OPTIMIZING HIGH LOCALITY MEMORY REFERENCES IN CACHE COHERENT SHARED MEMORY MULTI-CORE PROCESSORS

A Dissertation
Presented to
The Academic Faculty

By

Suk Chan Kang

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2019

OPTIMIZING HIGH LOCALITY MEMORY REFERENCES IN CACHE COHERENT SHARED MEMORY MULTI-CORE PROCESSORS

Approved by:

Dr. Sudhakar Yalamanchili, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Linda M Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Ada Gavrilovska
School of Computer Scienece
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Scienece
Georgia Institute of Technology

Date Approved: January 10, 2019

To my God and family

ACKNOWLEDGEMENTS

This thesis could be completed with the help of many individuals. Above all, I would like to express my deepest appreciation to my parents, Joo Shin Kang and Kwang Yeon Kim, for their love, trust, and sacrifice for me. I would like to thank my brother, Dr. Sang Yop Kang, for his warm brotherhood and encouragement. My special thanks should also go to my maternal grandmother, Gab Nieu Kim, my late maternal grandfather, Hyung Chul Kim, and my late paternal grandmother, Kyung Jieom Kim, for being my heart-warm grand parents who are full of love. I must also thank my aunt Kwang Woo Kim and my uncle Dr. Sung Mou Cho, for their generous support in my life.

I would also like to extend my deepest gratitude to Dr. Sudhakar Yalamanchili who is my advisor. As my role model, he demonstrates that an outstanding scholar can also be a nice and generous person with quiet charisma. Every one-on-one meeting with him was simply the precious opportunity to get myself cheered up significantly and to clearly learn the meaning of a advisor. He always respected me as his PhD student, listened to and encouraged me, gave exceptional technical solutions and ingenious suggestions, and patiently waited for my research progress with profound belief in my work. Hence, “who is your PhD thesis advisor?” has been and also will be one of my favorite question, ever since I first met with him.

I would also like to extend my gratitude to late Dr. George Riley who is the former ECE’s associate chair for graduate affairs, for enabling me to have Dr. Sudhakar Yalamanchili as my new advisor. I am so sorry that I was too busy even to know that Dr. George Riley passed away recently. Occasionally, I remind myself of the touching words he gave me when I was in the midst of hard time looking for a new advisor.

I would like to extend my sincere thanks to Dr. Linda M Wills, Dr. Ada Gavrilovska, Dr. Tushar Krishna, and Dr. Santosh Pande, for serving as my kind committee members, which is my honor. I must especially thank Dr. Linda M Wills, for generously giving me the insightful guidance, thorough and frequent proof-reads of my thesis, and emotional support, even though she herself was extremely busy. I cannot appreciate her enough because her help encouraged me extensively and enabled me to remain calm while I was preparing my thesis and final defense. I learned valuable and useful knowledge used for my thesis from the courses which Dr. Ada Gavrilovska and Dr. Santosh Pande taught. I also learned many valuable and critical knowledge from Dr. Tushar Krishna when he kindly answered my abrupt unscheduled questions related to my research, in the hall ways and lab space.

Thanks should also go to Dr. Jongman Kim who is my former advisor at Gerogia Tech and his colleague Dr. Chrysostomos Nicopoulos at University of Cyprus, as my former coworkers with whom I had precious time in my life.

I very much thank Dr. Daniela Staiculescu and Ms. Tasha M Torrence at ECE Graduate Affairs Office, for their ever consistent kind help and advice.

Additionally, I must very much appreciate Korean Government, for having granted me the scholarship while I studied at Carnegie Mellon University as a master student. I am so proud of the support.

Finally, I am extremely grateful to my God. My PhD program was the time when I realize that you always stay with me. It is you who let me get over all the hardships and eventually complete my PhD degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODES

ACRONYMS

2D 2-dimensional.

CAS compare-and-swap.

CCSM cache coherence shared memory.

CF caching flag.

CoW copy-on-write.

CSL centralized spin-lock.

CSLV centralized spin-lock variable.

DDA data-decoupled architecture.

FIFO first-in-first-out.

FSMO false shared memory object.

ILP instruction level parallelism.

IPC inter-process communication.

ISA instruction set architecture.

LKM loadable kernel module.

LRSW least recently spin-waiting.

LSQ load/store queue.

MRSW most recently spin-waiting.

NUMA non-uniform memory access.

OoO out-of-order.

OS operating system.

PIPT physically indexed physically tagged.

PLMD pure local memory data.

RMW read-modify-write.

ROI region of interest.

SM shared memory.

TLB translate look-aside buffer.

ToS top of stack.

TS test-and-set.

TSO total store ordering.

TTS test-and-test-and-set.

VA virtual address.

vCPU virtual CPU.

VIVT virtually indexed virtually tagged.

SUMMARY

Optimizing memory references has been a primary research area of computer systems ever since the advent of the stored program computers. The objective of this thesis research is to identify and optimize two classes of high locality data memory reference streams in cache coherent shared memory multi-processors. More specifically, this thesis classifies such memory objects into spatial and temporal false shared memory objects. The underlying hypothesis is that the policy of treating all the memory objects as being permanently shared significantly hinders the optimization of high-locality memory objects in modern cache coherent shared memory multi-processor systems: the policy forces the systems to unconditionally prepare to incur shared-memory-related overheads for every memory reference. To verify the hypothesis, this thesis explores two different schemes to minimize the shared memory abstraction overheads associated with memory reference streams of spatial and temporal false shared memory objects, respectively. The schemes implement the exception rules which enable isolating false memory objects from the shared memory domain, in a spatial and temporal manner. However, the exception rules definitely require special consideration in cache coherent shared memory multi-processors, regarding the data consistency, cache coherence, and memory consistency model. Thus, this thesis not only implements the schemes based on such consideration, but also breaks the chain of the widespread faulty assumption of prior academic work. This high-level approach ultimately aims at upgrading scalability of large scale systems, such as multi-socket cache coherent shared memory multi-processors, throughout improving performance and reducing energy/power consumption. This thesis demonstrates the efficacy and efficiency of the schemes in terms of performance improvement and energy/power reduction.

CHAPTER 1
INTRODUCTION

The modern cache coherence shared memory (CCSM) multi-processor architecture presents a shared memory (SM) abstraction to enable software designers to leverage the intuitive, convenient, and flexible SM programming model. However, the SM abstraction not only requires support by operating system (OS)s and compilers, but also incurs overhead in utilizing the underlying micro-architectures. For example, the versatile technique of creating virtual address (VA) synonyms in SM requires the OS provided inter-process communication (IPC) interfaces (e.g. shared memory segments in Linux), corresponding compiler support, virtual-to-physical address translation support, and cache coherence and memory consistency enforcement while performing synchronization operations on accesses to the shared synonymed memory. The resultant cache coherence and memory ordering (for memory consistency) traffic can eventually strain the interconnection networks. These overheads can grow with increasing core count and eventually overwhelm any intended performance speed-up and power/energy efficiency advantages, reducing scalability in the end. A socket based CCSM non-uniform memory access (NUMA) system will be more sensitive to the scalability challenges, due to its large average memory latency resulting from crossing of socket borders.

We observe that shared memory accesses can be distinguished based on their locality properties. In particular, we find that a significant number of accesses that are not shared, are treated as shared accesses and therefore incur all of the overheads associated with the SM abstraction. On the other hand, we also take notice that the extremely concurrent accesses to shared memory data naively make such data the hot spots of the overheads associated with the SM abstraction. Therefore, this thesis explores opportunities to optimize such high-locality memory accesses, by bypassing the aforementioned SM overheads in multi-processors. This high-level approach ultimately aims at upgrading system scalability by improving performance and energy/power efficiency.

1.1 False Shared Memory Object

This thesis hypothesizes that the policy of treating all the memory objects as being permanently shared significantly prevents modern CCSM multi-processors from further optimizing the access of high-locality memory objects. Accordingly, we observe that there are two classes of memory objects whose accesses exhibit high locality and can be subject to optimization to improve scalability. To better identify such objects, we develop the concept of the spatial/temporal false shared memory object (FSMO):

- ***Spatial FSMO*:** memory object which can be spatially isolated from the SM domain, due to its private scope
- ***Temporal FSMO*:** memory object which can be temporally isolated from the SM domain, until the critical moment

The FSMO concept is motivated by the clear discrepancy between the design principle of the parallel SM programs and the actual SM domain operation of the CCSM multi-processors. The parallel SM programs are optimized to maximally perform the computation locally, while minimally generating the global SM transactions (e.g. loop parallelization usually targets the outer loops, rather than the inner ones). Meanwhile, the contemporary CCSM multi-processors too conservatively treat every memory object as a permanently shared one, getting ready to incur the SM overhead mechanisms. This high false positive rate in identifying actual shared memory objects becomes the origin of the redundant SM overheads. For instance, the CCSM multi-processors prohibit even the qualified private memory operations from being executed in the full out-of-order (OoO) manner, constraining them to memory ordering rules regulated by the SM consistency model.

1.2 Two Target Memory Reference Types

This thesis focuses on the following two high-locality FSMOs:

1. **Pure Local Memory Data:** source of ***NO*** shared memory references (neither inter-thread/process nor intra-thread/process), spatial FSMOs

2. **Centralized Spin-Lock Variables:** source of ***SEVERE*** shared memory references (inter-thread/process), temporal FSMOs

Listing 1.1 and 1.2 depict examples of the two memory objects.

```
int foo() {
  /*
   * Pure local memory data "bufA[10][20][30]" and "r"
   * Scope: function foo() alone
   */
  char bufA[10][20][30];
  int r;

  /*
   * NOT pure local memory data "bufB[16]"
   * Scope: function foo() and its sub-function bar()
   */
  char bufB[16];

  ...
  r = bar(bufB);
  return r;
}
```

Listing 1.1: The first target memory object type addressed in this thesis: **Pure local memory data** (`bufA[10][20][30]`, `r`).

```
/*
 * Centralized spin−lock variable ”lock” inside the spin−waiting loop.
 * Other threads/processes can share it, to concurrently read/update it.
 */
do {} while(lock == LOCKED_CONDITION);
 ...
```

Listing 1.2: The second target memory object type addressed in this thesis: **Centralized spin-lock variables** (`lock`).

1.2.1 Pure Local Memory Data

The pure local memory data (PLMD) are the memory objects accessed **exclusively within** the functions where they are declared and, consequently, are not referenced from outside the function scope. We classify PLMD as spatial FSMOs because their spatial function frame scope is private to their threads.

In handling the PLMD, the contemporary CCSM multi-processors have been wasting a significant portion of performance and energy. Specifically, the accesses to the PLMD can perfectly avoid SM overheads such as dealing with VA synonyms and regulating memory consistency. Additionally, if keeping CPU-affinity scheduling, virtually no coherence checks are required,

To address this inefficiency of losing performance and energy improvement opportunities, this thesis proposes to treat the PLMD as spatial FSMOs which extend the architected registers in CCSM multi-processors. Even though there could be other spatial FSMOs than the PLMD in one running program, the PLMD are relatively easy to filter from the run-time VA function frame stacks.

Note that the raw function frame stacks can contain minor NON-PLMD memory objects, as well. For example, "`char bufB[16]`" in Listing 1.1 is *not* PLMD, because the `bar()` function references it with the pointer `bufB`. The portion of the PLMD references can be *roughly* estimated, through counting the references to the VA stack data.

Prior work [1] shows that stack references account for an average of 56% of all memory accesses in SPEC CPU 2000 integer benchmarks (single-threaded workloads). The thesis research in Chapter 3 also shows that on average around 50% of all memory accesses in PARSEC benchmarks (multi-threaded workloads) belong to stack references. However, it is not straight-forward to detect/decouple the references to the stack data (i.e. the superset of the PLMD), let alone to the PLMD. For this reason, prior work optimizing the stack data references including [2, 3, 4, 1, 5] all rely on function-critical faulty assumptions. Chapter 4 explains where the assumptions come from, the subtleties of the run-time VA stacks, and the proper safeguards for the issues.

1.2.2 Centralized Spin-Lock Variables

The centralized spin-lock variable (CSLV) is a hot spot memory object on which all the lock-waiter threads concurrently spin-wait. The example of the centralized spin-lock (CSL) design is the ticket spinlock. We classify CSLVs as temporal FSMOs because they can temporally delay appearing on the CPU node until the thread acquires the lock (this will be explained further in Chapter 5).

The spinlock is generally the best synchronization option to protect short sized shared data in memory. However, the spin-waiting synchronization easily becomes the most massive source of concurrent fine-grain cache coherence operations of the typhoon-like cache line bounces, and should consider the memory consistency issues. Ironically, it is only after going through this heavy SM synchronization overhead that the concurrently competing threads can safely access the very short sized shared data. In socket based CCSM NUMA systems, the overhead becomes more complicated: it not only comes from the interconnection network bandwidth limit (throughput) to handle cache coherence contention, but also from the latency penalty of moving cache lines across the socket boundaries. For this reason, the existing sophisticated scalable software spinlocks are still not good enough for the large scale socket based CCSM NUMA multi-processors [6]. Unfortunately, the spinlocks using CSLVs are notorious as the worst design in terms of the overhead.

To address this synchronization overhead issue (in terms of both bandwidth and latency), this thesis proposes to handle CSLVs as temporal FSMOs which only the lock-owner thread can see (while other threads cannot) in the local cache memory. This is designed to work without destroying the underlying memory consistency rule.

1.3 Thesis Statement

The key ideas, insights, and challenges lead to the following **thesis statement:** *Significant performance improvement is possible by identifying and optimizing high locality reference streams in cache coherent shared memory multi-core processors.*

1.4 Thesis Contributions

The principle contribution of this thesis is identification and optimization of two classes of high locality reference streams (PLMD and CSLVs) that significantly affect performance in CCSM multi-processors. The specific contributions are as follows.

1. It develops the concept of false shared memory objects (FSMOs). Contemporary CCSM multi-processors unconditionally assume all memory references to be permanently shared accesses and have them restricted by the SM related overheads. The concept of FSMOs is the main insight of this thesis enabling optimization of memory accesses to PLMD (spatial FSMOs) and CSLVs (temporal FSMOs).

2. It describes how CCSM multi-processors can effectively extend the architected registers cost-efficiently, by taking advantage of spatial FSMO attributes of the PLMD, if they can be properly detected and decoupled from the rest of the memory objects [5].

3. It presents fundamental insights on how future OSes and compilers can effectively and correctly exploit the benefits of the run-time VA stack data as a superset of PLMD

[7].

4. It proposes the contrarian idea of enabling the centralized spinlock to accomplish the ultimate minimalism in cache line bouncing, by handling CSLVs as temporal FSMOs. The idea aims to unleash the power of the centralized spinlock design.

This thesis is organized as follows. Chapter 2 goes through the important prior work related to this research. Chapter 3 (the first PLMD optimization) presents a novel architecture for L1 data helper caches in CCSM multi-processors, to realize contribution 2. Chapter 4 (the second PLMD optimization) details the subtleties of the run-time VA stacks, to realize contribution 3. Chapter 5 (the CSLV optimization) illustrates the new hardware-supported Linux kernel default spinlock design, to realize contribution 4. Finally, Chapter 6 concludes the thesis.

CHAPTER 2
LITERATURE REVIEW

2.1 Memory Reference Stream Decoupling

This research topic has been important and popular, ever since the advent of the stored-program computers. Harvard mark-1 [8] is the first stored-program computer which employs two separate memory reference pathways to instruction and data streams. As such, it became the root of the prevalent Harvard architecture systems. The separate instruction and data pathways help increase the memory referencing throughput, by enabling simultaneous accesses to the two streams. However, it is generally not straight-forward to perfectly decouple memory reference streams.

Quite a few works attempt to further decouple **data** memory reference streams. The Compiler Controlled Memory (CCM) scheme [9] decouples memory references into the two separate memories of the main memory and the compiler controlled memory (CCM). The register spilled memories generated by compilers are fed into the CCM, to prevent them from causing the cache pollution problem. However, the CCM scheme is not proper for the contemporary general purpose systems, because on every context switch, the data stored in the CCM should be flushed into the main memory. Therefore, the research is better suited for specialized hardware using dedicated memories, such as DSP devices.

Additionally, the Stack Cache [10], Access Region Locality (ARL) [2], Stack Value File (SVF) [1], Reverse (Register) Integration [3], Selective Snoop Probe (SSP) [4] , Specialized Stack Cache and Pseudo Set-Associative Cache (SSC and PSAC) [11], and Separate Stack-Based Memory Organization [12] techniques all demonstrated the potential benefit of decoupling the data memory references into stack/non-stack streams, in terms of performance improvement and energy/power reduction. However, these approaches have function-critical hazards because they are all based on faulty assumptions: (1) the stack/non-stack memory data regions can be perfectly dichotomized and (2) the raw VA stack memory regions are only composed of PLMD which have the register-file-like properties. This is described further in Chapter 4 and [7]. Interestingly, the Safe Stack scheme [13, 14] (which enhances the security of the run-time function call stacks) can help provide the safeguards for these previous approaches [10, 2, 1, 3, 4, 11, 12]: for each function call stack, the compiler transform technique creates a new separate "safe" stack and moves all the PLMD of the original stack there.

2.2 Virtual Cache Memory Architecture

Virtual caches [15] work using virtual address tags and/or indexes, avoiding the virtual to physical address translation overhead. The idea is to achieve fast cache access time and energy/power reduction by avoiding accesses to the translate look-aside buffer (TLB). However, these advantages cannot be obtained for free because homonym and synonym (aliasing) issues must be resolved [15, 16] for data consistency. Homonym memory objects are the objects in different VA spaces (processes) having the same virtual addresses but which are mapped (translated) into different physical addresses. Conversely, synonym memory objects are the objects mapped (translated) into the same physical addresses but which have different virtual addresses. Normally, virtual caches can resolve homonym issues relatively simply, for instance, by tagging cache lines with the address space identifiers (ASIDs) or flushing the cache memory when context switches take place. However, synonym objects are mostly impossible to classify without proper help from the OS. To make matters worse, synonym objects can exist within the same virtual address space (i.e. intra-process synonyms) as well as across different VA spaces (i.e. inter-process synonyms) [15, 16]. Therefore, simple solutions such as flushing the cache memory on context switches can not resolve all synonym issues.

The hardware-only solutions to resolve synonym issues have to rely on looking up or tracking down the synonymed cache lines, with significant energy and area overheads. A simple preferred design to resolve the synonym issue is to prohibit any duplicate synonym cache lines from residing in the cache memory [17, 18]. That is, on a cache miss, the cache controller invalidates all the other synonymed cache lines after looking up the candidate lines. Meanwhile, Woo, et al. [19] propose a scheme to use bloom filters [20] to minimize the overhead of this looking-up approach: the bloom filter early notifies if the virtual cache memory contains the synonym line of the requested cache line access, to indicate whether or not the energy-consuming synonym look-up process is still necessary.

The software-supported solutions to resolve synonym issues can have the OS restrict page allocation, albeit at the cost of increased page fault rates [15]. Other more advanced ways involve utilizing the OS-provided synonym page mapping information. An example is the Opportunistic Virtual Caching (OVC) technique [21]. To that end, the work enhances the Linux kernel's virtual address range allocator, to inform the supporting hardware whether the user-space pages are currently free from the read-write synonym page mappings. More specifically, when a user-space page has the read-write synonym page mappings, the enhanced virtual address range allocator sets the highest order bit (i.e. 48th bit, VA_{47}) of the virtual address of the page. Note that the VA_{47} bit is always kept unset for user-space pages, in the current 48 bit canonical form X86-64 virtual address space [22]. The proposed OVC cache memory has physical tags as well as virtual tags, to handle the default physical caching mode operation and cache coherence mechanism. Each line of the OVC cache also has ASIDs to resolve the homonym issue. The scheme requires only modifying 240 lines of the Linux kernel code.

2.3 Spinlocks And Linux Kernel

This section focuses on the three first-in-first-out (FIFO) spinlocks (MCS lock [23, 24, 25], ticket spinlock [23, 24], and qspinlock [25]) to explain the evolution of the contemporary default Linux kernel spinlocks. A spinlock design should satisfy the "4-byte lock variable" and “4-level stacked context spin-wait” requirements, to be employed as the default Linux kernel spinlock. The 4-byte lock variable requirement prohibits most of the sophisticated queuing spinlock schemes (e.g. the MCS lock design [23, 24, 25], CLH lock [26], hierarchical CLH lock [27], and K42 lock [28]) from being employed as the Linux kernel default spinlock. In addition, the 4-level stacked context spin-wait requirement hinders any novel lock accelerator techniques supporting only 1-level lock context (e.g. the Misar technique [29]) from being employed for the Linux kernel.

2.3.1 Ticket Spinlock

The ticket spinlock [23, 24] does not build an explicit linked list to conduct the FIFO lock operation. Instead, the lock and unlock operations update the tail/head ticket number, respectively. The lock-competing threads have to spin-wait, unless their uniquely assigned ticket numbers become equal to the current head ticket number. Therefore, the ticket spinlock is a centralized spinlock (CSL) design: all the lock-waiter threads spin on the shared lock variable. Like other centralized spinlocks, the ticket spinlock has the advantage of using simple code. Hence, the ticket spinlock performs the best, out of all the software spinlocks, in case of experiencing no lock contention [24]. On the other hand, due to the heavy cache coherence contention surrounding the lock variable, ticket lock definitely does not scale with increased CPU counts, especially in the socket based CCSM NUMA systems [24]. The Linux kernel employed the ticket spinlock as the default spinlock, until the qspinlock replaced it.

