QDo: A Quiescent State Callback Facility

by

Adrian Tam

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

QDo: A Quiescent State Callback Facility

Adrian Tam

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2006

In systems that only have short-lived threads, it is possible to define a Quiescent Point (QP(t)), relative to time t, as the earliest time after t where no thread exists that was alive at time t. Quiescent points have a number of uses, such as memory reclamation for lock free objects.

QDo is a facility that allows clients to register their interest for QP(t). QDo then monitors and notifies interested parties when a quiescent point is established. We have designed, implemented and analyzed the performance of the QDo facility. QDo works by checking the number of active threads in the previous epoch per processor. When that number reaches zero, a local quiescent point is established. QDo then communicates with other processors to establish a global quiescent point.

Our experiment shows that callback latency and overhead is highly dependent on how frequently the QDo facility monitors for quiescent point. Checking frequency of 10ms results in 2% overhead for SDET benchmark.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Shared memory multiprocessors (SMPs) are becoming increasingly prevalent. With the emergence of chip multiprocessors (CMPs) and simultaneous multi-threading processors (SMTs), one can even expect standard desktop computers and game consoles to dominantly be multi-processors within a few years. Today, Intel sells the Core Duo processor [43], AMD has released the Opteron processor [5], IBM has the Cell processor [44] and Sun sells the Niagra processor [46]. The Intel Core Duo processor, already employed in the Apple iMac notebook, has two execution cores within a single processor. The AMD Opteron processor, designed for servers and workstations, also has two cores on a single die. It supports both simultaneous 32 and 64 bits computations. The Sony Playstation 3 game console uses IBM Cell Processor as its main central processing unit (CPU). Each cell processor has one Power Processing Element (PPE) and eight Synergistic Processor Elements (SPE). The PPE is a dual-threaded, dual-issue 64 bit Power-Architecture compliant core that can run conventional operating system while each SPE is a reduced instruction set computer (RISC) processor that operates on local store memory. The Sun Niagra processor, also known as UltraSPARC T1, is designed for thread rich applications. Each processor has up to eight 4-way multi-threaded cores. Based on the current trend, it is not inconceivable

that within five years, low-end computers will be 8-way or higher multiprocessor machines.

Software for multiprocessor platforms is significantly more complex than multiprocessing software for uniprocessors. A key aspect is that synchronization is needed to protect consistency of common data structures that might be accessed in parallel. The cost of synchronization is high and, relative to processor speeds, has become higher over the years. Small et. al. measured that under heavy load, NetBSD 1.2 can spent 9% to 12% of its time executing synchronization operations [76]. McKenney et. al. demonstrated that lock efficiency, as measured by the speed of synchronization instruction divided by the speed of a non-synchronization instruction, has decreased by an order of magnitude from 1984 to 2001 [55]. Synchronization overhead stems from two sources: the overhead of executing atomic operations (for example test-and-set instructions) and contention overhead (such as spinning or queuing delays) that arises when multiple threads wish to execute a critical section at the same time.

Over the years, there has been a significant amount of research aimed at reducing synchronization overhead. Some of the research targets the restructuring of software so as to reduce the amount of data that is shared, and with it, the need for synchronization. For example, the K42 research team developed a structure for the K42 operating system where an object can be decomposed into multiple representatives while preserving a single unified interface [8]. Objects can customize its allocation by its partitioning, distributing and replicating its data so as to maximize locality and minimizing sharing.

Other researches have developed lock-free data structures, including lock free quajects [52] and lock free linked list [32]. Lock free data structures do not require explicit synchronization, such as locking, in order to protect the structures' integrity. In addition, priority inversion is avoided. Instead, through sequences of atomic operations, they ensure that a consistent picture is seen by all processors. One key problem with lock free structures is that it is difficult to know when it is safe to destroy an object. For example, even if an

object is dequeued from a linked list with no synchronization, it is difficult to know when it is safe to free the object without using traditional locking mechanisms, since other parallel threads may still be accessing the dequeued object.

Still other researches have developed techniques that exploit expected access behaviour to reduce synchronization overhead in the common case, but with higher overhead in the uncommon case. An example of this is the Read-Copy-Update (RCU) mechanism developed by Paul McKenney, and now a part of the Linux 2.6 kernel. RCU is optimized for read mostly data structures [54]. Reads do not block nor require atomic synchronization primitives. Writes, on the other hand, incur large latencies because their executions are delayed to known safe points [56].

1.1 Problem Statement

Operating systems, databases, web servers and other system software are typically demand driven, where threads execute only on stimulus from outside the system. For example, an operating system experiences activity only when applications issue a system call or there is an interrupt. Typically, threads that execute on behalf of outside parties are short-lived, namely for the duration required to service the request.

In systems that only have short-lived threads, it is possible to define a Quiescent Point (QP(t)) relative to time t as the earliest time after t where no thread exists that was alive at time t. Quiescent Points can be exploited in a number of ways. For example, for lock free data structures, it is safe to destroy at time QP(t) objects that were dequeued at time t, because no thread that potentially had a reference to the dequeued object can possibly still be executing. Similarly, for RCU, it provides guarantees on when all threads are able to see the effects of changes to the updated data structure [54]. However, detecting Quiescent Points in an efficient manner is not trivial.

In this dissertation, we describe the design and implementation of a Quiescent Point callback facility called QDo. Clients of the QDo facility can register quiescent point callbacks, and the QDo facility issues the callback the next time quiescent point is reached. We explore different ways of implementing QDo and assess their performance characteristics.

1.2 Motivation

Many problems can take advantage of QDo facility. Here we describe a few example uses.

One obvious use for QDo is memory reclamation for lock free objects. In theory, an object's memory can be reclaimed immediately after it is deleted. However, other threads executing on another processor might still have access to the deleted memory via transient references [62]. QDo provides a means to detect when other threads can no longer access the memory occupied by the object. Thus, memory reclamation can be implemented as callbacks after quiescent point is reached.

Hot swapping is another problem that can benefit from using the QDo facility. Hot swapping supports live upgrades to running system component and allows software to be more adaptable by changing its behaviour at run time. This is achieved by swapping one component for another component [25, 41]. QDo simplifies hot swapping by identifying when it is safe to swap out the existing object - the point at which no other thread will reference the old object.

Another use of QDo is hash table resizing. To maintain efficiency and avoids collisions, hash tables should be resized when the number of items in the table is larger than a threshold [23]. One common method to protect hash table integrity is to lock hash table entries and prevent other threads from modifying the hash table during resizing. However, this will slow the common read and write paths, since every access will require lock synchronization. As an alternative, we can use a two step process to resize the table. For most of

the time, reading and writing does not require a lock. At time t, we ask the QDo facility to schedule the resizing of hash table after QP(t). From time t to time QP(t), reading and writing to a hash table requires acquiring a lock.

1.3 QDo Design Goals

The QDo facility has four main goals. They are, in order of importance,

1. Low Overhead
2. Low Latency for Quiescent Callback
3. Scalability
4. Available in Both User and Kernel Space

Low overhead means that the latency and throughput of other non-QDo related calls should not be affected significantly by the presence of the QDo facility, especially when there are no outstanding callbacks. If the overhead were high, software designers would be reluctant to incorporate the facility into their systems.

Another important requirement for QDo is to have low latency for Quiescent callbacks. Latency in the case refers to the time between placing callback request onto the QDo facility and the execution of callback. There are several factors that affect latency, including system load, the number of outstanding QDo requests as well as the duration of callbacks. Our goal is to reduce the latency due to QDo's detection and callback mechanisms.

Scalability is another important requirement. The QDo facility was developed for the K42 Operating System, and K42 is designed for large scale shared memory multiprocessors. Hence, QDo must be as scalable as K42 with overhead and latency not increasing

disproportionately relative to the number of processors. Without requiring scalability, QDo may become the choking point for K42 in larger multiprocessor systems.

Finally, the QDo facility should be available both at the application level and the kernel level. In particular, many important kernel structures, such as lock free objects, can benefit from the QDo facility. Hence, we need to extend the facility to beyond user level code.

1.4 Summary of Results

QDo's callback latency is highly dependent on how frequently the QDo facility monitors whether quiescent point has been reached. Callback latency is low when each processor frequently monitors for quiescent condition. In addition, the method of communicating a processor's state with other processors also influences callback latency. Taking regular snapshot of other processors' state produces the lowest latency.

Overhead is also dependent on how frequently QDo facility monitors whether quiescent point is reached. Frequent checking results in high overhead. Based on our micro-benchmark experiments, we determine that a checking frequency of 10ms results in 1% overhead in the thread creating and destruction path. We have confirmed that this translates to about 2% overhead when running the SPEC Software Development Environment Throughput (SDET) benchmark.

1.5 Structure of Dissertation

This dissertation focuses on the design and implementation of QDo, a Quiescent Point callback facility for the K42 Operating System. Chapter 2 provides background information that enables the reader to understand the remainder of the dissertation. In addition, it presents work related to this research. Chapter 3 describes the design of the QDo facility,

while Chapter 4 presents implementation details. Chapter 5 includes and analyzes results of tests with micro-benchmarks and macro-benchmarks. Concluding remarks and future work is presented in Chapter 6. Appendix A provides a glossary of terminology definitions.

Chapter 2

Background and Related Work

This chapter first discusses some of the prior art in the area of synchronization and then gives an overview of the K42 operating system for which the QDo facility was designed.

2.1 Related Work

2.1.1 Synchronization

Memory Consistency Models

Leslie Lamport demonstrated in 1979 that maintaining data correctness within a multi-processed program executing on a modern multiprocessor does not require that every data access be atomic [49]. Instead, a program can be proved to be correct if it obeys a few rules on its operation sequences. These rules include:

1. "All write operations to a single memory cell by any one process are observed by all other processes in the order in which the writes were issued".

2. "A synch command causes the issuing process to wait until all previously issued memory accesses have completed".

3. "A read of a memory cell that resides in the process's cache precedes every operation execution issued subsequently by the same process".

Programs that follow these rules are said to adhere to the strictly sequential consistency model.

Since then, there has been much research on further relaxing shared memory access requirements [2, 3, 6, 29, 45, 64]. Weaker memory consistency models, including "cache consistency" [30], "processor consistency" [4], "weak consistency' [24]', "release consistency" [29] and "entry consistency models" [17] can achieve performances improvements on the order of 10 to 40 percent over the strictly sequential consistency model [64]. Modern compiler technologies have taken advantage of these weaker consistency models and made parallel accesses more efficient [15, 86]. However, explicit synchronization, such as memory barriers and locks are often still needed to prevent race conditions around critical regions.

Locks

Improving lock implementation can also improve shared memory multiprocessors (SMP) performance. For example, Mellor-Crummey et. al. stipulated that hot spot memory contention during busy waiting can be eliminated by having each processor spin on a local variable [60]. This technique is easily implementable on modern processors as it only has two requirements. First, a fetch_and_Φ operation (i.e., an operation that reads, modifies, and then writes a memory location atomically) needs to be available. Common processor instructions, such as test_and_set, fetch_and_store, fetch_and_add, and compare_and_swap satisfy this requirement. Second, there has to be a memory hierarchy in which each processor is able to read some portion of shared memory without using the interconnection network. This is the case on many modern SMP systems. Although Mellor-Crummey

et. al. 's approach reduces communication overhead during synchronization significantly, McKenney et. al. showed that locally accessed spin-locks are still not ideal from a scalability point of view [55]. CPU resources are spent on servicing memory latency, as each processor still has to modify the same global lock.

Tullsen et. al. seeks to reduce locking overhead through fine-grained synchronization on simultaneous multi-threading (SMT) processors by taking advantage of communicating threads executing in parallel on a single processor [80]. Tullsen et. al. 's scheme hands over a lock from one thread to another efficiently by recording the address of blocked instructions in a thread-shared hardware, known as lock-box. The drawback of his scheme, however, is that this method is only applicable to SMT processors and requires specialized hardware.

Preemption-safe locking addresses performance degradation on time-slicing multiprogrammed systems due to the preemption of processes holding locks [47]. While the lock-holding process is preempted, other processes waiting on the lock are unable to perform useful work. Solutions include avoidance (i.e., not preempting a process that is holding a lock) and recovery (blocking a process that fails to get a lock). These techniques require the kernel and scheduler to cooperate with the waiting processes. Michael et. al. asserts that non-blocking preemption-safe locks outperform both ordinary and preemption-safe locks [63].

Specialized Locks

Numerous specialized locks have been proposed that improve efficiency for either read-mostly or write-mostly data. Reader-writer lock (RWLock) improves read-mostly data synchronization by allowing multiple readers to read the shared data in parallel [48]. Each reader is required to acquire a reader lock but this lock may be shared with other readers. On the other hand, the writer lock is exclusive and excludes other writers and readers.

To increase parallelism and reduce communication overhead, the reader-writer lock can be combined with Mellor-Crummey et. al. 's localized spin lock's approach [59]. In this scheme, each interested thread adds itself to the RWLock list and spins on its local variable. When the thread releases the lock, it updates its neighbor's spin variable to indicate the lock is released.

Lock Free Approaches

Lock free data structure has been proposed to improve synchronization efficiency and to prevent deadlock situations. Instead of using locks, lock free structures use atomic instructions and careful ordering of instructions to maintain consistency [38, 82]. The challenge is to support all operations on the data structure without using locks, and for this, atomic instructions such as compare and swap are required. By definition, lock free structures also avoid priority inversion, where a lower-priority process is preempted while holding a lock needed by a higher priority process.

A distinction is made between "lock-free" structures and "wait-free" structures. Lock-free structures only guarantee that some, but not necessary all, "processes will complete an operation in a finite number of steps" [35]. On the other hand, wait-free structures guarantee that all processes will "complete an operation in a finite number of steps" [36, 78].

In the operating system domain, Masslain et. al. was perhaps the first to use lock free structures to implement an entire operating system. The Synthesis system is based on lock-free quajects that must fit in one or two words [52]. These structures are updated atomically using Compare And Swap (CAS) instructions and Double Compare And Swap (DCAS) instructions. The drawback of this technique is that it limits data structure size to one or two words.

In the Cache Kernel operating system [22], Greenwald et. al. implemented the system

kernel and supporting libraries with non-blocking data structure through the use of Type-Stable Memory Management (TSM) [31]. Each non-blocking data structure is associated with a descriptor similar to a version number. To modify the shared structure, both the data and the descriptor has to be atomically updated successfully. The atomic update, implemented using DCAS instructions, fails if the descriptor is modified by another thread before committing the update.

Both Massalin et. al. 's and Greenwald et. al. 's method require the use of an atomic Double-Compare and Swap instruction. DCAS is hardware specific and not widely available. Although software techniques [16, 51] have been proposed to emulate DCAS functionality, these techniques are $O(N^3)$ or $O(NM^2)$, where N is the number of processes per processor that needs to synchronize and M is the size of the data [7].

Harris et. al. shows how a lock free linked list can be updated with only two compare-and-swap (CAS) instructions, instead of with a double compare and swap (DCAS) instruction [32]. Deletion is divided into two separate steps - logical deletion and physical deletion. Logically deleted elements are marked as deleted and are removed from structures they may be part of. However, they may continue to be accessed by concurrently running threads. Physical deletion is performed only after there are no longer any references to the logically deleted nodes. To determine that there are no pointers to the logically deleted elements, an exhaustive search through the list is done [33]. This technique is only applicable to the linked list data structure.

Michael et. al. points out that there are two challenges in using lock free data structure [61]. First, the system does not know when it is safe to assume the object can no longer be accessed by any thread, and hence when it is safe to perform memory reclamation. Second, many systems using lock free data structures suffer from the so called ABA problem [42, 81]. ABA problem refers to the fact that atomic primitives such as CAS do not necessarily guarantee that no other threads have modified the protected data. Let us assume that one

thread changes the shared data's value from A to B. It is possible that another thread may have changed the compared data back to the original value A and CAS do not realize protected data had been changed (hence the term ABA problem). Thus, programmers cannot solely rely on comparing lock free data with expected values in determining whether writes are safe [82].

Safe Memory Reclamation

The Safe Memory Reclamation (SMR) scheme proposed by Michael et. al. makes it easy to determine safe points for memory reclamation in lock free data structure [21, 61, 62]. Whenever a thread accesses a shared lock free object, it adds itself to a list of single writer multi-reader pointers, known as hazard pointers (HP). When a thread no longer accesses a shared object, it removes that shared object from its hazard pointer list. A shared object's memory can safely be reclaimed only if it is not on any thread's HP list. The disadvantage of this technique is that it is not efficient for read accesses [55] since SMR requires scanning of every thread's HP list. As discussed earlier, operating systems are typically demand driven and have a large number of threads. These threads are dynamically created and typically short-lived. Hence, scanning these threads incurs large overhead costs.

Transactional Memory

Transactional memory, first proposed by Herlihy et. al. , is a hardware extension of a multi-processor cache-coherence protocol that allows read-modify-write operations to be defined and customized [38]. Herlihy's scheme associates writes to multiple words of memory in the local cache into a single transaction. The transaction is committed permanently only if no other transactions have updated any location in the transaction's data set. Otherwise, the transaction is aborted and the writes are discarded. Although transactional memory is

similar to database transactions, Herlihy's transactions are designed to be short-lived. In addition, only a small number of memory locations are expected to be written within a single transaction. Transactional memory requires changes to the existing cache coherence protocol and hence appropriate hardware support.

Software transactional memory (STM), on the other hand, is designed for general purpose processors [37, 73]. Similar to Herlihy et. al. 's transaction memory, updates to multiple words of memory can be associated into a single transaction. To start a transaction, ownership of all accessed data must be acquired successfully by writing to the shared memory's ownership vector. Otherwise, the transaction's acquired ownerships are released and the transaction is discarded. The drawback of STM is that its performance is inferior to using locks [66]. Note that STM requires ownership for both read and write operations. Thus, in the case of having multiple readers and no writers, extensive synchronization is still required. Moreover, since ownership is shared by all processors, applications will suffer from high cache traffic [55].

