國 立 交 通 大 學

資訊科學與工程研究所

碩士論文

為非耦合軟體管線所設計的鎖無關且尊重快取機制之軟體佇列

Lock-free cache-friendly software queue for decoupled software pipelining

研究生：陳韋任

指導教授：楊武 博士

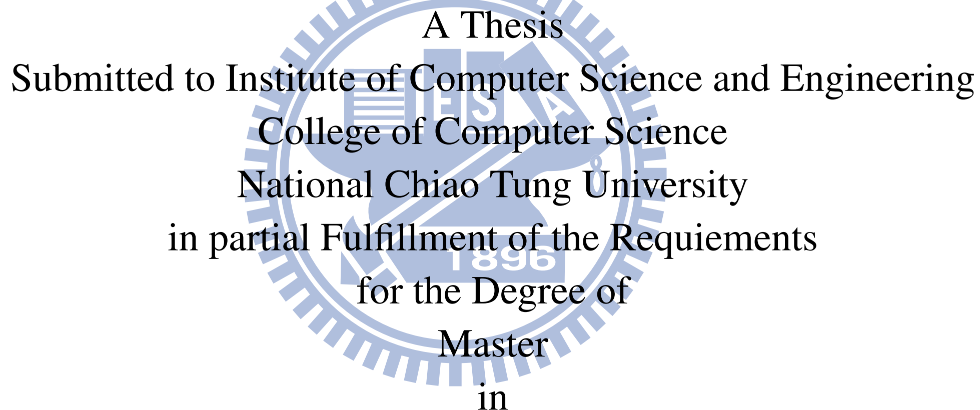
中華民國 九十九 年 九 月

爲非耦合軟體管線所設計的鎖無關且尊重快取機制之軟體佇列
Lock-free cache-friendly software queue for decoupled software pipelining

研究生：陳韋任 Student : Chen Wen-Ren

指導教授：楊武 博士 Advisor : Wuu Yang, Ph.D.

國 立 交 通 大 學
資訊科學與工程研究所
碩士論文



A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requiements
for the Degree of
Master
in

Computer Science and Engineering
September, 2010
HsinChu, Taiwan, Republic of China

中華民國 九十九 年 九 月

爲非耦合軟體管線所設計的鎖無關且尊重快取機制之軟體佇列

研究生：陳韋任　　　　　　　　指導教授：楊武 博士

國立交通大學資訊科學與工程研究所

摘要

近幾年來，不論是在伺服器或個人電腦領域，多核心平台皆已成爲主流。平行化是能充份利用多核心平台所提供額外計算能力的其中一種方法。 然而，一般應用程式具有複雜的資料與控制相依性，此種特性使得傳統平行化技術，如: DOALL 和 DOACROSS，無法應用於其上。 *Decoupled Software Pipelining* (DSWP) 是一新的平行化技術，其擁有平行化一般應用程式的潛能。然而，DSWP 的成功有賴於處理器之間高速的傳輸與同步。 目前 DSWP 在商用多核心平台上的效能表現並不理想。其主因在於處理器之間傳輸與同步基本上有賴於鎖相關、不尊重快取機制的軟體方式達成。 此種作法將會大幅抵消 DSWP 所可能帶來的好處。

我們爲 DSWP 提出一個鎖無關、尊重快取機制的軟體佇列。一個鎖無關且尊重快取機制的實作需要考慮到記憶體子系統的兩個不同面向，記憶體一致性 (memory coherence) 和記憶體一貫性 (memory consistency)。我們舉例說明忽略或混淆前述兩個不同面向將如何導致不正確或無效率的實作。之後，我們提出一個正確且有效率的實作， 並同時給出了詳細的解釋。由於平行程式本質上的不確定性，傳統的測試技術無法用來證明其正確性。我們同時以非正式和正式的方法討論我們實作上的正確性。

Lock-free cache-friendly software queue for decoupled software pipelining

Student : Chen Wen-Ren **Advisor : Wuu Yang, Ph.D.**

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Multicore has become a trend on server and client computers in recent years. Parallelization is one way to fully utilize the computing power provided by multicore architectures. Most applications of interest have complex data and control dependency, which make traditional parallelization techniques, such as DOALL and DOACROSS, inapplicable. *Decoupled Software Pipelining* (DSWP), a new parallelization technique, shows its potential on parallelizing general applications. However, its success relies on fast inter-core synchronization and communication. On commodity multicore platforms, the performance of current DSWP disappoints us since the overhead involving lock-based, cache dishonored software approach offsets the benefit from DSWP.

We present a lock-free, cache-friendly software queue designed for DSWP. A lock-free, cache-friendly solution need take two different aspects of memory system, memory coherence and memory consistency, into consideration. We show how inattention to these two aspects leads to incorrect or inefficient solutions. We also present our approach to providing a correct and efficient solution with detailed explanation. Due to the nondeterministic nature of parallel programs, traditional testing techniques cannot be used to fully verify the correctness of the implementation. We also discuss the correctness of our implementation both in informal and formal ways.

致謝

首先，我要感謝我的指導教授，楊武博士。沒有他的意見和建議，這篇碩士論文無法以它現在的面貌呈現出來。我感謝他在我研究上一路的指引。此外，他極有耐心修改我的論文，使得這篇論文能論述的有條理，易讀易懂。對此，我獻上我最誠摯的謝意。

我要感謝資涵。他在我研究的路上給予了許多建議和幫助。是他指引了我研究最初的方向。我還要感謝徐慰中教授和黃廷祿教授，他們分別在我研究所和大學時代給予我許多指導。

我還要感謝程式語言與系統實驗室的全體成員。在我低潮的時候包容和鼓勵我。願大家畢業後人生路上一帆風順。願將來的學弟妹能順利畢業。願大家以後能經常相聚。

我最深的感謝要獻給我的家人。沒有他們無間斷的支持，我無法克服一切走到這一步。我還要感謝徐如樺小姐在我研究上的一路陪伴。最後，我要對我的貓，小子，獻上最深最深的感謝和思念。牠已離我而去，僅以此篇論文在此紀念牠。

The work reported in this paper is partially supported by National Science Council, Taiwan, Republic of China, under grants NSC 96-2628-E-009-014-MY3, NSC 98-2220-E-009-050, NSC 98-2220-E-009-051, and 99-2219-E009-013 and a grant from Sun Microsystems OpenSparc Project.

Table of Contents

摘要 i

Abstract ii

致謝 iii

Table of Contents iv

List of Figures v

List of Tables vi

1 Introduction 1

2 Introduction to DSWP 4

3 Memory coherence and consistency 9
- 3.1 Memory coherence . . . 9
- 3.2 Memory consistency . . . 10

4 The Lock-Free and Cache-Friendly Properties 13
- 4.1 The Lock-Free Properties . . . 13
- 4.2 The Cache-Friendly Property . . . 17

5 Implementation and Verification 20
- 5.1 Implementation . . . 20
- 5.2 Verification . . . 24

6 Experiment 26
- 6.1 Platform and Benchmark . . . 26
- 6.2 Result and Discussion . . . 28

7 Related work 31

8 Conclusion and future work 33

Bibliography **35**

List of Figures

2.1 Linked-list traversal loop 5
2.2 DSWP transformation example 5
2.3 DOACROSS and DSWP schedules 6
2.4 Synchronization array hardware model 7
2.5 Transit and COMM-OP delays 8

4.1 The part of Dekker's algorithm achieving mutual exclusion 15
4.2 Valid executions of Fig. 4.1 on a sequentially consistent multiprocessor system 15

5.1 Class QueueBuffer data members 21
5.2 Class QueueBuffer member functions 22

6.1 Before applying DSWP with OpenMP and QueueBuffer 28
6.2 After applying DSWP with OpenMP and QueueBuffer 30

List of Tables

6.1 Experiment Platforms 27

6.2 Benchmark Programs 27

6.3 Experiment Result 29

Chapter 1

Introduction

Since 2003, uniprocessor performance has no more significant improvement [1]. Historically, microprocessors have been designed to improve the performance by exploiting instruction-level parallelism (ILP). Techniques used to take advantage of ILP, such as deep pipelines, multiple instruction issue, speculation, and out-of-order execution, etc., have led to complex processor design with high power consumption. Recent years, major microprocessor manufacturers, form Intel and AMD to Sparc and PowerPC, all turn to multicore architectures.