2.3.2 MCS Lock

The MCS lock [23, 24, 25] is the representative scalable queuing spinlock design, which conducts the FIFO lock operation. Each lock-waiter thread spin-waits on its own per-thread local “qnode,” instead of on the centralized shared lock variable. The per-thread qnodes are manually declared inside the function stacks of the lock-competing threads, and passed as an argument to the lock()/unlock() API functions, so the design can build a linked listed queue structure of them.

However, the MCS lock has some native critical problems and/or potential vulnerabilities. First, the lock variable unavoidably becomes much larger than 4 bytes, because it should contain VA pointer values pointing to the current tail thread qnode. This is the reason why the MCS lock has not yet been employed in the Linux kernel as the default spinlock, even though it scales nicely. The other queuing locks such as CLH lock [26], hierarchical CLH lock [27], and K42 lock [28] have a similar problem. Second, the unlocking operation (in addition to the locking operation) imperatively uses the atomic memory operation, when it detects no successor lock-waiter thread at the moment. This has everything to do with the multiple lock-competing threads concurrently managing the linked-listed qnode queue structure. The centralized spinlock designs such as ticket spinlock basically do not use any atomic memory operation, when releasing the lock. Third, the adjacent predecessor and successor lock-waiter threads should move their qnode cache lines between them, in a ping-pong way. This can be a heavy overhead taking long latency, if the threads are running on the large scale socket based CCSM NUMA systems [6].

2.3.3 Linux qspinlock

Even though the MCS lock scales excellently, it cannot be employed as the default Linux kernel spinlock, just because the design cannot meet the “4-byte lock variable” requirement. Alternatively, as of the kernel version 4.2, Linux kernel substitutes the qspinlock [25] for the previous default ticket spinlock.

As a matter of fact, the qspinlock is not the pure queuing spinlock, but is the workaround design to make the MCS lock fit into the 4 byte small lock variable. To this end, the qspinlock is designed to work in the form of a baseline centralized spinlock (CSL) which is enhanced with an internal MCS lock. The baseline centralized spinlock variable (CSLV) is referenced for the locking, unlocking, and spin-wait operations. The internal MCS lock removes the cache coherence contention surrounding the baseline CSLV, by letting only one lock-waiter thread spin-wait on the CSLV. The 4 byte small lock variable size is facilitated by the fact that the internal MCS lock establishes the queue structure with the per-CPU qnodes, instead of with the per-thread qnodes. Note that unlike the per-thread qnodes, the per-CPU qnodes are viable only while CPU-migrations are disabled (e.g. during the spin-wait iterations). Consequently, this design ends up having a "two-step" spin-wait operation: the first step spin-waits on the per-CPU qnode of the internal MCS lock, while the second step spin-waits on the baseline CSLV. Due to this complex design, the qspinlock naturally can incur much more cache line bounces than the original MCS lock, especially accompanying a number of atomic memory operations.

CHAPTER 3
(PLMD 1) PURE LOCAL MEMORY DATA CACHE: EFFECTIVE ARCHITECTED REGISTER FILE EXTENSION FOR MULTI-PROCESSORS

The original research document of this thesis work was published in [5].

In general, memory accesses to the VA stack area data (which are a superset of the PLMD) play an instrumental role in overall system performance. Research in this domain has shown that real workloads tend to access the VA stack area at a high frequency, albeit within a small memory footprint [10, 1], because a process/thread is always running within a function during its execution, and, usually, function frames tend not to be very deep. It is precisely this observation that has led to various designs for the single-processor systems that handle memory accesses to the stack area at a higher level in the memory hierarchy, in order to improve performance and reduce energy consumption.

This thesis research demonstrates that special L1 PLMD helper caches can significantly boost the performance of one primary and dominant multi-threaded workload running on a CCSM multi-processor, while actually reducing its energy consumption. This helper cache introduces the “address space filtering” technique which filters the VA spaces (i.e. processes) and subsequently handles only the PLMD of the selected virtual address space.

The idea exploits the spatial FSMO attribute of the PLMD, and the ultimate goal is to present the exclusively selected multi-threaded process the L1 helper cache structure. This cache is extremely fast and perfectly isolated from the SM domain, just like the architected register file. The scheme targets the modern workstation and server environments that run a single, dedicated multi-threaded application workload per machine.

The PLMD helper cache is implemented as a virtually indexed virtually tagged (VIVT) cache structure, and minimizes the overhead incurred in resolving virtual-cache-related artifacts (e.g. the synonym and homonym issues), in a cost effective way. This is mainly due to the fact that the scheme grants exclusive access to the PLMD stream of only one multi-threaded process at a time, as previously introduced.

This chapter first shows a conceptual description of the PLMD cache-based approach in Section 3.1. It then provides background in Section 3.2 on its context. Section 3.3 details its implementation and Section 3.4 evaluates the performance and energy efficiency of the PLMD helper cache technique, demonstrating the usability of this design in future CCSM multi-processors. The PLMD cache mechanism is especially available to systems that run one primary and dominant multi-threaded application per machine.

Note that this thesis work makes the assumption which all major related prior work including [10, 2, 1, 3, 4, 11, 12] rely on: the stack area data are all PLMD, and their memory references can be perfectly decoupled from the other memory reference streams. Chapter 4 explains the myths and realities of this assumption, and also explains how to correctly filter and decouple the PLMD references from the stack area data stream.

3.1 Insights and Contributions

3.1.1 Pilot Design to Effectively Extend Architected Registers of CCSM Multi-Processors

The architected registers are the fastest storage areas in a system. Moreover, on CCSM multi-processors, storing data in architected registers creates a more significant impact on performance than storing them in memory: the architected register operations participate in the full OoO execution, while memory operation reordering is restricted by the shared memory consistency model of the systems. This overhead has the potential to create the anomaly that single-threaded programs which require *no* shared memory operations run much faster on the OoO single-processors than on the OoO CCSM multi-processors.

Meanwhile, it is nearly impossible to extend the architected registers of existing machines, because they already have the maximum number of architected registers which their fixed instruction set architecture (ISA) allows. This thesis proposes a cache-based design

Table 3.1: The OoO opportunities of the architected register operations, PLMD helper cache memory operations, and regular memory operations on the single-processor and CCSM multi-processor systems

Regular Memory Operations	
Single-Processors	Full OoO
CCSM Multi-Processors	Limited re-ordering allowed by shared memory consistency model

Architected Register Operations	
Single-Processors	Full OoO
CCSM Multi-Processors	Full OoO

PLMD Helper Cache Memory Operations	
Single-Processors	Full OoO
CCSM Multi-Processors	Full OoO

which has a similar effect as extending the architected register file. This effect would remarkably increase the opportunities of the full OoO executions of the selected one primary and dominant multi-threaded process, especially on an ISA with relatively small number of architected registers (e.g. X86 or X86-64 [30]).

The L1 PLMD helper cache architecture proposes a **pilot** design of the "architected-register-like" cache memory for the CCSM multi-processors, which is (1) extremely fast and (2) perfectly isolated from the SM domain. The idea does not require changing the ISA to deliver the architected register file extension effect. To achieve the fast speed, the design profits from the VIVT cache memory structure. To achieve perfect isolation from the SM domain, the design utilizes the spatial FSMO attribute of the PLMD. The "isolation from the SM domain" property frees the PLMD helper cache from both the memory ordering regulation and the cache coherence interference (by enforcing "CPU-Affinity" scheduling). Additionally, the architected register "extension" includes the effect of storing even the bulky data (e.g. data structures and arrays) in the virtually extended architected registers (i.e. PLMD helper caches), which is generally not possible with the actual architected registers.

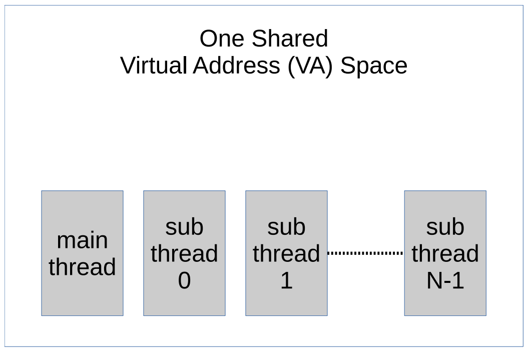
Table 3.1 recaps the OoO execution opportunities of the architected register operations, PLMD helper cache memory operations, and regular memory operations.

3.1.2 Cost-Effective and Flexible VIVT Cache Implementation

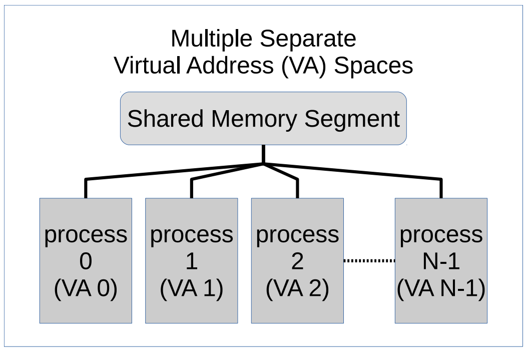
The L1 PLMD helper cache architecture manually filters out all VA spaces except the main VA space (process). Thus, the run-time PLMD of this lone process become the exclusive occupants of the PLMD helper cache. This address space filtering technique automatically removes the synonym and homonym problems of the employed VIVT cache structure, without incurring additional expensive overhead.

The address space filtering technique also presents the PLMD helper cache with the unlimited freedom for the VIVT cache configuration. Note that virtually indexed cache designs usually prefer limiting the cache set size to be equal to or smaller than the virtual address page size of the OS [21, 31, 32]. This design policy intends to take advantage of enabling the virtual indices to actually work as physical indices, skipping the address translation overhead. However, the resulting virtually indexed cache memories have the only restrictive and bizarre option of adding more associativities, to increase their size, which ironically increases the access latency and tag-matching power/energy consumption.

3.2 Background



(a) Multi-threaded



(b) Shared memory IPC

Figure 3.1: The two prevalent virtual address space-sharing types

There are two prevalent methodologies that are typically utilized in shared-memory architectures: the multi-threaded methodology and the shared memory inter-process communication.

Multi-threaded methodology

Multi-threaded methodology operates on the principle of one main thread and its N sub-threads residing in a single VA space and comprising one process (abstractly visualized in Figure 3.1a). The main thread creates its sub-threads using a “clone()” system call. Consequently, the main thread and its sub-threads share the page table, global variable area, and file descriptor. The principle of confinement within a single VA space aims to alleviate the heavy overhead of context switching during parallel execution. However, the main thread and all sub-threads must have their own separate resources, such as stack area (user-level/kernel-level) and registers. User-level thread libraries such as POSIX Threads [33] and OpenMP [34] are based on this concept. In the x86 architecture, the main thread and its sub threads use the same “cr3” register to refer to the page table for the single shared VA space. Note that this “cr3” register-sharing activity by a multi-threaded process can easily be monitored by a full-system simulator such as Simics [35]. This is employed in the evaluation framework of this thesis described in Section 3.4.

Shared memory inter-process communication (IPC)

Shared memory IPC follows a different philosophy, which does not allow as much resource sharing as possible. Instead, this type imposes strict VA space protection among processes, while permitting one shared area, aptly called the “shared memory segment” (illustrated in Figure 3.1b). The only shared resource is this shared memory segment. A programmer may utilize this methodology by creating N processes with each one having a different VA space. The processes interact through the OS-granted shared memory segment. The N processes are not aware of each other’s existence, because they are conventionally created by a “fork()” system call from the “shell.” However, processes can communicate with each other through the shared memory segment. The “POSIX shared memory segment” is an example of this methodology. One downside is that this programming type usually causes synonym problems when using VIVT caches. (One can easily verify this phenomenon by simply observing that each process uses a different VA pointer value to point to the shared memory segment.) While the VA pointers of the different processes will have different values, they still physically point to the same “shared memory segment” location.

The VIVT PLMD cache architecture proposed in this thesis work is targeted at the multi-threaded style (Figure 3.1a) of address space-sharing.

3.3 Implementation

3.3.1 Manual Virtual Address Space Filtering

The PLMD cache is the quintessential embodiment of a “small-but-fast” cache that can satisfy the stack area requirement of high frequency accesses within a confined memory footprint. Moreover, owing to its small size, the PLMD cache can elevate system performance with minimal power overhead. Its VIVT cache structure can offer significant advantages over a PIPT equivalent. Most of the additional benefits emanate predominantly from the elimination of address translation:

- Larger allowed size for the “small-but-fast” cache: PIPT caches require address translation on every memory access. Therefore, the address translation path latency from the TLB to the cache would inevitably lower the maximum size that would still allow for a single-cycle hit. On the contrary, by removing the TLB access, a VIVT design enables larger cache sizes that are still able to achieve a single-cycle delay.

- Lower energy consumption: typically, the TLB has large (or full) associativity. This high associativity comes at the cost of energy expensive tag-matching operations. Instead, a VIVT cache can skip this process, as well as further overhead associated with a TLB miss.

However, it is imperative to address the pathological scenarios of synonyms and homonyms, which incur undue overhead in typical VIVT caches. In an effort to extract all the inherent benefits of VIVT caches, while minimizing the adverse side effects, the PLMD helper cache manually filters out all VA spaces except one main VA space (process). Thus, the run-time stack areas of this lone process become the exclusive occupants of the PLMD cache. This design naturally favors modern workstation or server environments that run one primary and dominant multi-threaded workload. The proposed VA space filtering approach within the PLMD helper caches results in the following important attributes:

- No synonyms: the inter-process synonyms are created (shared) through the SM domain, while by definition, the PLMD are the spatial FSMOs which are isolated from the SM domain, due to the scope. Note that, even within one VA space, the intra-process synonyms can still be deliberately created by the programmer, as described in [15, 16]. However, the PLMD are also free from this kind of synonym, due to their function frame stack scope.

- No homonyms: since other VA spaces (processes) are filtered out of the PLMD cache, they do not interfere with the selected, active VA space.

3.3.2 The Implementation of The Proposed VIVT PLMD Cache for Multi-Processors

Figure 3.2 illustrates the proposed mechanism. It depicts the location of the PLMD cache. The non-PLMD L1 data caches are implemented as physically indexed physically tagged (PIPT), whereas the accompanying PLMD caches are VIVT. It is important to note that the PLMD cache is small enough to guarantee a hit access latency of a single CPU cycle, while the regular L1 caches (Data/Instruction – for non-stack region) require two to three CPU cycles for a cache hit (in line with most L1 caches found in commercially available CPUs today). For example, the 32 KB L1 data cache of the Intel micro-architecture (codename

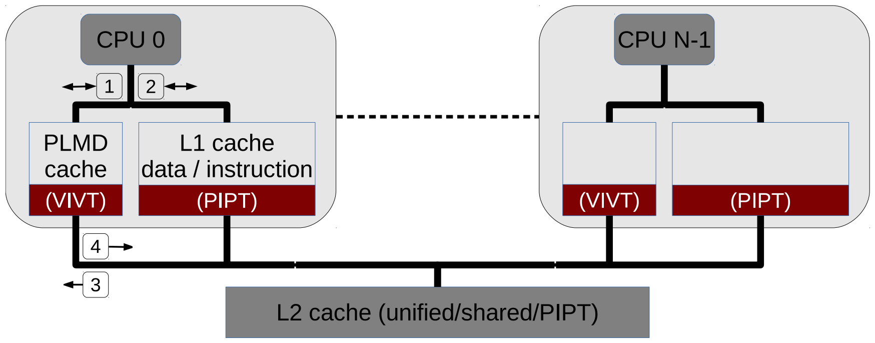


Figure 3.2: Implementation of the proposed VIVT PLMD cache for multi-core systems

Sandy Bridge) has a 4-cycle access time [36].

VA Space Filtering in Detail

The proposed VIVT PLMD cache revolves around the notion of VA space filtering. This technique allows one of the running applications to receive preferential treatment through an exclusive access to the PLMD cache. Any application can be chosen by the system administrator to receive this special status. Naturally, the selected application is one that tends to dominate all other running applications at a particular point in time. Once a program is granted this status, its VA space will be granted exclusive access of each core's PLMD cache in order to boost its overall performance.

This VA space filtering is achieved through the "cr3" register. The "cr3" register in the x86 architecture is the register pointing to a VA space's page table, and can be used as a unique VA space identifier. Hence, a system administrator can deliver a specific "cr3" register value to all the CPUs, in order to boost the performance of (potentially) one primary and dominant multi-threaded workload. This can be achieved through the invocation of a special command. Note, that this command is not a new ISA instruction, since the "cr3" register is already accessible to the programmer. Instead, operating system designers may create a new command that manages the newly proposed PLMD cache resources through the use of "cr3" register value manipulations. This command would, essentially, work in a fashion similar to the Unix "nice" command, which modifies the CPU scheduler priorities of all running processes in order to assign more CPU time to specific processes. Upon a change in the value of the “cr3” register, all VIVT PLMD caches flush their dirty cache lines into the Level 2 cache. This process prepares the PLMD caches to commence the handling of the newly selected VA space. The “cr3” VA space filtering mechanism was modeled in our evaluations using the Simics “Magic” instruction.

L1 Data Memory Reference Stream Decoupling

During VA space filtering, data cache accesses are directed in two separate pathways (designated by the labels “1” and “2” in Figure 3.2). Pathway “3” in Figure 3.2 denotes line fetching from the L2 cache, caused by L1 PLMD cache misses that require TLB lookup. Pathway “4” in Figure 3.2 denotes dirty write-backs from the L1 PLMD caches to the L2 cache, which also require TLB lookup. The stack area memory references of the active VA space are directed to path “1” (i.e., to the PLMD cache), while all other memory references are directed to path “2” (i.e., the regular L1 data cache).

3.4 Evaluation

3.4.1 Simulation Limitation

This section evaluates the performance improvement and energy reduction coming from these architected-register-like features of the L1 PLMD helper caches: [1]

- The cache intends to handle only the decoupled spatial FSMO data which are isolated from the SM domain.
- The cache functions fast with single cycle latency.
- The caches are implemented in the VIVT architecture to significantly reduce TLB accesses

[1]Due to the simulator limitation, the effect of full OoO execution opportunities (Section 3.1) of the PLMD helper caches is ***not*** evaluated.

Table 3.2: The simulation configuration of the two designs under evaluation

Simulation Configuration			
L1D Design (Regular + PLMD)	Component	Size/Number	Information
Baseline (32 KB + 0 KB)	L1 (data)	32 KB; 64 B; 2 way	3-cycle hit; write back/allocate
	L1 (instruction)	32 KB; 64 B; 2 way	3-cycle hit
	L2 (unified/shared)	512 KB; 128 B; 8 way	12-cycle hit; write back/allocate
	Main Memory	2 GB	218-cycle stall
	CPU Cores	8	Multi-threaded workloads each 16 sub-threads
	Coherence Protocol	-	MESI
Proposed (16 KB + 2 KB)	L1 (data)	16 KB; 64 B; 2 way	3-cycle hit; write back/allocate
	L1 (stack)	2 KB; 64 B; 2 way	1-cycle hit; write back/allocate
	L1 (instruction)	32 KB; 64 B; 2 way	3-cycle hit
	L2 (unified/shared)	512 KB; 128 B; 8 way	12-cycle hit; write back/allocate
	Main Memory	2 GB	218-cycle stall
	CPU Cores	8	Multi-threaded workloads each 16 sub-threads
	Coherence Protocol	-	MESI

3.4.2 Simulation Framework

In order to comprehensively evaluate the operational efficacy and efficiency of the proposed L1 PLMD cache, we employ Simics, a full-system simulator developed by Wind River [35]. Simics provides a "Magic" instruction that enables the simulated software on the target platform to deliver events to the simulator itself. The "Magic" instruction is incorporated as a special No Operation (NOP) instruction within the Simics's target ISA, and, hence, can be directly implemented in real computer systems. We use the "Magic" instruction to correctly acquire each multi-threaded benchmark's "cr3" register value, which is essential to the VA space filtering mechanism. The benchmark applications used in the simulations come from the PARSEC benchmark suite [37]. PARSEC is a popular benchmark suite containing parallel workloads from various emerging applications that are considered representative of next-generation shared-memory programs for multi-processors. Furthermore, all individual benchmarks can be executed in a multi-threaded mode and are, therefore, suitable for the proposed PLMD helper cache architecture for multi-processors.

The simulated target is the Simics "Tango machine" in a multi-core setting. This machine models an Intel Pentium 4 processor ($x86$-440bx machines). We simulate a system with *8 processing cores* running the Fedora Core 5 Operating System (with Linux kernel 2.6.15, including SMP support).

3.4.3 Designs Under Evaluation

Two designs are compared, by using the full-system simulation framework; namely, a "Baseline" design and the "Proposed" architecture. The "Baseline" system serves as the reference point and employs a 32-Kbyte L1 data cache per core, which is *twice as large* as the one in the "Proposed" setup (i.e., 16 Kbytes per core). Of course, the "Proposed" design has an additional 2-Kbyte L1 VIVT PLMD cache, for a total of 18 (16 + 2) Kbytes of L1 data cache per core. Despite this unfair comparison that tilts strongly in favor of the baseline design, it will be demonstrated in the following sub-sections that the proposed design comfortably outperforms the generic architecture, despite the significantly smaller L1 data cache. Both the 32- and 16-Kbyte L1 data caches (of the baseline and proposed architectures, respectively) have a 3-cycle hit latency. The 2-Kbyte PLMD cache has a single-cycle hit latency, similar to the one proposed in [11]. Table 3.2 summarizes the two design configurations.