2.1.2 Read-Copy Update

Read-Copy Update (RCU), proposed by Paul McKenney et. al. , is a reader-writer synchronization mechanism optimized for read mostly data [55, 56, 57]. Read operations, which can be concurrent, do not require any explicit synchronization. Therefore, this technique is ideal for read mostly workload because reads incur no overhead.

RCU works by requiring the writer first to make a copy of the data it wishes to modify. Subsequently, the writer modifies the copy and then constantly monitors whether it is safe to replace the previous data with the updated copy, a time known as quiescent point. Quiescent point (QP(t)) is the first point in time at which it is known that no other threads will be able to access the old data. Quiescent state is the system's condition at which it is known that

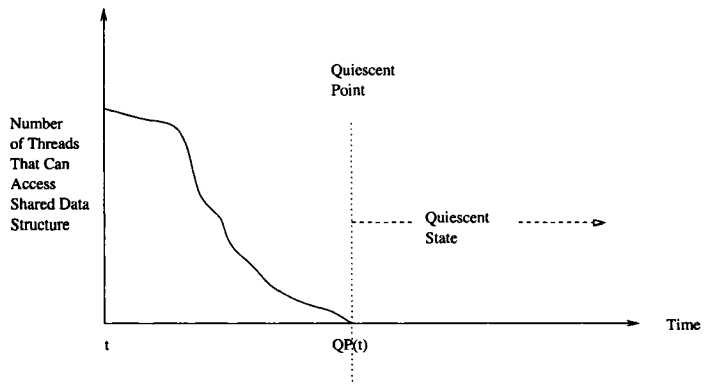

Figure 2.1: Relation Between Number of Threads, Quiescent Point and Quiescent State

no other threads will be able to access the old data. This state, as illustrated in Figure 2.1, may be either directly provided by the readers or indirectly observed by the system. For the direct case, read-side signals must be provided to indicate when the reader has completed. For the indirect case, quiescent state can be deduced by observing the state of participating threads.

Let us go through a simple example to see how RCU works. Assume that a thread wants to modify an element B from a lock free linked list, as seen in Figure 2.2. First, a copy of Element B (B-I) is created and updates are made on Element B-I, as seen in Figure 2.3. Next, the linked list is updated so that all new accesses will see Element B-I instead of Element B, as seen in Figure 2.4. This can be done with a single write in this case. At this time, there may still be threads that continue to access Element B, as depicted in Figure 2.5. Thus, even after Element B is removed from the list, the element's memory cannot be immediately reclaimed. Element B's memory can only be reclaimed when we know that no threads will reference this element, as seen in Figure 2.6. After quiescent state is reached, the element's memory can safely be reclaimed, as shown in Figure 2.7. Note that the modifying client reads, copies and then updates the copy, hence the term RCU.

Some methods, such as RCU, need to defer some actions, such as object reclamation,

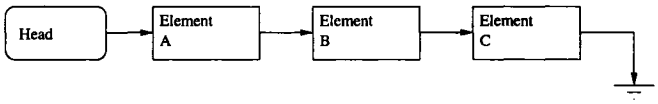

Figure 2.2: Linked List at Time 0

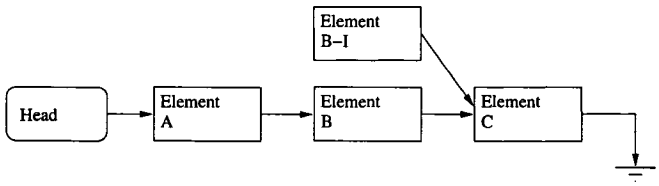

Figure 2.3: Create Copy of Element B-I

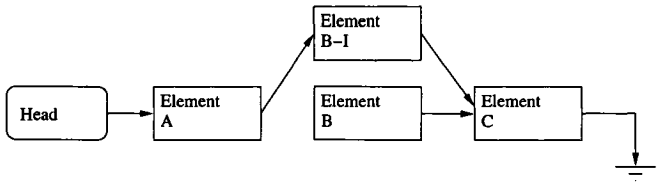

Figure 2.4: Linked List is Updated with Element B-I

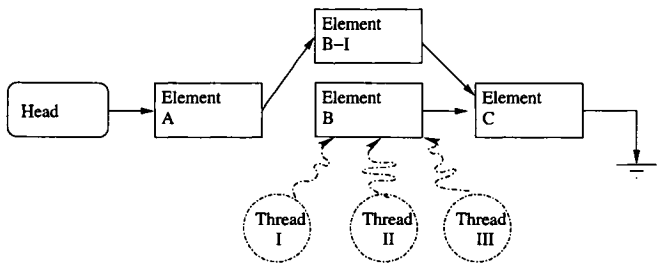

Figure 2.5: Threads Continue to Access Element B

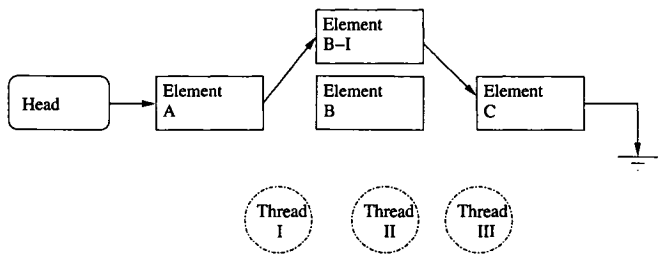

Figure 2.6: Quiescent State for Removing Element B

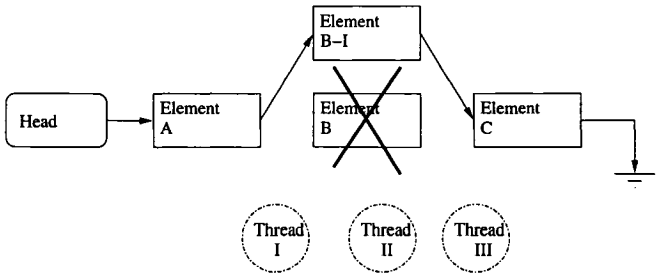

Figure 2.7: Memory Reclamation of Element B

to a later point when it is safe to do so. We generally refer to the execution of these actions as callbacks. In this case, callbacks are method calls invoked when quiescent state is established. In the previous example, Element B's memory is reclaimed when such a callback occurs.

Numerous RCU implementations have been been proposed for the Linux 2.6 operating system [14, 68, 69, 70]. These implementations differentiate themselves primary along two design issues: i) how they detect quiescent state on each processor and ii) how the processors communicate amongst each other to establish global multiprocessor quiescent state. For example, some of the implementations that have been proposed are:

1. RCU-Poll:

   Local Quiescent Detection: A counter (quiescent counter) is used to record when idle-loop and context switch has occurred in a non-preemptive kernel. Local quiescent is established when this counter has increased.

   Global Communication: Each processor regularly polls other processor's quiescent counter. Global quiescent is established when each processor's counter has been incremented.

2. RCU-Taskq:

Local Quiescent Detection: Local quiescent is established when a per-CPU kernel daemon schedules itself. The scheduling of kernel daemon forces context switch to occur in a non-preemptive kernel.

Global Communication: RCU facility forces other processors to schedule their RCU kernel daemon to be executed. The originating kernel then waits until all processors have executed the RCU kernel daemon. Note that quiescent state is *forced* instead of *observed* from other processors.

3. RCU-Preempt:

Local Quiescent Detection: A per-CPU counter (preemption counter) is used to keep track of the number of preempted tasks. Local quiescent is established when each tasks that was running or preempted at time t exited or voluntarily switched context.

Global Communication: Global quiescent is established when each CPU's number of running/preempted tasks reaches zero. This is observed by reading other processor's preemption counter.

2.2 K42

K42, a descendant of the Tornado operating system [8, 27, 28], is a research operating system that focuses on performance and scalability for large scale shared memory NUMA multiprocessor [9, 40, 79]. It is being developed by IBM Watson Research with collaboration from the University of Toronto and other universities.

K42's basic structure is depicted in Figure 2.8 [10]. Similar to Mach[1] and L4[34], K42 uses a micro-kernel architecture rather than a traditional monolithic kernel. The kernel

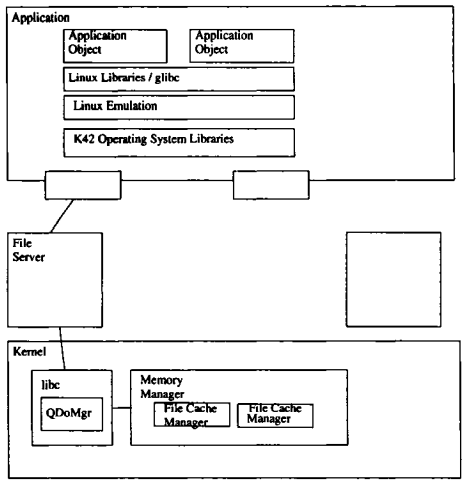

Figure 2.8: K42 Basic Structure

only provides basic support such as memory management, process management, interprocess communication (IPC) infrastructure and base scheduling. The libc library facility is available to both the user (application) and kernel layers. Additional functionality, such as file server, name server and socket server, are implemented as user level servers. Each server has an exclusive virtual address space. Communication with the micro-kernel and other servers is through IPC mechanism [10, 11, 28].

To achieve good scalability, K42 uses an object oriented approach. Virtual and physical resources are represented by objects, which allows locality to be maximized. In addition, only local locks are used to protect the relevant object and no global locks are used. Thus, centralized bottlenecks are avoided and lock contention is minimized [40, 79].

2.2.1 Clustered Object

An important structure used in K42 is the clustered object. Clustered object is a partitioned object model that allows an object to be decomposed into multiple representation objects

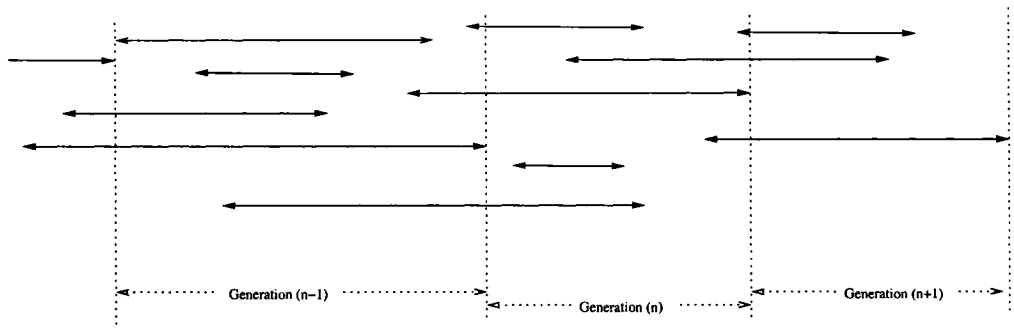

Figure 2.9: Generation Definition for K42

that have a single unified external interface [8]. Decomposing into multiple representation objects allows easy construction of objects that are partioned, distributed or replicated across different virtual processors (VPs). Thus, unrelated requests to different resources can be processed independently because resources are not shared across different VPs.

Using Clustered Objects, K42 takes locality to the extreme. For example, n threads of the same process independently but concurrently faulting on pages of a common mapped-in file causes no access to any shared data in the kernel, and causes no accesses to shared locks [9].

2.2.2 Generation Count

K42 uses an event-driven model for its micro-kernel and its servers. Threads, which can be created with low overhead, are created to service events when a system call is invoked or when an interrupt occurs. The threads are terminated when the servicing of the event completes. Thus, threads within the micro-kernel and server layers are by design short lived. K42 groups threads that are created at approximately the same time into epochs, also known as generations [28, 40]. Figure 2.9 illustrates the notion of generations for K42. A generation is defined as the time when the previous generation ended to the time when the last thread associated with the previous generation is terminated.

K42 keeps track of the number of active threads in an address space with a reference counter, on a per virtual processor basis. When a thread is created, it is assigned to the current generation and the reference count for the current generation is incremented. When a thread is terminated, the terminated thread's assigned generation reference count is decremented. If the generation's reference count reaches zero, a new generation is started and the current generation number is incremented. This design requires K42 to keep track of both the previous generation's reference count and the current generation's reference count for each virtual processor.

The clustered object infrastructure uses the generation count facility for its RCU-like garbage collection. When an object is to be deleted, it is marked as such, but memory reclamation is deferred for the clean up daemon. The clustered object manager (COSMgrObject) creates one cleanup daemon thread for each VP. The cleanup daemon thread sleeps for 2 milliseconds and can be woken up earlier if the system is low on memory. Once woken up, the daemon thread checks to see if any clustered object's memory is safe to be reclaimed. Reclamation is deemed to be safe when both its temporary and persistent references have been removed. Temporary references are clustered object references that are held privately by a single thread. Persistent references are clustered object references that are stored in shared memory and can be accessed by multiple threads. Practically speaking, for a uniprocessor, an object's memory can be reclaimed when two local generations have passed after it is logically deleted.

For the multiprocessor case, a token scheme, is used to establish safe points for memory reclamation. The token scheme is illustrated in Figure 2.10. A virtual processor hands off its token to the next virtual processor only when its cleanup daemon thread detects that two generations have passed since the virtual processor receives the token. Thus, we can establish that all temporary references are removed when the token had visited each VPs once.

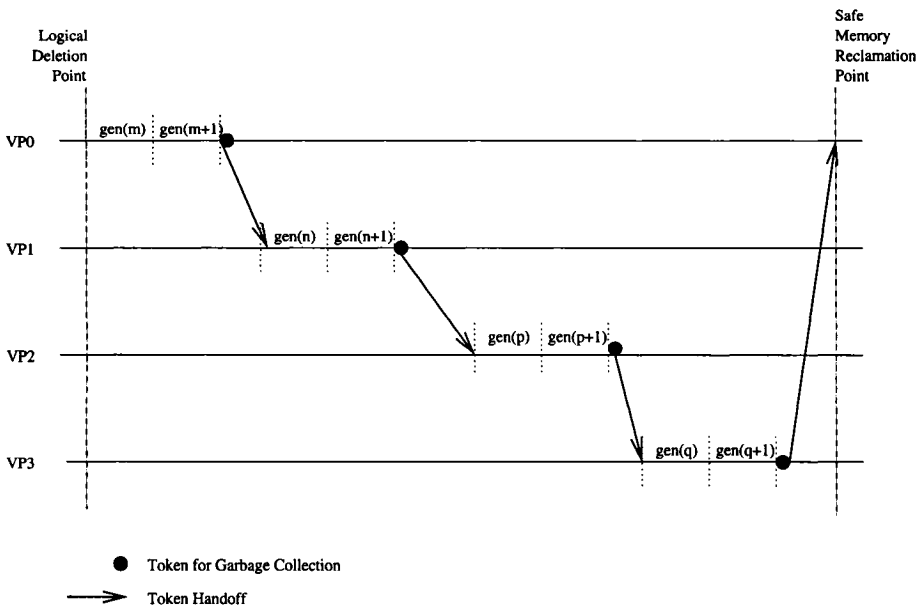

Figure 2.10: Clustered Object Garbage Collection for Multiprocessor

Chapter 3

Design of QDo Facility

In this chapter, we present our design for the QDo facility and the rational behind it. Section 3.1 presents high-level design decisions we made. Section 3.2 discusses how new QDo requests are registered. Section 3.3 describes how quiescent states are detected. Section 3.4 presents the design decisions made in executing the callbacks.

Chapter 4 will describe the implementation of the QDo facility. Chapter 5 will provide the experimental results.

3.1 High Level Design

Three high level design decisions heavily influence the overall design of the QDo facility:

1. the functional decomposition of the facility;

2. per-address space handling of QDo requests; and

3. supporting processor subsets for quiescent state establishment.

Each of these design decisions are discussed in this section.

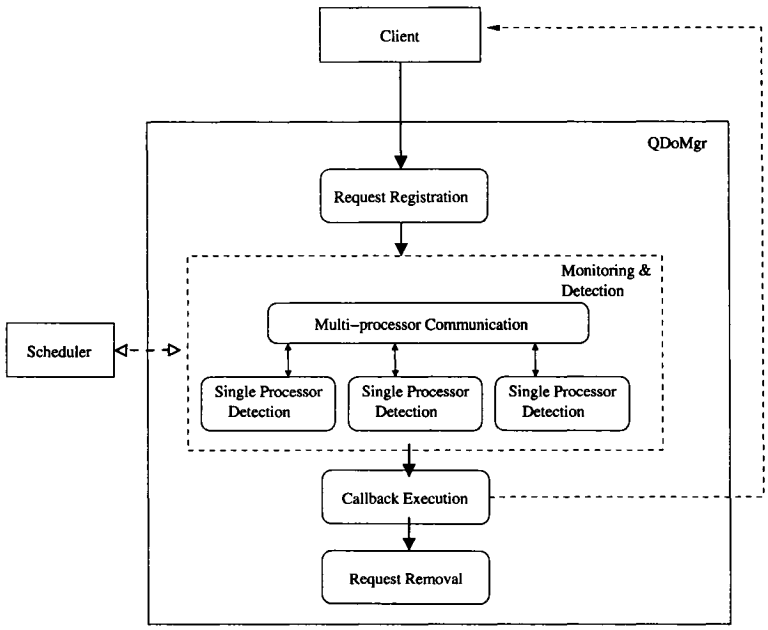

Figure 3.1: Life Cycle of QDo Processing

3.1.1 Functional Decomposition

In order to understand the functional decomposition of the QDo facility, let us walk through the life cycle of a QDo request in K42, as illustrated in Figure 3.1. First, a client places its request for a QDo callback by invoking a function at the QDo management system (QDoMgr). When the QDoMgr function is invoked, the 'Request Registration' component records the interest. The 'Monitoring & Detection' component continuously monitors all outstanding QDo requests and detects when quiescent state is reached for each of the requests. Once quiescent state for a request is reached, the associated request is forwarded to the 'Callback Execution' component, which schedules a callback for execution and the corresponding request is deleted from the Registration component. After the callback is executed, the request is deleted from the QDoMgr system.