While multicore architectures can improve the performance of multi-threaded applications, they are not as useful for single-threaded applications. One of the most significant current research topics is to parallelize sequential applications into multi-threaded ones that can take advantage of the multicore architectures. Conventional parallelization techniques, such as DOALL and DOACROSS, concentrate on parallelizing counted loops that manipulate very regular, analyzable structures. These techniques for parallelizing such patterns are used routinely in scientific and numerical domains with good results [2]. Unfortunately, the patterns do not appear in general applications. General applications usually have complex control flow, recursive data structures, or irregular pointer-based memory accesses which renders the above parallelization techniques inapplicable.

A new parallelization technique, called *Decoupled Software Pipelining* (DSWP), shows its potential on parallelizing general applications [3]. DSWP parallelizes a loop by partitioning the loop body into pipeline stages. A process consist of producer threads (PROD) and consumer threads (CONS). Theoretically, the advantage of DSWP compared with DOACROSS is that the inter-core communication latency was hidden in DSWP by overlapping computation and communication. In practice, the overhead in synchronization and communication between threads is so large that it negates the above advantage. Thus, the success of DSWP greatly relies on fast inter-core synchronization and communication. Rangan, R., et al. proposed a hardware, called *synchronization array* (SA), that makes inter-core communication as cheap as a load or store on L2 cache [3].

Due to the lack of hardware support for inter-core communication, DSWP on commodity multicore platforms did not achieve significant performance improvements.

On multicore platforms, communication between PROD and CONS is via a software queue. Traditionally, since the software queue is shared among threads, programmers must use lock-based synchronization primitives, such as a mutex, to prevent data race. Such a coarse-grained locking approach can significantly offset the parallelism brought up by DSWP.

In this paper, we present a *lock-free* approach so that PROD and CONS can access the software queue concurrently *without* any lock. However, providing a lock-free approach that is both correct and efficient is not as easy as it might like. Two different aspects of the memory system must be addressed while writing correct and efficient lock-free code. The first aspect, called *memory coherence*, defines *what* values can be return by a read operation. The second aspect, called *memory consistency*, determines *when* a written value will be returned by a read. Memory consistency is often confused with memory coherence. Coherence and consistency are complementary concepts. Coherence defines the behavior of reads and writes to the *same* memory locations, while consistency defines the behavior of reads and writes to *different* memory locations.

Memory consistency is a dark corner hidden from programmers writing parallel programs by using lock-based synchronization primitives. For those who want to bypass those lock-based synchronization primitives, full understanding of memory consistency is the key to writing correct lock-free code. It is easy to write lock-free code that only *appears* to work, but it is very difficult to write correct one.

When performance matters, programmers need take memory coherence into consideration. Unnecessary memory traffic introduced by maintaining memory coherence degrades performance. Cache-friendly approaches should be taken in order to avoid as much unnecessary memory traffic as possible.

In this paper, we first give an introduction to memory consistency and memory coherence. We use Dekker's algorithm, a classical textbook mutual exclusion algorithm, as an example to demonstrate how memory consistency issue could break Dekker's algorithm. Tools and approaches are introduced for writing lock-free, cache-friendly code. Finally, we present our implementation, a C++ template class `QueueBuffer` and verify its correctness both in informal and formal ways.

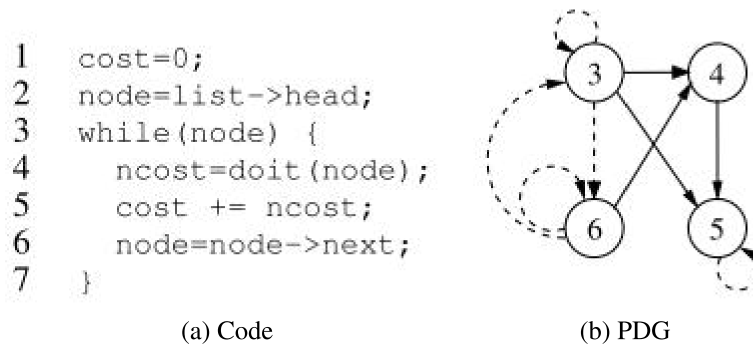
The rest of this paper is organized as follows: Chapter II gives an introduction to DSWP. Chapter III gives an introduction to memory coherence and consistency. Chapter IV describes the ways to achieve the lock-free and cache-friendly properties. Chapter V presents the implementation of our C++ template class `QueueBuffer` and verifies its correctness. Chapter VI presents experiments. Chapter VII discusses related work. Chapter VIII is the conclusion.

Chapter 2

Introduction to DSWP

Today, there are three non-speculative loop parallelization techniques: DOALL [2], DOACROSS [2], and DSWP [3]. Among these techniques, only DOALL yields speedup proportional to the number of cores. Nevertheless, programmers often find that DOALL may not be applicable in general-purpose code. For instance, consider the code in Fig. 2.1a, which is linked-list traversal loop. Figure 2.1b is the corresponding program dependence graph (PDG) which contains control and data dependences. Those dependences involving recurrences are denoted as dashed lines in Fig. 2.1b. Dependence recurrence is also a cross-iteration dependence, but not vice versa. For example, edge $6 \rightarrow 4$ is a cross-iteration dependence but does not involve a dependence recurrence. Because statement 3, 5, and 6 are each part of a dependence recurrence, loop iterations cannot be independently executed in separate threads. This makes DOALL inapplicable.

On the other hand, both DOACROSS and DSWP can parallelize the above linked-list traversal loop. The difference between DOACROSS and DSWP lies in the way the parallelism is achieved. DOACROSS distributes all loop iterations on different cores. DSWP, however, partitions the loop body, and each core is responsible for a particular piece of the loop body across all iterations.



(a) Code (b) PDG

Figure 2.1: Linked-list traversal loop

The DSWP algorithm partitions the loop body into pipeline stages. The process consists of producer threads (PROD) and consumer threads (CONS). The algorithm has three main steps. First, the algorithm constructs the program dependence graph (PDG) for the loop to be parallelized. The PDG contains all control and data dependencies in the loop. Second, all dependence recurrences (i.e., those dependences form loops) in the PDG are recognized by constructing its strongly connected components (SCCs). A SCC will be the basic scheduling unit. Finally, the SCCs are distributed on different cores while ensuring there is no cyclic dependence between cores. Figure 2.2 shows the PDG in Fig. 2.1b after applying the DSWP algorithm. There are three SCCs node 4, node 5, and the combination of nodes 3 and 6.