3.4.4 Overall Performance Evaluation

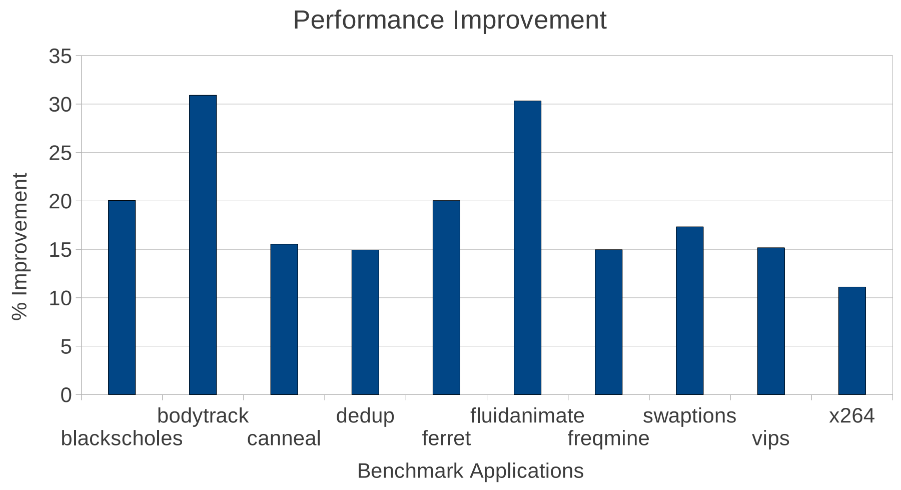


Figure 3.3: Overall performance improvement over the baseline design

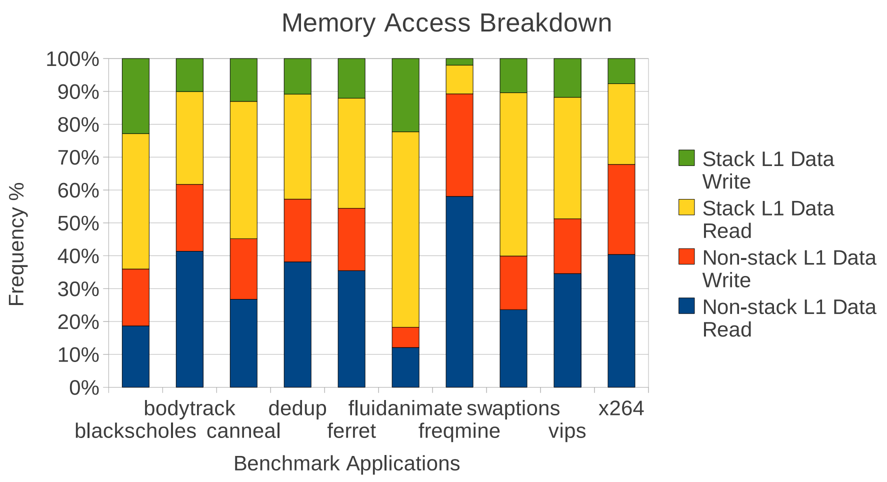
Figure 3.3 corroborates the prior assertions that the proposed VIVT PLMD cache instills dramatic performance improvements over the much larger L1 data cache of the baseline design. The (simple arithmetic) average overall performance improvement was found to be 19% for the 10 benchmark applications tested. Of course, the performance boost is attributed to the presence of the decoupled “small-but-fast” VIVT PLMD cache. Figure 3.4a depicts the breakdown statistics of the memory access patterns. One can notice the large percentage of accesses served by the PLMD cache. In fact, around 50% (on average) of all memory accesses can be directed to the small PLMD cache. Such a high percentage of accesses would easily justify the incorporation of the proposed VIVT PLMD cache in future multi-core systems.

3.4.5 TLB Access Behavior

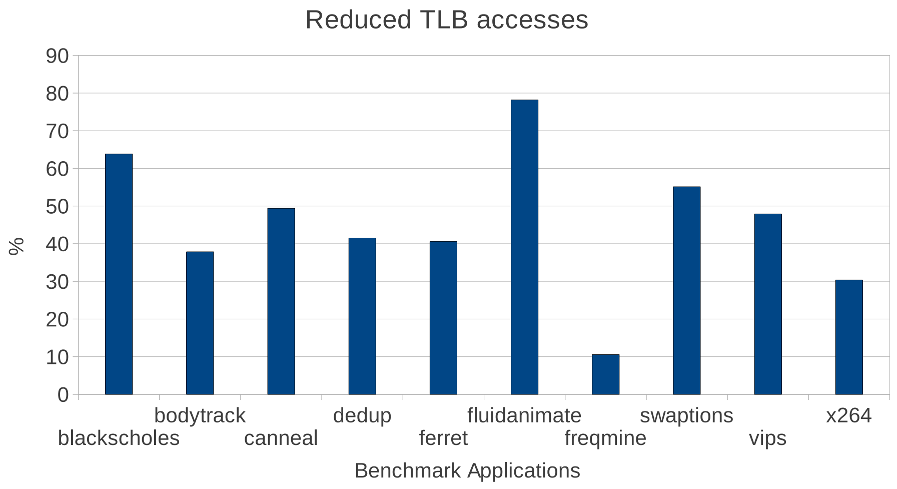
Figure 3.4a can also help us comprehend the TLB access behavior of the two designs, which is presented in Figure 3.4b. Since the PLMD cache read/write hit ratios are very high, the TLB access reduction percentages shown in Figure 3.4b are almost identical to the sum of the “Stack L1 Data Write” and “Stack L1 Data Read” portions in Figure 3.4a. It is important to note that the TLB access reductions in Figure 3.4b are a direct consequence of the VIVT PLMD cache hits, which obviate the need for TLB accesses. In fact, a TLB access is only needed in the infrequent cases of a PLMD cache read/write miss and a PLMD cache dirty write-back, (as illustrated by pathways “3” and “4”, respectively, in Figure 3.2). On average, the proposed design reduces the amount of TLB accesses by 45.5%. This substantial reduction in TLB references will, in turn, translate into a reduction in energy consumption. As previously described in Section 3.3.1, TLB accesses are energy expensive, due to the high associativities that typically characterize TLB implementations.

3.4.6 L1/L2 Statistics Analysis (Multi-Level Benefits of PLMD Caches)

As expected from the smaller L1 data cache (18 KB vs. 32 KB) of the proposed architecture, the L1 data cache hit ratio is noticeably lower, as compared to the baseline design. Figure 3.5 illustrates the L1 data read/write hit ratios of the two evaluated configurations.



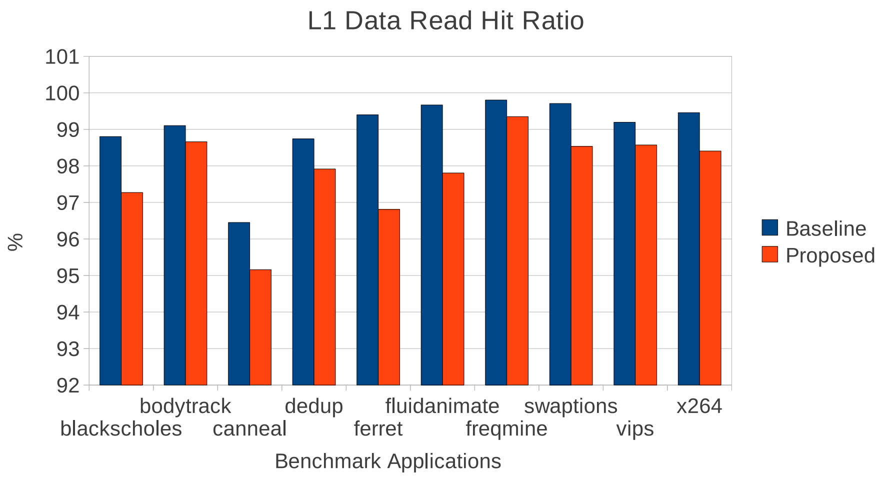
(a) Frequency breakdown of the memory area accesses



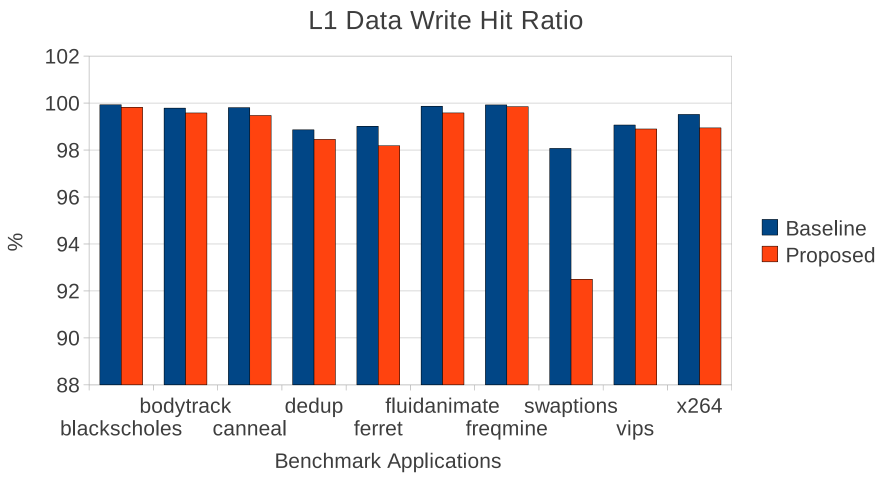
(b) Reduction in TLB accesses in the proposed (16KB + 2KB PLMD cache) design over the baseline (32K and no PLMD cache) design

Figure 3.4: Memory access pattern analysis

In the case of the proposed design, the L1 ratios refer to the *combined* hit ratios of the regular L1 data cache *and* the PLMD cache. Naturally, the lower L1 data hit ratios in the proposed architecture result in increased L2 cache accesses.

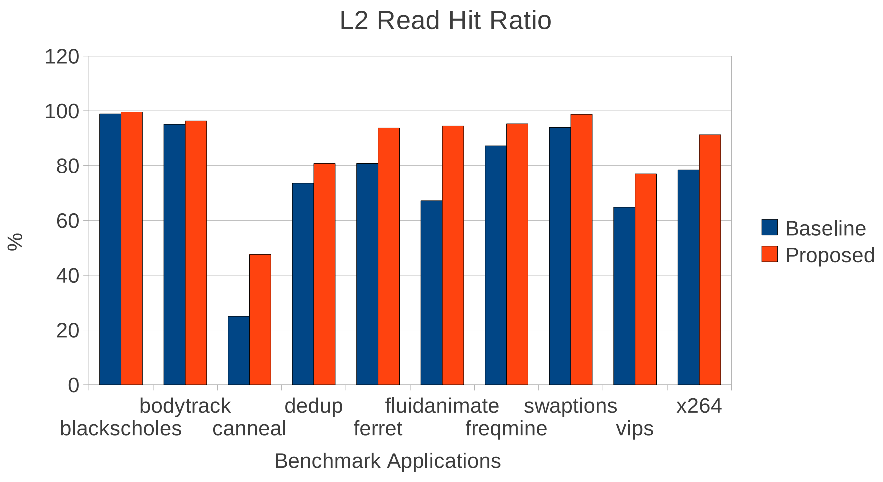


(a) L1 data read hit ratios

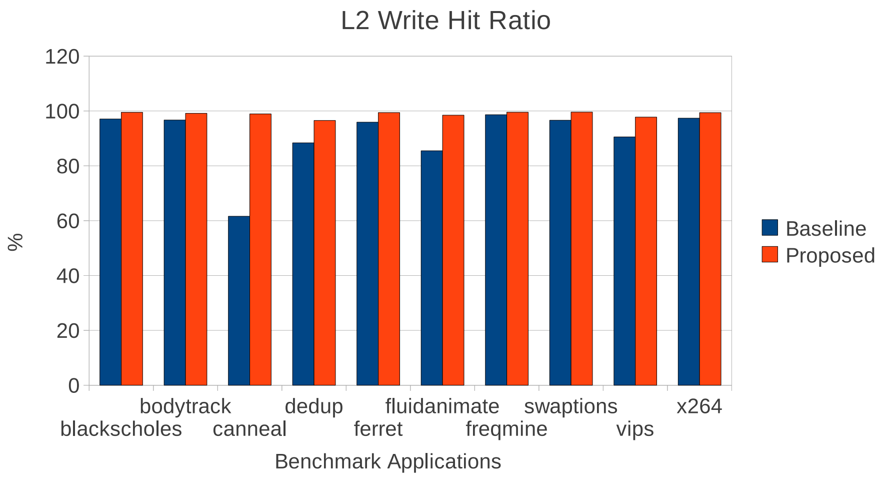


(b) L1 data write hit ratios

Figure 3.5: L1 data cache read/write statistics of the two evaluated architectures



(a) L2 read hit ratios



(b) L2 write hit ratios

Figure 3.6: L2 cache read/write statistics of the two evaluated architectures

Figure 3.6 shows the effect of the presence of the PLMD cache on the L2 cache behavior. Despite the elevated number of L2 cache accesses in the proposed design (caused by the lower L1 hit ratios), the existence of the PLMD cache beneficially alters the operational behavior of the L2 cache. As illustrated in Figure 3.6, all benchmark applications running on the proposed design experience increased read and write hit ratios in the L2 cache. This is because the PLMD cache's operation leads to the precipitation of a strong partitioning effect against data access thrashing between the L1 and L2 caches. This partitioning effect enables, for example, the "canneal" benchmark to exhibit an overall performance improvement of over 15%, even though it suffers from a relatively low L1 read hit ratio, as shown in Figure 3.5. This result is very important, since it showcases the *multi-level benefits* of the proposed PLMD cache. Its presence not only lowers the average L1 data cache hit latency, it also improves the performance of the L2 cache. Hence, the combined effect on overall performance far outweighs the reduction in L1 hit ratios (resulting from the smaller overall L1 regular data cache size).

3.5 Summary

The PLMD accesses have a profound effect on overall system performance, since application workloads tend to access the stack area data (i.e. the superset of the PLMD) at a very high frequency. It is apparent that contemporary ubiquitous CCSM multi-processor machines will continue to be front-runners in exploiting thread-level parallelism, in the future. Thus, it becomes imperative to devise a methodology that efficiently extends the architected registers in multi-processors, by making use of the spatial FSMO attribute of the PLMD. Toward this end, this thesis proposes the first PLMD cache architecture geared toward CCSM multi-core processors.

The presented PLMD cache employs a manual filtering technique that selects the run-time PLMD references of only one VA space (process) for caching into the per-CPU L1 VIVT helper caches. The filtering process very simply helps remove the synonym/homonym issue overheads and set-size limitations of VIVT caches, while, at the same time, it enables the extraction of all associated performance and energy advantages. The proposed PLMD cache mechanism is especially amenable to contemporary workstation and server systems that aim at running one primary and dominant multi-threaded application per machine.

The evaluation using a full-system simulation environment conclusively proves the efficacy and efficiency of the proposed VIVT PLMD cache for multi-core settings. Overall, this research certifies the viability of such helper PLMD caches in future CCSM multiprocessors and demonstrates their powerful capabilities.

CHAPTER 4
(PLMD 2) SUBTLETIES OF RUN-TIME VIRTUAL ADDRESS STACKS

The original research document of this thesis work was published in [7].

The run-time virtual address (VA) stack has some unique properties, which have garnered the attention of researchers. The stack one-dimensionally grows and shrinks at its top, and contains data that is seemingly local/private to one thread, or process. Accordingly, the optimization of accesses to local variables on run-time VA stacks has been widely investigated, focusing on these properties. Prior research indicates that real workloads tend to access the VA stack area with high temporal/spatial locality and within a small memory footprint [10, 2, 38, 11, 1]. This attribute directly stems from the fact that a process/thread always runs within a particular function (e.g., main()) whose frame is usually not very deep, and a stack simply grows and shrinks in one dimension at its top. Moreover, the VA stack data is assumed to be private to one thread/process.

These well defined attributes of the VA stack can be utilized to help improve the performance and energy/power consumption of the highest level of the memory hierarchy. The common practice of past work in this area is to first decouple the raw VA stack data from the entire data memory stream. However, the sophisticated behavior of contemporary operating systems (OS) and compilers not only prohibits this straight-forward data decoupling by VA, but it also invalidates the assumption that stacks are private to only one thread/process. This problematic phenomenon gives rise to potential function-critical hazards, which have (so far) been ignored in the literature. Hence, the behavior of run-time VA stacks should be fully investigated under the new light of contemporary OSes and compilers, so that its nuances do not lead to unexpected (at best), or incorrect (at worst) behavior.

This thesis work aims to demonstrate how conventional wisdom pertaining to the run-time VA stack fails to capture some critical subtleties and complexities. It first explores two widely established assumptions surrounding the VA stack area. Then, it demonstrates why these assumptions are invalid, and highlights the potential hazards regarding data consistency, shared memory consistency, and cache coherence. Finally, it suggests safeguards against these hazards, which helps correctly filter and decouple the PLMD objects from the stack address space. Overall, the work explores the function-critical issues that future operating systems and compilers should address to effectively help reap all the benefits of using run-time VA stacks.

4.1 Contributions

4.1.1 Breaking Chain of a Widespread Erroneous Assumption

There is a widespread assumption underlying the run-time VA stack area memory references, which all major prior work depends upon, including [10, 2, 1, 3, 4, 11, 12]. Moreover, much of this work is interconnected, building on concepts developed by the others, so it was difficult to isolate the sources of the hazards. For instance, the Stack Cache [10] and Access Region Locality (ARL) [2] inspired the SVF technique [1], while the SVF technique (and possibly Reverse Integration [3]) inspired the SSP mechanism [4].

However, the contemporary shared memory programming models are much more sophisticated and complicated than the erroneous assumption of prior work. This study breaks the chain of the misconception, by exploring the subtleties and nuances of VA stacks. In particular, it makes 3 contributions:

- It invalidates the assumption of a rigid dichotomy between the run-time stack area and non-stack areas, by demonstrating that OSes generate infrequent, yet non-negligible, VA-related aliases.
- It shows how the shared memory programming model allows VA stack data to also be visible across other threads/processes.

- It explains the potential hazards that these two realities may cause, regarding data consistency, memory consistency, and cache coherence.

4.1.2 Safeguards for Related Prior Work

This study also describes some safeguards which require the hardware-software interplay against these hazards. The safeguards can revive related prior research, by demonstrating how to correctly filter and decouple the PLMD objects from the run-time VA stack area data.

4.2 Background

The VA stack area stores: (1) spilled private local data (variables/arrays/memory allocations) exceeding the architected register file capacity; (2) function frame elements (arguments, the return address, and the caller/callee saved register context); and (3) local data to be shared with other threads/processes (more in Section 4.3.2). The first two types tend to dominate the run-time VA stack with high temporal/spatial locality. This is primarily why prior research in this area handled the run-time stack data (identified simply based on the VA) assuming the same properties as those exhibited by the register file.

User-level thread runtime libraries (e.g., Pthreads and OpenMP) operate on the principle of one main thread and its N sub-threads residing in a single VA space and comprising one process. The main thread and its sub-threads share some resources, such as page table, global variable area, and file descriptor. The principle of confinement within a single shared VA space aims to alleviate the heavy overhead of context switching during parallel execution. However, the main thread and all sub-threads must also have their own separate resources, such as VA stack area and registers. During the lifetime of a process (main thread), the per-thread stack memories are dynamically allocated/deallocated using mmap()/munmap() system calls (with flags like "MAP_STACK") on the creation/termination of those threads. This per-thread dynamic stack memory allocation strategy could

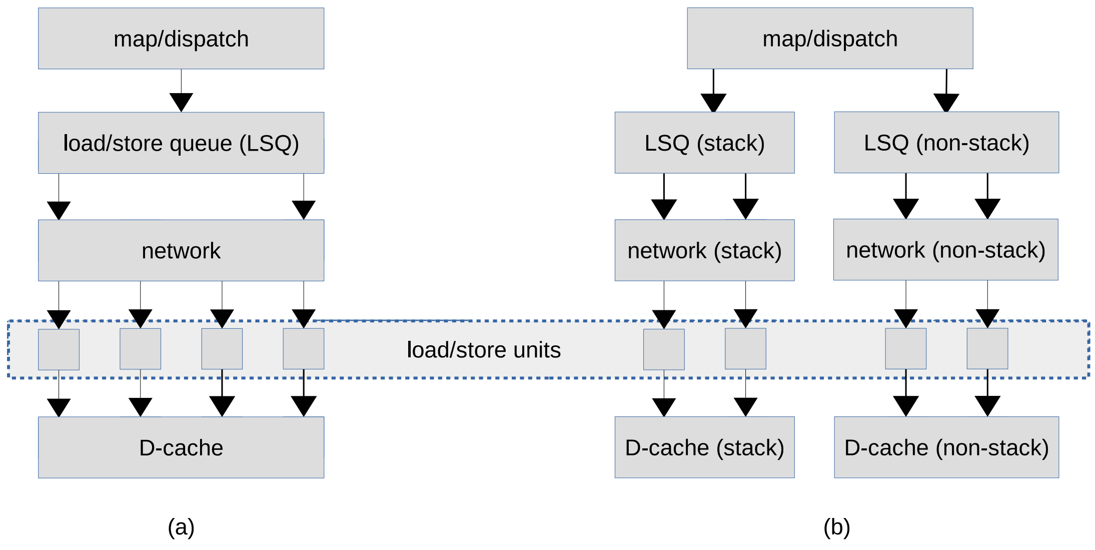


Figure 4.1: Diagram of an example pipeline of (a) a conventional superscalar processor and (b) a data-decoupled architecture. This is obtained from [10] and is redrawn here.

use the same VAs as those of non-stack memory allocations in a temporally interchangeable manner. The per-thread stack and non-stack memory can be allocated in a spatially "sandwiched" manner.