Within the 'Monitoring & Detection' component, there are two sub-components, namely 'Single Processor Detection' and 'Multi-processor Communication'. 'Single Processor De-

tection' is responsible for detecting whether quiescent state is reached within the current processor and 'Multi-processor Communication' is responsible for communicating state information between different processors. We will further explore these sub-components in subsequent sections.

3.1.2 Per Address Space QDoMgr

In our design, QDo requests are processed on a per address space basis. In other words, each application in the user space has its own QDoMgr, and QDo requests from different applications are handled separately. The QDoMgr in the kernel spaces will only monitor for QDo requests from the kernel. This decision was made for a number of reasons.

Firstly, having a per address space QDoMgr, allows for specialization. The current implementation is designed primarily for demand-driven servers, where threads are relatively short-lived (to process a client request), and hence quiescent state is based on generations. This may not be suitable for all applications, and through specialization it may be possible to implement alternative means of identifying quiescent state. For example, in applications with long running threads, it may be advantageous to have the client identify the critical sections or mark its known safe points.

Secondly, K42 addresses security in a first class manner. Objects in different address spaces communicate via an Interprocess Communication (IPC) infrastructure, and processes from different address spaces do not share data directly [11]. Hence, we only need to be concerned with threads within the same address space in establishing quiescent state.

Thirdly, executing callbacks to different address spaces brings in unnecessary complexity. In particular, QDoMgr would have to keep track of the state of different address spaces.

Finally, K42 already provides the *ActiveThrdCnt* facility, that is part of the user level scheduler, and counts threads' generation on a per address space basis [28]. Managing QDo

on a per address space basis allows us to reuse this existing infrastructure without major changes.

The drawback of processing QDo requests on a per address space basis is that the total overhead of the monitoring and detection component is proportional to the number of address spaces, since each space has to manage its own QDoMgr and performs regular checks for quiescent state.

3.1.3 Processor Subset Support

The final high-level design decision we made was to support processor subsets when establishing quiescent state. That is, clients are allowed to specify the set of virtual processors on which quiescent state has to be achieved before the callback is executed. This is different from the Linux 2.4 and 2.6 implementations, where quiescent state must be established system wide (i.e., on all processors) before callback is initiated [68, 69, 70, 71]. K42's clustered object system allows clustered objects (CO) to be created with representatives instantiated on just a subset of virtual processors [8, 13]. Requiring that quiescent state be established globally would be inefficient, as a significant number of COs only ever instantiate one or a small number of representatives.

3.2 Request Registration

The 'Registration' component is responsible for accepting QDo requests and recording them within the QDoMgr system.

```
SysStatus QDoMgr::QDo((void *) callback,
                      uval64 parameter,
                      VPSet quiescenceSet,
                      VPSet executeSet);
```

Figure 3.2: QDo's registration interface

3.2.1 Interface

In designing the QDo interface, we settled on making a single function, QDo(), available to clients. QDo's registration interface is shown in Figure 3.2. Two obvious parameters are the callback function to be executed when quiescent state is reached and a single argument for the callback function. In the figure, the parameter *callback* is a subroutine pointer to the callback function. The callback parameter, specified as a 64 bit integer, is defined as *parameter*. Furthermore, *quiescenceSet* is the set of processors on which quiescent state must be reached before a callback can occur. Finally, through the *executeSet* parameter, the QDo client can specify the set of processors on which the callback function should be executed on when quiescent state is reached. Application sometimes may require a callback to be executed on only one processor, as in the case for memory reclamation, or they may require a callback to be executed on every processor, as in the case for hot-swapping [77].

If the registration is successful, the method's return code is 0 to indicate success.

3.2.2 Example Usage

We will go through a simple example to demonstrate how QDo is called. In this example, shown in Figure 3.3, a lock free linked list needs to delete a specified element. This is implemented in the method *deleteElement*. Once quiescent is established, QDoMgr will execute the specified callback method *callbackDeleteElement* to free the element's memory. We do not know how many other virtual processors may have access to the deleted

```
void LockFreeLinkedList::callbackDeleteElement(uval data) {
        /* callback when quiescent state is established */
        delete (Element *) data;
}

void LockFreeLinkedList::deleteElement(Element* element) {
        /* logically delete the specified element */

        /* Here, we will search what are the left and right
       nodes with respect to the element */
        do {
          left_node = search(element, right_node);
        } while CompareAndStore(left_node->right,
                        element, right_node);

        /* Find out the quiescent set and execution set */
        VPSet quiescentSet;
        quiescentSet.addVP(Scheduler::GetAllVP());
        VPSet executionSet;
        executionSet.addVP(Scheduler::GetVP());

        /* schedule a callback to physically delete the
                element when quiescent state is reached */
        DREFGOBJ(QDoMgr)->QDo(&callbackDeleteElement,
                        (uval) element,
                        quiescentSet, executionSet);
}
```

Figure 3.3: Example of how QDo might be called by a lock-free linked list object

element. Thus, quiescent state has to be established across all virtual processors. On the other hand, the callback only needs to be called on one processor - a good candidate is the current virtual processor.

3.2.3 Batching

Another request registration design issue is whether to support batching of QDo requests or to only support individual requests. Batching is currently implemented in a majority of the

Linux's RCU implementation [56]. It involves grouping QDo requests that are registered in approximately the same time together so that a single quiescent state will launch multiple callbacks. The benefit of batching is that detection and monitoring costs (which can be significant) is shared among multiple requests. However, batching does not allow users to have the flexibility of specifying the virtual processor subsets when establishing quiescent state.

3.3 Monitoring & Detection of Quiescent State

The 'Monitoring & Detection' component is responsible for establishing when quiescent state is reached for outstanding QDo requests. Requests are handed off to the 'Callback Execution' component when callbacks are safe to execute. To simply our design, we divide this part into two subcomponents, namely *single processor detection* and *multiprocessor communication*. The *single processor detection* subcomponent detects when quiescent state is reached locally with respect to a particular VP. In order to achieve quiescent state on multiple processors, it is necessary to aggregate quiescent state information from all the processors of the set for which global quiescent state must be reached. For this, *multiprocessor communication* is necessary. This subcomponent is concerned with coordinating the results from single processor detection in forming a consensus of when quiescent state is reached among all relevant VPs.

3.3.1 Single Processor Detection

Detecting Local Quiescent State

The design chosen for QDoMgr is based on generation counts, as described in Section 2.2.2. Local quiescent state is established for a request when the local generation is incremented

at least twice after the request was registered with the QDo facility. At that time, all existing threads that had existed before registration will no longer exist. We selected this approach because of its advantages. First, checking of quiescent state occurs naturally as part of each thread's destruction process – when a thread is destroyed, the thread destructor checks whether the number of active threads from previous epoch reaches zero. Leveraging this facility reduces checking overhead. Second, this method is not dependent on programmer discipline. A programmer's failure to declare an object as shared would not affect proper establishment of quiescent state. The drawback is that this design can result in longer callback latency. Quiescent state cannot be achieved until all threads that existed at time of registration have terminated, even if these threads do not access the shared object. In addition, this method is only suitable for servers that only have short-lived threads; long-lived threads would prevent quiescent state from being reached. For our environment – K42 – this choice is appropriate, because as discussed in Section 2.2.2, K42 is demand driven and its threads are designed to be short-lived.

There are many alternative designs in detecting and monitoring local quiescent state. One way is to use a reference counter to keep track of the number of threads currently accessing a shared object [50, 72]. Before a shared object is accessed, the accessing thread increments the object's associated reference counter. This counter is decremented by the thread when the object will no longer be accessed. When the counter reaches zero, we know that it is safe to launch a callback with respect to the target object. Using a reference counter would require client code to be changed so that the shared object can be monitored. Reference counters and semaphores are similar in that both techniques provide access control by counting the number of client accessing the target object [75]. However, reference counters are intended to count threads accessing the shared object, while semaphores only count threads accessing a critical section.

The advantage of using reference counters is the short latency because a reference count

being decremented to zero is the theoretical definition of the associated quiescent point. On the other hand, using reference counters results in larger overhead because the counter has to be updated whenever a thread wishes to access the target object [84]. Furthermore, this technique relies on the discipline of the programmer: if a programmer forgets to increment the reference counter before accessing the shared object, quiescent state may be incorrectly established, leading to a race condition; similarly, we will never achieve quiescent state if a programmer forgets to decrement the reference counter after accessing the shared object.

Another way to establish quiescent state is to use a strategy based on hazard pointers, as suggested by Michael [62]. Before accessing a shared object, the accessing thread registers the node of the shared object onto its list of hazard pointers. Each thread will write to its own list and this list is accessible to all other threads. Threads waiting for an object's quiescent state will regularly scan all other thread hazard pointer lists. Quiescent state can be established when the shared object's node does not appear on any other thread hazard pointer list. Scanning for threads is more efficient than using reference counters, as threads do not experience write contention when registering the shared object. On the other hand, this technique still relies on the discipline of programmers. Furthermore, it is difficult to determine how frequently the threads list should be scanned. Large overhead arises when scanning occurs too frequently while infrequent scanning results in larger latency.

For non-preemptable operating system kernels, we can also detect quiescent state by noticing when the operating system reaches a known safe point. This is similar to how RCU is implemented in Linux [56, 57]. This design does not rely on programmer discipline. Examples of safe points in a non-preemptable kernel are idle loop and context switches. Idle loop is a known safe point because we know that there are no active kernel threads. Context switches are also a known safe point because kernel threads are prohibited from holding locks across a context switch. Therefore, they are also prohibited from holding pointers to sensitive data structure [57].

Idle processor

One problem we face when using generation count to establish quiescent state is that it fails to detect quiescent state if the processor becomes idle - in that case, the generation count is not increased. There are a few ways to address this issue.

The solution chosen for QDoMgr is to use a soft timer to regularly schedule QDoMgr. If a generation change has not occurred within a certain period of time, then QDoMgr would invoke the scheduler to see whether the number of active threads in the current generation is zero [1]. If this is the case, the generation count is increased. This design incurs a small amount of overhead, especially when the processor is not idle. There are two drawbacks to the soft timer scheme. First, if there are long running threads which prevent generation changes from happening within the scheduled period, then the timer may fire off even when local quiescent state is not reached. We do not envision this to be a problem because K42 is not designed for long running threads. Second, the time taken to establish quiescent state is longer than it needs to be when system is idle. In this case, the latency is determined by the check frequency. Future work will have to investigate how this particular case can be further optimized.

An alternative way is to provide hooks within the idle loop that prompts the QDoMgr to check for outstanding requests. At this point, no useful work is being done and QDoMgr can check the scheduler's blocked thread queues. If there are no blocked threads, QDoMgr can conclude quiescent state is reached. However, K42 uses a two level scheduler [12] and different address spaces have a separate QDoMgr. It is unclear whether hooks to multiple QDoMgr will cause significant overhead for the scheduler.

Another solution is to run a background loop thread within the same address space. The

[1] We have already discounted the scheduler/daemon thread when counting the number of active threads in the current generation

thread would run with the lowest priority and thus only executes when there are no other threads to execute. This thread would activate and deactivate itself to trigger generation changes. However, the problem with this technique is that K42's scheduler is not fully implemented at this time, as it only supports 2 classes of priority - CPU bound and I/O bound. Therefore, we cannot declare a specific thread as having a lower priority. Running the background thread as I/O bound or CPU bound has high overhead because it competes for CPU resources with useful threads.

3.3.2 Multiprocessor Communication

The multiprocessor communication subcomponent is responsible for coordinating the individual processors that detected quiescent state locally so as to establish global quiescent state. In the next few paragraphs, we analyze some of the possibilities. Without loss of generality, but for the sake of simplicity, the analysis is done with the assumption of one virtual processor per processor and one core per processor. For this analysis, we assume there are a total of m requests distributed evenly across n processors and each request requires quiescent on an average of r processors. We have implemented all four possibilities in the K42 Operating System and we will analyze their performance in Chapter 5.

Single Token Ring

One possibility is to use a token ring, similar to the garbage collection scheme used in K42, as discussed in Section 2.2.2. A token is passed continuously from one processor to the next. Global quiescent state is established when the token has passed through every processor at least once. The token is handed off to the next processor when the current processor reaches local quiescent state relative to the time the token was received. There are three advantages with this technique. First, communication overhead cost is in the order

of O(n), where n is the number of processors. No matter how many outstanding requests the system has, only 1 exchange per processor is required. Explicit synchronization between different processors is not required. Second, this design has small space overhead. Token passing requires 1 byte per processor to store the token, and 1 byte per request, to store whether the specific request had seen the token. Thus, space overhead is on the order of O(n)+O(m).

On the other hand, token passing incurs large latencies. Since the token has to pass through every processor and local quiescent establishment is not done in parallel, latency is on the order of O(n). In addition, this design is not flexible. Even when the client has specified that it is only interested in local quiescence on two processors, it is still necessary to wait until all processors have serially established local quiescence.

Multiple Token Rings

From above paragraphs, we know that a single token ring does not allow for efficient processor subset processing. One possibility to overcome this shortcoming is to have multiple different specialized token rings, as illustrated in Figure 3.4 [58, 65]. Each request has an associated token and the token is only passed serially through the specified processors. Similar to the single token ring case, processors only hand off a token when local quiescence is established. Each processor may hold multiple tokens at the same time. Global quiescence is established when all specified processors for that particular request have reached local quiescence.

As can be seen, this arrangement is flexible and allows clients to specify what processor to wait on. An unresponsive processor will not affect requests that do not have the processor in its quiescenceSet. In addition, assuming each request requires an average of r processors, the average latency is O(r). If r is much smaller than n, then the latency for multiple token rings is much smaller than the single token case. As discussed in Section 3.1.3, many

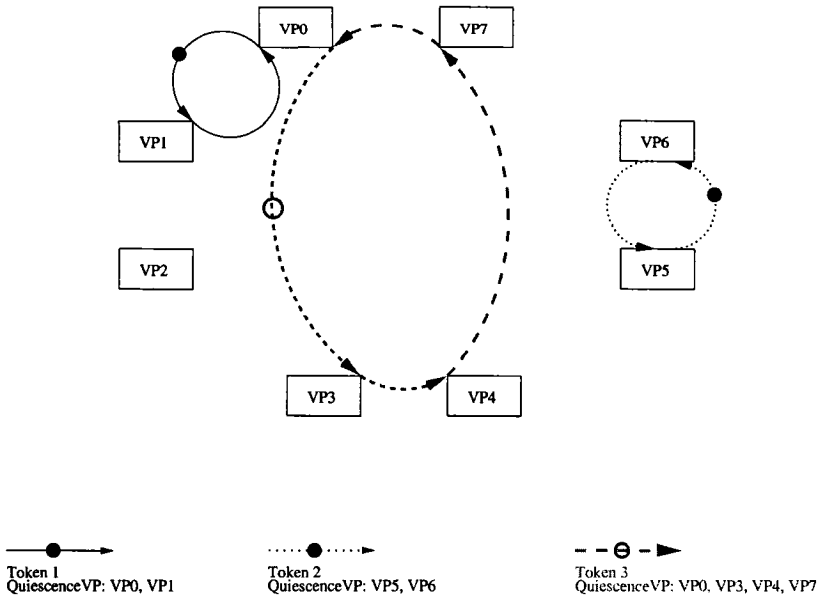

Figure 3.4: Multiple Token Rings

applications in K42 only needs to wait a subset of all processors due to the clustered object design.

The advantages of this scheme comes with higher costs, however. There are m token rings and each token ring requires communication with r virtual processors. Hence, the average communication cost is O(mr). In addition, when each request is specified to be quiescent on every processor (i.e., r equals n), communication costs increase to O(mn) and latency is O(n). This design requires space overhead in the order of O(mr). Since the number of requests a processor has to process is not known, token storage has to be allocated dynamically. Further, explicit synchronization techniques are required for coordinating the passing of tokens. As tokens are dynamically allocated in lists and multiple processors can write to the same list concurrently, we need to prevent race conditions by either using locks or having lock free lists.

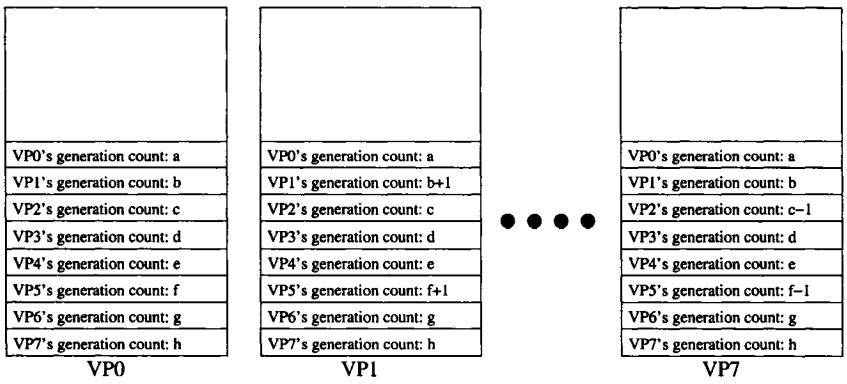

Figure 3.5: Snapshot Multiprocess Communication Method

Snapshot

Snapshot is another way to communicate between multiple processors, as shown in Figure 3.5. Each processor regularly takes snapshots of other processors' generation counts (A convenient time is when generation change occurs). Global quiescence for a particular request is established when each relevant processor's local generation number has been incremented at least two times.

Consider the following example. Let us assume that we have an eight processors system. We have a QDo request that is interested in VP2, VP3, VP4 and VP5. First, the request for a callback is registered on VP2 and QDoMgr enqueues the request. After that, the request will wait until the next time snapshot is taken. QDoMgr will then store this snapshot with the QDo request, as illustrated in Figure 3.6. Callback can be executed when the generation count for VP2, VP3, VP4 and VP5 has been incremented by at least two, as seen in Figure 3.7.