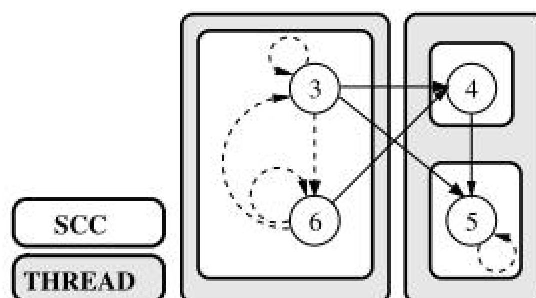


Figure 2.2: DSWP transformation example

Figure 2.3a and Figure 2.3b give the parallel execution schedules for DOACROSS and DSWP respectively. Comparing the schedules of DOACROSS and DSWP, we will find that

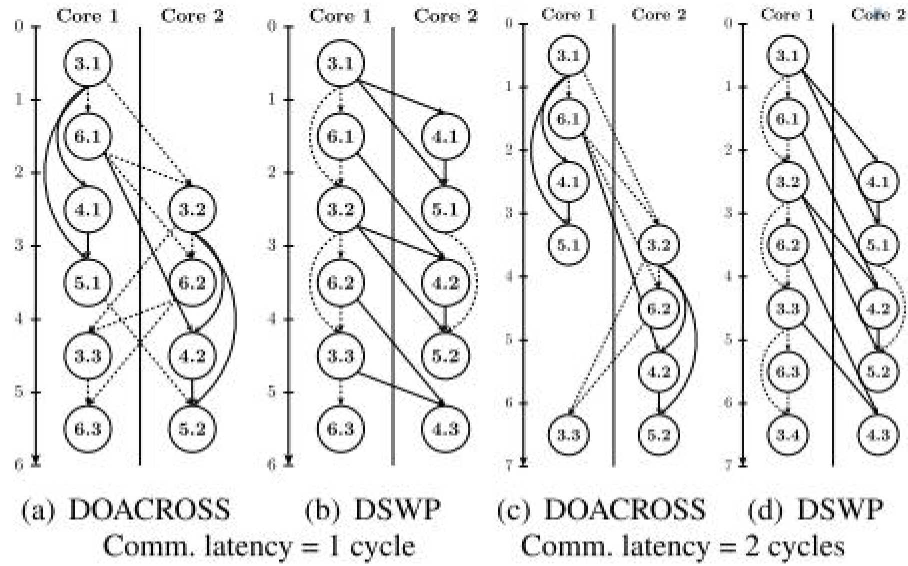


Figure 2.3: DOACROSS and DSWP schedules

DSWP provides significantly more latency tolerance than DOACROSS. In Figure 2.3, the nodes are numbered with both static instruction numbers and iteration numbers. For instance, the node 3.1 denotes instruction 3 in the first iteration. The communication latency is the time for transmitting a data value from one core to another. When the communication latency is one cycle, both DOACROSS and DSWP complete one iteration every two cycles. As the communication latency becomes longer, as shown in Fig. 2.3c and Fig. 2.3d, DSWP still completes one iteration every two cycles. DOACROSS, however, need three cycles to complete one iteration.

The success of DSWP relies on a hardware, called *synchronization array* (SA), shown in Fig. 2.4. [3], which make inter-core communication as cheap as a load or store to L2 cache. The synchronization array works as a set of low-latency queues associated with dependence numbers. The instruction-set-architecture (ISA) extension of SA is a set of blocking queues accessed via the `produce` and `consume` instructions. PROD and CONS communicate with one another through SA by using the `produce` and `consume` instructions. The `produce` instruction takes a dependence number and a register as operands. The value in the register

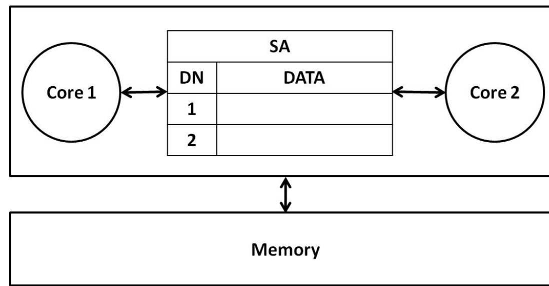


Figure 2.4: Synchronization array hardware model

is enqueued in the queue identified by the dependence number. The `consume` instruction dequeues data in a similar fashion [3, 4].

The inter-core communication overheads can be divided into *transit delay* and *communication operation delay* (COMM-OP) [5]. The transit delay refers to the time taken to transfer a data value from one core to another. The COMM-OP delay refers to the time taken to enqueue (dequeue) data into (from) the queue.

Figure 2.5 illustrates the effect of COMM-OP and transit delays on the DSWP performance. In order to send a data value from thread A to thread B, thread A first must ensure the queue is not full, then fills the queue with the data value. Those operations occur during the time line segment labeled the COMM-OP delay for thread A. After the transit delay has elapsed, thread B must check the queue is not empty before consuming the data value from the queue. During the transit delay, thread A can continue its own work. In other words, the transit delay is tolerated by overlapping computation and communication.

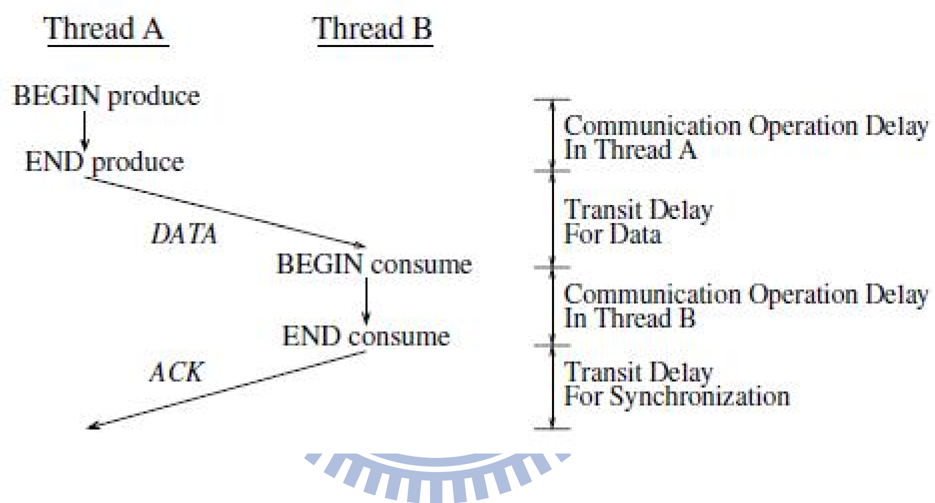


Figure 2.5: Transit and COMM-OP delays

Chapter 3

Memory coherence and consistency

Programmers need to distinguish two aspects of the memory system behavior while writing parallel programs.

The first aspect, called *coherence*, defines *what* value can be returned by a read operation. The second aspect, called *consistency*, determines *when* a written value will be returned by a read operation. Coherence and consistency are complementary concepts. Coherence defines the behavior of reads and writes to the *same* memory locations, while consistency defines the behavior of reads and writes to *different* memory locations [6].

3.1 Memory coherence

Since memory accesses are among the slowest of a CPU's operations, a multi-level memory hierarchy is introduced into the computer system. Each level in the memory hierarchy is smaller and faster than the next lower level. Cache is below CPU registers in the memory hierarchy. When data is not found in the cache, it must be fetched from memory and placed in the cache before continuing. A processor brings into the cache a *cache line* during every memory operation. A cache line is usually 64 bytes on modern multiprocessor systems. On a multiprocessor system, every processor has its own local cache. There are usually multiple copies of the same

data in different local caches. A *cache coherence protocol* is used to maintain the coherence of multiple caches. Cache coherence protocols are classified according to the techniques used to tracking the states of the shared data blocks. There are two broad categories: *directory-based protocols* and *snooping protocols*.

