd Int'l Symp. on Microarchitecture (MICRO)*, pages 469–480, December 2009.

[35] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Matthew D. Sinclair Boris Shingarov, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery.

[37] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *ACM SIGOPS Operating Systems Review*, 26(2):108, 1992.

[38] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Int'l Symp. on Workload Characterization (IISWC)*, pages 35–46, September 2008.

[39] CORPORATE Motorola, Inc. *M68000 family programmer's reference manual*. Motorola Inc., 1991.

[40] Srishty Patel, Rajshekar Kalayappan, Ishani Mahajan, and Smruti R. Sarangi. A hardware implementation of the mcas synchronization primitive. In *2017 Design, Automation, and Test in Europe (DATE)*, pages 918–921, March 2017.

[41] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *34th Int'l Symp. on Microarchitecture (MICRO)*, pages 294–305, December 2001.

[42] Ravi Rajwar and James R Goodman. Transactional lock-free execution of lock-based programs. In *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 5–17, October 2002.

[43] Alberto Ros and Stefanos Kaxiras. Non-speculative store coalescing in total store order. In *45th Int'l Symp. on Computer Architecture (ISCA)*, pages 221–234, June 2018.

[44] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, April 2016.

[45] Marina Shimchenko, Rubén Titos-Gil, Ricardo Fernández-Pascual, Manuel E. Acacio, Stefanos Kaxiras, Alberto Ros, and Alexandra Jimborean. Analysing software prefetching opportunities in hardware transactional memory. *Journal of Supercomputing (SUPE)*, 78(1):919–944, January 2022.

[46] Jaswinder P. Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, (1):5–44, March 1992.

[47] Standard Performance Evaluation Corporation. SPEC CPU2017, 2017.

[48] T. B. Strøm and M. Schoeberl. Hardlock: A concurrent real-time multicore locking unit. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 9–16, 2018.

[49] Michael E Thomadakis. The architecture of the nehalem processor and nehalem-ep smp platforms. *Resource*, 3(2):30–32, 2011.

[50] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 212–222, 1992.

[51] Andrew Waterman and Krste Asanovic. The risc-v instruction set manual, volume i: Unprivileged isa document, version 20190608-baseratified. *RISC-V Foundation, Tech. Rep*, 2019.

[52] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[53] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. Parsec3.0: A multicore benchmark suite with network stacks and splash-2x. *SIGARCH Comput. Archit. News*, 44(5):1–16, February 2017.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2520

Editor: The Dean of the Faculty of Science and Technology

Distribution: publications.uu.se
urn:nbn:se:uu:diva-552947

ACTA UNIVERSITATIS
UPSALIENSIS
2025