The snapshot method is flexible in that we will only wait for the processors we are interested in. Unrelated processors have little effect on the request latency. In addition, snapshot's latency is mostly on the order of O(1) because each processor is waited on in parallel. The worst case latency is slight larger than latency of the worst processor. Since

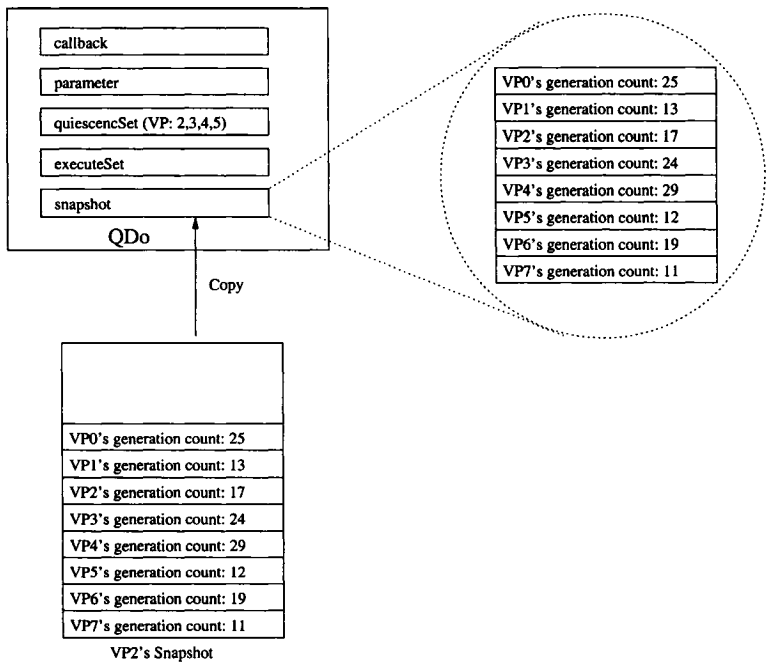

Figure 3.6: Snapshot Stored in QDo

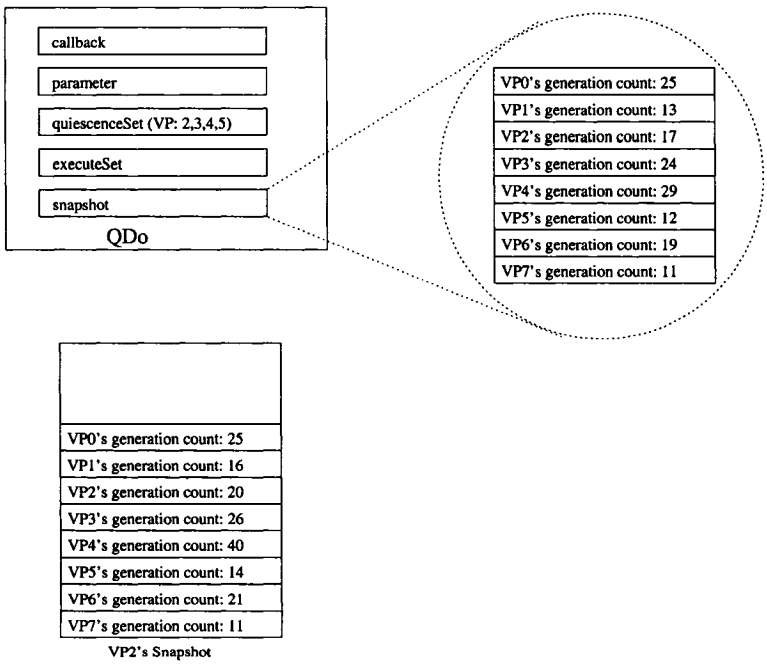

Figure 3.7: Callback is Safe to be Executed

the maximum number of processors is known, snapshots can be stored in a simple static data structure such as an array. Further, no matter how many outstanding requests the system has, only 1 exchange per processor is required per virtual processor pair. Hence, synchronization is not required when communicating between processors.

The drawback of this scheme is that our communication costs are increased to $O(n^2)$. Each processor has to periodically communicate with each other processor to obtain its generation count. Although we expect the overhead associated with copying one word of memory per processor to be small, the number might add up and affect scalability. Second, the storage overhead is larger than with the other techniques. Every processor has to store a snapshot of all other processors' state. In addition, each outstanding request has to store a copy of the snapshot. Therefore, the storage overhead is in the order of $O(n^2)$ + O(mn).

Diffracting Tree

A diffracting tree-like structure can be used to communicate quiescent state information among multiple processors [53, 74, 87]. In this scheme, each virtual processor is represented by a leaf node in a binary diffracting tree. Each leaf node publishes its own quiescent state information and makes this information available to its parent node. Each non-leaf (parent) node, in turn, recursively aggregates the quiescent information from its children and makes the information available to its parent. Once the root node notices that all virtual processors have achieved local quiescent state, it increments the global generation count. All virtual processors will then reset their quiescent state information.

For an example, let us assume we have a system with eight processors, as shown in Figure 3.8. A QDo request has been registered on VP3. The first thing we want to do is to wait until the global generation number increases. After the global generation number increases, VP2 checks to see whether it achieves local quiescent state. If so, it polls whether VP3 achieves local quiescent, as illustrated in Figure 3.9. At the same time, once VP0 and

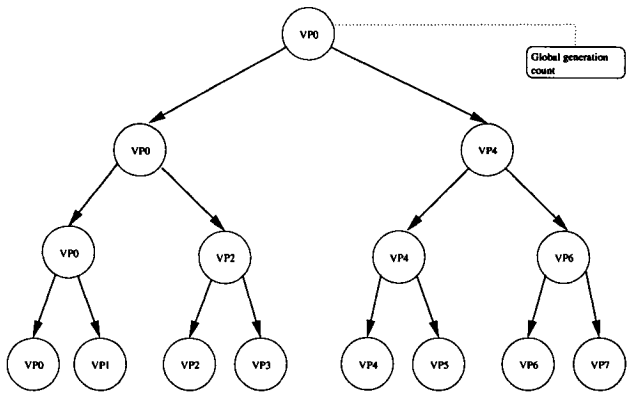

Figure 3.8: Diffracting Tree

VP1 achieve local quiescence, VP0 polls VP2 for the state of VP2 and VP3, as shown in Figure 3.10. Finally, once VP0, VP1, VP2 and VP3 reach local quiescence, VP0 polls VP4 for the state of VP4, VP5, VP6 and VP7. This is illustrated in Figure 3.11. VP0 increments the global generation number when all virtual processors reaches local quiescence. After the global generation number increments, global quiescence is declared and callbacks can be executed.

The latency of quiescence communication is on the order of O(log n). This is because most of the communication can be done in parallel. In addition, since information on other processors' state is aggregated, we do not have to have multiple copies of other processors' state for each VP. Instead, each processor only needs to have 1 copy of other processors' state and each request only needs to allocate 1 additional word to store the global generation number. Thus, the memory overhead is O(n)+O(m). Furthermore, communication overhead is limited to order of O(n) because each processor only communicates with 1 other processor at a time. An additional benefit is that this technique does not require explicit synchronization. The maximum number of processors is known in advance and only 1 word is exchanged at a time.

The drawback of this technique is that we do not have the ability to customize the

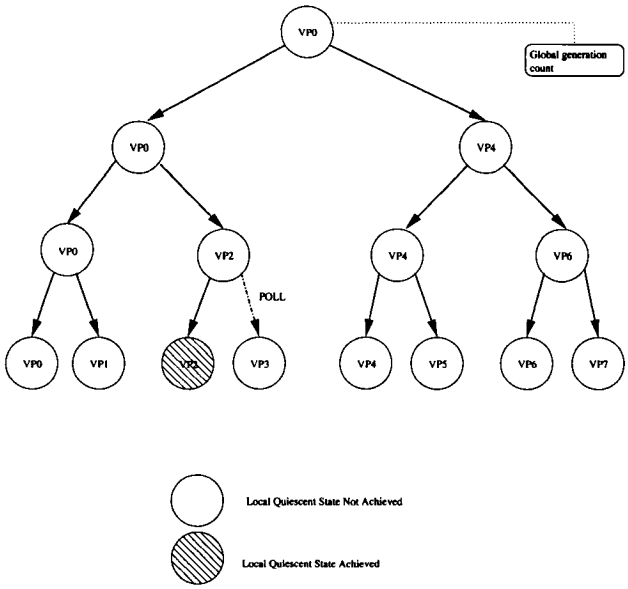

Figure 3.9: VP2 Achieved Local Quiescence

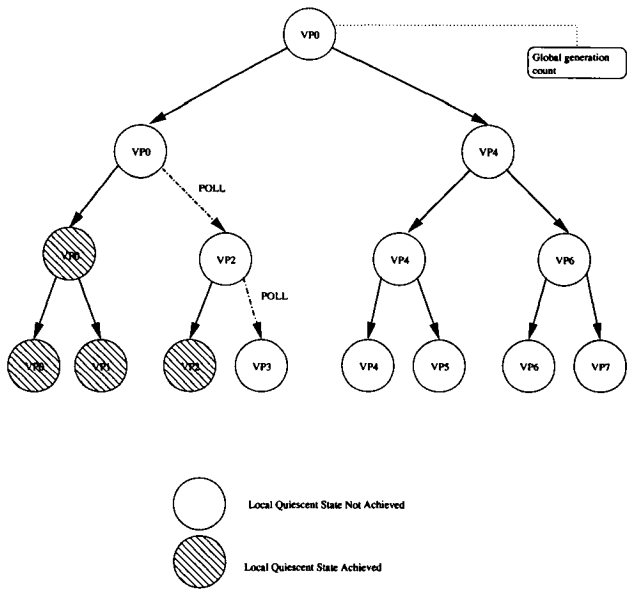

Figure 3.10: VP0, VP1 and VP2 Achieved Local Quiescence

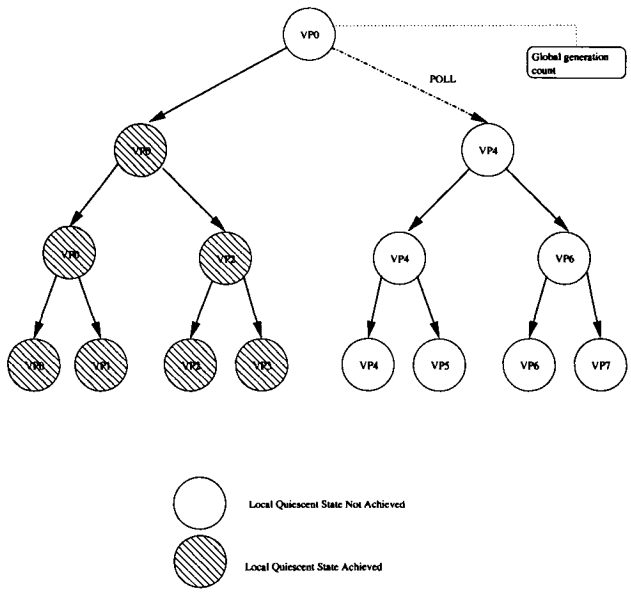

Figure 3.11: VP0, VP1, VP2 and VP3 Achieved Local Quiescence

processors subset we are interested in for each request. We need to wait for local quiescence even for processors that have no bearing on the shared object. This contradicts one of the requirements made in Section 3.2.1. In addition, communication overhead is not distributed evenly across all processors. In our example, VP0 suffers higher overhead because it needs to communicate with VP1, VP2 and VP4 in establishing global quiescence.

Summary

We have summarized our design trade-offs in Table 3.3.2. As it can be seen, different designs have different trade-offs. Snapshot has the lowest latency and allows flexibility in specifying which processors to wait on. However, it also has the highest space overhead of all the possible choices. On the other hand, the diffracting tree has a low communication and space overhead while it has a latency of O(log n). The drawback is that the diffracting tree is not flexible in customizing the quiescence communication route.

Thus, if latency is an important constraint, then snapshot is the best option. However,

Design	Latency	Communication Overhead	Space Overhead	Require Explicit Synchronization	Customize Processors to Wait On
Single Token Ring	O(n)	O(n)	O(n)+O(m)	No	No
Multiple Token Ring	O(r)	O(mr)	O(mr)	Yes	Yes
Snapshot	O(1)	$O(n^2)$	$O(n^2)$+O(mn)	No	Yes
Diffracting Tree	O(log n)	O(n)	O(n)+O(m)	No	No

Table 3.1: Trade-offs for Multiprocess Communication Techniques

if we need to minimize overhead, then a diffracting tree should be used. Single token is an inferior option when compared to diffracting tree because of its larger latency with the same order of magnitude of overhead. Multiple token rings are feasible only when the expected number of request is much lower than the number of processors. Otherwise, the cost of synchronization out-weights all potential benefits.

3.4 Callback Execution

The 'Callback Execution' component is responsible for launching the callback after the 'Monitoring & Detection' component declares that quiescent state has been reached. After the callbacks are scheduled for execution, requests are removed from the QDoMgr system. One decision we need to consider is whether the callbacks should be executed as separate threads.

After careful analysis, we decided that callbacks are executed as separate threads. Although the overhead of executing callbacks in separate threads may be relatively expensive when the callback is simple. This overhead is mitigated by K42's design decision of having low overhead for thread creation. Furthermore, even if a callback causes the executing thread to die, other callbacks are not affected.

On the other hand, executing callbacks with the same thread lowers the overhead because the scheduler does not have to be involved. However, this technique is potentially unsafe because QDoMgr has no control over the integrity of the callback or the quality of the callback code. Callbacks can encounter programming errors that cause this one thread to crash so that later requests may not have their callback executed. In addition, if a callback runs for a long time, later callbacks are delayed because the scheduler cannot preempt execution with tasks scheduled in the same thread.

Chapter 4

Implementation

4.1 Overview

QDoMgr is fully implemented in the K42 Operating System as part of the scheduler module. It consists of about 3000 lines of C++ code. In addition, about 40 lines of existing K42 code were modified, mainly to initialize QDoMgr during address space creation.

4.2 High Level Description

At a high level, QDoMgr operates as follows: When a caller registers a new QDo request with QDoMgr, QDoMgr enqueues this new request into a local unordered linked list. In addition, various system states (such as generation count) are recorded. As described in Section 3.3.1, QDoMgr continuously monitors these requests until local quiescent state is established. This is accomplished by comparing the current local generation number with the recorded generation number. Then, these QDo requests will be moved to a separate unordered local list where they await global quiescence, as discussed in Section 3.3.2. When global quiescent state is established for a particular QDo request, the scheduler schedules

the QDo's callback method for execution. After the scheduler has scheduled the callback, its associated QDo is removed from the QDoMgr's lists and is deleted.

There are a few particularly interesting and non-trivial implementation issues, such as how QDoMgr should be structured and how to initialize QDoMgr. These issues are discussed in this chapter.

4.3 Clustered Object

One particular interesting implementation issue is how QDoMgr should be structured. QDoMgr's structural efficiency has a major impact on the scalability of managing QDo requests. To maximize potential concurrency and minimize overhead, we want to maximize the locality of individual object members as much as possible and minimize the amount of sharing. Therefore, we have implemented QDoMgr as a clustered object. Clustered object allows us to divide objects into shared and non-shared implementations. Shared members are implemented within the *Root Object*. Non-shared members are implemented within *Representative Object* with one representative local to each virtual processor.

QDoMgr is implemented as a *well known clustered object* in the libc library. *Well known clustered objects* are clustered objects that are statically allocated instead of dynamically allocated [9]. By implementing QDoMgr as a well known clustered object, we know that there is only one copy of QDoMgr running per VP per address space. This knowledge allows us to make a few assumptions in simplifying the synchronization and callback mechanics. For example, we can assume that once *ActiveThrdCnt* detects that there are no active threads from the previous generation, and the generation number can be incremented, we will know which QDoMgr to notify.

QDoMgr is implemented in libc because one of our design goals, as explained in Section 1.3, is to make the QDo facility available to both user and kernel spaces. Implementing

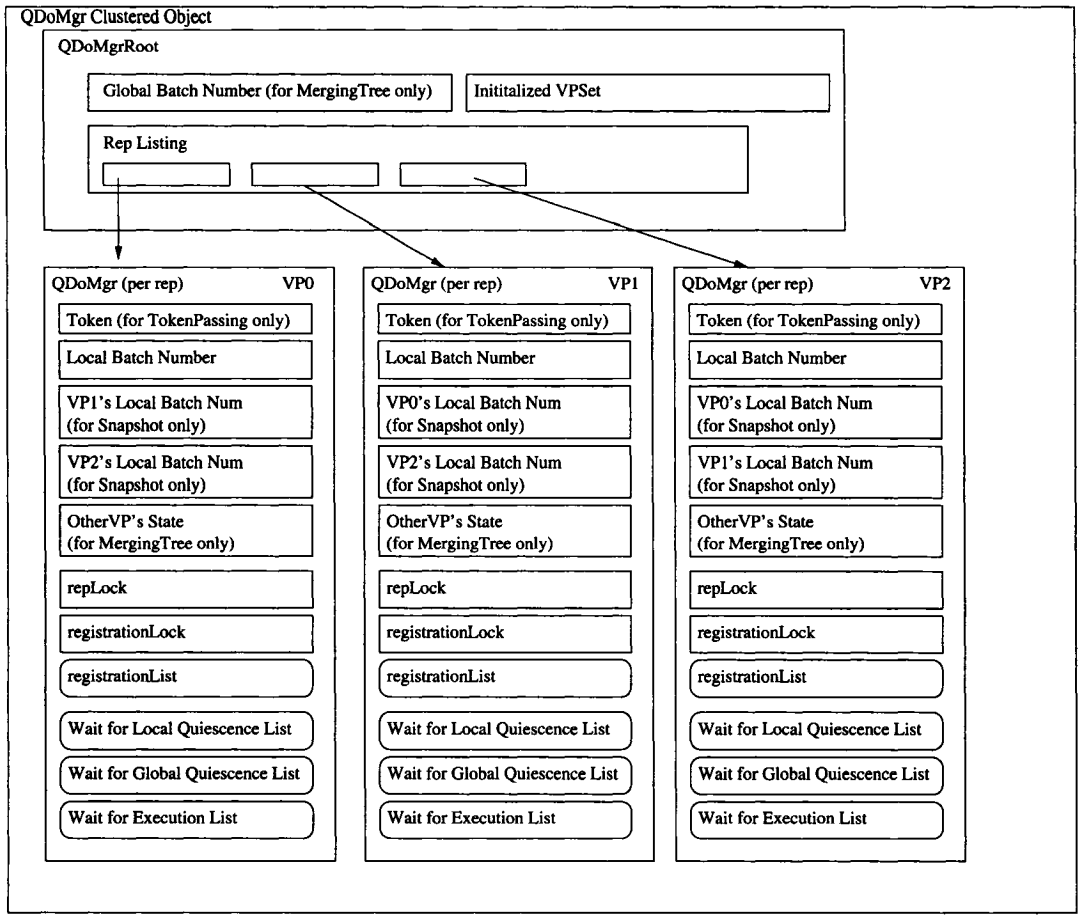

Figure 4.1: Major Data Structure Used in QDo

QDoMgr within the library allow us to use the same code for both the kernel and user space.