The designs of most prior research attempting to optimize the stack area data accesses can be abstracted as the stack/non-stack data-decoupled architecture (DDA) which is introduced in the Stack Cache design [10]. More specifically, the DDA abstraction has the partitioned stack/non-stack two-way pipelines from the specific stages (e.g. load store queue, OoO execution units, or cache memories), so as to safely exploit handling the stack data as the PLMD objects. Figure 4.1 illustrates the example pipeline of (a) a conventional superscalar processor and (b) a data-decoupled architecture. The PLMD helper cache architecture (Chapter 3) of this thesis has the partitioned L1 data cache memories, as an example implementation of the DDA abstraction.

4.3 Insights: Myths and Realities of Run-Time Stacks

The intricacies and nuances resulting from the interactions between the run-time stack and the OS/compiler invalidate two fundamental and widely accepted assumptions: (a) it is straight-forward to decouple VA stacks (dichotomy of stack/non-stack area), and (b) the VA stack data is private to one thread/process (privacy of VA stack data).

4.3.1 Myth1: Dichotomy of Stack/Non-Stack Area

In reality, the VA stack/non-stack areas can overlap, due to the following attributes:

Synonyms Reality (for Dichotomy Myth)

Actually, the stack/non-stack dichotomy myth resorts to identifying the data memory reference streams simply based on the VA, which is by nature the origin of enormous aliases. Under a contemporary OS, one physical memory frame is easily mapped by its virtual address page aliases, in the form of synonyms. The representative cases are “inter-process” (among different VA spaces) synonyms through the OS-provided shared memory segments, or shared files mapped by the mmap() function. However, an OS also allows “intra-process” (within one VA space) synonyms for flexible VA usage. For instance, the mmap() function presents even user code with a way to create a new mapping in the virtual address space of its calling process.

Moreover, the OS sometimes uses temporary synonyms in the kernel virtual address space to access user memory. For example, to process a Direct I/O request (used by databases) that bypasses the OS’s page cache, the kernel copies user data through a kernel address-space synonym. Additionally, kernel space synonyms are also used during a copy-on-write (CoW) page fault to copy content of the old page to the newly allocated page. Notice that Direct I/O is an example of heavy usage of a kernel address-space synonym, and the CoW fault is an indispensable service throughout the process lifetime. Figure 4.2 illustrates this aliased mapping, where the OS kernel generates a synonym of user stack data within its non-stack VA space when for example a Direct I/O or CoW operation is needed.

Listing 4.1 is a working code snippet using the Direct I/O technique.

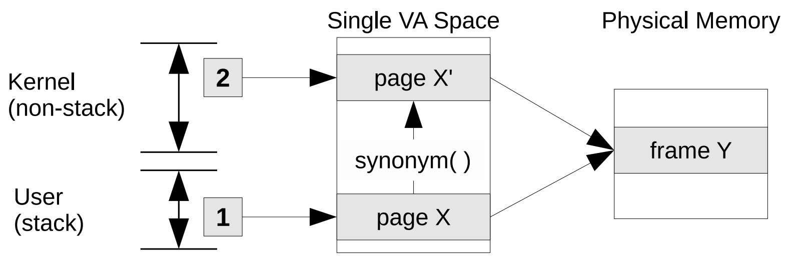


Figure 4.2: Kernel-space synonyms are observed in contemporary OSes.

```
1   int main()
2   {
3    int fd, ret;
4    #define BUFSIZE 1024
5    char buf[BUFSIZE] __attribute__((aligned(0x200)));
6
7    memset(buf, 0, sizeof(buf));
8    if ((fd = open("./file-to-read", O_RDONLY | O_DIRECT)) <0) {
9      printf("Failed to open %s\n", strerror(errno));
10     exit(1);
11   }
12
13   while ((ret = read(fd, buf, BUFSIZE))) {
14     if (ret < 0) {
15       printf("Failed to read %s\n", strerror(errno));
16       exit(1);
17     }
18
19     printf("%s\n", buf);
20   }
21
22   close(fd);
23  }
```

Listing 4.1: Direct I/O working example code

The code displays the content of the “file-to-read” file using Direct I/O, with the flag `O_DIRECT` (Line 8). The Direct I/O technique makes a synonym alias inside the kernel (non-stack), using the kmap*() function and accesses the local (in stack) buffer `buf[]` (Line 5) through the alias. Therefore, the `read` (Line 13) and `printf()` (Line 19) functions use two different paths to access the same data of the `buf[]` array. The `read()` function will fill the `buf[]` through the kernel synonym VA alias, while the `printf()` function will directly access `buf[]` through the regular stack VA. Direct I/O is a representative example of using a heavy amount of kernel synonyms.

Listing 4.2 is the actual CoW code obtained from the Linux kernel 3.14.

```
static inline void cow_user_page(struct page *dst, struct page *src, unsigned long va,
        struct vm_area_struct *vma)
{
        debug_dma_assert_idle(src);

        /*
         * If the source page was a PFN mapping, we don't have
         * a "struct page" for it. We do a best-effort copy by
         * just copying from the original user address. If that
         * fails, we just zero-fill it. Live with it.
         */
        if (unlikely(!src)) {
                void *kaddr = kmap_atomic(dst);
                void __user *uaddr = (void __user *)(va & PAGE_MASK);

                /*
                 * This really shouldn't fail, because the page is there
                 * in the page tables. But it might just be unreadable,
                 * in which case we just give up and fill the result with
                 * zeroes.
                 */
                if (__copy_from_user_inatomic(kaddr, uaddr, PAGE_SIZE))
                        clear_page(kaddr);
                kunmap_atomic(kaddr);
                flush_dcache_page(dst);
        } else
                copy_user_highpage(dst, src, va, vma);
}
```

Listing 4.2: The "cow_user_page()" function in the "mm/memory.c" file of Linux kernel 3.14

The final end products of the CoW operation are the "two complete different page instances (of the new `struct page* dst` and the old `struct page* src`) of threads or processes." The OS kernel creates the virtual address alias (`kaddr`) *temporarily* (Line 12) using the `kmap*()` function inside the kernel space and then destroys it quickly (Line 23). This alias of `kaddr` is used to point to the destination (new) page `struct page *dst`, during the copy operation (Line 21).

Reallocation/Remapping Reality (for Dichotomy Myth)

The cross-regional reallocation (of one virtual page) and remapping (to one physical frame) which the OS page allocator and memory management subsystem perform can destroy the mutually exclusive access to one memory object from either the VA stack, or the non-stack address space.

- Stack/non-stack page reallocation (Figure 4.3a). The OS page allocator can interchangeably allocate the same virtual page for the per-thread VA stack page and for the non-stack page. Thus, a virtual page that once belonged to the VA stack can belong to the VA non-stack, and vice versa.

- Stack/non-stack page remapping (Figure 4.3b). Conversely, the OS memory management subsystem can interchangeably get the same physical frame mapped for the per-thread VA stack page and for the non-stack page. Thus, a physical frame that once belonged to the VA stack page can belong to the VA non-stack page, and vice versa.

Note that the per-thread stack and non-stack pages can be allocated in a spatially "sandwiched" manner, in the VA space.

4.3.2 Myth2: Privacy of the VA Stack Data

Each thread of a process has its own individual run-time VA stack. Unfortunately (and contrary to popular belief), this does not mean that the VA stack data is always non-visible

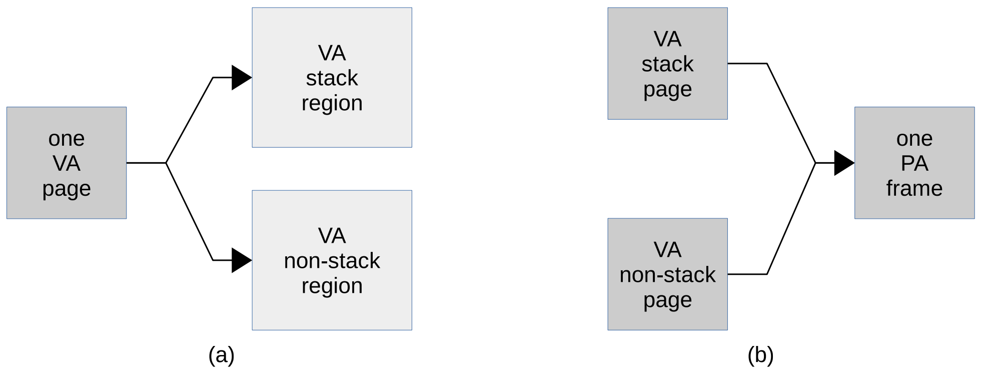


Figure 4.3: Page reallocation (a) and remapping (b).

to other threads/processes, as will be explained shortly. Please note that in order to make VA stack data visible to other threads/processes, compilers cannot help but forcibly spill (commit) the local data into the VA stack area (so that VAs can be assigned to leverage the shared memory model), even when there are enough available architected registers.

```
void foo() {
  double dotp;
  ...
  dotp = 0.0;    // initialization of ‘‘dot product’’
  # pragma omp parallel shared ( N, X, Y ) private ( i )
  # pragma omp for reduction ( + : dotp )
  for (int i = 0; i < N; i++ )
    {dotp += X[i] * Y[i];}
  ...
  ... = dotp * dotp;    // use of dotp
```

Listing 4.3: An *OpenMP* pseudo-code snippet evaluating the variable `dotp` in a fork-join task model.

Scope Reality (for Privacy Myth)

Within one VA space, virtual memory is always shared (e.g., through pointers) and, hence, different thread stacks are not protected from other threads. On the contrary, registers are never shared [39]. Since every thread can access every memory address within the process address space, one thread can read, write, or even completely wipe out another thread's stack [40]. This, one – entirely visible – VA space across multiple threads helps perform parallel operations, as illustrated in Listing 4.3. This parallel code snippet calculates the variable `dotp`, by using the `OpenMP`'s fork-join task model, i.e., its default style. All sub-threads spawned inside the foo() function can see `dotp`, even though it is a local variable defined in the foo() function. The fork-join model is merely a standardized example case. Programmers can use more complicated multi-threaded programming styles, such as the thread-pool model, thereby leveraging this scope capability.

The great actual example of the non-private stack data is the per-thread qnodes of MCS lock [41]. Each per-thread qnode is declared as the automatic variable inside the run-time VA stack of each lock-competing thread, and is passed as the arguments to the MCS lock functions. The adjacent predecessor and successor lock-waiter threads have to share their per-thread qnodes with each other, to perform the queuing lock mechanism which reads and updates each other's qnode.

Additionally, the non-private (i.e. shared) stack data accesses are also observed in the code of the real benchmark suites, as in Listing 4.4.

```
void CPearlInfEngine::ParallelProtocol()
{
    if( m_maxNumberOfIterations == 0 )
    {
        SetMaxNumberOfIterations(m_numOfNdsInModel);
    }

    int i, j;
    int converged = 0;
    int changed = 0;
    int iter = 0;
    const CGraph *pGraph = m_pGraphicalModel->GetGraph();
    int nNodes = m_connNodes.size();
    int nAllMes = m_messagesFromNeighbors.size();

    ..... <deleted part> .....

    messageVecVector newMessages(m_messagesFromNeighbors);

    ..... <deleted part> .....

    while ((!converged)&&(iter<m_maxNumberOfIterations))
    {
        //delete all old data
        //work with new data
        if (iter > 0)
        {
#ifdef _OPENMP
#pragma omp parallel for
#endif
            for( i = 0; i < nAllMes ; i++)
            {
                for( j = 0; j <m_messagesFromNeighbors[i].size(); j++)
                {
                    delete m_messagesFromNeighbors[i][j];
                    m_messagesFromNeighbors[i][j] = newMessages[i][j];
                }
            }

            //compute beliefs
            changed = 0;
#ifdef _OPENMP
#pragma omp parallel for schedule(dynamic) reduction(+:changed)
#endif
            for( i=0; i < nNodes; i++ )
            {
                if( !m_areReallyObserved[m_connNodes[i]])
                {
                    message tempBel = m_beliefs[m_connNodes[i]];
                    ComputeBelief( m_connNodes[i] );
                    changed += !tempBel->IsEqual( m_beliefs[m_connNodes[i]],
                        m_tolerance);
                    delete tempBel;
                }
            }
            converged = !(changed);
        }
```

Listing 4.4: A code example (pnlPearlInferenceEngine.cpp) from the BioParallel benchmark suite [42]

In Listing 4.4, the variable `changed` is declared as the VA stack data inside the CPearlInfEngine::ParallelProtocol() function and initialized to zero (Line 10 and 41). Then, the variable is shared and processed by sub-threads of the parallel loop that has the `#pragma omp parallel reduction` directive (on Line 43).

Inter-Process Synonym Reality (for Privacy Myth)

Actually, the inter-process communication (IPC) programming model allows even VA-stack data to be visible across different processes. For example, multiple processes can map shared files (e.g., using the mmap() function), or OS-provided shared-memory segments (e.g., using the POSIX shmat() function) into their own stacks. Accordingly, these mappings require proper pre-allocated space inside the VA stacks of the associated processes. Listing 4.5 is the example working code for these mappings which shows that the Linux `mmap()` function is so versatile and flexible.

```
void
foo (int fd, size_t len)
{
  /* 4 KByte (0x1000) buffer in VA-stack, aligned to 4 KByte page */
  char buffer[0x1000] __attribute__ ((aligned (0x1000)));
  char *mapped_addr;

  mapped_addr = mmap (buffer, len, PROT_READ,
            MAP_PRIVATE | MAP_FIXED, fd,
            0 /* offset for mmap() must be page aligned */ );
  // successful "mmap" ?
  assert (mapped_addr != MAP_FAILED);
  // successful mmap spot ?
  assert (mapped_addr == buffer);

  do
    {
      /* write the file into the shared memory segment */
      write (STDOUT_FILENO, mapped_addr, len);
    }
  while (0);

23   munmap (mapped_addr, len);
24 }
25
26
27 int
28 main (int argc, char *argv[])
29 {
30   int fd;
31   struct stat sb;
32   size_t length;
33
34   fd = open (argv[1], O_RDONLY);
35   // successful file "open" ?
36   assert (fd != -1);
37   // successful file size obtaining ?
38   assert (fstat (fd, &sb) != -1);
39
40   length = sb.st_size;
41
42   foo (fd, length);
43
44   close (fd);
45 }
```

Listing 4.5: Working code with the mmap() function mapping a file inside the VA stack

The `foo()` function declares the `char buffer[]` array (the target VA memory location) inside its stack memory space (Line 5). Then, the `mmap()` function maps the shared file to the `char buffer[]` array (Line 8). The important point is to allocate the target VA stack memory location at 4 Kbyte page granularity (in alignment and size, i.e., 4, 8, 16, ... Kbytes), as shown on line 5 of Listing 4.5. Recap that a similar alignment attribute was also applied to the `char buf[]` in Listing 4.1 (Direct I/O). The mapping technique with the shared-memory segment, using the `shmat()` function can also be employed, for the same purpose.

4.4 Insights: Potential Hazards

This section explains certain function-critical hazards that the OS and compiler peculiarities outlined in Section 4.3 give rise to the aforementioned stack/non-stack two-way data-decoupled architecture (DDA). Note that all hazards described below may lead to functional incorrectness (i.e., erroneous results), regarding data consistency, memory consistency, and cache coherence. The safeguards in terms of the micro-architecture, OS, and compiler are suggested after these hazards are analyzed.

4.4.1 Dichotomy Affects Data Consistency and Dependencies

Synonym Issue

The OS-driven synonym issue of Section 4.3.1 invalidates the assumption that stack and non-stack data streams can be perfectly decoupled. In other words, data can be accessed from either, or both, of these two areas. Consequently, duplicated data streams will occupy the stack/non-stack two-way DDA pipelines, thereby violating data consistency and dependencies. Compilers and hardware cannot resolve this problem on the fly, unless the OS provides assistance.

Reallocation/Remapping Issue

The reallocation/remapping issue of Section 4.3.1 can leave stale residue memory objects in the stack/non-stack two-way DDA pipelines and cache memories. The stale cache lines can overwrite the fresh lines in the merging point (e.g. lower level shared cache memories) when they are dirty-written-back to there.

Once the absolute dichotomy myth is broken, the system can hardly benefit from employing the two-way stack/non-stack DDA pipelines, because it then requires the intervention between the two-way pipelines, to ensure data consistency and enforce dependencies. Particularly, data coherence between the two-way DDA pipelines should be maintained. This intra-node cache coherence will ironically complicate the design, especially because the mechanisms should also consider memory ordering between the intra-node two-way DDA pipelines, in multi-processor environments. For instance, the load/store queue (LSQ) snooping operation is required between the two DDA LSQes of the same CPU node.

4.4.2 Privacy Affects Cache Coherence and Memory Consistency

The OS- and compiler-induced scope and synonym issues of Section 4.3.2 create false positive inter-thread/process visibility in the VA stack data. While most of the VA stack data is purely private to one thread/process, a small amount is not. This false positive visibility requires the VA stack data to be subject to cache coherence and memory consistency. Note that there is neither register coherence, nor register consistency regulation, because registers are not accessed by VA, but by name, and, hence, they are private to one thread/process.

Cache Coherence Issue (on One Stack Data)

To illustrate this, take the scenario depicted in Listing 4.3 again. The variable `dotp` is defined in the foo() function. Then, the compiler would forcibly spill (store) `dotp` onto the VA stack (on Line 2) for the fork-join sub-threads to see it, and a register will be finally filled (load) with `dotp`'s updated value (on Line 10), after the parallel part (Lines 5 to 8). If the initializing store on Line 4 and the parallel part (Lines 5 to 8) of Listing 4.3 are not concurrently subject to cache coherence, the system could get the incorrect value of `dotp` on Line 10.

Another example is the MCS lock [41] which uses the per-thread qnodes. As explained in Section 4.3.2, each per-thread qnode is declared as the automatic variable inside the run-time VA stack of each lock-competing thread, and is passed as the arguments to the MCS lock functions. The adjacent predecessor and successor lock-waiter threads have to share their per-thread qnodes with each other, to perform the queuing lock mechanism which reads and updates each other's qnode. Therefore, without the proper cache coherence on the per-thread qnodes, the MCS lock design would not work at all.

Memory Consistency Issue (between Stack/Non-Stack Data Streams)

The two-way stack/non-stack DDA pipelines intend to remove the intervention between the two data streams. However, this no intervention combined with the shared stack data references makes the multi-processor implementations of the architecture unprotected to memory-order distortions between the two data streams. This hazard becomes more exacerbated if the two-way stack/non-stack DDA pipelines employ the OoO execution scheme: the case further complicates (i.e. decouples) the OoO execution pipeline and its cache memories which are the two main factors already notorious for affecting memory ordering. Note that adding any memory ordering intervention between the two-way stack/non-stack DDA pipelines can adversely affect their performance optimization goal, as explained in Section 4.4.1.

One example explaining this memory-order distortion hazard is the MCS lock, once again. Recall that the per-thread qnodes are declared inside the run-time VA stack, while the critical sections are mostly declared in the non-stack area. Thus, the multi-processor implementation of the two-way stack/non-stack DDA pipelines can easily distort the memory write operations on the per-thread qnodes and on the critical sections in a random manner, which destroys the spinlock synchronization.

Listing 4.6 depicts another scenario where such memory-order distortion takes place in the multi-processor implementation of the two-way stack/non-stack DDA pipeline architecture.

```
/* sharedData: declared in global area */

/* Thread to be spawned for parallel processing */
void processing_thread(lock_t *lptr) {
....
```

```
 6    /* Synchronization */
 7    spin_lock(lptr);     // *lptr = LOCKED,    (acquire the lock)
 8    sharedData = ...;    // updating sharedData
 9    spin_unlock(lptr);  // *lptr = UNLOCKED, (release the lock)
10  ....
11  }
12
13  void parallel_produce_cb() {
14    lock_t lock;
15
16    /* spawning parallel threads */
17    thread_create(..., processing_thread, (void *)&lock);
18    ....
19  }
```

Listing 4.6: A code snippet of the run-time binding parallel callback function `parallel_produce_cb()` to be called by the producer of a "producer-consumer" style program.

The code describes `parallel_produce_cb()` which is the run-time binding parallel callback function to be called by the producer of a "producer-consumer" style program. Let the program declare `sharedData` in the global area such as OS-provided shared memory segments, so the producer and consumer can share the data. The call back function spawns multiple instances of the `processing_thread()` thread to conduct some parallel processing (Line 17), and finally has all those sub-threads synchronized when updating the `sharedData` (Line 7 to 9). The synchronization uses X86-TSO-architecture-style spinlocks because spinlocks are the best choice for the extremely short critical section. Note that because it is a run-time binding callback function, the `parallel_produce_cb()` function deliberately declares the `lock` variable inside its stack (Line 14), instead of inside the global shared data structure containing `sharedData`. Basically, this is the most recommended way where the "fork-join" style multi-threaded function can benefit from allocating its lock variables as local automatic variables. The other inefficient alternative way is dynamically allocating and deallocating the lock variables in the global memory location, on every run-time "fork-join" operation.

The multi-processor implementation of the two-way stack/non-stack DDA pipelines which realizes dichotomy myth would easily distort the memory ordering between Line 7 and 8 or between Line 8 and 9, in the random order among processors. This is due to the fact that `lock` is in the stack VA space, while `sharedData` is in the non-stack VA space. This distortion could then break the synchronization of `sharedData`. Modern processor designs, such as X86-TSO, have stronger memory consistency models to ease the coding process of programmers and compilers, regarding the memory ordering issues.