Cache coherence protocols, however, do not answer the question of *when* a processor sees the value that has been updated by another processor. Since processors communicate through shared variables, the above question becomes the ordering that must be enforced among reads and writes to different locations by different processors.

3.2 Memory consistency

The most commonly used memory consistency model is *sequential consistency*, formally defined by Lamport as follows [7]:

> [A multiprocessor is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Lamport also proposed two sufficient conditions for sequential consistency are [7]:

1. Each processor issues memory requests in the order specified by its program.

2. Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

The first condition is called the *program order*, which means that the processor may not change the order of memory operations specified by the source program. The *program order* maintains all four possible orderings: R $\rightarrow$ R, R $\rightarrow$ W, W $\rightarrow$ W, and W $\rightarrow$ R. The form X $\rightarrow$ Y

means that memory operation X must complete before memory operation Y is done. R and W denote read and write operations, respectively. The second condition is called *atomicity*, which means that memory operations issued by all processors to the same memory location should be served in the FIFO order. This implies that earlier memory operations being served will not be interrupted by later ones.

Although sequential consistency is simple and intuitive, it constrains many hardware and compiler optimizations. Therefore, several relaxed memory consistency models have been proposed as alternatives to sequential consistency. The relaxed memory consistency models are classified according to the read-and-write orderings that are relaxed. For example, relaxing the W $\rightarrow$ R ordering yields models known as *total store order* (TSO) and *processor consistency* (PC).

There is a trade-off between *programmability* and *performance* while discussing memory consistency. The stronger the memory consistency model, the less challenging it is for programmers, but the more limited it is for hardware efficiency. Relaxed memory consistency models also provide programmers with mechanisms for overriding such relaxations. For example, different kinds of memory fences are provided to enforce ordering constraints on memory operations issued before and/or after the memory fence [8, 9].

Different combinations of hardware and operating systems may have different memory consistency models. For example, Sun SPARC has a weaker memory consistency model under Linux (which uses *relaxed memory order* or RMO) than under Solaris (which uses *total store order* or TSO) [10, 11]. Besides, programming language specifications usually grant compilers the license to presume that the code is single-threaded. This is true for older languages such as Fortran, C, C++, that are not designed with parallel programming in mind. Compilers, therefore, are free to reorder instructions. This could break the code which appears correct at the source level.

Inattention to the memory consistency issue can result in parallel programs that run correctly on uniprocessors or hardware with Hyper-Threading, but fail when run on multi-threaded hardware with *disjoint* caches [12].

Chapter 4

The Lock-Free and Cache-Friendly Properties

In this chapter, we show how the memory consistency issues may break Dekker's algorithm, a classical mutual exclusion algorithm. Tools used to writing lock-free code is introduced. Cache thrashing, which implies low cache utilization due to unnecessary memory traffic, hurts performance. We present common cache-friendly strategies that could be adopted when performance matters.

4.1 The Lock-Free Properties

Since DSWP exploits fine-grained pipeline parallelism in applications [4], fast synchronization and communication mechanisms are necessary. Use a lock-based approach, such as `pthread_mutex_lock`, to achieve synchronization is infeasible, as such a coarse-grained locking approach can significantly offset the parallelism brought up by DSWP. Besides, lock-based approaches come with a lot of pitfalls, from deadlocks and livelocks to priority inversion to convoying [13]. With lock-free approaches, synchronization is achieved through lower-level tools rather than mutex locks [14].

Lamport [15, 7] proved that a queue for SPSC (single producer single consumer) can be accessed concurrently without explicit lock only if the multiprocessor system is *sequentially consistent*. Although Lamport's conclusion seems heartening, most hardware and compilers used today do not provide the necessary sequential consistency [9, 16]. We will show how ordering and atomicity might not hold on modern multiprocessor systems and give an example demonstrating how the ordering issues break Dekker's mutual exclusion algorithm.

We discuss the ordering issue first. From the programmers' point of view, a computer system consists of a processor, memory, and the IO subsystem. In practice, there is a multi-level memory hierarchy since the significant speed gap between processor and memory. Each level in the memory hierarchy is smaller, faster than the next lower level. Memory accesses are expensive operations compared to other CPU's operations. In order to improve the performance of sequential programs, compilers, microprocessors, and caches put much emphasis on optimizing memory reads and writes. They may reorder, insert, or remove memory reads and writes in order to avoid or delay memory accesses.

Here we explain how the ordering issue could break Dekker's mutual exclusion algorithm on multiprocessor systems. Figure 4.1 gives the part of Dekker's algorithm achieving mutual exclusion. In Figure 4.1, X and Y represent different memory locations; r1 and r2 are registers of P1 and P2 respectively. Figure 4.2 gives three possible execution orders that illustrate the possible final values of r1 and r2 on a sequentially consistent multiprocessor system. Clearly, it is not possible that both X and Y are zero at the end of execution. This fact ensures mutual exclusion.

Compilers are allowed to reorder memory operations involving different memory locations. Since store operations cost much more time than does load operations, compilers will try to schedule memory loads early. The instructions which depend on those memory loads can be executed as soon as possible due to the early memory loads scheduling. Take Figure 4.1 as an example, a compiler might reorder the independent memory operations in the threads so that

Initially X = Y = 0	
Thread 1 on P1	Thread 2 on P2
X = 1; r1 = Y;	Y = 1; r2 = X;

Figure 4.1: The part of Dekker's algorithm achieving mutual exclusion

Execution 1	Execution 2	Execution 3

X = 1;
r1 = Y;
Y = 1;
r2 = X;
	Y = 1;
	r2 = X;
	X = 1;
	r1 = Y;
		X = 1;
		Y = 1;
		r1 = Y;
		r2 = X;

r1 == 0 r2 == 1	r1 == 1 r2 == 0	r1 == 1 r2 == 1

Figure 4.2: Valid executions of Fig. 4.1 on a sequentially consistent multiprocessor system

the memory loads can be executed early.

In addition, modern processors nearly always use a (hardware) *store buffer* to avoid waiting for the store instruction to complete. This means that a later read operation might reach memory before an earlier store operation. Both compilers and hardware optimization make the outcome of r1 == 0 and r2 == 0 possible, and hence may break Dekker's algorithm.

Next we discuss atomicity. On modern multiprocessor systems, atomicity is not always guaranteed. Consider what may happen if one thread assigns 1000000 to a 32-bit integer variable X on a 16-bit processor while the other thread reads that variable X? The assignment is translated into two hardware `store` instructions, one for each 16-bit half-word for the constant 1000000. Without an appropriate lock mechanism, the other thread might see an "intermediate" value. Note that common hardware does not guarantee bit-, byte-, or word-stores are atomic. Any shared data that could be modified has to be protected in some way.

Existing low-level tools used to realize lock-free operations include explicit memory fences

(e.g., `mb()` in Linux), special API calls (e.g., `InterlockedExchange` in Windows), and various special *atomic types*. Many of them are tedious or difficult to use. Worse still, their varieties imply that lock-free code is not portable.

In recent years, the computer industry gradually adopts *ordered atomic variables* as the main tool to write lock-free code in major programming languages and OS platforms. In short, ordered atomic variables are safe to read and write by several threads simultaneously without any explicit locking. Ordered atomic variables guarantee the following properties:

- The read and write operations are guaranteed to be executed under some ordering rules defined by programming languages and libraries.

