The Dissertation Committee for Wei-Lun Hung
certifies that this is the approved version of the following dissertation:

# Asynchronous Automatic-Signal Monitors with Multi-Object Synchronization

Committee:

Vijay K. Garg, Supervisor

Christine Julien

Sarfraz Khurshid

Neeraj Mittal

Dewayne E Perry

Keshav Pingali

# Asynchronous Automatic-Signal Monitors with Multi-Object Synchronization

**by**

**Wei-Lun Hung, B.S., M.S.**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2016

Dedicated to my family.

# Asynchronous Automatic-Signal Monitors with Multi-Object Synchronization

Wei-Lun Hung, Ph.D.
The University of Texas at Austin, 2016

Supervisor: Vijay K. Garg

One of the fundamental problems in parallel programming is that there is no simple programming paradigm that provides mutual exclusion and synchronization with efficient implementation at the same time. For monitor [Hoa74, Han75] (lock-based) systems, only experienced programmers can develop high-performance fine-grained lock-based implementations. Programmers frequently introduce bugs with traditional monitors. Researchers have proposed transactional memory [HM93, ST95], which provides a simple and elegant mechanism for programmers to atomically execute a set of memory operations so that there is no deadlock in transactional memory systems. However, most of transactional memory systems lack conditional synchronization supports [WLS14, LW14]. Hence, writing multi-threaded programs with conditional synchronization is rather difficult. In this dissertation, we develop a parallel programming framework that provide simple constructs for mutual exclusion and synchronization as well as efficient implementation.

Our framework includes four components. The first part is AutoSynch, which introduces automatic signaling monitor with an efficient implementation. Most programming languages use monitors with explicit signals for synchronization in shared-memory programs. Requiring programmers to signal threads explicitly results in many concurrency bugs due to missed notifications, or notifications on wrong condition variables. By using our `monitor` object and the `waituntil` statement, programmers are able to write simpler parallel programs than before. The second component is ActiveMonitor, which enhances `monitor` objects with asynchronous executions. Tradition montors inhibit parallelism by enforcing serial executions of critical sections, and thus the performance of parallel programs with monitors scales poorly with number of processes. By using our system, programmers can increase the parallelism of their programs without any extra effort. The third part enables multi-object synchronization. We introduce the `multisynch` construct for multi-object mutual exclusion, which lets the system determine the order of locking multiple objects. Furthermore, we allow `waituntil` to take global predicates that across multiple monitor objects. Our method allows efficient monitoring of the conditions without any global lock. The last part introduces logical composition operations, `OR`, `AND`, `selectone`, and `selectall`. With our logical operations, programmers avoid reinventing the wheel since they can easily reuse well-developed concurrent objects together. Our experimental results indicate that our implementations outperform lock-based and transactional memory implementations on most of the test cases.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Multicore hardware is now ubiquitous. Programming these multicore processors is still a challenging task due to bugs resulting from concurrency and synchronization. Although there is widespread acknowledgement of difficulties in programming these systems, it is surprising that by and large the most prevalent methods of dealing with synchronization are based on ideas that were developed in early 70's [Dij68,Hoa74,Han75]. For example, the most widely used threads package in C++ [Str00], pthreads [But97], and the most widely used threads package in Java [GJS$^+$14], java.util.concurrent [Lea05], are based on the notion of monitors [Hoa74, Han75](or semaphores [Dij65, Dij68]). Therefore, we propose a new approach, based on asynchronous automatic signaling monitor that allows gains in productivity of the programmer as well as gain in performance of the system.

In this dissertation, we develop a system and programming paradigms that are simpler than current parallel programming methods by shifting many programming tasks from programmers to our system. These include automatic notification of threads for conditional synchronization, creation of additional

threads for asynchronous execution, multi-object synchronization, and composition operations for monitor objects. By shifting these decisions to system, our goal is to make the parallel programming not only less error prone but also faster. To achieve the goal, our framework includes the following four components.

The first part is AutoSynch, which introduces automatic signaling monitor with an efficient implementation. Most programming languages use monitors with *explicit signals* for synchronization in shared-memory programs. Requiring programmers to signal threads explicitly results in many concurrency bugs due to missed notifications, or notifications on wrong condition variables. AutoSynch eliminates such concurrency bugs by removing the burden of signaling from the programmer.

The second component of our framework is called ActiveMonitor, which improves parallelism by exploiting asynchronous execution of critical sections. The original design of monitor enforces blocking (synchronous) executions for a thread to execute critical sections. Therefore, even if multi-core resources are available, threads are forced to wait for accessing critical sections; thus, the benefit of multi-core devices is limited. ActiveMonitor allow asynchronous execution for monitor objects to increase performance of the overall system.