4.5 Safeguards for Stack/Non-Stack Decoupled-Data Architecture

4.5.1 Suggested Safeguards to Achieve Real Dichotomy

Synonym Reality

For the "kernel-space" and "inter/intra-process" synonym issues, we can borrow from Basu et al. [21] the VA encoding counter-measure for their opportunistic virtual caching (OVC), and we can enhence the technique. That is, an OS can encode the VA of each synonym page without breaking the existing VA mechanism, to specify the stack/non-stack VA region information of the very original page to which the synonym page is mapped. The solution requires that the OS is given enough unused VA room (as in 64-bit Linux) to encode the VA. For 64-bit Linux, the augmented OS VA range allocator can assign each synonym page with a special VA which contains this encoded original region information in any of the unused bits (between the 49th VA_{48} bit and the 64th VA_{63} bit). This selective VA encoding of synonyms does not overlap with the existing VA space. Then, the hardware VA-bound check can eventually identify whether the origin of the synonym page is either a VA stack, or non-stack page, by the selectively encoded bits. This identification finally directs the synonym pages to the correct decoupled (stack or non-stack) pipeline.

Recall that the stack/non-stack DDA machines can be free from the intra-process read-write synonym issue between the run-time stack and non-stack areas, if they feed the PLMD (filtered from the raw VA stack data) to the decoupled stack pipeline, as explained in Section 3.3.1. On the contrary, Basu et al. [21] classified a process having the user-space intra-process synonyms as the offending process and turned off their OVC for such cases.

Reallocation/Remapping Reality

To resolve the reallocation/remapping issue, the OS should invalidate the associated residue cache lines of either data stream in the pipeline(s) of all CPUs, whenever the reallocation/remapping happens. Fortunately, these invalidations do not need a dirty write-back into the lower memory sub-system.

4.5.2 Suggested Safeguards to Achieve Real Privacy

Both of the scope issue and inter-process synonym issue of Section 4.3.2 can be eliminated, by adopting a compiler transformation technique. For instance, the compiler can create the new VA stack area instance per the existing one, and then filter and move all the PLMD of the original existing stack area to the newly created stack instance.

Fortunately, the required compiler transformation technique can be directly borrowed from the safe stack scheme [13] which creates said new "safe stack" instances, to enhance the security of the function call frame stacks. It is also easy for hardware to detect and decouple the safe stack data references from the rest of the data references, because the scheme uses the "hardware-based instruction-level safe region isolation" technique. Therefore, the micro-architecture can identify the safe stack data accesses, by the dedicated segment register used to access the safe stack.

4.6 Summary

This study aims to capture the intricacies and nuances of the hazardous interplay between the run-time VA stack and modern OSes and compilers. Unlike popular belief and practice, the demonstrated problematic scenarios indicate that the VA stack data are just the superset of the PLMD objects, and, thus, cannot be directly treated as the architected register extensions. Additionally, this study suggests simple safeguards and modifications to the OS and/or compiler to eliminate the function-critical hazards of treating the VA stack data as the PLMD. The full-system vertical study presented in this work aims to guide the future OS and compiler designs, such that the previously unaddressed issues pertaining to the use of the VA stack are prevented from leading to erroneous behavior.

CHAPTER 5
(CSLVs) QT SPINLOCK: QUEUING TICKET SPINLOCK FOR LINUX KERNEL

The spinlock is generally the best synchronization option to protect the short sized shared data in memory. In this regard, the importance of the spinlocks inside the OS kernels can not be emphasized enough, because the kernels have a lot of essential short sized data shared by the multiple concurrent kernel threads [43, 24]. Every spinlock design runs with the seemingly simple spin-waiting loops. However, the spin-waiting synchronization is all about the memory consistency issues and easily becomes the most massive source of the concurrent fine-grain cache coherence operations generating the typhoon-like cache line bounces. Ironically, it is only after going-through this heavy SM synchronization overhead that the concurrently competing threads can safely access the very short sized shared data.

The ticket spinlock is a simple centralized design which out-performs other software locks, until the lock contention makes the lock variable the cache coherence hot spot [24]. Meanwhile, the queuing lock is the localized design which paradoxically executes by far the best under the lock contention.

This research proposes the queuing ticket (QT) spinlock which is specially designed for the Linux kernel. As the name implies, the QT spinlock is the transformed ticket spinlock design which silently performs the queuing lock operations as well. To this end, it works with light-weight micro-architectural support. The QT spinlock ultimately reduces the spinlock synchronization overhead, by restricting each lock-competing thread to start caching the lock variable "only when" it acquires the lock. The key enabler of this overhead reduction is handling the ticket spinlock variable (the CSLV) as the temporal FSMO, until the thread finally exits the spin-waiting iteration, as introduced in Section 1.2.2. In this way, the QT spinlock which is the centralized spinlock design can even better scale than the state-of-the-art software locks, especially on the large scale CCSM NUMA systems.

5.1 Insights and Contributions

5.1.1 Ultimate Minimalism in Cache Line Bouncing

The QT spinlock design proposes the contrarian idea of enabling the centralized spinlock (CSL) to out-perform all the other sophisticated spinlocks, by handling the centralized spinlock variable (CSLV)s as the temporal FSMOs. The main insight is that (1) the centralized spinlock design has the shortest critical-path of cache line bounces, and (2) the Linux kernel has the *"CPU-Affinity Spin-Wait"* property (explained later in Section 5.2.6). The design aims at unleashing the power of the centralized spinlock design.

As described previously, it is ironic that the spinlock synchronization provokes much higher overhead than its short sized critical section does. The temporal FSMO concept helps resolve this with these features:

- The concept restricts each lock-competing thread to cache the lock variable line, "only after" eventually acquiring the lock.

- The concept makes the thread perform the spin-wait operation on the proper hard-wired value which keeps the spin-wait loop iterating, until receiving the wake-up signal from the lock-owner thread.

- The concept makes the threads send the required signals to the spin-waiting CPUs, through the direct CPU-to-CPU message-passing micro-architecture, instead of through the SM domain.

In this way, the QT spinlock design simply incurs the **N** times lock variable cache line moves, in the lock contention case having the **N** lock-waiting threads (assuming no thread CPU migration). In other words, the lock variable cache line moves only on each lock ownership CPU migration. More specifically, the QT spinlock suppresses the unnecessary moves of the lock variable cache line, among the lock-owner and lock-waiter threads, which are caused by both of the true and false sharing effects. The introduced temporal FSMO concept helps the QT spinlock accomplish this overhead reduction mechanism without damaging the underlying SM consistency model.

This is by far the ultimate counter-measure to alleviate the interference of the cache line moves surrounding the CSLV. Related prior work of such a counter-measure includes the evolution from the test-and-set (TS) spinlock into the test-and-test-and-set (TTS) spinlock [44].

5.1.2 Small Lock Variable Size

The QT spinlock brings in the beneficial by-product of great significance that its lock variable becomes small enough (e.g. 4 bytes). This is by virtue of the fact that the QT spinlock is basically the ticket spinlock design whose lock variable is a CSLV composed of the simple ticket head/tail value pair. The spinlock variable size seemingly does not look vital for the most lock designers. However, the lock variable size determines the eligibility of the lock design in the target programs: it is common practice to embed the lock variables in the very shared data structure for them to protect, with the cache line alignment being kept for the best performance. As a consequence, the codes can have trouble re-shaping the shared data structure, with even the subtle lock variable size increment (e.g. notably "struct page" of the Linux kernel [25]). Note that the Linux kernel cannot adopt almost all the sophisticated scalable software spinlock schemes (e.g. the MCS lock [41], CLH lock [26], hierarchical CLH lock [27], and K42 lock [28]) as the default lock, because of the 4 byte lock variable size limitation.

5.1.3 Proper Approximate Simulation Model

In fact, it is very difficult to evaluate the spin-waiting synchronization designs using software multi-processor simulators. Most of the simulators are optimized for the macro-scopic cache coherence simulation to avoid taking an outrageously long simulation time. In contrast with this, the spin-waiting synchronization could be the most massive source of the micro-scopic fine-grain cache coherence operations accompanying the typhoon-like cache line bounces.

Therefore, this thesis work explores the essentials which the simulators should consider to properly approximate the evaluation of the spin-wait synchronization performance.

5.2 Background

Despite looking quasi-simple, the spin-waiting synchronization primitives are the main source of sophisticated programming techniques, such as employing inline assembly atomic memory operations, and handling memory ordering issues of the memory consistency model.

This section aims to help understand the contribution of the proposed QT spinlock design by explaining the evolution of the main-stream spinlock designs. Also presented is the knowledge required to understand the spinlock designs inside the Linux kernel.

5.2.1 Test-and-Set (TS) Spinlock: Centralized Spin-Lock

```
lock_t Atomic_TestAndSet(lock_t *ptr) {
    lock_t rv = *ptr;
    *ptr = LOCKED;
    return rv;
}

volatile lock_t lock = UNLOCKED;

void spin_lock(lock_t *lptr) {
  /* spin-waiting loop */
  do {} while(Atomic_TestAndSet(lptr) == LOCKED);
}

void spin_unlock(lock_t *lptr) {
  *lptr = UNLOCKED;
}
```

Listing 5.1: A pseudo-C code of the test-and-set (TS) spinlock.

Listing 5.1 depicts the pseudo-C code of the test-and-set (TS) spinlock. The TS spinlock is a naive centralized spin-lock (CSL) design notorious for generating the worst cache coherence contention traffic, among all the spinlock designs ever proposed. On every spin-wait iteration (Line 11), the TS spinlock executes the heavy atomic memory exchange operation of the `Atomic_TestAndSet()` instruction which "always" launches the write-invalidate messages for the lock variable cache line. This way, each spin-wait iteration not only suffers from, but also creates the cache miss penalties for the lock variable, induced by the heavy atomic memory exchange instruction.

5.2.2 Test-and-Test-and-Set (TTS) Spinlock: Centralized Spin-Lock

```
 1 volatile lock_t lock = UNLOCKED
 2
 3 void spin_lock(lock_t *lptr) {
 4   /* nested spin-wait loops */
 5   do {
 6     do {} while (*lptr == LOCKED);              // spinning on the cache line
 7   } while (Atomic_TestAndSet(lptr) == LOCKED); // spinning on atomic test-and-set
 8 }
 9
10 void spin_unlock(lock_t *lptr) {
11   *lptr = UNLOCKED;
12 }
```

Listing 5.2: A pseudo-C code of the test-and-test-and-set (TTS) spinlock.

Listing 5.2 depicts the pseudo-C code of the test-and-test-and-set (TTS) spinlock. The TTS spinlock is a CSL design which can remarkably reduce the cache coherence contention traffic of its predecessor TS spinlock [44]. To that end, the TTS spinlock simply adds an inner spin-wait loop (Line 6) to the existing TS spin-wait loop (Line 5 to 7), making the two loops nested. The added inner loop spin-waits on the "cached" lock variable, so as to enable the thread to delay executing the TS spinlock operation until the lock variable is released. In short, the TTS spinlock is the "trylock" version of the TS spinlock. That is to say, each spin-waiting thread first confirms that the lock is currently released. If it is true, the thread executes the heavy atomic memory exchange instruction to try to acquire the lock.

5.2.3 Ticket Spinlock: Centralized Spin-Lock

```
typedef struct lock {
  ticket_t headTicket, tailTicket;
} lock_t;

lock_t lock;
lock.headTicket = lock.tailTicket = 0;

lock_t Atomic_XchgAdd(lock_t *ptr, int increment) {
  lock_t rv = *ptr;
  *ptr += increment;
  return rv;
}

void spin_lock(lock_t *lptr) {
  lock_t lock_status = Atomic_XchgAdd(lptr, (0x1 << TAIL_TICKET_SHIFT_OFFSET));

  /* lock acquisition decision */
  if (lock_status.headTicket == lock_status.tailTicket)
    return;

  /* spin-wait loop */
  ticket_t myticket = lock_status.tailTicket;
  do {} while(lptr->headTicket != myticket);
}

void spin_unlock(lock_t *lptr) {
  lptr->headTicket += 1;
}
```

Listing 5.3: A pseudo-C code of the ticket spinlock.

Listing 5.3 depicts the pseudo-C code of the ticket spinlock. The ticket spinlock is a CSL design which works in the perfect fair FIFO fashion, unlike the TS and TTS spinlock. The lock variable is plainly composed of the `headTicket` and `tailTicket` value pair (Line 1 to 3), both of which are initialized to the same value. Basically, the `spin_lock()` and `spin_unlock()` functions respectively increment the `tailTicket` and `headTicket` values in memory, by one. Hence, the gap between the two values means the total number of the lock-owner thread and lock-waiter threads, at the moment. Incrementing the `tailTicket` value requires the `Atomic_XchgAdd()` instruction which performs the atomic "exchange-and-add" memory operation on the lock variable in memory (Line 15). In other words, it simultaneously returns the current lock variable in memory and increments its `tailTicket` field in memory. The return value of the `Atomic_XchgAdd()` instruction is used to make the lock acquisition decision (Line 18), and to set the `myticket` value (Line 22). Hence, each spin-wait thread is assigned the current `tailTicket` field of the lock variable as its unique `myticket` value. The proposed QT spinlock makes use of the property that this kind of atomic memory instruction can easily decide the lock acquisition status. Hence, the QT spinlock gets the atomic instruction augmented with additional necessary micro-ops, for this purpose.

The great improvement of the ticket spinlock over the TS and TTS spinlock is that the spin-waiting loop incurs ***no*** atomic memory operation. The loop just compares the `myticket` value (which is stored in a register) with the current `headTicket` value (which is stored in memory). However, this spin-wait loop operation still makes the lock variable cache line the heavy hot spot in memory.

5.2.4 General Queuing Spinlock Scheme

As the name implies, the queuing spinlock designs operate with the lock-owner and lock-waiter threads organized in the queue structure. Each lock-waiter thread spin-waits on its own local `locked` value contained in its per-thread "qnode." Therefore, the spin-wait induced notorious cache coherence contention surrounding the centralized spin-lock variable (CSLV) disappears. The queue is basically a linked-listed structure of the per-thread qnodes, and is established and managed by the concurrently running lock-owner and lock-

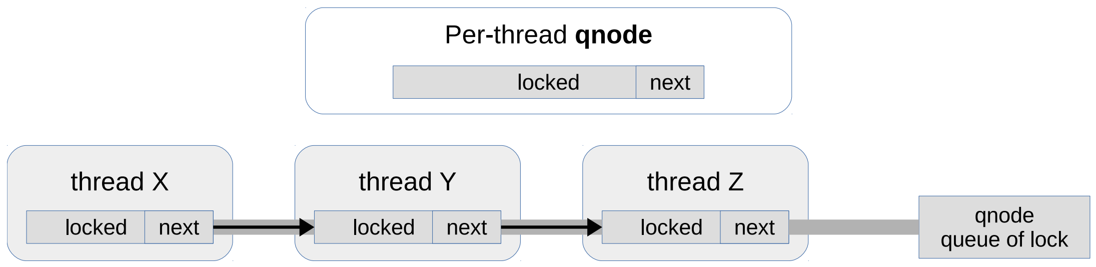


Figure 5.1: Illustration of the per-thread qnodes establishing a spin-wait queue of a lock.

waiter threads, in a back-to-back manner. In other words, the qnode of the newly coming lock-waiter thread becomes the tail qnode, while the qnode of the lock-owner thread is the head qnode. The lock-owner thread performs the unlock operation by releasing the `locked` value in the qnode of its successor lock-waiter thread. Thus, the queuing locks deliver perfect fairness for the spinlock "releasing" operation. Figure 5.1 depicts the per-thread qnodes which establish the spin-wait queue of a lock. The thread X is the lock-owner thread and thread Y, Z are the lock-waiter threads.

5.2.5 MCS Lock: Queuing Lock

```
typedef struct qnode {
  struct qnode *next;
  byte locked;
} qnode_t;

typedef qnode_t* mcslock_t;

mcslock_t lock = NULL;

qnode_t* Atomic_Xchg(mcslock_t* ptr, qnode_t* new_ptr) {
  qnode_t* rv = *ptr;
  *ptr = new_ptr;
  return rv;
}

qnode_t* Atomic_CmpAndSwap(mcslock_t* ptr, qnode_t* old_ptr, qnode_t* new_ptr) {
  qnode_t* rv = *ptr;
  if(*ptr == old_ptr)
    *ptr = new_ptr;
20
21   return rv;
22 }
23
24 void spin_lock(mcslock_t *lptr, qnode_t *mynode) {
25   qnode_t *predecessor;
26
27   mynode->next = NULL;
28   predecessor = Atomic_Xchg(lptr, mynode);
29
30   if (predecessor) {
31     mynode->locked = LOCKED;
32     predecessor->next = mynode;
33
34     /* spin-waiting loop */
35     do {} while (mynode->locked == LOCKED);
36   }
37 }
38
39 void spin_unlock(mcslock_t *lptr, qnode_t *mynode) {
40   if (!mynode->next) {
41     if (Atomic_CmpAndSwap(lptr, mynode, NULL) == mynode)
42       return;
43     do {} while (!mynode->next);
44   }
45   (mynode->next)->locked = UNLOCKED;
46 }
```

Listing 5.4: A pseudo-C code of the MCS spinlock.

Listing 5.4 depicts the pseudo-C code of the MCS lock. The MCS lock is the representative queuing spinlock design which establishes the linked-listed structure of the per-thread qnodes, as explained in Section 5.2.4. Actually, the lock variable is just the VA pointer variable pointing to the per-thread qnode (Line 6), and is initialized to the NULL pointer (Line 8). The `spin_lock()` and `spin_unlock()` functions keep the lock variable pointing to the current tail qnode, using the `Atomic_Xchg()` and `Atomic_CmpAndSwap()` atomic memory instructions (Line 28 and 41, respectively). Each per-thread qnode contains the local "`locked`" value (on which the thread spin-waits) and the `next` VA pointer variable pointing to the successor per-thread qnode (Line 1 to 4). The per-thread qnode is manually created by the programmer as the automatic variable inside the function frame stack of the lock-competing thread, and the VA pointer variable pointing to it is passed to the `spin_lock()` and `spin_unlock()` API functions as an argument (Line 24 and 39).

This way, the MCS lock separates the shared lock variable from the local per-thread `locked` variables on which each thread actually spin-waits. This separation shines best in the lock contention case because the `spin_unlock()` function even skips referencing the shared lock variable, and directly accesses the local per-thread qnode to let go of the successor lock-waiter thread (Line 45). On the contrary, with no outstanding lock-waiter thread, the MCS lock runs much slower than the ticket spinlock [24] because releasing the lock should use the atomic memory instruction on the shared lock variable (Line 40 to 44).

This sophisticated queuing spinlock design can help the MCS lock get over the cache coherence contention issues. However, the advantage can be extensively lost on the large scale CCSM multi-processors. The per-thread qnode cache lines should move in a ping-pong manner between the adjacent predecessor and successor lock-waiting threads, with long latencies (especially when crossing the processor sockets).

5.2.6 Important Properties of Spinlock Operation inside Linux Kernel

CPU-Affinity Spin-Wait

In the contemporary Linux kernel, it is guaranteed that the spinlock disables the scheduling preemption, during the spin-wait iterations. In other words, no CPU migration can happen while the kernel thread spinlock performs the spin-wait operation. This property enables the queuing spinlock code to spin-wait on the "per-CPU" qnodes, instead of on the "per-thread" qnodes. Adopting the per-CPU qnodes brings-in these advantages for the Linux kernel queuing spinlock design:

- The per-CPU qnodes can stay persistent in memory once created, while the per-thread qnodes should be manually created on demand, inside the thread function frame stack.

- The per-CPU qnodes can use the smaller CPU ID number (instead of the longer VA pointer value which the per-thread qnodes use), when constructing the linked-listed queue structure.

This property gives both of the Linux qspinlock and the proposed QT spinlock the means to shrink the lock variable into 4 bytes.

Deadlock-Free Stacked Spin-Wait

Actually, the Linux kernel spinlock designs are not as simple as the naive tight loops. At any given moment, one CPU can spin inside up to the 4-level stacked spin-wait loops of the different contexts: the normal thread and the software/hardware/non-maskable interrupt handlers [25]. This property (combined with the *"CPU-Affinity Spin-Wait* property") introduces the abstraction of the **per-CPU 4-level spin-wait stack**. The stack grows when an interrupt service both preempts the current spin-waiting context and ends up spin-waiting on its own lock. The stack shrinks after the interrupt service finishes spin-waiting. Because of this property, it is mostly impossible to employ any innovative synchronization accelerator which considers only 1-level locking operation (e.g. [29]), for the Linux kernel spinlock.

However, it is strictly inhibited that any multiple stacked contexts running on one CPU spin-wait on the same lock variable because this becomes the deadlock hazard between the higher and lower level contexts: for example, the lower-level context and higher-level context wait for each other when the lower-level context is owning the lock, while the higher-level one with priority waits for the lock to be released.

When the perfect fair lock functioning in the FIFO fashion contends, the abstraction of the **spin-wait queue** for the contended lock is established. As a result of the *"Deadlock-Free Stacked Spin-Wait"* property, each entry of the spin-wait queue of one lock spin-waits

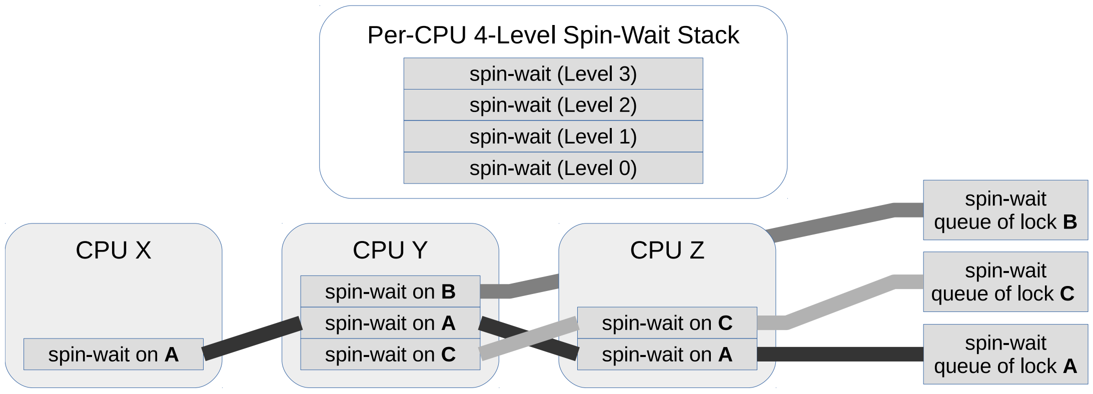


Figure 5.2: Illustration of the abstraction of the per-CPU 4-level spin-wait stack and per-lock spin-wait queue.

on each different CPU. Figure 5.2 depicts the abstraction of the per-CPU spin-wait stack and per-lock spin-wait queue, which are explained in this Section. The lock A, B, and C are now contended and form their spin-wait queues, respectively. In the lock A's spin-wait queue, CPU X is the least recently spin-waiting (LRSW) CPU of lock A, while CPU Z is the most recently spin-waiting (MRSW) CPU of lock A.