- Each read or write operation on an ordered atomic variable is guaranteed to be atomic, all-or-nothing.

Many programming languages and libraries now support ordered atomic types that assure ordering and atomicity:

- Java provides ordered atomic types under the `volatile` keyword (e.g., `volatile int`), and solidifies this support in Java 5 (2004).

- .NET added them in Visual Studio 2005, also under the `volatile` keyword (e.g., `volatile int`).

- ISO C++ added them to the C++0x Draft Standard in 2007, under the templated name `atomic<T>`.

- Intel® Thread Building Blocks library provides template classes `atomic<T>`, which implement atomic operations in the C++ style.

Programmers need to know the ordering rules defined by the programming languages and libraries when using the ordered atomic type. Besides, data types and functions which are atomic

might not define an ordering rule. For example, functions `atomic_inc()` and `atomic_add()` provided by Linux kernel only guarantee atomicity but *not* ordering [9]. Using them to write lock-free code without any ordering enforcement (e.g., memory fence) is completely wrong.

4.2 The Cache-Friendly Property

Cache on multicore systems, as a two-side sword, can provide significant performance improvement or degradation. On the bright side, since cache offers much faster access speed than memory, cache improves performance. On the other hand, cache thrashing, which implies low cache utilization due to mismanaged memory traffic, hinders performance. A reason for cache thrashing is false sharing. False sharing arises from the use of an invalidation-based cache coherence algorithm with a single validity bit per cache line. False sharing occurs when a cache line is invalidated and a subsequent reference causes a miss because some word in the cache line, other than the one being read, is modified. If the modified word and the word being read are different, the miss is due to false sharing.

Take DSWP as an example. Without the synchronization array, communication between PROD and CONS must occur between cache and memory. If PROD (CONS) enqueues (dequeues) one data element at a time, false sharing might occur. Consider the following scenario: CONS intends to read a word `A` in cache line `L` that is shared with PROD after PROD writes a different word `B` in the same cache line `L`. CONS will have a miss due to false sharing.

The above false-sharing miss could be avoided if PROD (CONS) enqueues (dequeues) one or several cache lines at a time. Besides, we could allocate variables into different chunks according to their locality in order to avoid false sharing. Based on the locality, variables can be broadly divided into the following categories [12]:

- Thread-private variables are private to a given thread and are not shared with other threads. Accesses to thread-private variables tend to be very fast and will not consume bus band-

width.

- Shared read-only variables are shared among multiple threads, but are not modified by them. Since there are no writes, every thread tends to keep its own copy in its local cache, as long as it fits.
- An exclusive variable is read and written by multiple threads, but is protected by a lock. Once a thread acquires the lock, the variable will be moved into that thread's local cache. At the meantime, no other thread can touch the variable.
- Wild-west variables are read and written by unsynchronized threads. Ordered atomic variables fall into this category.

We prefer thread-private variables and shared read-only variables as they have lower impact on the bus and do not need synchronization. Moreover, variables of different localities should not be mixed on the same cache line. Putting a shared read-only variable and a wild-west variable on the same cache line impedes access to the shared read-only variable because accesses to the wild-west variable will cause the cache line ping-ponging between threads.

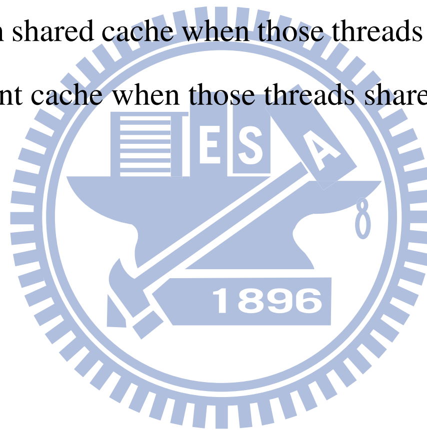
The techniques used to layout those variables in a specific way include padding and alignment [17]. Padding simply puts a block of useless bytes between variables as needed. Alignment, however, is not so straightforward. Compilers align variables according to some rules. For example, an `int` variable will be aligned to its native word size.

Compiler vendors provide programmers with various mechanisms to change compilers' alignment rules. For example, GCC has a syntax extension `__attribute__ ((aligned()))`. In order to unify alignment mechanisms, ISO C++ adds a new language syntax `attribute` to the forthcoming C++0x Standard [18]. One of the `attribute` tokens defined in the standard is `align`, which specifies the alignments as programmers require.

Thread migration is another reason of cache thrashing in addition to false sharing. Consider the following scenario: Core 1 and core 2 share no cache. Thread A runs on core 1, then is

blocked for waiting for an event. When the event occurs, thread A might be scheduled to run on core 2 by the OS. It is possible that some data modified by thread A still stay in the cache of core 1. The data must be flushed from the cache of core 1, then placed in the cache of core 2. The cost is even higher when core 1 and core 2 belong to different groups on Non-Uniform Memory Access (NUMA) machines.

Processor affinity, which binds a thread to a particular core, is one way to solve the above problem. If data is left in the cache from the previous time the thread ran, then cache thrashing caused by thread migration can be avoided. Processor affinity works well for single-threaded applications, but may not always work for multi-threaded applications. There are two rules of thumb when applying processor affinity on multi-threaded applications: First, binding threads to one core or on cores with shared cache when those threads have shared data. Second, binding threads to cores with disjoint cache when those threads share no data.



Chapter 5

Implementation and Verification

In this section, we present our implementation of the `QueueBuffer` class, and explain its correctness both in informal and formal ways.

5.1 Implementation

We present the C++ template class `QueueBuffer`, which is a lock-free, cache-friendly, STL-compliant queue. Figure 5.1 shows the data members of class `QueueBuffer`. The most important data members are `m_front` and `m_back`, which are shared, mutable variables. Compilers do not yet support ordered atomic types defined by the forthcoming C++0x Draft Standard. So we declare them as ordered atomic variables by using the template class `atomic<T>` provided by Intel® Threading Building Blocks library. An `atomic<T>` class supports atomic read, write, fetch-and-add, fetch-and-store, and compare-swap operations. For reads and writes, their default memory fences are `acquire` and `release`, respectively.

Hence, the ordered atomic read is also known as a acquiring read, and the ordered atomic write is also called a releasing write.

An `acquire` fence prevents memory operations *after* the fence moves over it. On the other hand, a `release` fence prevents memory operations *before* the fence moves over it. We will

```
Class QueueBuffer {
  <align> const size_type m_cap;
  const size_type m_cds;
  const value_type m_eof;
  allocator_type m_alloc;
  pointer m_buf;

  // padding
  char pad1[CACHE_LINE_SIZE - 2*sizeof(m_cap) -
    sizeof(m_eof) - sizeof(m_alloc) - sizeof(m_buf)]

  tbb::atomic<size_type> m_front;

  // padding
  char pad2[CACHE_LINE_SIZE - sizeof(size_type)]

  tbb::atomic<size_type> m_back;
}
```

Figure 5.1: Class QueueBuffer data members

show how the ordered atomic variables allow us to access the queue concurrently without locks later.