Some of the member structures used in implementing QDoMgr are illustrated in Figure 4.1. As it can be seen, only few member structures are shared across multiple virtual processors. The majority are contained within per virtual processor representatives.

4.3.1 Root

Only a small amount of the object data is implemented within the Root Object. First, the variable *initializedVPSet* records on which VPs the QDoMgr has been initialized. This vari-

able is updated at most once per virtual processor. Hence, we do not need to be concerned with the overhead of sharing this variable, even though it needs to be protected by a lock. Another shared variable in the root is the global batch number needed for the diffracting tree communication algorithm. This variable is shared across multiple virtual processors, but is updated by only 1 virtual processor, so, explicit synchronization is not required and the cost of sharing is low.

4.3.2 Representatives

We will now discuss some of the data implemented within Clustered Object Representatives. First, *token* is a variable used by the single token ring communication algorithm to denote whether a virtual processor currently holds the token. The variable is set by the previous virtual processor holding the token and is reset by the local virtual processor. No other VP accesses this variable. No explicit synchronization is needed, given the protocol of using this variable.

Another variable that is implemented in the Representatives is the local generation number. Recall from Section 3.3.1 that the local generation number is used to determine whether local quiescent state is reached. Each VP continuously updates its local generation number, but at the same time, other VPs periodically copy this local generation number and store it in their local memory. On the surface, it would appear that the local generation number should be implemented in the Root. However, it is desirable that each VP has the flexibility in deciding when to update its view of the other processors' local generation number. This allows us to reduce unnecessary cache-line invalidation.

The third data structure implemented within Representatives is a table of virtual processors' quiescent state. This table is used in the snapshot communication method. The table is only accessed and updated by the local VP. Each VP's table's entries are different.

Finally, QDoMgr has a number of unordered linked lists that are used to hold incomplete QDo requests [1]. These linked lists are designed for temporary storage and are only accessed by the local virtual processor. Some implementations require local locks. These locks are primary used in coordinating the queuing of new QDo requests. However, in the case of the multiple token ring communication algorithm, we also use these lock to coordinate the communication of tokens. We will further discuss the trade-offs in Section 4.8.1.

4.4 Initialization

Another interesting issue is how to start up QDoMgr – more specifically, when to initialize QDoMgr. At a first glance, we would expect that we need to initialize QDoMgr as soon as the address space is created to ensure we do not miss any threads created during process initialization. However, it turns out that it is sufficient to initialize QDoMgr before the scheduler enables the system call and IPC (Inter-process communication) entry points. The ActiveThrdCnt facility (which counts the number of threads in the current generation) is already initialized very early on (as a matter of fact, before context switch is enabled). Thus, all threads are accounted for when calculating the number of active threads in the current generation. In addition, applications using the QDoMgr facility are by definition designed for deferred method callbacks [55]. Thus, QDoMgr still functions correctly and its performance is not significantly affected if its initialization is delayed to until just before IPC and system call entry point is enabled.

[1] Incomplete QDo Requests refer to QDo requests of which their callbacks have not been scheduled to be executed. They may or may not have reached quiescent state.

4.5 Delay Checking

To improve performance, we also need to consider how to minimize the overhead when QDo service is not required. One technique we employ is to delay the regular checking of quiescent state until the first QDo request has been placed. This is achieved by only starting the checking daemon (which will be explained in Section 4.7) after the method QDoMgr::QDo() is called for the first time. This way, only programs that make use of QDoMgr's service have the overhead of quiescent state detection. The drawback is that once QDo requests have been placed in an address space, checking overhead will continue to be incurred. Determining how to further reduce the checking overhead is deferred for future work.

4.6 Fork

We also need to consider how QDoMgr should behave during program forks. In theory, after a process forks, the child process should get a copy of the stack, memory, open file descriptors, etc. from the parent process [75]. However, we do not want the child to have a copy of the parent's incomplete QDo requests. Instead, the child should be started with an empty QDo request list. The reason we do not want the child process to inherit incomplete QDo requests from its parent is that, after a process forks in K42, its child does not inherit the parent's threads (except the thread that executes fork). Therefore, the parent's active thread count (both for the current and previous generation) should have no bearing on the child's quiescent state. This implies that the parent's QDo request should be invalid with respect to the child's process.

There are a few solutions to address this issue. The solution we employ is to simply reset the child QDoMgr's incomplete callback queues as part of the child's post fork-

ing cleanup. While doing so, we will not encounter any race conditions because regular checking of incomplete requests has not begun. Furthermore, as the child process has not yet resumed execution, no new QDo requests can be placed on the child's QDoMgr. This is accomplished by adding a line in the child's post forking handle routine (ProgExec::ForkChildPhase2) to re-initialize QDoMgr's data structure.

One alternative solution is to delay forking until all QDo callbacks are launched. This solution has many problems. First, large latency is introduced as both the parent and child program has to be delayed. Second, it is unclear whether we need to block QDoMgr from accepting new requests as we wait for incomplete QDo callbacks to be completed. Otherwise, forking might be infinitely delayed.

4.7 Quiescent State Monitoring Daemon

In our implementation, we use a daemon to monitor system state to detect quiescent state for each outstanding request. The alternative is to schedule a new thread to check for quiescent state every time a callback request is registered. In deciding whether a daemon is the best option, we need to evaluate different trade-offs. First, using a daemon allows us to easily ensure that only 1 thread is monitoring quiescent state on each virtual processor. This assurance is important because it simplifies our intra virtual processor synchronization. We will further discuss this issue in Section 4.8.2.

Second, using a daemon allows us to reduce the possibility that quiescent states are triggered by the checking thread's termination. Let us assume that there are no active threads in the address space. If we schedule a new thread to check for quiescent state, then the new checking thread will become the only active thread in the address space. Once this new checking thread is terminated, the scheduler notices that the active thread count drops to zero. This triggers the scheduling of checks for outstanding callbacks. On the other

hand, using a daemon will not have this problem because the daemon thread is already discounted from the active thread count. It may appear that frequent checking is desirable as latency is reduced. However, unnecessary checking takes away the scheduler's quantum from other address spaces' processes.

4.8 Synchronization

Explicit synchronization is required for both intra-virtual processor and inter-virtual processor communication. Multiple threads on the same virtual processor can *concurrently* enqueue callback requests on the same QDoMgr. In addition, we need to coordinate the passing of tokens for multiple token rings communication, as discussed in Section 3.3.2. Explicit synchronization between multiple VPs is not needed for Single Token Ring, Snapshot and Diffracting Tree communication algorithm. This is because only one memory word is exchanged per virtual processor pair and this is done in a coordinated fashion with one VP doing the updating and the other reading. There are three ways to approach synchronization, namely using explicit locks, disabling the scheduler, or using a lock free technique. We consider each technique in more detailed in the subsections that follow. All three techniques have been implemented in QDoMgr and their performance implications will be discussed in Chapter 5.

4.8.1 Locks

The first way we can enforce consistency is to use locks. In our case, this is done by executing the atomic instruction FetchAndOrSync. Separate locks are used to govern the placing of new QDo requests and the communication of tokens. The locks are implemented on a per virtual processor basis – that is, each VP has its own set of locks because each VP

has its own separate lists for new requests. Thus, placing new requests on separate VPs should be independent of each other. Furthermore, we need to have a different lock for token passing because we do not want to block threads from placing QDo requests while token passing takes places.

Although QDoMgr has multiple locks, deadlock is prevented by following a few programming practices. First, each thread will not hold more than 1 lock at a time. Second, while the lock is held, the holder will not be blocked. This is achieved because updates do not depend on other objects that are protected by locks.

The advantages of locks are that they are simple and easy to use. In addition, they can be used for both intra- and inter-virtual processor synchronization. However, locks can have high overhead. Although we have implemented locks on a per virtual processor basis, multiple virtual processors may still contend for the same lock.

4.8.2 Lock Free List

Another way to coordinate intra- and inter-processor communication is to use lock free lists. Deadlock is avoided because no thread is blocked using this technique. In our implementation, we have used Harris et. al. 's lock free linked list technique [32]. As discussed in Section 2.1.1, the lock free linked list is updated by the atomic instruction Compare-And-Swap. This technique has similar problem as locks in that en-queuing and dequeuing requires communication with other processor, resulting in high overhead.

Readers may wonder whether using a lock free list would result in a circular dependency. Recall that one use of QDo is to delete elements of lock free lists. Circular dependency is avoided, however, by satisfying three conditions. First, although multiple threads may add to the list, only the daemon removes entries from the list. Second only one instance of the daemon is running per address space per virtual processor. From that, we

can conclude that there are multiple producers and a single consumer for every QDo lock free list. Third, all producers will only add elements to the head of the list. These three conditions allow us to infer that no thread will have any temporal references to deleted elements. Producers do not have temporal references to QDo lock free linked list elements because these threads do not have to traverse the link list. On the other hand, since there is only one consumer, the only thread traversing the linked list is the thread that deletes elements. Consequently, we do not need QDoMgr to defer deletion to safe points and circular dependency is avoided.

4.8.3 Disabling the Scheduler

Disabling the scheduler can also prevent race conditions between threads running on the same virtual processor. This is accomplished by preventing the executing thread from being scheduled out before it has completed its task, ensuring atomicity within the virtual processor. This technique has some limitations, however. First, we must be careful that the thread do not invoke any blocking calls or incurs any page fault while the scheduler is disabled. Second, this technique does not apply to inter-virtual processor communication. As a result, disabling the scheduler cannot prevent race conditions for the multiple token rings communication algorithm. On the other hand, this technique has low overhead. Moreover, it should be noted that synchronization across multiple processors are not needed for the *Single Token Ring*, *Snapshot* and *Diffracting Tree* algorithms.

4.9 Multiple Processors Callback

4.9.1 Method of Notifying Callbacks

The first decision we need to make is how to notify other virtual processors that they should execute callbacks. One possibility, and the one we have adopted, is to send an asynchronous inter-processor message to the target virtual processor. When this happens, the multiprocessor message system will send an interrupt to the target VP notifying a message is pending. A thread is then created on the target VP to execute the callback. The advantage of this technique is that the latency for completing callbacks on multiple processors is reduced as callback requests are received immediately. Furthermore, virtual processors do not have to regularly poll for pending callbacks because the inter-processor messages are delivered using a push model. On the other hand, the overhead of receiving an inter-processor message is high in that a potentially useful running process may be interrupted.

An alternative is to enqueue pending callback requests on the target VP's to-execute list. The target VP regularly polls this list to determine whether there are pending callback requests. This alternative has overhead even when there are no pending callbacks because it uses a polling method. However, the overhead from each pending callback is low because the target virtual processor is not interrupted. Instead, the callbacks can be launched when QDoMgr has the scheduler's quantum.

4.9.2 Serial vs. Concurrent Callback Execution

Recall that callers may request callbacks to be executed on multiple virtual processors. We need to determine whether it is best to concurrently execute callbacks from the same QDo request on multiple processors or to execute the callbacks in sequence. The solution we have employed is to execute callbacks on each target VP serially. QDoMgr from one VP

notifies the next specified virtual processor that callbacks are to be executed only after the local callback has been executed. The disadvantage is that this technique has a latency of O(q), where q is the number of virtual processors on which the callbacks are to be launched. On the other hand, lower overhead is achieved by reducing the need for write side synchronization. Since the callbacks are launched serially, these callbacks will not compete to acquire write lock with each other.

An alternative solution is to execute callbacks concurrently. Executing callbacks concurrently has O(1) latency. However, this technique has high communication overhead: QDo Callbacks would likely need to use explicit synchronization, such as locks, to protect critical regions. Thus, callbacks from the same QDo request will compete for the same lock. It is not beneficial for the same QDo callbacks to execute concurrently at multiple virtual processors.

Chapter 5

Experimental Evaluation

In this chapter, we present the micro-benchmark and macro-benchmark measurement results obtained from our implementation.

5.1 Definition

Before discussing our experimental results, we first define some terminologies. A diagram illustrating the relationship between a thread's state and various measurement terms is provided in Figure 5.1.

- Callback Latency - Latency between the point in time when invoking the function to register a QDo callback request to the point in time the callback has completed execution.
- Communication Latency - The time difference between the point in time when local quiescent state is established and the point in time when global quiescent state is established.
- Detection Latency - Time difference between the point in time when the monitoring

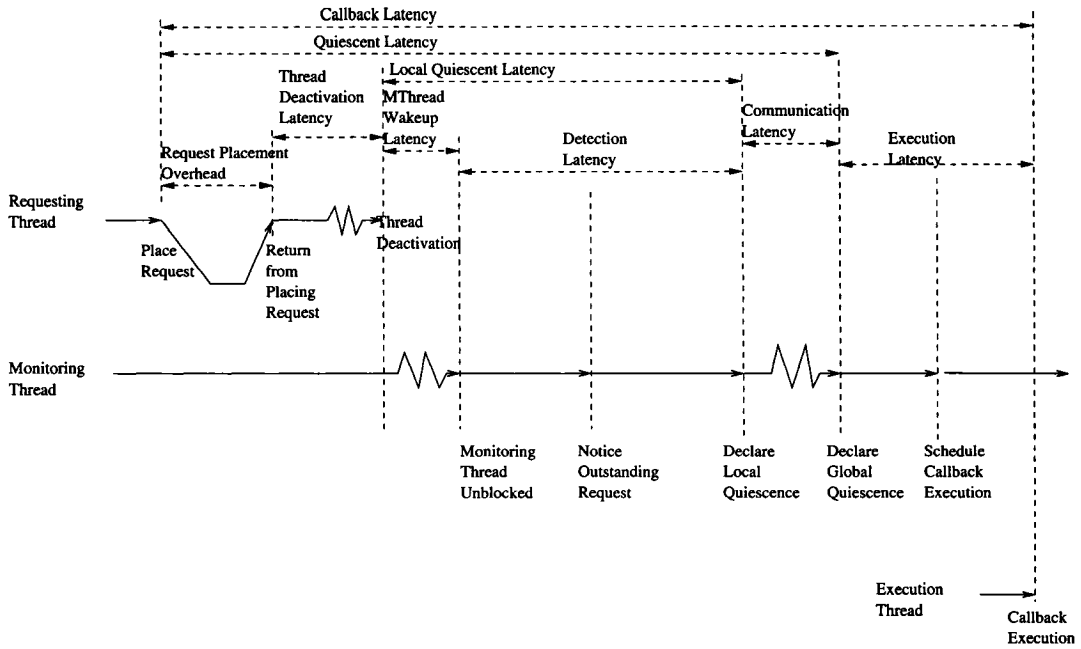

Figure 5.1: Measurement Terminologies

thread is woken up to the point in time when local quiescent is established.

- Execution Latency - Time difference between the point in time when global quiescent state is established to the point in time when the callback has completed execution.
- Execution Thread - The thread that is launched to execute the scheduled callback.
- Local Quiescent Latency - Time difference between when the requesting thread is deactivated to when local quiescence is established.
- Monitoring Thread - QDo manager daemon thread that checks whether there are outstanding QDo requests, and if so, whether quiescent state has been reached for these requests.
- MThread Wakeup Latency - Time difference between the point in time when the requesting thread is deactivated to when the monitoring thread is unblocked.

Parameter	Value
Processor Type	Power 3 (630+)
Processor Clock Frequency	375 MHz
Number of Processors	4
Data Cache	64 KB
Instruction Cache	32 KB
L2 Cache Size	4 MB per processor
Main Memory Size	1489732 KB

Table 5.1: Characteristics of IBM 270 Workstation

- Quiescent Latency - Time difference between when a callback request is registered to when global quiescent is established.
- Request Placement Overhead - Time difference between the moment when a requesting thread places a callback request to the moment when the requesting thread resumes normal execution.
- Requesting Thread - The thread that requests callback to be executed upon quiescent state.
- Thread Deactivation Latency - Time difference between the moment when requesting thread places a QDo callback request to the moment when that requesting thread is deactivated.

5.2 Experimental Setup

Our experimental results are obtained by running the K42 Operating System on a IBM 270 Workstation. The characteristics of the 270 Workstation are summarized in Table 5.1 [19].

For the micro-benchmark experiments, our file server will be Network File System (NFS) [39]. To remove noise and secondary effects, NFS validation is turned off and K42 thinwire polling is disabled. The experiments are written as user level programs. This

allows us to verify that the QDo facility is functional in user space. The only processes active during experimental runs are the kernel, baseServers, bash, procfs (process filesystem), systcl (system control) and the micro-benchmark process.

To eliminate the effect of disk IO on macro-benchmarks, we use Ram File System (ramfs) as our file server [83]. In addition, K42 thinwire polling is disabled. Both the micro-benchmark and macro-benchmark results are recorded with the K42 tracing facility [85].