N Maximum Spin-Wait Queue Entries for N CPU System

The above *"Deadlock-Free Stacked Spin-Wait"* property makes it apparent that the maximum possible entry number of the spin-wait queue of one lock is equal to the total number of CPUs in the system. This property helps understand how small the lock variable could be. For instance, if the ticket spinlock is required to support 256 CPUs, each of the `headTicket` and `tailTicket` field of the lock variable can be only 8 (i.e. $log_2(256)$) bits.

5.2.7 Linux qspinlock: Centralized Spin-Lock Enhanced with Internal Queuing Lock

```
typedef struct qnode {
  struct qnode *next;
  byte locked;
} qnode_t;

/* persistent per-CPU qnodes (this code assumes 1-level stack for simplification) */
```

```
qnode_t perCPUQnodes[NUM_CPUS];

typedef struct qspinlock {
  cpuID_t tailCPUid;
  byte locked;
  ........
} qspinlock_t;

qspinlock_t lock;

void spin_lock(qspinlock_t *lptr) {
  qnode_t *predecessor, *successor;
  ........
  /* enqueue to the predecessor qnode */
  predecessor = &perCPUQnodes[lptr->tailCPUid];
  predecessor.next = &perCPUQnodes[myCPUid];
  ........
  /* spin-wait loop (two-step) */
  do {} while (perCPUQnodes[myCPUid].locked == LOCKED); // MCS spin-wait
  do {} while (lptr->locked == LOCKED);                  // CSL spin-wait
  ........
  /* lock the lock variable */
  lptr->locked = LOCKED;
  ........
  /* release the successor qnode */
  successor = perCPUQnodes[myCPUid].next;
  successor->locked = UNLOCKED;
  ........
}

void spin_unlock(qspinlock_t *lptr) {
  lptr->locked = UNLOCKED;
}
```

Listing 5.5: A simplified pseudo-C code of the Linux kernel qspinlock. This code assumes the 1-level per-CPU qnode stack for simplification.

Listing 5.5 depicts the simplified pseudo-C code of the Linux qspinlock. Because the entire code is highly complicated, Listing 5.5 only highlights the part activated in the lock contention case where the predecessor and successor lock-waiter threads exist.

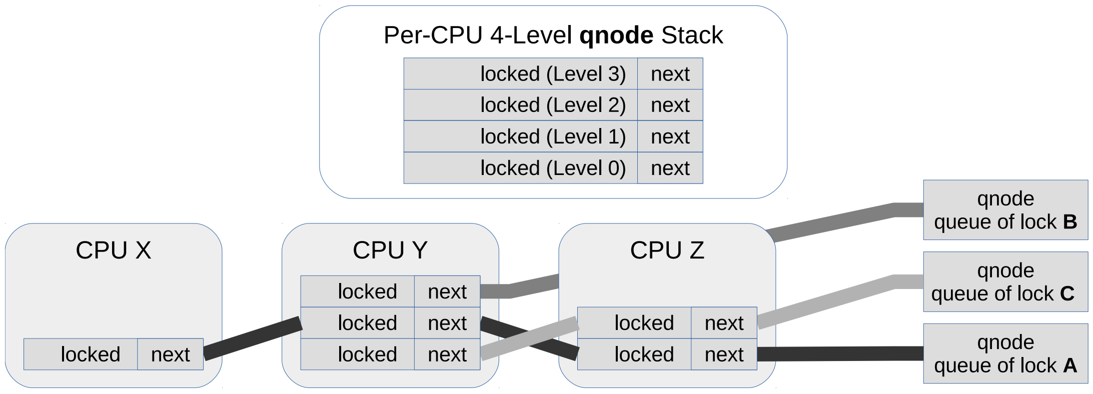


Figure 5.3: Illustration of the per-CPU 4-level qnode stack and per-lock qnode queue of the Linux qspinlock.

Contrary to its name, the qspinlock is not the pure queuing spinlock because it is invented as a result of hard effort to make the MCS lock fit into the 4 byte small lock variable. Instead, the qspinlock is designed in the form of the baseline centralized spinlock which is enhanced with the internal MCS lock.

Baseline Centralized Spinlock

To accomplish the small lock variable, the qspinlock is basically designed as a centralized spinlock (CSL): note that the qspinlock contains the `locked` element in the lock variable (Line 11) and accesses it for both of the spin-wait operation (Line 26) and the lock release operation (Line 38). In the `spin_lock()` function, the lock-owner thread locks the baseline CSLV before entering the critical section (Line 29). The baseline CSLV is released by the `spin_unlock()` function of the lock-owner thread (Line 38).

Internal MCS Lock

The qspinlock removes the cache coherence contention on its baseline centralized spinlock variable (CSLV), by internally employing the MCS lock. More specifically, all the lock-waiter threads are synchronized, so only one of them can be selected as the successor lock-owner which finally can access the baseline CSLV. The internal MCS lock also helps achieve the small lock variable size by using the per-CPU qnode technique explained in

Section 5.2.6: the lock variable has the `tailCPUid` field (instead of the long VA pointer variable pointing to the per-thread qnode), for the internal MCS lock operation (Line 10). Figure 5.3 illustrates the per-CPU 4-level qnode stacks and per-lock qnode queues of lock A, B, and C. They are implemented by following the abstraction of the per-CPU 4-level spin-wait stack and per-lock spin-wait queue (Section 5.2.6).

The internal MCS lock operates only inside the `spin_lock()` function. When the calling thread detects its predecessor lock-waiter thread, the thread enqueues its per-CPU qnode to the qnode of the predecessor lock-waiter thread (Line 22). Additionally, when the calling thread detects its successor lock-waiter thread, it dequeues (releases) the per-CPU qnode of the successor lock-waiter thread, before entering the critical section (Line 33).

Two-Step Spin-Wait

Unlike the per-thread qnodes, the per-CPU qnodes are limitedly viable only while CPU-migrations are disabled (e.g. during the spin-wait iterations). Consequently, the qspinlock design ends up having the two-step spin-wait operation: the first step spin-waits on the per-CPU qnode of the internal MCS lock (Line 25), while the second step spin-waits on the baseline CSLV (Line 26). The per-CPU qnodes are similar to the per-thread qnodes of the normal MCS lock (Line 1 to Line 4). However, the per-CPU qnodes are organized as the per-CPU 4-level stack, to meet the *"Deadlock-Free Stacked Spin-Wait"* Linux kernel property explained in Section 5.2.6, and stay persistent in memory (Line 7). Remember that the per-thread qnodes are manually created on demand as the local data, in the thread function frame stack. Even though not shown in Listing 5.5, the qspinlock code has several points which could get a bunch of atomic memory instructions executed on the fly, for the two-step spin-wait technique. Overall, the qspinlock is a twisted design, which means that the design is the analogy of the large complex tree (code) planted in the small container (lock variable).

Table 5.1 compares and summarizes all the spinlock designs explained in this Section.

Table 5.1: Summary of the representative spinlock designs

Fairness of Releasing Contended Lock	
TS	Random
TTS	Random
Ticket	Perfect FIFO
MCS	Perfect FIFO
Qspin	Perfect FIFO
Atomic Memory Operations	
TS	Atomic_TestAndSet(): on every spin-wait loop iteration
TTS	Atomic_TestAndSet(): on every outer spin-wait loop iteration
Ticket	Atomic_XchgAdd(): one time in the `spin_lock()` function
MCS	Atomic_Xchg(): one time in the `spin_lock()` function Atomic_CmpAndSwap(): one time in the `spin_unlock()` function (only with NO successor spin-waiter thread)
Qspin	Atomic_CmpAndSwap(): multiple times (including in loops) in the `spin_lock()` function
Variable(s) Spin-Waiting On	
TS	Lock variable (CSLV)
TTS	Lock variable (CSLV)
Ticket	Lock variable (CSLV)
MCS	Per-thread local qnode
Qspin	1st step: per-CPU local MCS qnode 2nd step: lock variable (CSLV)
Minimum Lock Variable Size for 256 CPUs (byte-addressing & 64 bit processors)	
TS	1 byte
TTS	1 byte
Ticket	2 bytes
MCS	8 bytes (sizeof generic VA pointer variable)
Qspin	4 bytes
Operations Requiring to Access Lock Variable	
TS	Locking / Unlocking / Spin-Waiting
TTS	Locking / Unlocking / Spin-Waiting
Ticket	Locking / Unlocking / Spin-Waiting
MCS	Locking / Unlocking (only with *no* outstanding successor spin-waiter thread)
Qspin	Locking / Unlocking / Spin-Waiting (2nd step)

5.3 Implementation of QT Spinlock

As defined in Chapter 1, the temporal FSMOs are the memory objects which can be temporally isolated from the SM domain, until the critical moment, so as to suppress the associated heavy SM transactions. The QT spinlock design exploits the advantage that the centralized spinlock variables (CSLVs) are the representative temporal FSMOs. That is to say, the design makes a CPU delay caching the lock variable, until the critical moment when the CPU eventually acquires the lock. Accordingly, the QT spinlock design simply incurs the **N** times lock variable cache line moves, in the lock contention case with the **N** lock-waiting threads, as mentioned in Section 5.1.1. In order to properly actualize the temporal FSMO attribute of the lock variable by exploiting this caching-off/on operation, the QT spinlock design employs these supports:

- Caching-flag
- QT instructions and QT messages
- Transformed ticket spinlock code

5.3.1 Caching-Flag (CF)

The caching flag (CF) governs the temporal FSMO attribute of the lock variable on each CPU node. When the CF is reset (offline), the CPU does not cache the lock variable. The CPU starts caching the lock variable when the CF is set (online) again.

As soon as detecting the lock acquisition failure, a CPU secures and initializes (resets) the CF. If the lock is contended, the associated CPUs construct the linked-listed system-wise CF queue for the lock. The CF eventually gets set again remotely in the message-passing manner by the lock-owner thread, when the thread releases the lock. Listing 5.6 describes the details on how the CFs are implemented on each CPU node, using the pseudo-Verilog code.

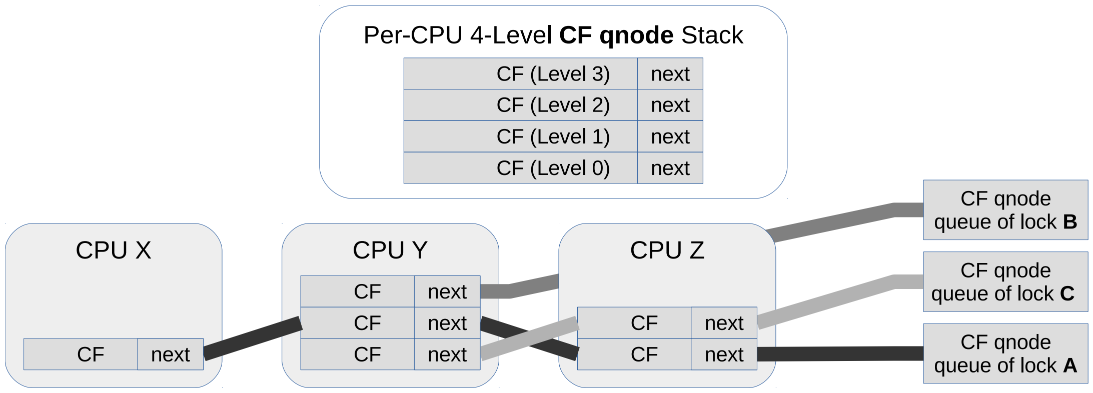


Figure 5.4: Illustration of the per-CPU 4-level CF qnode stack and per-lock CF qnode queue.

```
 1 /* location of CF: 8 bit */
 2 typedef struct {
 3   reg [5:0] CPUid;              // CPU of per-CPU CF stack
 4   reg [1:0] index;              // index (level) in per-CPU CF stack
 5 } cflocation_t;
 6
 7 /* entry of per-CPU CF stack: 9 bit */
 8 typedef struct {
 9   reg CF;                       // local caching-flag
10   cflocation_t next;            // location of successor CF
11 } cfentry_t;
12
13 /* per-CPU CF stack: 36 bit storage (i.e. 4 X 9 bit) */
14 typedef struct {
15   cfentry_t entry[3:0];         // 4-level
16   reg[1:0] tos;                 // top of stack
17 } cfstack_t;
```

Listing 5.6: The pseudo-Verilog code describing the per-CPU CF stack

Per-CPU (4-Level) CF Qnode Stack

The QT spinlock also follows the *“Deadlock-Free Stacked Spin-Wait”* property of the Linux kernel (Section 5.2.6). Accordingly, each CPU forms the per-CPU 4-level CF qnode stack (Line 14 to 17), which implements the abstraction of the per-CPU 4-level spin-wait stack (Section 5.2.6). Hence, the per-CPU CF qnode stack grows when a context spin-waits, and shrinks when the context finishes the spin-wait operation. The location of each CF qnode is defined by its 2-dimensional (2D) coordinates of (`CPUid`, stack `index`) (Line 2 to 5). The storage overhead for this per-CPU CF qnode stack is so small. For instance, one CPU requires 38 (i.e. 4 X 9 + 2) bits for its per-CPU CF qnode stack, if the system has up to the total 64 CPUs.

Per-Lock CF Qnode Queue

If the lock is contended, the associated CPUs construct the linked-listed system-wise CF qnode queue for the lock, which implements the abstraction of the spin-wait queue (Section 5.2.6). Thanks to the *"CPU-Affinity Spin-Wait"* property of the Linux kernel (Section 5.2.6), the adjacent CF qnodes can be linked-listed by using the CF qnode location. Recall that the software qnodes cannot help using the longer VA pointer values for this purpose.

Figure 5.4 illustrates the per-CPU 4-level CF qnode stacks and per-lock CF qnode queues of lock A, B, and C.

5.3.2 QT Instructions and QT Messages

QT Instructions

In order to correctly handle the lock variable as the temporal FSMO controlled by the CF value, the QT spinlock design introduces the four QT instructions which are the new X86-64 ISA memory instructions.

QT Messages

Two QT instructions have to update the content of the tail or head CF qnode of the lock. For this purpose, they introduce the QT messages which are the new light-weight CPU-to-CPU message passing micro-architecture similar to the cache coherence write-update or write-invalidation messages. Using the QT message is a reasonable technique because the actual data required to manage the per-lock CF qnode queue are small enough.

The associated QT instructions send the target CF qnodes the QT messages as part of their memory write operation, just as sending the cache line sharer CPUs the write-invalidate messages. The QT message gets the location of the target CF qnode from its QT instruction which directly reads the location from the lock variable in memory. To this end, the transformed ticket spinlock code for the QT spinlock has its lock variable contain the location of the head and tail CF qnodes of the lock. Thus, using the QT message results in a great advantage over the scalable software queuing spinlocks which rely on the long-latency mechanisms of bouncing the entire cache lines. It is known that such expensive and redundant cache line ping-pongs can bottleneck the performance of even the sophisticated scalable software queuing locks, on the large scale CCSM NUMA multi-processors [6].

Section 5.3.3 explains the QT instructions and QT messages in detail, along with the QT spinlock code.

5.3.3 Transformed Ticket Spinlock Code

It requires insight to take the traditional ticket spinlock code to the next-level version which fits into the QT spinlock idea. Thus, we first come up with the "generic ticket spinlock" concept, and then create the transformed code for the QT spinlock design, as one implementation of the generic ticket spinlock.

Generic Ticket Spinlock. For each lock-waiter thread to be assigned a unique `myticket` value, the `headTicket` and `tailTicket` field pair of the lock variable must be properly updated. Therefore, the actual implementations of the system can be diverse and are distinguished by two factors: (1) which data forms the `headTicket` and `tailTicket` field pair and (2) which action updates the field pair, on the locking/unlocking operations. From this perspective, the traditional ticket spinlock employs the simple two integer values for the first factor, and the increment (by one) action for the second factor.

QT Spinlock Design. For the first factor, the design makes the lock variable `headTicket` and `tailTicket` fields comprise the location of the head and tail CF qnodes of the lock, respectively: each of the fields also contains the trivial "toggle" metadata. For the second factor, the design updates the `headTicket` and `tailTicket` field pair, utilizing the queuing operations carried-out by the QT instructions and QT messages.

Listing 5.7 is the pseudo-C code of the transformed ticket spinlock code which the QT spinlock employs. It illustrates the lock variable, `spin_lock()` and `spin_unlock()` functions, and use of the QT instructions. Basically, the code structure is almost the same as that of the traditional ticket spinlock (Section 5.2.3). The main transformed parts are placed along with where the four QT instructions are used (they have the "`QT_`" prefix in the names).

```
typedef struct cflocation{
  byte index:2 /*2bit*/, cpuID:6 /*6bit*/;
} cflocatation_t;

typedef struct ticket{
  byte toggle;
  cflocation_t CFLocation;
} ticket_t;

typedef struct lock{
  ticket_t headTicket, tailTicket;
} lock_t;

void spin_lock(lock_t *lptr) {
  lock_t lock_status = QT_Atomic_ToggleEnqueue(lptr);
16
17    /* lock acquisition decision */
18    if(lock_status.headTicket == lock_status.tailTicket)
19      return;
20
21    /* spin-wait loop */
22    ticket_t myticket = lock_status.tailTicket;
23    do {} while(QT_Load(&lptr->headTicket) != myticket);
24
25    QT_Dequeue(&lptr->headTicket.CFLocation);
26  }
27
28  void spin_unlock(lock_t *lptr) {
29    QT_WakeUp(&lptr->headTicket);
30  }
```

Listing 5.7: The pseudo-C description of the transformed ticket spinlock code for the QT spinlock. The code employs the QT instructions which have the "`QT_`" prefix in their names. This example is for systems having up to 64 CPUs.

Lock Variable Fields

The `headTicket` and `tailTicket` of the lock variable consist of these sub-fields:

`CFLocation` (Line 7). This is the newly adopted facilitator field of the QT spinlock. It holds the location of the head and tail CF qnodes of the lock (Line 2). The adjacent successor and predecessor lock-competing threads refer to this location (instead of performing heavy operations such as looking-up the directory memory), to directly send the QT messages to their target CF qnodes.

`toggle` (Line 6). It is nothing but the minimum legacy of the ticket value of the traditional ticket spinlock. The `spin_lock()` and `spin_unlock()` functions respectively toggle this field of the `tailTicket` and `headTicket`. Actually, the QT spinlock is designed to be able to compress this field into as small as 1 bit, so the code can support as many CPUs as possible. However, Listing 5.7 specifies the field to be 1 byte (consequently supporting 64 CPUs), for the program efficiency in the byte-addressing machine: if this value becomes smaller than 1 byte, Line 29 should become the atomic read-modify-write (RMW) memory operation.

QT_Atomic_ToggleEnqueue instruction (`spin_lock()`*, Line 15)*

It is the augmented substitute of the `Atomic_XchgAdd()` instruction which the traditional ticket spinlock code uses. The instruction basically (1) toggles the `tailTicket.toggle` field, and (2) returns the properly processed lock status value which gives the thread the lock acquisition result (Line 18) and unique `myticket` value (Line 22).

The instruction is augmented to determine if the thread acquires the lock or not, using the simple comparison logic with the current lock variable value. The further augmented changes are activated when the thread fails to acquire the lock, in this sequence:

- The instruction allocates and initializes (resets) the new CF qnode, and pushes the CF qnode into the local per-CPU CF qnode stack to make it the local top of stack (ToS) CF qnode. As a result, the local per-CPU CF qnode stack grows.
- The instruction checks the `tailTicket.CFLocation` of the lock variable, to get the location of the tail CF qnode of the lock. When it detects the tail CF qnode with the check, it enqueues the local ToS CF qnode to the tail CF qnode, by using the QT message: the instruction sets the `next` filed of the tail CF qnode to the location of the local ToS CF qnode.
- The instruction updates the `tailTicket.CFLocation` field of the lock variable to the location of the local ToS CF qnode. To suppress the cache line bounce, it accomplishes this memory write operation in the **write-update** manner, but does not yet get the lock variable cached on the local CPU node at this moment. This way, the instruction handles the lock variable as the temporal FSMO, enabling the lock variable cache line to exist only on the lock-owner thread CPU node at almost all

times.

- The instruction has the updated `tailTicket.CFLocation` field contained in its return value, so its calling spin-waiting thread is assigned that field as its unique `myticket` value.

QT_Load instruction (`spin_lock()`*, Line 23)*

It switches between the “offline” and “online” memory load operation modes, depending on the local ToS CF value. When the CF is reset (offline), it just returns the **hard-wired** value which keeps the spin-wait loop iterating. When the CF is set (online), it acts as the normal memory load instruction. The CF is reset by the `QT_Atomic_ToggleEnqueue` instruction, as explained previously, and is set again remotely by the `QT_WakeUp` instruction of the lock-owner thread. This way, it handles the lock variable as the temporal FSMO, enabling the lock variable cache line to exist only on the lock-owner thread CPU node at almost all times.