Since false sharing will cause unnecessary memory traffic, we take some steps to avoid false sharing. First, the software interfaces emulating the `produce` and `consume` instructions have been designed to take false sharing into consideration. Therefore, PROD (CONS) will enqueue (dequeue) multiple (`m_cds`) data elements whose total size is a multiple of the cache-line size at a time. Second, according to the locality as illustrated in section 4.2, we group class `QueueBuffer` data members into different chunks that are multiples of the cache line size and aligned on cache line boundaries by using alignment and padding. Data members are aligned to the cache-line boundary. Padding is added between different groups of data members. Most compilers do not yet support `attribute` defined by the forthcoming C++0x Draft Standard. Therefore, we have to use compiler-specific syntax extensions to control a compiler's alignment operations.

Figure 5.2a and Figure 5.2b show the class `QueueBuffer` member functions called by

```
void push(Iter begin, Iter end,
  bool last) {

size_type local_back = m_back;

const size_type next_back =
  (local_back + m_cds) % m_cap;

while ( next_back == m_front)
  ; // spinning if queue is full

// insert data into m_buf

m_back = next_back;
}
```

(a) push

```
void front(Iter begin) {

const size_type local_front
  = m_front;

while (local_front == m_back)
  // spinning if queue is empty
  ;

// retrieve data from m_buf

m_front =
  (local_front + m_cds) % m_cap;
}
```

(b) front

Figure 5.2: Class QueueBuffer member functions

PROD and CONS, respectively. We use `m_back` and `m_front` to check if the queue is empty or full. The queue is empty when `m_back` == `m_front` and is full when (`m_back` + `m_cds`) % `m_cap` == `m_front`. We now show how `atomic<T>` allows us to access the queue concurrently without locks. First, although `m_back` and `m_front` are shared, mutable variables, they are safe to be read and written by PROD and CONS simultaneously without locks as atomicity is guaranteed by `atomic<T>`.

Second, as pointed out by [19, 20, 21], there is an ordering problem between PROD and CONS. CONS might see that `m_back` has been incremented by PROD before it sees the change to the corresponding `m_buf` slot if the memory consistency model is very relaxed. The solution proposed by [19] is to use mutexes to flush the cache. We do not need mutexes here since the ordered atomic write associated with a `release` fence which guarantees the change of `m_buf` slot occurs before it. We show the queue can be accessed concurrently without locks.

When writing parallel programs, we need to understand the ordering rules provided by programming languages and libraries. For the problem in Dekker's algorithm as demonstrated in Section 4.1, acquiring reads and releasing writes do not fix the problem. Instead, the fix needs to stop reads from floating backwards over writes, but acquiring reads can nonetheless float backwards over releasing writes.

As shown in Figure 5.2a and Figure 5.2b, m_back and m_front are copied to local variables local_back and local_front before checking if the queue is empty or full. Since m_back and m_front are only written by PROD and CONS, respectively, we are free to do so. Because local variables can be cached in registers or cache, doing so can avoid reloading the ordered atomic variables unnecessarily.

There is one last thing that needs to be considered. In DSWP, both PROD and CONS are in a continuous loop of accessing the queue. We need to terminate CONS when PROD produces no more data. When writing a sequential program or a parallel program with locks, we can associate an EXIT flag with the queue. When PROD produces no more data, the queue's EXIT flag is set to true. When CONS tries to dequeue data, if the queue is empty and the EXIT flag is true, then CONS exits the loop.

The above approach might not work for a lock-free parallel program. A typical pattern for lock-free programs is do the work off to the side, then *publish* each change to the shared data with a single ordered atomic write or compare-and-swap [22]. The difficulty of writing lock-free code is that we are only allowed to use a *single* ordered atomic write or compare-and-swap operation to update the shared object. In our case, PROD will update m_back after inserting data elements into the m_buf slot. And CONS will update m_front after retrieving data elements from the m_buf slot. We cannot update m_back and the EXIT flag at the same time.

There is another approach, however. Stopping CONS can be accomplished by having PROD pass an EOF through the queue. When CONS receives an EOF, CONS exits the loop.

The member function push_back_n in the QueueBuffer class requires a parameter, last_iter, set by PROD when there is no more data. Then push_back_n will insert an m_eof whose value is a template argument provided by programmers. The m_eof has to be inserted with the last chunk of data. If not, since the front_n member function will return m_cds data unconditionally, it will return some garbage data.

5.2 Verification

```
unsigned head, tail : 16;
unsigned cds : 16 = 1;
unsigned capacity : 16 = 16;

active proctype producer() {
  do
  :: !((head + cds) % capacity == tail) ->
        printf("Produce\n");
     head = (head + cds) % capacity;
  od
}

active proctype consumer()
{
  do
  :: !(head == tail) ->
        printf("Consume\n");
     tail = (tail + cds) % capacity;
  od
}
```

Program 1: SPIN model for class QueueBuffer

As indicated in [23], the non-deterministic nature of a parallel system makes it hard to use traditional testing techniques to verify the parallel system. Our lock-free queue belongs to this case. Hence, it is insufficient to run test cases to verify the correctness of our lock-free queue. Instead, the implementation should be modelled and verified in a formal way. In this paper, we use the SPIN model checker [24] to verify our design.

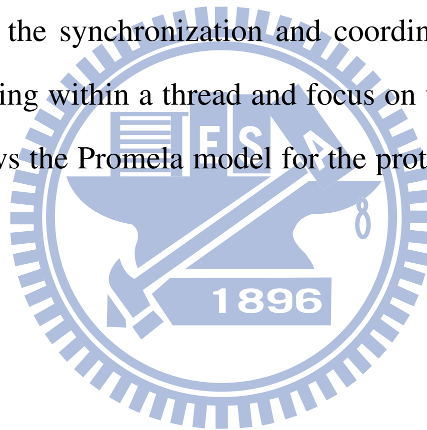
The tool we use to check our design is called SPIN, and the specification language that it accepts is call Promela (Process Meta-Language). Promela is a process modelling language whose intended use is to verify the logic of parallel systems. It is not to be an implementation language like C/C++, but a system description language for building verification models. Therefore, Promela focuses on modelling of processes, synchronization and coordinations, but

not on computations.

A Promela program is composed of processes, values and message channels. Processes are global objects that represent concurrent entities of the parallel system. Variables and message channels can be defined either globally or locally within a process. Processes specify their behavior; global variables and message channels together define the environment in which processes run.

SPIN simulates the operation of a parallel system by running its Promela program. This is the SPIN simulation mode. SPIN also has verification mode. SPIN can convert a Promela program into a C source code which is a verifier. Once the verifier discovers flaws in the parallel system, we can rely on the SPIN simulator to display the error traces.

Since Promela models the synchronization and coordination between processes, we can ignore the activities occurring within a thread and focus on the interactions among concurrent processes. Program 1 shows the Promela model for the protocol used by the producer and the consumer.


1896

Chapter 6

Experiment

This section describes the platforms and benchmarks we used in our implementation of the `QueueBuffer` class.

6.1 Platform and Benchmark

Experiments were conducted on an IBM System x3400 Server [25] and a Sun SPARC Enterprise T5120 Server [26]. The IBM System x3400 Server is equipped with two quad-core Intel Xeon E5335 processor. Each pair of cores in a quad-core Intel Xeon E5335 processor share a 4 MB L2 cache [27]. The Sun SPARC Enterprise T5120 Server equipped with one UltraSPARC T2 processor which has eight cores. All cores in a UltraSPARC T2 processor share a 4 MB L2 cache divided into eight banks [28]. Table 6.1 shows the detailed parameters of our experimental platforms.