5.3 Latency Micro-benchmark

The latency micro-benchmark consists of a user level thread invoking the QDo facility to schedule a callback when quiescent state is reached. Code snippet of this micro-benchmark is provided in Figure 5.2. Our measurement is taken from 5 separate runs and each run consists of the user level thread scheduling a callback 500 times.

In the next few paragraphs, we will explore how latency is affected by factors such as the choice of communication protocol (as discussed in Section 3.3.2), QDoMgr monitoring thread's soft timer's frequency (as discussed in Section 3.3.1), synchronization techniques (as discussed in Section 4.8).

5.3.1 Effect of Communication Algorithm

We will now determine the choice of communication algorithms' effect on callback latency. In this experiment, we hold the other parameters, such as the monitoring thread frequency and synchronization technique, constant. Details of the setup for this experiment can be found in Table 5.2.

As discussed in Section 3.3.2, we expect the snapshot communication algorithm to have

```
void
UsrQDOTestCB(uval num) {
    /* This function is executed when quiescent
     * state is reached */

    /* We take a timestamp when we start this function*/
    if (num == 0)
       TraceStart();

    /* QDo will wait for quiescent state across
     * all processors and will execute the callback
     * on this processor */

    VPSet quiescentSet;
    VPSet executionSet;
    quiescentSet.addVP(Scheduler::GetAllVP());
    executionSet.addVP(Scheduler::GetVP());

    if (num < 500) {
        /* UsrQDoTestCB will call itself 500 times */
         DREFGOBJ(TheQDoMgrRef)->qdo(UsrQDoTestCB,
                                     num+1,
                                     quiescentSet,
                                     executionSet);
    }

    TraceEnd();
    /* we take a timestamp of when this
     * function finishes */
}
```

Figure 5.2: Code Snippet of Latency Measurement Process

Parameter	Value/Choice
Monitoring Thread's Soft Timer Frequency	10ms
Number of Processors	2 and 4
Synchronization Technique	Disable Scheduler
Method of Executing Callback	Launch a New Thread for Each Callback

Table 5.2: Setup for Measuring Latency for Different Communication Algorithms

Number of Processors	Multi-Token	Snapshot	Token	Tree
2 VP	5.127756 (0.138102)	10.101238 (0.0315741)	20.202603 (0.0534543)	20.144856 (0.0524735)
4 VP	12.976357 (0.244489)	10.084384 (0.0211059)	29.837121 (0.451504)	21.05772 (0.18661)

Table 5.3: Average Callback Latency (in ms) for Different Communication Algorithms

the lowest callback latency, followed by the tree communication algorithm. The token and multi-token communication should have similar callback latency.

We ran our experiments on a IBM 270 workstation and the results are summarized in Table 5.3. The standard error is denoted in brackets. The result is different than what we had expected. To understand why this is the case, we will analyze the individual components that make up callback latency. Recall from Figure 5.1, callback latency consists of five components, namely request placement overhead, thread deactivation latency, local quiescent latency, communication latency and execution latency. Thread deactivation latency is dependent on the requesting thread and is outside of QDoMgr's control. We will now examine the other four components' impact.

Request Placement Overhead

The request placement overhead result is summarized in Figure 5.4. We expect the request placement overhead to be small and independent of the communication algorithm. This expectation has been verified. Each communication algorithm's request placement over-

Number of Processors	Multi-Token	Snapshot	Token	Tree
2 VP	3149.20 (12.73)	3371.26 (16.74)	3445.16 (17.43)	3427.66 (16.53)
4 VP	3236.94 (14.61)	3375.96 (16.80)	3427.66 (16.44)	3446.56 (17.04)

Table 5.4: Average Request Placement Overhead (in ns) for Different Communication Algorithms

head is about 3400 ns. Since the request placement overhead is small, it is not the source of callback latency's discrepancy.

Local Quiescent Latency

The second component is local quiescent latency. Local quiescent latency encompasses the time from when the requesting thread is deactivated to when local quiescent state is established. In theory, this value is independent of the communication algorithm used. The measured results are presented in Table 5.5. As can be seen, local quiescent latency is, contrary to initial expectation, dependent on the communication algorithms. Snapshot communication has higher local quiescent latency than tree communication. This is the case because, for the snapshot communication algorithm, QDoMgr needs to take snapshots of other processors before checking whether local quiescent state is reached. This allows us to have an accurate picture of other processors' state at the time when QDo request is *registered.* Otherwise, we will not notice if other processors have already reached local quiescent during our wait for local quiescent state. Nevertheless, since the local quiescent latencies are in the range of 7 to 20 microseconds, they are not the main contributing factor for callback latency's discrepancies.

Number of Processors	Multi-Token	Snapshot	Token	Tree
2 VP	9566.45 (32.84)	11268.96 (18.96)	8753.28 (33.93)	7553.42 (16.39)
4 VP	9874.34 (32.41)	15979.2 (19.41)	8715.68 (17.90)	7586.92 (14.98)

Table 5.5: Average Local Latency (in ns) for Different Communication Algorithms

Number of Processors	Multi-Token	Snapshot	Token	Tree
2 VP	5.112565 (0.138)	10.084824 (0.032)	20.185612 (0.054)	20.131964 (0.049)
4 VP	12.960751 (0.245)	10.063246 (0.021)	29.820415 (0.452)	21.04462 (0.182)

Table 5.6: Communication Latency (in ms) for Different Communication Algorithms

Communication Latency

Another component is communication latency. As can be seen in Table 5.6, communication latency has a major effect on callback latency. We expected the multi-token communication algorithm to have similar communication latency as the token communication algorithm. However, this is not the case. The multi-token algorithm's latency is, however, much smaller than the token algorithm's latency. In fact, multi-token performs better for 2 VP than our expected best algorithm for latency, namely snapshot communication algorithm.

The reason for these apparent discrepancy becomes clear when considering the effect of soft timers. Recall from Section 3.3.1 and Section 4.7 that generation changes do not occur when the processor is idle. Thus, we use a soft timer to regularly schedule the monitoring thread to monitor quiescent state. For this experiment, the processors are mostly idle and the soft timer is set to 10ms. In the case of the snapshot communication algorithm, as shown in Figure 5.3, after the requesting thread registers a callback request with QDoMgr

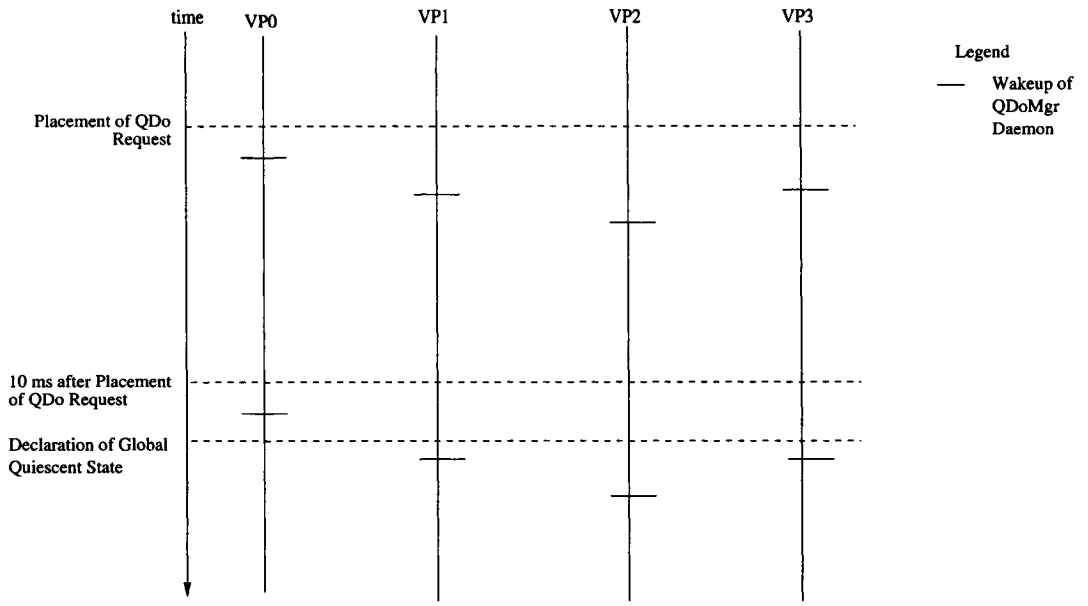

Figure 5.3: Relationship Between Communication Latency, Wakeup Frequency and Snapshot Communication Method

on VP0, VP0's monitoring thread wakes up and takes a snapshot of other VPs' generation count. This monitoring thread then sleeps for 10ms because there are no active threads and generation changes do not occur. In the meantime, during VP0's 10ms sleep time, other VPs' monitoring threads wake up and increment their generation counts. After the 10ms expires, VP0's monitoring thread wakes up and finds that other processors have reached local quiescent state. Thus, global quiescent state can be deduced only after 10ms.

The situation is different in the case of the multi-token communication algorithm, which is depicted in Figure 5.4. Similar to the snapshot communication algorithm, the requesting thread registers a callback request on VP0. First, the monitoring daemon established local quiescent state on VP0. Then, before VP0's monitoring thread goes to sleep for 10ms, it passes the token to the next specified processor. Global quiescent state is established when the token has passed through all specified processors. One important thing to note is that the token does not have return to VP0 before global quiescent state can be deduced.

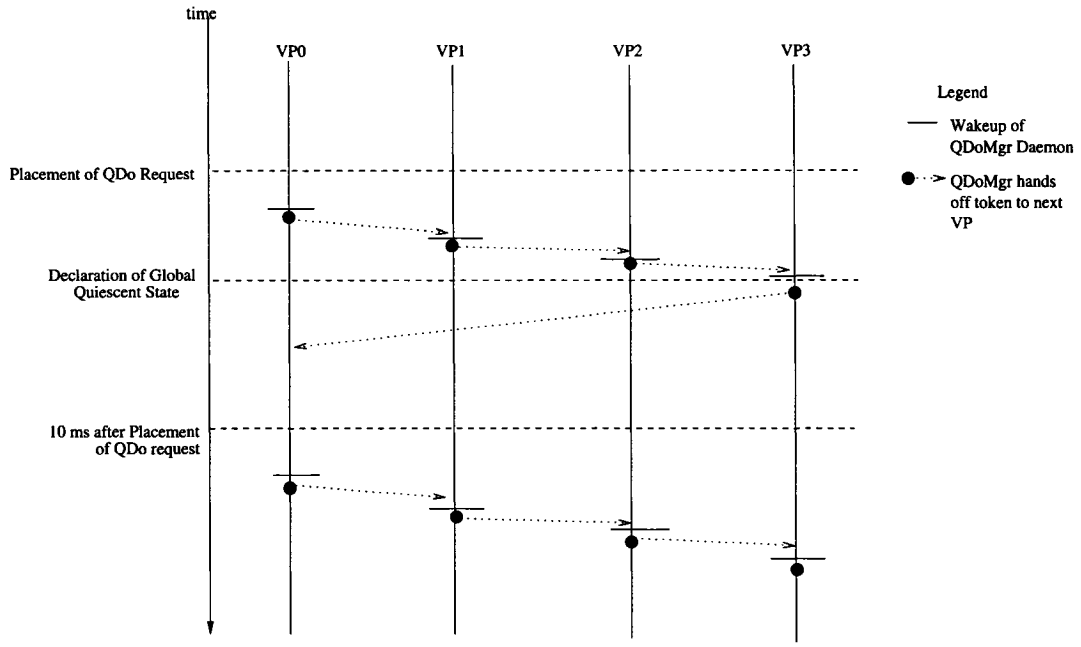

Figure 5.4: Relationship Between Communication Latency, Wakeup Frequency and Multi-Token Communication Method

Instead, VP3's monitoring thread establishes that global quiescent state is achieved after the token has arrived VP3. Thus, depending on the timing of other VPs' monitoring thread, it is possible that quiescent state is achieved before VP0 wakes up again. This explains why the communication latency for multi-token communication can be shorter than the communication latency for snapshot communication in the case of 2 VPs.

Now let us consider at the case of the token communication algorithm, which is illustrated in Figure 5.5. Similar to the multi-token communication algorithm, the requesting thread registers a callback request on VP0. However, in contrast with the multi-token case, other processors may hold the token when QDo request is placed. A processor must be visited by the token twice before deducing that quiescent state was achieved. Hence, in the case when the processors are idle, the minimum communication latency is 20ms.

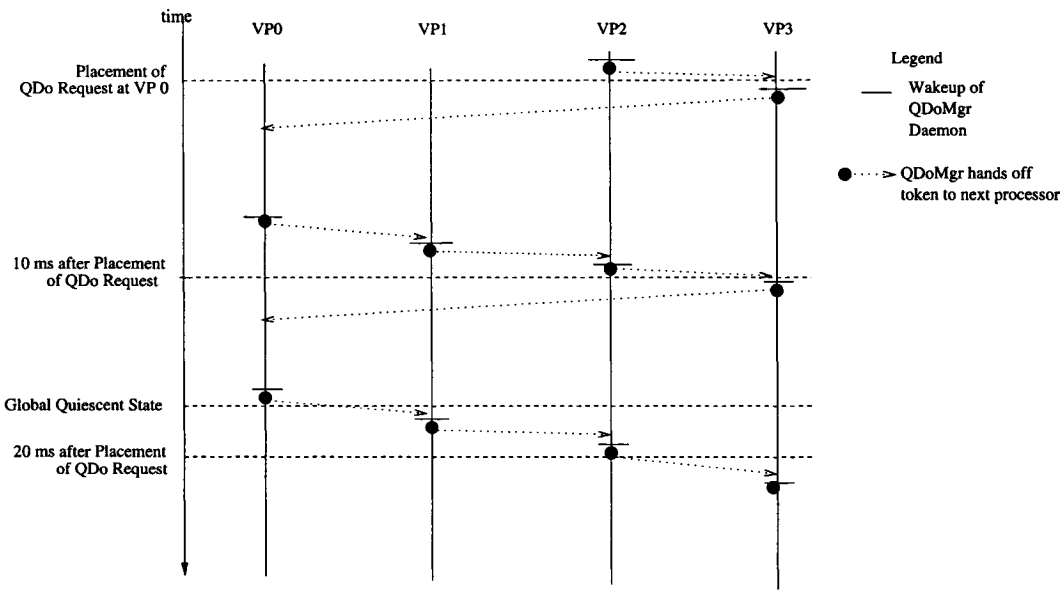

Figure 5.5: Relationship Between Communication Latency, Wakeup Frequency and Token Communication Method

Number of Processors	Multi-Token	Snapshot	Token	Tree
2 VP	1763.146293 (12.04589)	1253.86 (5.989889)	4272.18 (8.645286)	1326.88 (6.296987)
4 VP	1792.14 (11.940121)	1264.74 (5.874614)	4041.22 (10.745737)	1520.3 (5.977098)

Table 5.7: Average Execution Latency (in ns) for Different Communication Algorithms

Execution Latency

The final component of callback latency is execution latency. The execution latency is summarized Table 5.7. As it can be seen, execution latency is mostly independent of the communication method and does not significantly impact callback latency. The token communication method has a higher execution latency than other communication methods because the executing processor has to pass the token to the next processor even after global quiescent is detected.

Parameter	Value/Choice
Communication Method	Snapshot
Number of Processors	2 and 4
Synchronization Technique	Disable Scheduler
Method of Executing Callback	Launch a New Thread for Each Callback

Table 5.8: Setup for Measuring Latency for Various Monitoring Thread Soft Timer's Frequency

5.3.2 Effect of Monitoring Thread's Soft Timer Frequency

Next, we will investigate how the monitoring thread's soft timer frequency affects callback latency. We would expect that callback latency is small when the soft timer is set to a small value. Recall from our previous analysis in Section 5.3.1, communication latency dominates callback latency and the monitoring thread's soft timer greatly affects communication latency. The setup for measuring the callback latency is summarized in Table 5.8 and the results are summarized in Figure 5.6.

As it can be seen from the graph, the result again is counter-intuitive in that when the soft timer frequency is very high (i.e., the soft timer period is very small), the callback latency increases rather than decreases. To understand this effect, we break the callback latency down into four different components for the four VPs, as illustrated in Table 5.9.