QT_Dequeue instruction (`spin_lock()`*, Line 25)*

It first pops the ToS CF qnode from the local per-CPU CF qnode stack, to retrieve the `next` field pointing to the head CF qnode of the lock. Then, it updates the lock variable `headTicket.CFLocation` field to the retrieved `next` value. Note that this projection of the local ToS CF qnode content onto the lock variable is indispensable at this point (i.e. right after the spin-wait loop). This is because the local per-CPU CF qnode stack relies on the *“CPU-Affinity Spin-Wait”* property of the Linux kernel, as explained in Section 5.3.1. That is, after the thread exits the spin-wait loop, the thread can experience the kernel preemption and, hence, can run on a different CPU node afterwards.

As a result of this instruction, the local per-CPU CF qnode stack shrinks. That is, its ToS CF qnode is finally deallocated from the local per-CPU CF qnode stack and dequeued from the CF qnode queue for the lock.

QT_WakeUp instruction (`spin_unlock()`*, Line 29)*

The instruction is basically a memory RMW instruction which toggles the `headTicket.toggle` field. Upon writing on the cache line, it silently reads the `headTicket.CFLocation` from the lock variable, to get the location of the head CF qnode of the lock. If it notifies that the head CF qnode exists, it remotely sets the head CF by sending the QT message, so the `QT_Load` instruction of the CPU of the head CF qnode should start handling the lock variable as the shared memory object.

Atomic Memory Instruction in `spin_unlock()` *Function*

Just like the traditional ticket spinlock, the QT spinlock does not require the atomic memory operation to release the lock. However, should the code support more than 64 CPUs, Line 29 has to become the atomic RMW memory operation, as mentioned previously.

Figure 5.5 illustrates the QT instructions which update the CF qnodes and lock variable `CFlocation` fields, in the lock contention case.

5.3.4 Intuitive View on Procedures in Lock Contention

Overall, the operations of the QT spinlock in the lock-contention case are portrayed as the round-trip of the location of the newly created local ToS CF qnode as the `CFLocation` value. The `spin_lock()` and `spin_unlock()` functions make the value traverse from the lock variable to the CF qnode queue, and back to the lock variable. The `QT_Atomic_ToggleEnqueue` instruction has its CPU become the departure CPU of the round-trip. The instruction makes its new local ToS CF qnode the new tail CF qnode of the lock, by storing its location in (1) the `next` field of the tail CF qnode, (2) the lock variable (`tailTicket.CFLocation` field), and (3) the local register (`myticket`). Then, when the tail CF qnode later moves to the head of the CF qnode queue for the

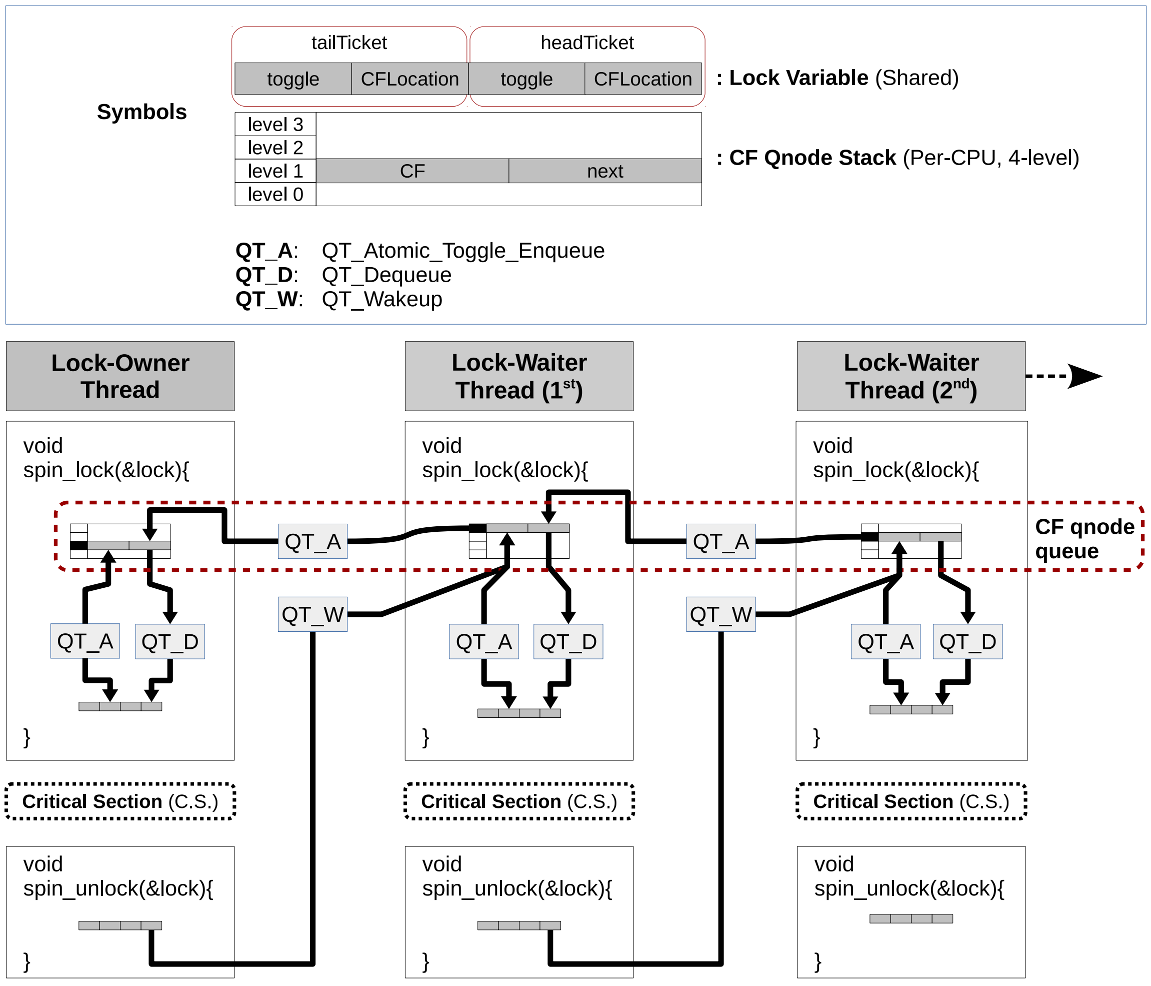


Figure 5.5: Sketch of the QT instructions which update the CF qnodes and lock variable `CFLocation` fields, in the lock contention case.

lock, the `QT_Dequeue` instruction projects the `next` field of the head CF qnode back on the lock variable (`headTicket.CFLocation` field), so the lock-owner thread can reference it to locate the departure CPU. When releasing the lock, the lock-owner thread has the `QT_WakeUp` instruction inform the departure CPU to read the updated lock variable. Then, the `CFLocation` value finally reunites with the `myticket` value on the departure CPU, and lets go of the CPU.

5.3.5 Impact on Underlying Memory Consistency Model

The QT instructions and QT messages are implemented as the memory instructions and the associated write-invalidate message equivalents, so the QT spinlock mechanism can still force them to abide by the employed shared memory consistency model. The simulation environment of this thesis work confirms this fact to some extent, because it runs with its own shared memory model which is based on the basic total store ordering (TSO) shared memory consistency model. The TSO model allows the store-load reordering, and the QT spinlock mainly modifies the memory load instruction inside the spin-wait loop (i.e. the `QT_Load` instruction). Thus, the simulator would immediately crash if any violation of the (allowed) memory reordering happens.

Currently, the QT spinlock code adopts the ticket spinlock code of Linux kernel 3.4.4 as the baseline of its transformed code. The micro-architectural supports (e.g. the QT instructions and QT messages) are modeled in two separate components of Marssx86 full-system simulator [45]: Qemu emulator [46] and PTLsim micro-architecture simulator [47].

5.4 Essentials Required for Proper Simulators

In truth, it requires utmost circumspection to select the proper multi-processor simulator, to evaluate the spin-waiting synchronization designs. It stems from the fact that no one simulator fits all workloads. Above all, most of the simulators are optimized for the macro-scopic cache coherence simulation because, otherwise, the simulation time will sky-rocket. In contrast with this, the spin-waiting synchronization could be the most massive source of the micro-scopic fine-grain cache coherence operations accompanying the typhoon-like cache line bounces. Therefore, we are supposed to empower the simulator to capture the micro-scopic fine-grain cache coherence events, just like the high-speed cameras can take sharp consecutive snapshots of the fast-moving objects by taking the extremely high frame rates. For this purpose, we focus on the next two essentials, while selecting and setting up the simulator for this research:

- the perfect-synchronization-based processor multiplexing, and
- the proper coherence message implementation.

5.4.1 Essential 1: The Perfect-Synchronization-Based Processor Multiplexing

We explain importance of the "perfect-synchronization-based" multiplexing by inversely making clear the danger of its prevalent relaxed alternative of the "time-quantum-based" idea. Basically, the time-quantum-based multiplexing is the legacy handed down by the high performance multi-processor **emulators**. That is, the longer time-quantum results in the faster (but the less concurrent) emulation.

Time-Quantum-Based Multiplexing **Emulators**

Here we quote the details on the time-quantum based emulator, directly obtained from the user guide of Simics [48] which is the representative high performance emulator:
"Ideally, Simics would make time progress and execute one cycle at a time, scheduling processors according to their frequency. However, perfect synchronization is exceedingly slow, so Simics serializes execution to improve performance. Simics does this by dividing time into segments and serializing the execution of separate processors within a segment. The length of these segments is referred to as the quantum and is specified in seconds (this

is similar to the way operating systems implement multitasking on a single-processor machine: each process is given access to the processor and runs for a certain time quantum). The processors are scheduled in a round-robin fashion, and when a particular processor P has finished its quantum, all other processors will finish their quanta before execution returns to P."

Time-Quantum-Based Multiplexing **Simulators**

Actually, quite a few multiplexing simulators are designed based on the existing high performance emulators. That is, such simulators extend the emulator (functional-frontend) with the accurate add-on latency model (timing-backend). Hence, it is natural that those multiplexing simulators inherit the time-quantum legacy, to speed-up the simulation. This speed-up is critically required because the execution performance of the real (or virtual) machines, emulators, and simulators degrade significantly in the listed order by multiple order-of-magnitude.

Notice that the time-quantum based simulators can still simulate the concurrent IO events (e.g. cache coherence operations), but at most, in the same limited macro-scopic manner that the **single-processor** can do. This can be easily understood by the above quote which mentions the analogy that "the time-quantum scheduling is similar to the way operating systems implement multitasking on a single-processor machine." A single-processor machine running the multitasking mechanism can get the multiple IO-bound (not CPU-bound) events initiated in the limited parallel fashion by switching one task to another as soon as encountering IO service failure. Therefore, the time-quantum based simulation absolutely achieves its speed-up in exchange of the opportunities to accurately simulate the extremely concurrent IO events.

Unfortunately, this simulation limitation for the concurrent IO events still sustains, no matter how short time-quantum is adopted. As a matter of fact, the ultimately short (e.g. a single-cycle long) time-quantum can help get the multiple concurrent events to happen in a more overlapped way. However, this counter-measure brings in another problem when we simulate a large scale multi-processor machine: the shorter the time-quantum becomes, the more likely is it to get the latency of each individual IO event considerably hidden by the long round-robin CPU multiplexing interval.

From this understanding, we can think of the time-quantum-based multiplexing scheme as the intrinsic simulator design defect when evaluating the micro-scopic fine-grain concurrent cache coherence events. In other words, the scheme can inhibit the timing-backend subsystems of the simulator from proving their full real worth, no matter how sophisticated and accurately the subsystems are designed.

5.4.2 Essential 2: The Proper Coherence Message Implementation

Almost all contemporary computer architecture simulators utilize the trick of embedding NO actual data on the cache lines, to substantially reduce the memory resource required to run. Instead, each memory operation directly (and instantly) accesses the shared global memory space, to actually read/write the data. Due to this trick, the multi-processor simulator could unconsciously deliver the *"too good to be true"* emulator-level short-latency and high-throughput simulation result, unless it carefully effectuates the cache coherence messages. Among all the coherence messages, the `getX` message (which each memory write operation generates) should be handled with the most special care, for the better correct simulation.

Built-In Synchronization Effect of `getX` *Message*

In the write-invalidation CCSM systems, writing on the cache line is rendered in the three actions of (1) securing the entire target cache line, (2) updating part of the secured cache line, and (3) sending out the invalidation message to the other sharer CPUs, if any. These three actions should be performed in the atomic manner (whether or not the line is true or false shared one), because the mechanism works with the cache line granularity, and the updated cache line becomes the only fresh source of the data. Therefore, the CCSM multi-processor systems grant the cache line write operation the built-in synchronization effect, which is verified by the code in Listing 5.8. The code uses the OpenMP directive to generate the false sharing cache line update inflation. The false shared array `False_Shared_Array[]` is aligned to the cache line (64 bytes in the case), to maximize the false sharing effect. Each spawned parallel thread is observed to successfully increment its own private array element (i.e. `False_Shared_Array[tid]`) by one, until reaching the number of `N`. However, we do not see the X86-64 `lock` prefix in the disassembled executable file, which means that the code works without using any explicit software synchronization.

```
int tid, i;

/* array on one cache line (aligned to 64 bytes cache line) */
int False_Shared_Array[NTHREADS] __attribute__((aligned(64))) = {0};

#pragma omp parallel private(i,tid)
{
  tid = omp_get_thread_num();

  for (i = 0; i < N; i++) {
    False_Shared_Array[tid] += 1;
  }
} /* end of parallel section */
```

Listing 5.8: The OpenMP code generating the false sharing cache line update inflation on the array `False_Shared_Array[]`.

The built-in synchronization effect (i.e. atomic memory operation) of the cache line write comes from its `getX` message which requests for the exclusive access to the cache line, against other memory read/write operations. Thus, the cache line write operations contribute to constructing the **critical path** of the cache line read/write bounces on the CCSM multi-processors: their `getX` messages constitute the dominant source of the stalled memory operations. For this reason, the mechanism how `getX` message works should be the most correctly modeled, to properly simulate the spin-waiting synchronization.

5.5 Simulator Setup

Actually, it is almost unnecessary to simulate the performance of the QT spinlock, because it arouses the already "determined" ultimately minimal **N** times cache line moves, in the lock contention case with the **N** lock-waiter threads, as explained in Section 5.1.1. Thus, we conversely focus on making the simulation demonstrate how badly the representative comparison control provokes the heavy cache coherence memory transactions. The Linux qspinlock is taken as the comparison control, because it is the work-around default spinlock choice of the contemporary Linux kernel.

5.5.1 Baseline Simulator (Marssx86)

We scout out Marssx86 [45] as the baseline simulator, among the popular contemporary multi-processor simulators, considering these required features carefully:

- ***Full-system multi-processor.*** The simulator should boot from and run the real Linux OS kernel because the default Linux kernel spinlocks have more sophisticated and complicated issues to design, validate, and simulate, than the user-level spinlocks.
- ***Open source X86-64 platform.*** The simulator should model the popular X86-64 machines and also should allow us to add the QT instructions, as the new ISA instructions.
- ***Perfect-synchronization-based processor multiplexing.*** The simulator should avoid using the time-quantum-based multiplexing scheme, to be able to capture the microscopic concurrent cache coherence events (Section 5.4.1).

Marssx86 is a multiplexing multi-processor simulator. However, it does not use the time-quantum-based multiplexing mechanism, to simulate the CPUs. Even though the simulator is designed by combining the Qemu emulator [46] and the PTLsim micro-architecture model [47], the relationship is not the functional front-end and timing back-end. Instead, the Qemu emulator is just used to fast-forward the target benchmark application until it reaches the region of interest (ROI). The ROI detection is accomplished with the magic instructions which can be embedded inside the benchmark codes. The ROI-detecting magic instruction immediately switches Marssx86 from running in the fast-forwarding emulation mode into running in the fine-grain micro-architecture simulation mode. The simulation mode executes its main loop which realizes the **perfect-synchronization-based processor multiplexing**, by having all the simulated virtual CPU (vCPU)s run on each single simulation cycle. Listing 5.9 describes the main simulation loop of the Marssx86 simulator.

```
/* single sim_cycle iteration */
for (;;) {
  ....
    /* all vCPUs run in the round-robin manner */
    foreach (i, coremodel.per_cycle_signals.size()) {
      ....
    }
    sim_cycle++;
    iterations++;
  ....
}
```

Listing 5.9: The main simulation loop of Marssx86 simulator. The loop has all the vCPUs run on every single simulation cycle. It is located in the `BaseMachine::run()` function of the `ptlsim/sim/machine.cpp` file.

However, the stock Marssx86 simulator has weak and improper simulation model for the shared memory subsystem. The main critical problems of the simulator and our corresponding counter-measures are:

- ***Imperfect cache coherence state updates.*** Even though Marssx86 simulator implements the cache coherence protocols which use the "MOESI" or "MESI" states, it is observed that the states embedded in each cache line do not get updated appropriately. To work around this, we introduce the **ideal directory memory** structure.

- ***No proper cache coherence message.*** Constituting an important part of the SM consistency model, the cache coherence mechanism itself requires synchronization operation. To achieve this, the contemporary CCSM multi-processors use the cache coherence messages. However, the Marssx86 simulator implements deficient cache coherence messages. Especially, the simulator does not perform any mechanism equivalent to the `getX` message which is the essential for the proper shared memory subsystem simulation of the CCSM multi-processors, as explained in Section 5.4.2. To work around this, we introduce the **ideal interconnection network** layer.

Without the proper counter-measures, these two problems easily mess-up the simulation. For instance, the stock Marssx86 simulator reports even the totally incorrect cache-to-cache transfer numbers. To achieve the proper approximate comparison evaluation of the spinlock design performance, we introduce the new simulation model which employs said ideal directory memory structure and ideal interconnection network layer, in Section 5.5.2.

5.5.2 Approximate Simulation Model: Critical-Path of Cache Line Bouncing

The primary goal of the counter-measures explained in this Section is get the simulator to report the correct cache-to-cache transfer numbers and approximate throughput performance reflecting the critical-path of cache line bouncing, when simulating the spin-waiting workloads.

Ideal Directory Memory

We implement the "ideal directory memory", by using the map container of C++ standard template library (STL). It contains the cache line granularity information which the "ideal interconnection network layer" requires, including the presence bit vectors and the status of on-going shared memory load/store operations. However, it does not explicitly control the memory operation ordering for the SM consistency model, which the regular directory memories are supposed to do as the centralized ordering point [49, 50]. This directory memory model is ideal, because it (1) frees each cache line from having and updating the local coherence state, and (2) it always experiences the directory look-up hits.

Ideal Interconnection Network

We implement the "ideal interconnection network" layer to have these features for the shared memory consistency model.

Cache coherence. The layer synchronizes the shared memory read/write operations on each cache line the same way the readers/writer lock (also known as shared-exclusive lock or multiple readers/single-writer lock) synchronization [51] works. That is, during a store buffer performs the `getX` operation to update data on one cache line, all the other CPUs are blocked to perform the load/store operations on the cache line and should wait until the `getX` operation completes. The layer mostly adopts the write-preferring priority, in order to remove the starvation of the memory write operations.

Memory ordering. The layer allows the maximum concurrency of processing the shared memory operations, by exploiting as much **cache-to-cache transfer level parallelism** as possible: the ideal interconnection network instantly (upon arrival) initiates all the requesting shared memory operations in the FIFO manner, and lets them run with the **uniform** cache-to-cache transfer latency. Remark that the instruction level parallelism (ILP) is also calculated with the uniform latency for the all instructions [52]. The uniform cache-to-cache transfer latency enables the ideal interconnection network layer to free the ideal directory memory from playing the centralized ordering point [49, 50] role. The ideal interconnection network layer also naturally reflects the effect that the baseline Marssx86 simulator has the per-CPU store-buffer which allows the re-ordering of the store-load operation.

This interconnection network layer is considered ideal because it allows the maximum concurrency with the shared memory operations, while it at least keeps the synchronization of the `getX` message associated events, which is definitely enforced in real systems (Section 5.4.2). Thus, the layer is optimized to evaluate the critical-path of cache line bounce dependencies.

Memory Consistency Model

The CCSM multi-processor simulators (and even the emulators) undeniably should have the proper SM consistency model to work accordingly. Otherwise, they can not run the unmodified softwares which are written and built following the specific SM consistency model. Moreover, the more sophisticatedly the simulators improve their event-driven attributes, the more likely is it that any violation of the SM consistency model can easily crash the simulator. As a recap, the approximate simulation model works with these features for the SM operations:

- It is the in-order machine model.
- Its ideal interconnection network layer by default effectuates the maximum concurrency of processing the shared memory operations in the FIFO order.
- Its ideal interconnection network layer synchronizes the memory read/write operations on each cache line, in the readers/writer lock synchronization manner.
- It has the per-CPU store-buffer which allows the re-ordering of the store-load operation.

Thus, the approximate simulation model works with its memory consistency model which is based on the simple TSO consistency model.