We manually applied DSWP on three linked-list traversal loops in three SPEC CPU2006 benchmarks [29]: 429.mcf, 450.soplex, and 453.povray. We also experimented on a micro-benchmark, called llubench, which focuses on linked-list manipulation [30]. Llubench was modified so that CONS does more computation [1]. More detailed information of the benchmarks

[1] See Figure 6.1

Machine Name	x86-64	NIAGARA
System	IBM System x3400	Sun SPARC Enterprise T5120
CPU	Intel Xeon E5335	Sun UltraSPARC T2
CPU / System	2	1
Core / CPU	4	8
Thread / Core	1	8
L1 I Cache	32 KB	16 KB
L1 D Cache	32 KB	8 KB
L2 Cache	4 MB	4 MB
Cache Line Size	64 B	64 B
Memory	10 GB	32 GB
OS	Linux 2.6.32	Solaris 10 10/09
Compiler	g++ 4.4	Sun Studio 12 u1 CC
Compiler Optimization	-O3	-xO5

Table 6.1: Experiment Platforms

Program Name	Function	Input Data Size	Description
429.mcf	refresh_potentiontial	test	linked-list traversal loop
450.soplex	forestPackColumns	test	linked-list traversal loop
453.povray	Build_Bounding_Slabs	test	linked-list traversal loop
llubench	main	[1]	linked-list traversal loop

[1] traverse 100 thousand nodes

Table 6.2: Benchmark Programs

is shown in Table 6.2.

PROD and CONS are extracted from the loops and combined with OpenMP [31] and the `QueueBuffer` class. Figure 6.1 and Figure 6.2 show the serial and parallel code for an example, respectively. Processor affinity is achieved with platform-specific mechanisms (e.g., `numactl` in Linux or `SUNW_MP_PROCBIND` in Solaris). Flawed parallel programs might run incorrectly on cores with disjoint caches or fail when aggressive compiler optimizations are turned on. In our experiments, we turn on all possible compiler optimization options, and also bind PROD and CONS to cores with/without shared cache. The execution results of all benchmarks were correct. And we use the `tick_count` class provided by Intel® Threading Building Blocks to measure wall-clock time.

```
while (trav != NULL) {
  accumulate += trav->count;
  if (dirty) {
    srand ( trav->count++ );
    double x = rand();
  }
  trav = trav->next;
}
```

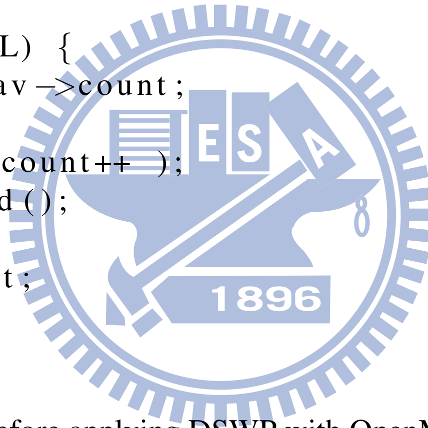


Figure 6.1: Before applying DSWP with OpenMP and QueueBuffer

6.2 Result and Discussion

Table 6.3 shows the execution times of the serial and parallel versions of the benchmarks. The parallel version is obtained by translating the benchmarks into the DSWP style. The parallel version shows significant slowdown compared to the serial version. As shown in Figure 9, the parallel version includes additional control flows and auxiliary data structures, which may offset the benefits brought up by DSWP. Besides, compared with *synchronization array* which accessed by machine instructions `produce` and `consume`, we need to use functions

	x86-64		NIAGARA	
	Serial	Parallel	Serial	Parallel
429.mcf	5.83s	9.55s	31.72s	161.71s
450.soplex	0.162s	0.172s	1.787s	1.792s
453.povray	3.34s	3.44s	29.238s	30.588s
llubench	0.276s	0.210s	0.0183s	0.0287s

Table 6.3: Experiment Result

`push_back_n` and `front_n` to access `QueueBuffer`. Although we can access the shared resource `QueueBuffer` concurrently without locks, the cost of function calls are still considered as overhead.

More efforts are needed to shorten the code sequences in functions `push_back_n` and `front_n`. Doing so makes compilers be able to inline function `push_back_n` and `front_n`. The cost of function calls, therefore, can be eliminated by the inlining.

One might think the shared cache could be a fast communication mechanism, compared with memory. As mentioned in [32], shared cache could be an alternative communication mechanism among cores when compared with much slower memory. However, in our experiments, we saw no benefit from shared cache. Apparently, we cannot obtain any benefit from shared cache automatically. Further detailed profiling is needed to answer why shared cache does not help. Programmers have no explicit control over cache usage, but only follow general principles. Programmers might have to craft non-portable code to fully utilize cache. Such an approach might need deep knowledge of the processor, cache, and memory system on a particular platform.

```
dswp::QueueBuffer<element *> qb;
const size cds = qb.get_cds();
#pragma omp parallel num_threads(2) shared(qb)
{
  if (omp_get_thread_num() == 0)
  {
    struct element *arr[8]; size_t i = 0;
    while (trav != NULL) {
      arr[i++] = trav;
      trav = trav->next;
      if (!trav)
        qb.push_back_n(arr, arr + i, true);
      else if (i == cds)
      {
        qb.push_back_n(arr, arr + cds); i = 0;
      }
    }
  }
  else if (omp_get_thread_num() == 1)
  {
    struct element *arr[8]; bool flag = true;
    while (flag)
    {
      qb.front_n(arr);
      for (size_t i = 0; i < cds; ++i)
      {
        if (arr[i])
        {
          accumulate += arr[i]->count;
          if (dirty) {
            srand( arr[i]->count++ );
            double x = rand();
          }
          else
          {
            flag = false; break;
          }
        }
      }
    }
  }
}
```

Figure 6.2: After applying DSWP with OpenMP and QueueBuffer

Chapter 7

Related work

Over the past few years, a lot of research results on lock-free queues for SPSC (single producer single consumer) have been published. Most of them [33, 34, 35] are variations of Lamport's queue [15]. However, they all overlooked the atomicity and ordering issues on multiprocessor systems. As illustrated in Section 4.1, none of them holds inherently on multiprocessor systems. Some mechanisms have to be adopted to ensure atomicity and ordering.

Therefore, we should carefully consider the atomicity and ordering properties when implementing a lock-free algorithm. We need to handle shared mutable variables carefully even if they are modified by only one thread. Modifying them without locks even when they are simple data types (e.g., `int`) is completely wrong. Declaring them as `volatile` variables is also problematic, at least for C/C++ [36, 37]. The `volatile` keyword in C/C++ does not ensure atomicity nor ordering. In fact, the purpose of the `volatile` keyword in C/C++ are: (1) allow access to memory mapped devices, (2) allow uses of variables between `setjmp` and `longjmp`, and (3) allow uses of `sig_atomic_t` variables in signal handlers. These are completely orthogonal to threads [38].

The other issue is performance. The `head` and `tail` indices in Lamport's queue are called *control variables*. We use control variables to check if the queue is empty or full. If there are

frequent communications between threads like DSWP, then the producer and consumer threads have to frequently access control variables from memory. The frequent communication results in poor performance.

In order to eliminate the overhead of accessing control variables, Giacomoni, et al. proposed the FastForward queue [39]. The FastForward queue uses the data stored in the queue rather than control variables to indicate the empty- and full-queue conditions. This technique is called *data/control coupling*. For example, if the data stored in the queue are pointers, then we can use NULL to indicate a buffer slot is empty. In this approach, the control variables (i.e., `head` and `tail`) are thread-local so that they can be kept in the cache.