We see that when the frequency is very high, the local quiescent latency, communication latency and execution latency increases substantially. After further investigation, we identify two reasons for the increase. First, the scheduler's overhead becomes significant when the soft timer is set to a small value. The executing thread and monitoring thread does not have a chance to execute to completion before the timer kicks in and interrupts the processor. Second, as the monitoring threads for different VP execute at approximately the same time, they start to interfere with each other. For example, when VP0's monitoring thread takes a snapshot of VP2's state, VP2's monitoring thread may be updating

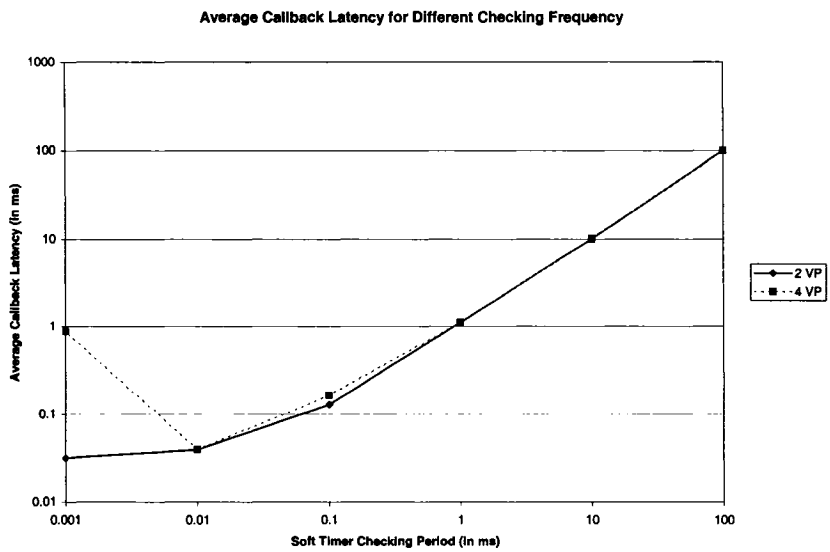

Figure 5.6: Actual Relationship between Callback Latency and Monitoring Thread's Soft Timer's Period

Monitoring Thread Soft-Timer Frequency	Request Placement Overhead	Local Quiescent Latency	Communication Latency	Execution Latency	Callback Latency
0.001	0.001752176	0.528700918	0.342516826	0.011369182	0.884845389
0.01	0.001619621	0.009483673	0.024451218	0.003561776	0.039615848
0.1	0.001577864	0.015248283	0.141788762	0.00356986	0.162684012
1	0.001470918	0.012766467	1.093163733	0.003537685	1.111435848
10	0.00149012	0.012957625	10.07524649	0.003835888	10.09403419
100	0.00159022	0.013337405	100.1158967	0.005080739	100.1364242

Table 5.9: Breakdown of Callback Latency for Different Monitoring Thread's Soft Timer Frequencies for 4 VP (in ms)

Parameter	Value/Choice
Monitoring Thread's Soft Timer Frequency	10 ms
Number of Processors	2 and 4
Synchronization Technique	Disable Scheduler
Method of Executing Callback	Launch a New Thread for Each Callback

Table 5.10: Setup for Measuring Callback Latency for Various Synchronization Method

Number of VPs	Disabled Synchronization	Lock Synchronization	Lock-Free Synchronization
2	10.08412401 (0.026502491)	10.79778994 (0.122462024)	10.77461108 (0.117888982)
4	10.09403419 (0.024663745)	10.78058096 (0.116344351)	10.78264814 (0.116212869)

Table 5.11: Callback Latency for Different Synchronization Techniques

VP2's generation. Therefore, although increasing the monitoring thread's soft timer frequency can lower callback latency, it is important not to set the monitoring thread checking frequency too high.

5.3.3 Effect of Synchronization Techniques

The third parameter we consider the effect of synchronization techniques on callback latency. We expect that disabling scheduler has the least latency because it does not require explicit synchronization communication with other processors, as discussed in Section 4.8. We have measured the callback latency for different synchronization techniques with setup summarized in Table 5.10 and the result is presented in Table 5.11.

The results confirm our expectation that disabling the scheduler produces the best latency results for both 2 VPs and 4 VPs case. One may wonder why the latency for 4 VPs is less than 2 VPs for lock synchronization. To answer this question, we need to realize that the results for 2 VPs and 4 VPs fall within the standard error. Thus, we cannot accu-

rately conclude the 4 VPs case has less latency than the 2 VPs case. Instead, we can only conclude 4 VPs has similar latency with the 2 VPs case.

5.4 Overhead Micro-benchmark

Recall from Section 1.3 that one of our design goal is to have low overhead. To evaluate whether we have achieved this goal, one of our micro-benchmark experiments measures how long it takes to schedule and execute 500 functions with and without QDoMgr running. A code snippet of the experiment is provided in Figure 5.7. This micro-benchmark gives us a good picture of the overhead because it exercises the scheduler to create and destroy threads. As such, when QDoMgr is enabled, QDoMgr will frequently be invoked to check whether quiescent state has beens achieved. We calculate overhead as follows:

$$Overhead = \frac{ExecuteTimeWithQDoMgr - ExecuteTimeWithoutQDoMgr}{ExecuteTimeWithoutQDoMgr} \quad (5.1)$$

5.4.1 Effect of Soft-Timer Frequency

We first examine the effect of the monitoring thread's soft timer frequency on overhead. The setup of our experiment is the same as our latency micro-benchmark described in Section 5.3.2. We expect overhead to grow exponentially with respect to the monitoring thread's soft timer frequency. The result is shown in Figure 5.8. As it can be seen, the overhead for 100ms and 10ms soft timer frequency is less than 1% while the overhead for a 0.01ms checking frequency is over 100%. This confirms our earlier assertion that the soft-timer frequency should not be set to a very low value.

```
void QDoCallback(uval done) {
        *done = 1;
}

void schedFunction(uval runLeft) {
  if (runLeft) {
    Scheduler::ScheduleFunction(schedFunction,
                                runLeft - 1);
  } else {
    TraceEnd();
        /* Stop the timer */
  }
}

int main() {
#ifdef HAVE_QDO_MGR
/* This part will only run when QDoMgr is running */

        uval done = 0;

    /* First, we want to call QDo to see the
       effect of QDoMgr on overhead.  If we
       do not place a QDo request, the QDoMgr
       will remain idle */

    VPSet quiescentSet;
    VPSet executionSet;
    quiescentSet.addVP(Scheduler::GetAllVP());
    executionSet.addVP(Scheduler::GetVP());

    DREFGOBJ(TheQDoMgrRef)->qdo(QDoCallback,
                                &done,
                                quiescentSet,
                                executionSet);
    while (! done) {
       /* Want to wait until callback is done */
       Yield();
    }
#endif // #ifdef HAVE_QDO_MGR

    TraceStart(); /* Start the timer */
    Scheduler::ScheduleFunction(schedFunction,
                                500);
    return 0;
}
```

Figure 5.7: Code Snippet of Overhead Measurement Process

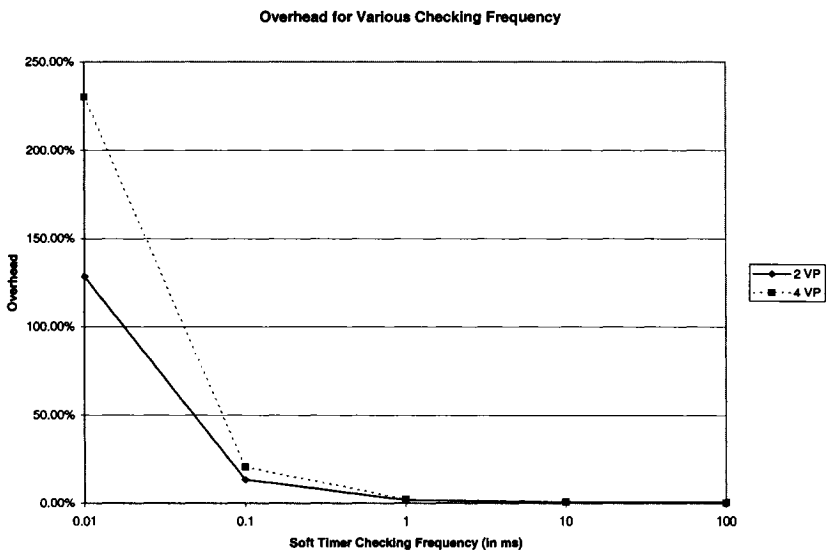

Figure 5.8: Relationship between Monitoring Thread's Soft Timer Frequency and Overhead

	Disabling Scheduler	Locked List	Lock-Free List
2 VP	0.67%	0.63%	1.16%
4 VP	0.59%	0.58%	1.08%

Table 5.12: Overhead for Different Synchronization Techniques

5.4.2 Effect of Synchronization Techniques

Next, we explore whether the choices of synchronization technique used affects overhead. Our experiment setup is the same as described in Section 5.3.3. Based on the results from Section 5.3.3 and Section 5.4.1, we expect that the overhead for all three techniques (Disabling the Scheduler, Using Locks and Lock-Free List) to be less 2 %. Nevertheless, disabling the scheduler should have the least overhead while lock free list should have the highest overhead. This is because the lock-free list requires extensive communication with other processors. It should be note that lock free linked list has an advantage of being deadlock free.

Our experimental results are summarized in Table 5.12. As expected, all three cases result in overhead of less than 2%. The Lock-Free linked list produces the worst results compared to the other three cases. On the other, Disabling the scheduler and locked list have similar overhead.

5.5 Distribution of Generation Period in Kernel Space

We now investigate the distribution of the global generation period under real work loads. This allows us to understand how QDoMgr behaves in a real server. To setup this experiment, we place a background thread in the kernel space, as described by the code snippet in Figure 5.9. The function BackgroundQDoLoop will invoke QDoMgr to schedule a callback on the same function on the local virtual processor once quiescent state is reached

across all virtual processors. We then measure the distribution of callback latency for the first 2000 QDo callbacks. At the same time, we run the SPEC Software Development Environment Throughput (SDET) benchmark [26]. The SDET benchmark simulates typical programs run by a software developer. Many kernel threads are created and destroyed when servicing SDET requests.

5.5.1 Communication Algorithms

The first parameter we consider is how the choice of communication algorithm affects the global generation distribution. The setup of this experiment is the same as the latency measurement experiment described in Section 5.3.1.

The experimental results are summarized in Figure 5.10 and Figure 5.11. As expected, the distribution with the snapshot communication algorithm is consistent for both the 2 VPs and the 4 VPs cases. With the token communication algorithm, the median callback latency is almost twice as high in the 4 VPs case than in the 2 VPs case. This matches our expectation in Section 3.3.2, namely that token algorithm's generation period grows proportionally with respect to the number of processors. With the multi-token algorithm, the median global generation period also increases from a 2 VPs to a 4 VPs system. This again matches our expectations where multi-token's callback latency grows with respect to the number of virtual processors.

5.5.2 Monitoring Thread's Soft-Timer Frequency

The second parameter we consider is how the monitoring thread's soft-timer frequency affects on the distribution of the global generation period. Our setup is the same as the experiment described in Section 5.3.2.

The experimental results are summarized in Figure 5.12, Figure 5.13, Figure 5.14 and

```
void BackgroundQDoLoop(uval numCall) {
/* This function will call QDoMgr to schedule itself
   once quiescent state is reached and we will measure
   the distribution of the callback latency.
*/

    VPSet quiescentSet;
    VPSet executeSet;
    quiescentSet.addVP(Scheduler::GetAllVP());
    executionSet.addVP(Scheduler::GetVP());

    if (numCall < 2000) {
        Trace(); /* measure distribution */
        DREFGOBJ(TheQDoMgrRef)->qdo(BackgroundQDoLoop,
                                    numCall+1,
                                    quiescentSet,
                                    executionSet);
    }

}

int main() {
    Scheduler::ScheduleFunction(BackgroundQDoLoop,
                                0);
    return 0;
}
```

Figure 5.9: Code Snippet of Measuring the Distribution of Generation Period in Kernel Space

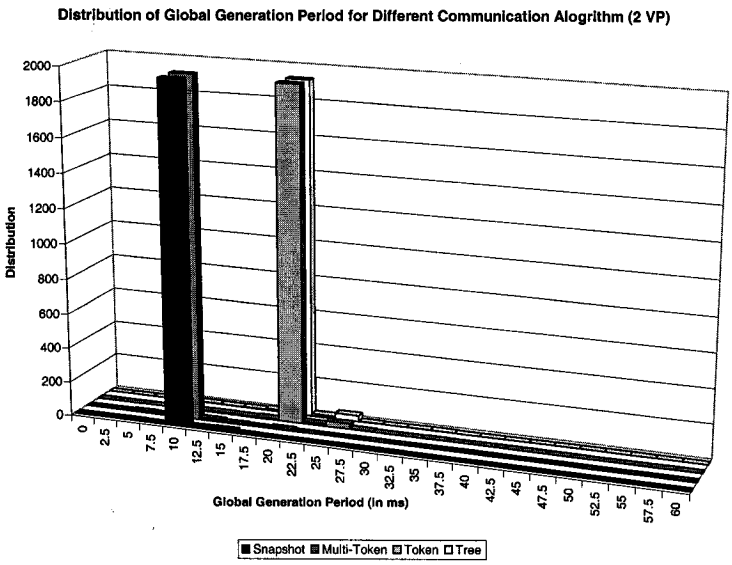

Figure 5.10: Histogram of Global Generation Period for Different Communication Algorithm (2 VP)

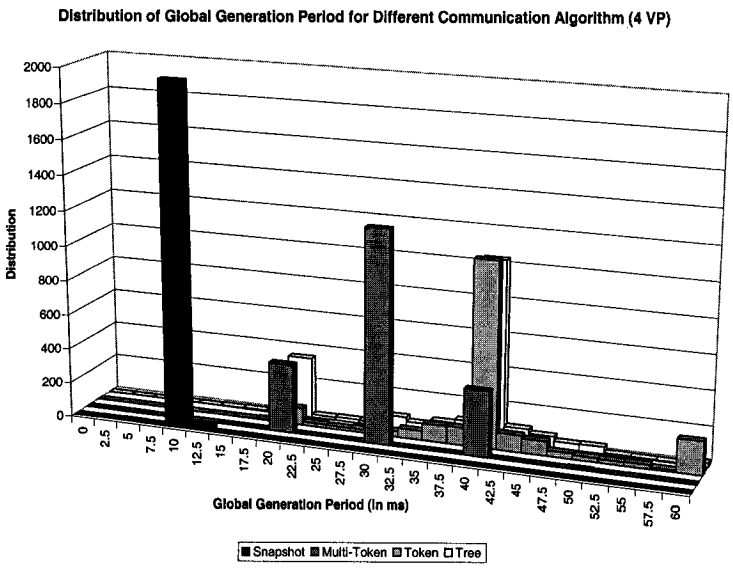

Figure 5.11: Histogram of Global Generation Period for Different Communication Algorithm (4 VP)

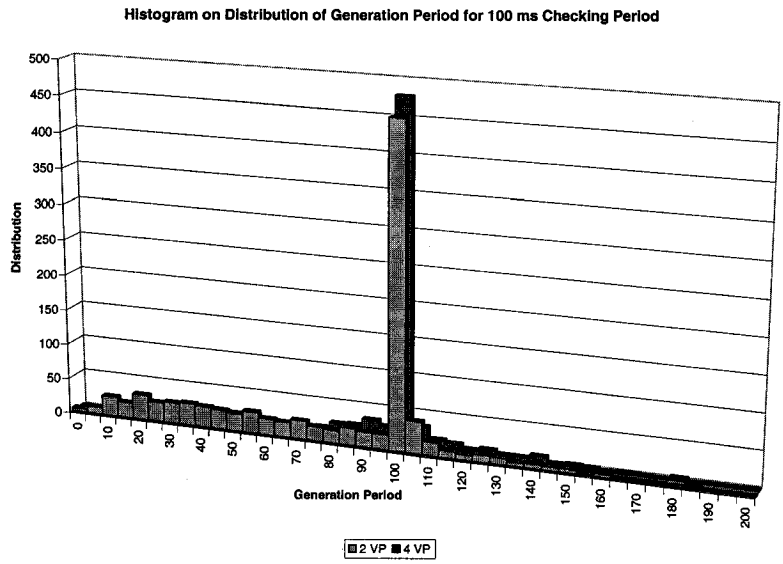

Figure 5.12: Histogram of Global Generation Period when Checking Frequency = 100 ms

Figure 5.15. Note that for a 100ms monitoring thread period, our experiment finishes before 2000 samples are taken. As expected, the generation period is large when the soft timer is set to a large value, and vice-versa. In addition, the histogram for soft timer frequency of 0.1ms is much more spread out. This suggests that for the SDET workload, the theoretical minimum limit of global generation period is between 0.1ms to 1ms.

5.6 SDET Macro-Benchmark Result

Now that the latency, overhead and distribution of global generation period for QDo callbacks is known, we investigate how the QDo facility impacts performance in a more realistic setting. For this experiment, we modified existing K42 code (HashNonBlocking and XHandleTrans module) to use the QDo facility. The HashNonBlocking module is a non-blocking hash lookup table implemented with RCU logic. This table is used for var-

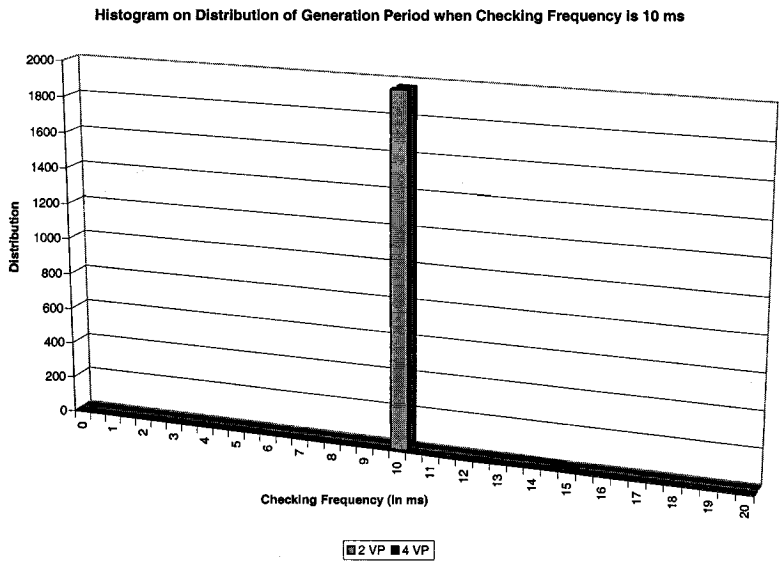

Figure 5.13: Histogram of Global Generation Period when Checking Frequency = 10 ms

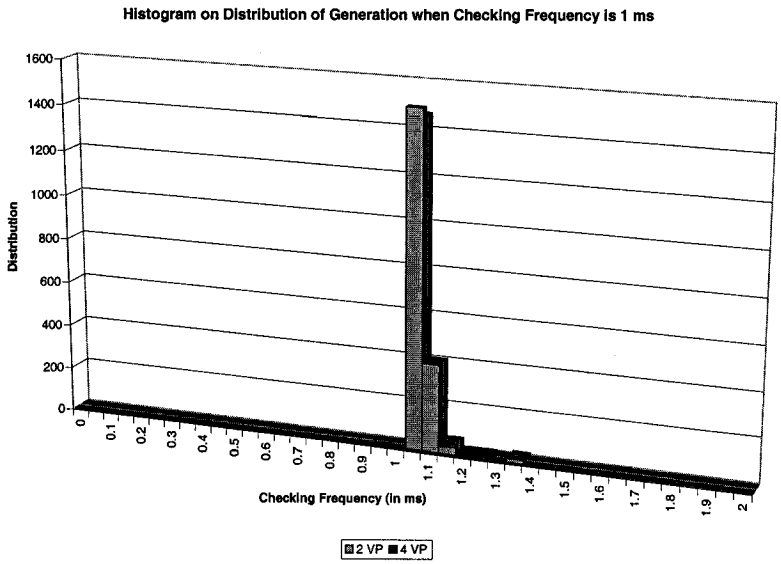

Figure 5.14: Histogram of Global Generation Period when Checking Frequency = 1 ms

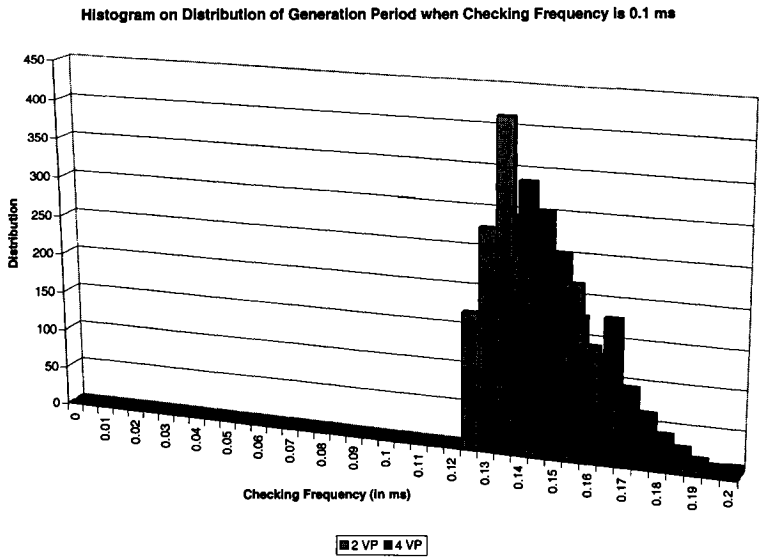

Figure 5.15: Histogram of Global Generation Period when Checking Frequency = 0.1 ms

ious purposes, such as maintaining a list of processes as well as the interface to service for registration and look up of mount points [18]. Before the changes, the HashNonBlocking module used K42's clustered object garbage collection's generation count to determine when it is safe to resize the hash table [1]. After the changes, the HashNonBlocking module uses the QDo facility to resize the hash table after global quiescence across all virtual processor is reached.