Table 5.2: Simulation configuration

Platform	
ISA	X86-64
Cache Memories and Main Memory	
L1-I/D	Private, 3-cycle latency
L2	Private, Unified (I/D), LLC, 15-cycle latency
RAM	136-cycle latency
CPU Structure	
Scheduling	In-order execution
Fetch width	2
Issues per cycle	2
Store buffer size	16
Commit buffer size	32
Shared Memory Subsystem Model	
Ideal Directory Memory	Cache coherence operations
Ideal Interconnection Network	Cache coherence and memory ordering operations
Cache-to-Cache Transfer	72-cycle latency (**uniform**)
Evaluated Target Software/Hardware	
Operating System	Linux (kernel ver. 3.3.4)
Comparison Designs	Linux qspinlock v.s. QT spinlock

5.6 Evaluation

5.6.1 Simulation Configuration

The performance of the QT spinlock and Linux qspinlock designs are compared. We choose the qspinlock (among all the other high performance scalable spinlocks) as the comparison control because it is the most recently adopted default Linux kernel spinlock which satisfies the 4 byte lock variable size rule. Thus, the qspinlock design serves as the reference point which demonstrates how badly the qualified software spinlocks drive the redundant shared memory transactions. The simulated target is the Marssx86 X86-64 model in a multi-core setting. This machine models an Intel Atom processor with the in-order pipeline scheduling. We simulate the systems with *4, 8, 12, 16, 20, 24, 28, 32 processing cores* running the Ubuntu Linux OS (with Linux kernel 3.4.4). Note that all the cache-to-cache transfers run with the **uniform 72-cycle** latency, as explained in Section 5.5.2. Table 5.2 summarizes the simulation configurations.

5.6.2 Simulation Workload: Steady State Extreme Lock Contention

The benchmark workload used in the evaluation is the simple but representative spinlock micro-benchmark application (similar to the ones used in [44, 24, 6]) which creates the worst case steady-state lock contention. The application creates the **N** kernel threads and has them concurrently compete to perform the read-modify-write (RMW) operations on the one cache line sized critical section, in a synchronized manner: each kernel thread simultaneously loops, to acquire the lock, update the critical section, and release the lock. We measure the throughput of the **1024 + 2N** times lock acquisitions and releases as the performance measure metric. The additional **2N** times is added to compensate for the fact that the simulator does *n*ot respond immediately when the magic instruction informs of the start and termination of the ROI. The "steady state" means the state where all the **N** lock-competing threads are either holding the lock or spin-waiting for the lock. This state is best for evaluating/comparing the critical-path of cache line bouncing of the qspinlock and QT spinlock designs. In this state, (1) the qspinlock induces its smallest cache-to-cache transfer traffic, and (2) both of the qspinlock and QT spinlock avoid the case where the lock-owner thread acquires the lock again as soon as the thread releases the lock. We write the benchmark application in the form of the loadable kernel module (LKM), so that the application can be attached to the Linux kernel on the fly. Additionally, to minimize any irrelevant kernel services which possibly interfere the benchmark application with preemptions, we get the simulated system to boot into the Linux kernel "Recovery Mode" which is a minimal kernel option. The Qemu emulator of the Marssx86 simulator fast-forwards the simulated system until it reaches the steady-state ROI where the required **N** number of the kernel threads are eventually created and ready to participate in the simulation. Then, the magic instruction of the Marssx86 which we embed in the benchmark application switches the simulator from the emulation mode to the fine-grain simulation mode.

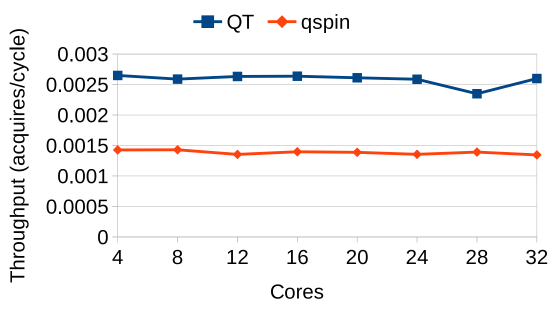
5.6.3 Unfair Simulation Model

As mentioned in Section 5.5.2, the simulation model has these ideal properties which by default allows the maximum concurrency of shared memory operations:

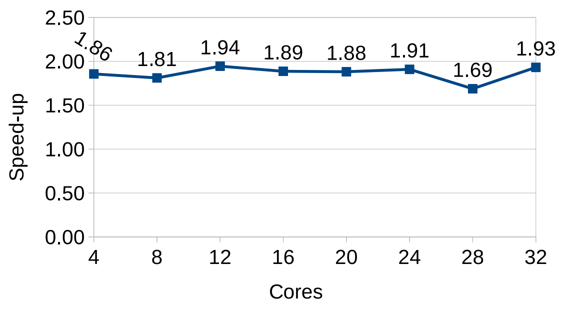
- Its ideal interconnection network instantly (upon arrival) initiates all the requesting shared memory operations in FIFO manner.
- Its ideal interconnection network lets such shared memory operations run with **uniform** cache-to-cache transfer latency, for all memory operations (i.e. memory load/-store and atomic-RMW instructions).

These properties are lopsidedly favorable to the Linux qspinlock design rather than to the QT spinlock design: they significantly hide the shared memory operation latencies of the former, while giving no benefit to the latter. Despite this unfair comparison that tilts strongly toward the Linux qspinlock design, it is demonstrated (in the following sub-sections) that the QT spinlock design comfortably outperforms the Linux qspinlock design.

5.6.4 Throughput Performance



(a) Throughput (acquires / cycle)

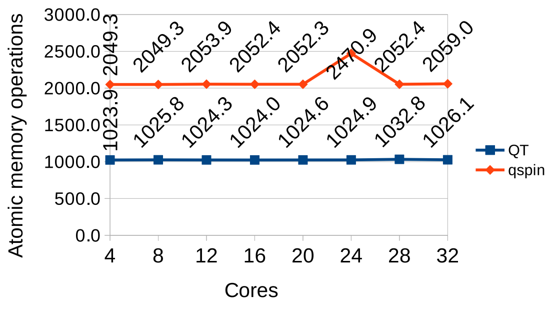


(b) **Speed-up** of throughput performance (QT spinlock over qspinlock)

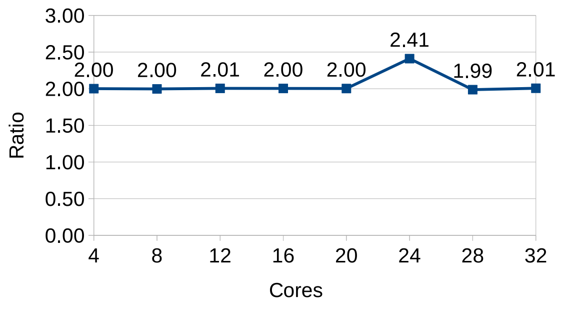
Figure 5.6: Lock acquisitions/releases throughput performance of the Linux qspinlock and QT spinlock designs.

Figure 5.6 displays the throughput performance of the Linux qspinlock and the QT spinlock designs obtained with increasing kernel thread (CPU) counts. Figure 5.6a shows that on the idealized approximate simulation model, both the Linux qspinlock and QT spinlock scale well, as the CPU count increases. However, it also corroborates the assertions that the QT spinlock design out-performs the Linux qspinlock even with the unfair simulation model (Section 5.6.3). Figure 5.6b gives the speed-up numbers of the throughput performance (i.e. the throughput of the QT spinlock over the throughput of the qspinlock), for varying number of CPU cores. The (simple arithmetic) average overall performance improvement is found to be 1.86 times speed-up. Of course, the performance boost is attributed to the ultimate minimalism in cache line bouncing of the QT spinlock design as explained in Section 5.1. The important relevant numbers are further provided in Section 5.6.5 and 5.6.6.

5.6.5 Atomic Memory Operations



(a) Number of run-time atomic memory operations

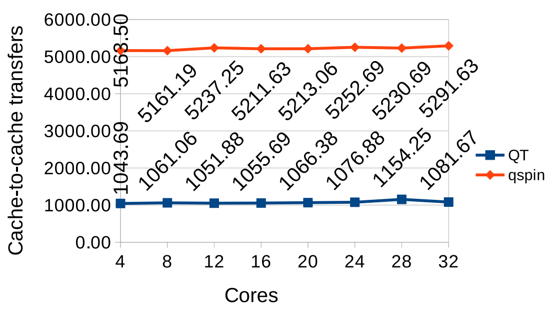


(b) **Ratio** of run-time atomic memory operations (qspinlock to QT spinlock)

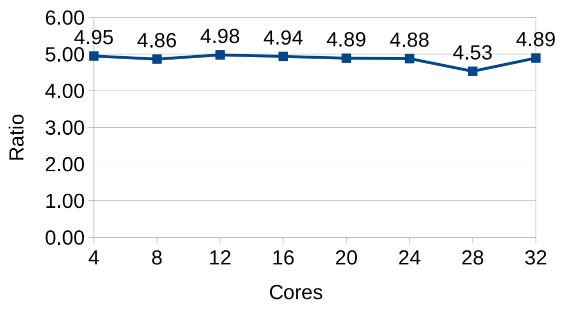
Figure 5.7: Run-time atomic memory operations used by the Linux qspinlock and QT spinlock.

Figure 5.7 displays the simulation results on the run-time atomic memory operations which the qspinlock and QT spinlock incur. Figure 5.7a shows the numbers of each design, while Figure 5.7b shows the ratio of the numbers (i.e. the number of the qspinlock to the number of the QT spinlock). Because the QT spinlock code is the transformed ticket spinlock code, the design is supposed to incur simply **1024** ± ***trivial error*** times atomic memory operations (refer to Table 5.1), which Figure 5.7a verifies. Meanwhile, the qspinlock incurs **twice** as many run-time atomic memory operations, with this "steady-state extreme" lock contention workload. It is very hard to estimate how many run-time atomic memory operations the qspinlock design incurs, using a static code analysis. However, the qspinlock clearly can incur more run-time atomic memory operations while the concurrently competing threads are in the middle of creating the spin-wait queue of the lock, than while the threads are eventually running in this "steady-state extreme" lock contention. This is due to the fact that the qspinlock uses "compare-and-swap (CAS)" based atomic memory operations to update the lock variable. The CAS operations are subject to being performed multiple times (usually being located inside a loop), when the lock variable is updated frequently by the concurrently competing multiple threads.

5.6.6 Cache-to-Cache Transfers



(a) Number of cache-to-cache transfers



(b) **Ratio** of cache-to-cache transfers (qspinlock to QT spinlock)

Figure 5.8: Run-time cache-to-cache transfers incurred by the Linux qspinlock and QT spinlock.

Figure 5.8 displays the simulation results on the run-time cache-to-cache transfers which the qspinlock and QT spinlock incur. Figure 5.8a shows the numbers for each design, while Figure 5.8b shows the ratio of the numbers (i.e. the number of the Linux qspinlock to the number of the QT spinlock).

The cache-to-cache transfer number in Figure 5.8a verifies the ultimate minimalism in cache line bouncing of the QT spinlock design (Section 5.1). One can see that the QT spinlock incurs **1024** ± ***minor error*** cache-to-cache transfers, for the **1024 + 2N** times lock acquisition and release operations. Additionally, the numbers in Figure 5.8a confirms that the Marssx86 simulator is modified and enhanced correctly with the counter-measures explained in Section 5.5.2.

The ratio of the cache-to-cache transfer numbers presented in Figure 5.8b demonstrates that the Linux qspinlock incurs an (simple arithmetic) average of 4.87 times more cache line bounces than the QT spinlock does. This ratio value exhibits the "two-step spin-waiting" technique of the qspinlock (Section 5.2.7): in the steady-state extreme lock contention, the qspinlock requires at least 4 cache line bounces among the lock-competing threads, per lock acquire/release operation. Note that the cache line bounces of queuing locks induced while running this workload constitutes the critical-path shared memory dependencies which become the main source of performance degradation.

As Section 5.6.3 emphasized, the adopted approximate simulation model is lopsidedly favorable to the Linux qspinlock design rather than to the QT spinlock design: it hides significant amount of cache-to-cache transfer latencies of the qspinlock. Therefore, the performance improvement (speed-up) evaluated in this Section must be considered the minimum improvement. That is, the real CCSM multi-processor systems would deliver much higher performance improvement with the QT spinlock design (over the Linux qspinlock), leveraging the ultimate minimalism in cache line bouncing contribution of the QT spinlock.

5.7 Summary

This study proposes the QT spinlock design, to help resolve the issue that the spinlock operations themselves induce much heavier SM overheads than the short critical sections which the very spinlocks are intended to synchronize. Under the lock contention, the QT spinlock design is ultimately free from cache coherence interference on the lock variable cache line.

Consequently, the lock-owner thread has the effect that no other thread competes for the lock variable cache line concurrently.

To accomplish this, the design identifies the ticket spinlock lock variable as a temporal FSMO inside the Linux kernel, and handles it accordingly by employing the transformed ticket spinlock code and the associated light-weight hardware support. Importantly, the design functions without breaking the underlying SM consistency model. Furthermore, because it is basically a centralized spinlock design, the QT spinlock can run with a sufficiently small (4 bytes) lock variable, which is the critical feature required for the Linux kernel default spinlock.

CHAPTER 6
DISSERTATION CONCLUSION

This thesis is dedicated to resolving the clear discrepancy between the design principle of the parallel SM programs and the actual SM domain operation of the CCSM multi-processors. The parallel SM programs are optimized to maximally perform the computation locally, while minimally generating the global SM transactions. Meanwhile, the contemporary CCSM multi-processors too conservatively treat every memory object as a permanently shared one which incurs SM overhead. This "one size fits all" style policy of the systems on handling the memory references has been hiding many opportunities to improve performance and energy/power efficiency.

In order to achieve this goal, the thesis develops the concept of spatial and temporal FSMOs, identifies their high-locality performance-affecting instances, and optimizes the identified references. Carrying out the goal requires the comprehensive understanding of the computer systems, including the micro-architecture, programming models, OSes, and compilers. Nonetheless, it can free the CCSM multi-processors from unconditionally paying unnecessary overhead cost when referencing the target FSMOs.

It is nearly impossible to propose a general way to identify and optimize references to such FSMOs because of the complexity of actual software and hardware computer systems. Therefore, this thesis focuses on optimizing the references to two well-defined FSMOs: the PLMD and CSLVs. However, this thesis work can be augmented and enhanced to discover other performance-affecting FSMO references so as to enable future high-performance and energy-efficient large scale CCSM multi-processors and their OSes.

REFERENCES

[1] H. H. S. Lee, M. Smelyanskiy, C. J. Newburn, and G. S. Tyson, “Stack value file: Custom microarchitecture for the stack,” in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 5–14.

[2] S. Cho, P.-C. Yew, and G. Lee, “Access region locality for high-bandwidth processor memory system design,” in *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 32, IEEE Computer Society, 1999, pp. 136–146.

[3] V. Petric, A. Bracy, and A. Roth, “Three extensions to register integration,” in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 35, IEEE Computer Society Press, 2002, pp. 37–47.

[4] C. S. Ballapuram, A. Sharif, and H.-H. S. Lee, “Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors,” in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII, ACM, 2008, pp. 60–69.

[5] S. C. Kang, C. Nicopoulos, H. Lee, and J. Kim, “A high-performance and energy-efficient virtually tagged stack cache architecture for multi-core environments,” in *2011 IEEE International Conference on High Performance Computing and Communications*, 2011, pp. 58–67.

[6] T. David, R. Guerraoui, and V. Trigonakis, “Everything you always wanted to know about synchronization but were afraid to ask,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13, ACM, 2013, pp. 33–48.

[7] S. C. Kang, C. Nicopoulos, A. Gavrilovska, and J. Kim, “Subtleties of run-time virtual address stacks,” *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 152–155, 2015.

[8] *Harvard Mark I*, `https://en.wikipedia.org/wiki/Harvard_Mark_I`, 2017 (accessed May 3, 2017).

[9] K. D. Cooper and T. J. Harvey, “Compiler-controlled memory,” in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII, ACM, 1998, pp. 2–11.

[10] S. Cho, P.-C. Yew, and G. Lee, “Decoupling local variable accesses in a wide-issue superscalar processor,” in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA ’99, IEEE Computer Society, 1999, pp. 100–110.

[11] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas, “L1 data cache decomposition for energy efficiency,” in *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ser. ISLPED ’01, Huntington Beach, California, USA: ACM, 2001, pp. 10–15, ISBN: 1-58113-371-5.

[12] M. Mamidipaka and N. Dutt, “On-chip stack based memory organization for low power embedded architectures,” in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, ser. DATE ’03, Washington, DC, USA: IEEE Computer Society, 2003, pp. 11 082–, ISBN: 0-7695-1870-2.

[13] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’14, USENIX Association, 2014, pp. 147–163.

[14] *Code-Pointer Integrity*, `http://dslab.epfl.ch/proj/cpi/`, 2014 (accessed May 4, 2017).

[15] M. Cekleov and M. Dubois, “Virtual-address caches. part 1: Problems and solutions in uniprocessors,” *IEEE Micro*, vol. 17, no. 5, pp. 64–71, 1997.

[16] ——, “Virtual-address caches.2. multiprocessor issues,” *IEEE Micro*, vol. 17, no. 6, pp. 69–74, 1997.

[17] *COMPAQ Alpha 21264 Microprocessor Hardware Reference Manual*, `http://www.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=21264hrm.pdf`, 1999 (accessed May 6, 2017).

[18] S. G. Tucker, “The ibm 3090 system: An overview,” *IBM Syst. J.*, vol. 25, no. 1, pp. 4–19, Jan. 1986.

[19] D. H. Woo, M. Ghosh, E. Özer, S. Biles, and H.-H. S. Lee, “Reducing energy of virtual cache synonym lookup using bloom filters,” in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES ’06, ACM, 2006, pp. 179–189.

[20] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[21] A. Basu, M. D. Hill, and M. M. Swift, “Reducing memory reference energy with opportunistic virtual caching,” in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA ’12, IEEE Computer Society, 2012, pp. 297–308.

[22] *x86-64*, `https://en.wikipedia.org/wiki/X86-64`, 2017 (accessed May 7, 2017).

[23] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[24] S. Boyd-wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, *Non-scalable locks are dangerous*.

[25] *MCS locks and qspinlocks*, `https://lwn.net/Articles/590243/`, 2014 (accessed March 3, 2017).

[26] T. Craig, “Building fifo and priority-queuing spin locks from atomic swap,” Tech. Rep., 1993.

[27] V. Luchangco, D. Nussbaum, and N. Shavit, “A hierarchical clh queue lock,” in *Proceedings of the 12th International Conference on Parallel Processing*, ser. Euro-Par’06, Springer-Verlag, 2006, pp. 801–810.

[28] M. A. Auslander, D. J. Edelsohn, O. Y. Krieger, B. S. Rosenburg, and R. W. Wisniewski, “Enhancement to the MCS lock for increased functionality and improved programmability,” pat. U.S. patent application 10/128,745, 2003.

[29] C.-K. Liang and M. Prvulovic, “Misar: Minimalistic synchronization accelerator with resource overflow management,” in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15, ACM, 2015, pp. 414–426.

[30] R. E. Bryant and D. R. O’Hallaron, *x86-64 Machine-Level Programming*, `https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf`.

[31] M. Parasar, A. Bhattacharjee, and T. Krishna, “Seesaw: Using superpages to improve vipt caches,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA ’18, Los Angeles, California: IEEE Press, 2018, pp. 193–206, ISBN: 978-1-5386-5984-7.

[32] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, `http://www.intel.com/content/dam/www/public/us/en/documents/`

manuals/64-ia-32-architectures-optimization-manual.pdf, 2016 (accessed March 3, 2017).

[33] POSIX standards IEEE.

[34] *OpenMP*, http://www.openmp.org/.

[35] *Wind River Systems*, http://www.windriver.com.

[36] *Intel Sandy Bridge Architecture*, http://www.realworldtech.com/page.cfm?ArticleID=RWT091810191937&p=7.

[37] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08, ACM, 2008, pp. 72–81, ISBN: 978-1-60558-282-5.

[38] Z. Fang, L. Zhao, X. Jiang, S.-l. Lu, R. Iyer, T. Li, and S. E. Lee, "Reducing l1 caches power by exploiting software semantics," in ISLPED 2012.

[39] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 2nd. Addison-Wesley, 2010, ISBN: 0136108040, 9780136108047.

[40] A. S. Tanenbaum, *Modern Operating Systems*, 3rd. Prentice Hall Press, 2007, ISBN: 9780136006633.

[41] M. Chabbi, A. Amer, S. Wen, and X. Liu, "An efficient abortable-locking protocol for multi-level numa systems," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17, ACM, 2017, pp. 61–74.

[42] A. Jaleel, M. Mattina, and B. Jacob, "Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads," in *The International Symposium on High-Performance Computer Architecture (HPCA), 2006.*, 2006, pp. 88–98.

[43] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, USENIX Association, 2010, pp. 1–16.

[44] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, Jan. 1990.

[45] A. Patel, F. Afram, S. Chen, and K. Ghose, “Marss: A full system simulator for multicore x86 cpus,” in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 1050–1055.

[46] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’05, Anaheim, CA: USENIX Association, 2005, pp. 41–41.

[47] M. T. Yourst, “Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator,” in *2007 IEEE International Symposium on Performance Analysis of Systems Software*, 2007, pp. 23–34.

[48] *Simics User Guide for Unix (Simics Version 3.0)*. Virtutech AB., 2007, p. 193.

[49] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh, *SCORPIO: A 36-Core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC with In-Network Ordering*, `https://scorpio.mit.edu/sites/default/files/images/ISCA_SCORPIO_Presentation.pdf`.

[50] ——, “Scorpio: A 36-core research chip demonstrating snoopy coherence on a scalable mesh noc with in-network ordering,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA ’14, 2014, pp. 25–36.

[51] P. J. Courtois, F. Heymans, and D. L. Parnas, “Concurrent control with ‘readers’ and ‘writers’,” *Commun. ACM*, vol. 14, no. 10, pp. 667–668, Oct. 1971.

[52] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, ISBN: 1-55880-069-8.