The FastForward queue solves the problem with Lamport's queue. However, it has its own limit. The correctness of the FastForward queue holds only under given assumptions on multiprocessor systems. Coupling data and control into a single operation is assumed to be atomic in the FastForward queue. Such an assumption is valid only for rare case (e.g., aligned native-word-size data). Besides, CONS termination might be a problem for the FastForward queue since we need find another special value as an `EOF`.

Chapter 8

Conclusion and future work

In this paper, we present a lock-free, cache-friendly C++ template class `QueueBuffer`. We show how memory coherence and consistency play important roles on writing a correct lock-free code. The difficulty in writing lock-free programs comes from the lack of a memory model in programming languages. Memory models make reasoning about the correctness of parallel programs much more formal and easier. Programmers do not need to worry that compilers and hardware might change the meaning presented by the source code. As an example, Java programmers can write parallel programs much easier and more comfortable since Java has a sequential-consistency memory model [40]. On the contrary, C/C++ have no memory model so far. The bright side is the forthcoming C++ Standard has already defined a memory model [18].

Cache, as a two-side sword, can provide significant performance improvement or degradation of applications. Since there is no explicit control over cache, it is a challenge for programmers to write cache-efficient programs without precise profiling. In recent years, shared cache on multicore systems has became an interesting topic. Tian mentioned that the shared cache could be an alternative communication mechanism among cores instead of the much slower traditional memory [32]. Using shared cache as an alternative communication mechanism, however, is not easy as it might look like. It is not clear if there is a way in which we

can ensure correctness with ordered atomic types and obtain performance promised by shared cache at the same time. How to use shared cache as a communication mechanism among cores depends on the underlying systems. For example, it is possible that the data written by PROD cannot reach shared cache on time. Then CONS will have a cache miss. This topic needs to be further explored.

Bibliography

[1] H. Sutter. (2005) The free lunch is over. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[2] R. Allen and K. Kennedy, Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers, 2002.

[3] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August, "Decoupled software pipelining with the synchronization array," in Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. IEEE Computer Society Washington, DC, USA, 2004, pp. 177–188.

[4] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2005, p. 118.

[5] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. August, and G. Cai, "Support for high-frequency streaming in CMPs," in Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2006, pp. 259–272.

[6] J. Hennessy, D. Patterson, D. Goldberg, and K. Asanovic, Computer architecture: a quantitative approach. Morgan Kaufmann, 2007.

[7] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess progranm," IEEE transactions on computers, vol. 100, no. 28, pp. 690–691, 1979.

[8] S. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," Computer, vol. 29, no. 12, pp. 66–76, 1996.

[9] P. E. Mckenney, "Memory ordering in modern microprocessors," Linux Journal, vol. 30, pp. 52–57, 2005.

[10] J. Reinders, Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc., 2007, pp. 122–129.

[11] P. McKenney and I. Beaverton, "Memory Barriers: a Hardware View for Software Hackers," 2009.

[12] S. Akhter and J. Roberts, Multi-core programming: increasing performance through software multi-threading. Intel Press, 2006, pp. 212–213.

[13] H. Sutter. (2005) The trouble with locks. [Online]. Available: http://www.drdobbs.com/cpp/184401930

[14] ——. (2009) volatile vs. volatile. [Online]. Available: http://www.drdobbs.com/hpc-high-performance-computing/212701484

[15] L. Lamport, "Proving the correctness of multiprocess programs," IEEE Transactions on Software Engineering, pp. 125–143, 1977.

[16] S. Adve and H. Boehm, "Memory models: a case for rethinking parallel languages and hardware," in Proceedings of the 28th ACM symposium on Principles of distributed computing. Citeseer, 2009, p. 2.

[17] S. Norton and M. DiPasquale, Thread Time: A Multi-Threaded Programming Guide. Prentice Hall PTR Upper Saddle River, NJ, USA, 1996, pp. 412–414.

[18] P. Becker. (2010) Programming languages - C++. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf

[19] C. SunSoft, Solaris multithreaded programming guide. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1995, pp. 117–118.

[20] H. Sutter. (2008) Lock-free code: A false sense of security. [Online]. Available: http://www.drdobbs.com/cpp/210600279

[21] Intel® Threading Building Blocks - Design Patterns, Intel Crop., 2010. [Online]. Available: http://www.threadingbuildingblocks.org/uploads/81/91/Latest\%20Open\%20So\%urce\%20Documentation/Design_Patterns.pdf

[22] H. Sutter. (2008) Writing lock-free code: A corrected queue. [Online]. Available: http://www.ddj.com/high-performance-computing/210604448

[23] S. Padidar, "Parallel Program Verification: A Brief Introduction," 2010.

[24] G. Holzmann, The SPIN model checker: Primer and reference manual. Addison Wesley Publishing Company, 2004.

[25] IBM System x3400, IBM, 2007. [Online]. Available: http://www-07.ibm.com/systems/includes/pdf/XSD02288USEN.pdf

[26] SUN SPARC ENTERPRISE T5120 SERVER, Oracle, 2009. [Online]. Available: http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/035999.pdf

[27] Quad-Core Intel®Xeon®Processor 5300 Series, Intel Crop., 2006. [Online]. Available: http://www.intel.com/Assets/en_US/PDF/prodbrief/xeon-5300.pdf

[28] UltraSPARC T2 supplement to the UltraSPARC architecture 2007, Sun Microsystems, Inc., 2007. [Online]. Available: http://opensparc-t2.sunsource.net/specs/UST2-UASuppl-current-draft-P-EXT.pdf

[29] J. Henning, "SPEC CPU2006 benchmark descriptions," ACM SIGARCH Computer Architecture News, vol. 34, no. 4, p. 17, 2006.

[30] C. Zilles, "Benchmark health considered harmful," ACM SIGARCH Computer Architecture News, vol. 29, no. 3, p. 5, 2001.

[31] L. Dagum and R. Menon, "Open MP: An Industry-Standard API for Shared-Memory Programming," IEEE Computational Science and Engineering, vol. 5, no. 1, pp. 46–55, 1998.

[32] T. Tian. (2007) Effective use of the shared cache in multi-core architectures. [Online]. Available: http://www.drdobbs.com/high-performance-computing/196902836

[33] Y. Zhang, K. Ootsu, T. Yokota, and T. Baba, "Clustered Decoupled Software Pipelining on Commodity CMP," in 14th IEEE International Conference on Parallel and Distributed Systems, 2008. ICPADS'08, 2008, pp. 681–688.

[34] P. Lee, T. Bu, and G. Chandranmenon, "A lock-free, cache-efficient shared ring buffer for multi-core architectures," in ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2009.

[35] T. Jablin, Y. Zhang, J. Jablin, J. Huang, H. Kim, and D. August, "Liberty Queues for EPIC Architectures," in Proceedings of the Eigth Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology (EPIC), 2010.

[36] B. Lewis and D. Berg, Multithreaded programming with java technology. Prentice Hall PTR, 2000, pp. 159, 357.

[37] R. Carver and K. Tai, Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs. John Wiley and Sons, 2006, pp. 54, 77.

[38] H. Boehm and N. Maclaren. (2006) Should volatile acquire atomicity and thread visibility semantics? [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2016.html

[39] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM, 2008, pp. 43–52.

[40] J. Manson, W. Pugh, and S. Adve, "The Java memory model," in Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 2005, pp. 378–391.