The XHandleTrans module provides support routines for the external object translation and invocation subsystem [77]. Before the changes, XHandleTrans used the clustered object garbage collection to physically delete freed object. With QDoMgr, we use the QDo facility to physically deleted the freed object upon quiescent state.

The setup of the experiment is listed in Table 5.13. Our measurements is summarized in

[1]Recall from Section 2.2.2 that K42's clustered object infrastructure uses generation count to implement RCU-like garbage collection

Parameter	Value/Choice
Monitoring Thread's Soft Timer Frequency	10 ms
Communication Algorithm	Snapshot
Number of Processors	2 and 4
Synchronization Technique	Disable Scheduler
Method of Executing Callback	Launch a New Thread for Each Callback

Table 5.13: Setup for Measuring SDET performance

Number of Processors	No QDo	With QDo	Percentage Difference
2	1567.4	1559.5	-0.50%
4	2979.5	2921.5	-1.95%

Table 5.14: SDET performance (in scripts/hour)

Table 5.14. It is disappointing that there is a slowdown when using QDo instead of performance improvement, although the slowdown is less than 2%. We still need to investigate the cause of this decrease, but believe it is due to the soft timer overhead.

Chapter 6

Conclusion

We have designed and implemented a quiescent point callback facility called QDo for the K42 Operating System. This facility allows i) clients to register interest in the quiescent point relative to the time of registration, ii) monitors the system for relevant quiescent point, and iii) initiates a client-specified callback when quiescent state has been achieved. In designing this facility, there were four design goals:

1. low overhead,
2. low latency for callbacks,
3. good scalability, and
4. having the facility available for both user and kernel space

To achieve our design goals, we have made a number of design decisions. Quiescent monitoring is done on a per address space basis. In addition, we partitioned the monitoring and detection of quiescent state into two separate steps - the monitoring of quiescent states on the local processor and the communication of these states with other processors to establish a global quiescent state. For the local processor, we use a soft timer to regular

wake up the QDo daemon thread and check whether at least two generation changes have occurred. To communicate quiescent information with other processors, we have devised four possible ways to establish for global quiescent state:

1. Using a single token ring for all QDo request.

2. Using one token ring for each QDo request.

3. Taking snapshots of other processors' state.

4. Using in a binary diffracting tree-like communication structure.

We implement the QDo management facility (QDoMgr) as a clustered object to minimize true and false data sharing. There are three choices in synchronizing data, namely temporary disabling the scheduler, using an explicit lock, and using a lock free list. In addition, QDoMgr will schedule each callback with a new thread.

Through our experiments, we determined that taking regular snapshots of other processors is the communication method resulting in the least amount of latency. In addition, the soft timer used in regularly waking up the QDoMgr daemon plays an important role in determining callback frequency and overhead. Setting the soft timer wakeup to less than 10us can greatly increase the overhead and renders the facility unusable. To reduce overhead and callback latency, it is best to temporary disable the scheduler in critical sections instead of using lock free list. For analyzing the performance impact on real application, we experimented with the SDET benchmark. Unfortunately, running SDET decreased performance by about 2%. We still need to investigate the cause of this decrease, but believe it is due to the soft timer overhead.

6.1 Future Work

We have a number of tasks that are left for future works, including:

1. Detection of quiescent state when there are no active threads – Currently, a soft timer is used to regularly wake up the monitoring daemon to determine whether quiescent state has been reached. However, this is not efficient. We need to examine how we can efficiently check for quiescent state in an idle processor.

2. Combine QDoMgr with the clustered object garbage collection system – Currently, the clustered object garbage collection uses a generation-based RCU collection scheme. We believe that the clustered object management system can use QDoMgr facility to do the garbage collection. This reduces overhead as there will be only 1 daemon monitoring for quiescent state.

3. Repeat the experiments on a variety of machines – We have only measured the performance on a 4 processor system. It would be interesting to repeat our experiments on a large machine (i.e., 24 ways or 64 ways) to determine whether our design assumptions continue to hold true as the number of processors increase.

Bibliography

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tavanian, and Michael Young. Mach: A new kernel foundation for unix development. In *Proceedings of the USENIX Summer Technical Conference*, Atlanta, GA, July 1986.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1995.

[3] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and System*, 15(1):182–205, January 1993.

[4] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA'93)*, pages 251–260, June 1993.

[5] AMD. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, October 2005.

[6] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[7] James H. Anderson and Mark Moir. Universal constructor for multi-object operations. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing (PODC)*, pages 184–193, August 1995.

[8] Jonathan Appavoo. Clustered objects: Initial design, implementation and evaluation. Master's thesis, University of Toronto, 1998.

[9] Jonathan Appavoo. *Clustered Object.* PhD thesis, University of Toronto, 2005.

[10] Jonathan Appavoo, Marc Auslander, Maria Burtico, Dilma Da Silva, Orran Krieger, Mark Mergen, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42: an open-source linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[11] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42 overview. Technical report, IBM Research, August 2002.

[12] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. Sceduling in k42. Technical report, IBM Research, August 2002.

[13] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, Ben Gamsa, Reza Azimi, Raymond Fingas, Adrian Tam, and David Tam. Enabling scalable performance for general purpose workloads on shared memory multiprocessors. Technical Report RC22863, IBM Research Report, 2003.

[14] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy-update techniques for system v ipc in the linux 2.5 kernel. In *Proceedings*

of 2003 USENIX Annual Technical Conference (FREENIX Track), pages 297–310, June 2003.

[15] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–176, Seattle, WA, March 1990.

[16] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the Thirteenth IEEE International Conference on Distributed Computing Systems*, pages 264–273, Los Alamitos, CA, May 1993. IEEE Computer Society Pres.

[17] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, September 1991.

[18] Suparna Bhattacharya and Dilma Da Silva. Towards a highly adaptable filesystem framework for linux. In *Proceedings of Ottawa Linux Symposium 2006*, volume One, pages 87–99, Ottawa, ON, 2006.

[19] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded powerpc processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, 2000.

[20] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA, second edition, 2003.

[21] John B. Carter, Chen-Chi Kuo, and Ravindra Kuramkote. A comparison of software and hardware synchronization mechisms for distributed share memory multiprocessors. Technical Report UUCS-96-011, University of Utah, Salt Lake City, UT, September 1996.

[22] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 179–193, Monterey, CA, November 1994.

[23] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 524–531, 1988.

[24] Michael Dubois, Christoph Scheurich, and Faye A. Briggs. Memory access buffering in multiprocessors. In *Proccedings of the Thirteenth Annual International Symposium on Computer Architecture*, pages 432–442, June 1986.

[25] Dominic Duggan. Type-based hot swapping of running modules. In *Proceedings of the Fifth International Conference on Functional Programming*, pages 62–73, 2001.

[26] Steven L. Gaede. Perspectives on the spec sdet benchmark. Technical report, Lone Eagle Systems Inc, January 1999. http://www.specbench.org/osg/sdm91/sdet/SDETPerspectives.html.

[27] Ben Gamsa. *Tornado: Maximizing Locality and Concurrency in a Shared-Memory Multiprocessor Operating System.* PhD thesis, University of Toronto, 1999.

[28] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system.

In *Proceedings of the Third Symposium on Operating System Design and Implementation*, New Orleans, LA, February 1999.

[29] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[30] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[31] Michael Greenwald and David R. Cheriton. The synergy between nonblocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 123–136, Seattle, WA, October 1996. USENIX Association.

[32] Timothy L. Harris. A pragmatic implementation of non-blocking linked-list. In *Proceedings of the Fifteenth International Symposium on Distributed Computing*, volume 2180 of Lecture Notes in Computer Science, pages 300–314, Springer-Verlag, October 2001.

[33] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the Sixteenth International Symposium on Distributed Computing*, pages 265–279, Springer-Verlag, October 2002.

[34] Hermann Hartig, Michael Hohmuth, Jochen Liedtke, Sebastian Schonberg, and Jean Wolter. The performance of u-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP'97)*, Saint-Malo, France, October 1997.

[35] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jaunary 1991.

[36] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[37] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, pages 92–101, Boston, Massachusetts, 2003. ACM Press.

[38] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, pages 298–300, New York, NY, May 1993. ACM.

[39] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[40] Kevin Hui. Design and implementation of k42's dynamic clustered object switching mechanism. Master's thesis, University of Toronto, 2000.

[41] Kevin Hui, Jonathan Appavoo, Robert Wisniewski, Marc Auslander, David Edelsohn, Ben Gamsa, Orran Krieger, Bryan Rosenburg, and Michael Stumm. Supporting hot-swappable components for system software. In *Proceedings of HotOS*, 2001.

[42] IBM T.J. Watson Research Center. *System/370 Principles of Operations*, 1983.

[43] Intel. *Intel Core Duo Processor and Intel Core Solo Processor on 65 nm Process Datasheet*, January 2006.

[44] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589 – 604, 2005.

[45] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture (ISCA'92)*, pages 13–21, Gold Coast, Australia, May 1992.

[46] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagra: A 32 way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, March - April 2005.

[47] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1997.

[48] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *Proceedings of the 1993 International Conference on Parallel Processing*, pages 201–204, St. Charles, IL, August 1993.

[49] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[50] Bernard Lang, Christian Queinnec, and Jose Piquer. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of*

Programming Languages, ACM SIGPLAN Notices, pages 39–50. ACM Press, January 1992.

[51] Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 314–323, San Diego, CA, USA, 2003. ACM Press.

[52] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, New York, NY, June 1991.

[53] Marios Mavronicolas, Marina Papatriantafilou, and Philippas Tsigas. The impact of timing on lineraizability in counting networks. In *Proceedings of Eleventh International Parallel Processing Symposium*, pages 684–688, 1997.

[54] Paul McKenney and J. D. Slingwine. Read-copy update: using execution history to solve concurrency prolbems. In *Proceedings of International Conference on Parallel and Distributed Computing and Systems*, October 1998.

[55] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, Oregon Health & Science University, 2004.

[56] Paul E. McKenney, Jonathan Appavoo, Audi Kleen, Orran Kreiger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Ontario, Canada, July 2001.

[57] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russel. Read copy update. In *Proceedings of the Ottawa Linux Symposium*, pages 338–367, Ottawa, Ontario, Canada, June 2002.

[58] Philip K. McKinley and Jane W. W. Liu. Group communication in multichannel networks with staircase interconnection topologies. In *Proceedings of ACM SIGCOMM Symposium 1989*, pages 170–181, Austin, Texas, September 1989. ACM.

[59] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113, Williamsburg, VA, USA, 1991. ACM Press.

[60] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *Proceedings of the Fourth Symposium Architectural Support for Programming Languages and Operating Systems*, pages 269–278, Santa Clara, CA, April 1991. ACM.

[61] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, July 2002.

[62] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel And Distributed Systems*, 15(6), June 2004.

[63] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, May 1998.

[64] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–27, January 1993.

[65] Thomas M. Parks, Jose Luis Pino, and Edward A. Lee. A comparison of synchronous and cyclo-static dataflow. In *Proceedings of IEEE Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, November 1995.

[66] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based program. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating System*, pages 5–17, October 2002.

[67] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly & Associates, Inc., second edition, June 2001.

[68] Rusty Russell. Re: 2.4.10pre7aa1. Linux Kernel Mailing List, September 2001.

[69] Rusty Russell. Re: [patch for 2.5] preemptible kernel. Linux Kernel Mailing List, March 2001.

[70] Dipankar Sarma. Re: [patch] per-cpu areas for 2.5.3-pre6. Linux Kernel Mailing List, February 2002.

[71] Dipankar Sarma and Paul E. McKenney. Making rcu safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*, pages 182–191. USENIX Association, June 2004.

[72] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[73] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Symposium on Priniciples of Distributed Computing*, pages 204–213. ACM, August 1995.

[74] Nir Shavit and Asaph Zemach. Diffracting tress. *ACM Transactions on Computer System*, 14(4):385–428, November 1996.

[75] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison Wesley Longman, Inc., fifth edition, November 1998.

[76] Christopher Small and Stephen Manley. A revisitation of kernel synchronization schemes. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, California, 1998.

[77] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Kreiger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proceedings of the Usenix Technical Conference*, pages 141–154, San Antonio, TX, June 2003.

[78] Håkan Sundell, Philippas Tsigas, and Yi Zhang. Simple and fast wait-free snapshots for real-time systems. In *Proceedings of the Fourth International Conference on Principles of Distributed Systems (OPODIS 2000)*, pages 91–106, 2000.

[79] David Tam. Performance analysis and optimization of the hurricane file system on the k42 operating system. Master's thesis, University of Toronto, 2003.

[80] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, pages 54–58, January 1999.

[81] John D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, October 1994.

[82] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 214–222, Ottawa, ON, Canada, August 1995. ACM. Erratum available at ftp://ftp.cs.rpi.edu/pub/valoisj/podc95-errata.ps.gz.

[83] An-I Wang, Geoffrey Kuenning, Peter Reiher, and Gerald Popek. The effects of memory-rich environments on file system microbenchmarks. In *Proceedings of the 2003 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, Montreal, July 2003.

[84] Paul R. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Surveys*, 1992.

[85] Robert W. Wisniewski and Brian Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proceedings of Supercomputing 2003*, Phoenix, AZ, November 2003.

[86] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6):737–753, June 1995.

[87] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

Appendix A

Glossary

Throughout the dissertation, we use various terminology describing the implementation of QDo. We will define these terminologies below:

- Batch - Grouping different callback requests together such that these requests have a common quiescent point. Global batch refers to grouping of requests from all virtual processors together. Local batch refers to grouping of requests from a single virtual processor.

- Callback - Function or method calls that will be executed when the operating system detects global quiescent state is reached.

- Callback Latency - Latency between the point in time when invoking the function to register a QDo callback request to the point in time the callback has completed execution.

- Completed Callbacks - QDo requests of which the callbacks had been executed in all the specified processors. These QDo calls are ready to be deleted.

- Grace Period - Time interval during which all CPUs and tasks have pass through at least one quiescent state [57].

- Incomplete QDo Requests - QDo requests of which the callbacks have not been scheduled to be executed. They may or may not have reached quiescent state.

- Kernel Daemon - A kernel process that runs in the background, without intervention by a user.

- Launched Callbacks - QDo requests of which the callbacks have just been executed by the scheduler. Once all the callback's instructions had been executed, the QDo requests became Completed Callbacks.

- Pending Callbacks - QDo requests of which quiescence have been established. The callbacks can be, but have not yet been, executed.

- Quiescent Latency - Time difference between the moment of placing a callback request to the moment when global quiescence is established.

- Quiescent Point - The first point in time at which it is known that no other threads will be able to access the old data.

- Quiescent State - The system condition, after QP(t) is reached, at which it is known that no threads will be able to access the old data. Global quiescent refers to the condition where it is known that no threads from all virtual processor will be able to access the old data. Local quiescent state refers to the condition where it is known that no threads from the local virtual processor will be able to access the old data.

- Safe Point - Program location at which it is known no threads are accessing a shared data structure.

- Scheduled Callbacks - QDo requests of which quiescent state had been established and the scheduler had scheduled the callbacks to be executed. However, the callbacks have not yet been executed. Once the first processor executes the callback, the requests became Launched Callbacks.

- Tasklet - A mechanism in Linux 2.4 kernel and above that schedules the execution of a kernel task to a later safe time. The tasklet is guaranteed to be executed only once and only on the processor it is first scheduled on [67].