In the third part of our framework, we deal with multi-object synchronization problems. Current monitor based systems require the programmers to manually determine the order of locking operations, and use global locks or perform busy waiting for operations that depend upon a condition that

spans multiple objects. We propose new monitor based methods that provide automatic signaling for global conditions that span multiple objects. First, we introduce the `multisynch` construct for multi-object mutual exclusion, which lets the system determine the order of locking multiple objects. Second, our system provides automatic notification for global conditions. Assuming that the global condition is a Boolean expression of local predicates, our method allows efficient monitoring of the conditions without any need for global locks.

Finally, our system solves the compositionality problem of monitor systems without requiring global locks. With current programming systems, solving this problem is extremely difficult [HS08]. An ad hoc way to deal with compositionality problem is by using a global lock. But this approach results in slower performance and poor scalability due to the global lock.

We have implemented our constructs on top of Java and have evaluated their overhead. Our results show that on most of the test problems, not only our code is simpler but also faster than Java's reentrant-lock as well as the Deuce transactional memory system.

## 1.2 Automatic-Signal Monitors

For conditional synchronization, both pthreads and Java require programmers to explicitly signal threads that may be waiting on certain condition. The programmer has to explicitly declare condition variables and then signal one or all of the threads when the associated condition becomes true. Using the wrong waiting notification (`signal` versus `signalAll` or notify versus

notifyAll) is a frequent source of bugs in Java multithreaded programs. In AutoSynch, there is no notion of explicit condition variables and it is the responsibility of the system to signal appropriate threads. This feature significantly reduces the program size and complexity. In addition, it allows us to completely eliminate signaling more than one thread resulting in reduced context switches and better performance. The idea of automatic signaling was initially explored by Hoare in [Hoa74], but rejected in favor of condition variables due to efficiency considerations. The belief that automatic signaling is extremely inefficient compared to explicit signaling is widely held since then and all the prevalent concurrent languages based on monitors use explicit signaling. For example, Buhr, Fortier, and Coffin claim that automatic monitors are 10 to 50 times slower than explicit signals [BBF+95]. We show in this dissertation that the widely held belief is wrong. The reason for this drastic slowdown in previous implementations of automatic monitor is that they evaluate all possible conditions on which threads are waiting whenever the monitor becomes available.

With careful analysis of the conditions on which the threads are waiting and evaluating as few conditions as possible, automatic signaling can be as efficient as explicit signaling. In AutoSynch, the programmer simply specifies the predicate $P$ on which the thread is waiting by using the construct `waituntil(`$P$`)` statement. When a thread executes the `waituntil` statement, it checks whether $P$ is true. If it is true, the thread can continue; otherwise, the thread must wait for the system to signal it. The AutoSynch system has a

condition manager that is responsible for determining which thread to signal by analyzing the predicates and the state of the shared object.

```
class BoundedQueue {
  Object[] items;
  int putPtr, takePtr, count;
  Lock mutex = new ReentrantLock();
  Condition notFull = mutex.newCondition();
  Condition notEmpty = mutex.newCondition();
  public BoundedQueue(int n) {
    items = new Object[n];
    putPtr = takePtr = count = 0;
  }
  public void put(Object item) {
    mutex.lock();
    while (count == items.length) {
      notFull.await();
    }
    items[putPtr++] = item;
    putPtr %= items.length;
    ++count;
    notEmpty.signal();
    mutex.unlock();
  }
  public Object take() {
    mutex.lock();
    while (count == 0) {
      notEmpty.await();
    }
    Object x = items[takePtr++];
    takePtr %= items.length;
    --count;
    notFull.signal();
    mutex.unlock();
    return ret;
  }
}
```

Figure 1.1: The bounded-queue using traditional Java

```
1 monitor class BoundedQueue {
2   Object[] items;
3   int putPtr, takePtr, count;
4   public BoundedQueue(int n) {
5     items = new Object[n];
6     putPtr = takePtr = count = 0;
7   }
8   public void put(Object item) {
9     waituntil(count < items.length);
10    items[putPtr++] = item;
11    putPtr %= items.length;
12    ++count;
13  }
14  public Object take() {
15    waituntil(count >= num);
16    Object x = items[takePtr++];
17    takePtr %= items.length;
18    --count;
19    return ret;
20  }
21 }
```

Figure 1.2: The bounded-queue using our approach

Fig. 1.1 and 1.2 show the difference between the Java and the our proposed implementation for the producer-consumer problem, also known as the bounded-buffer problem [Dij65, Dij71]. In this problem, producers put an item into a shared queue, while consumers take an item out of the queue. The `put` function has a parameter `item`; the `take` function has no parameter. There are two requirements for synchronization. First, every operation on a shared variable, such as `items`, should be done under mutual exclusion. Second, we need *conditional synchronization*; a producer must wait when the queue is full, and a consumer must wait when the queue is empty. The explicit-signal bounded-queue is written in Java. Programmers need to explicitly associate conditional predicates with condition variables and call `signal` (`signalAll`) or `await` statement manually. Note that, the `unlock` statement should be done in a `finally` block, `try` and `catch` blocks are also need for the `InterruptedException` that may be thrown by `await`. However, for simplicity, we avoid the exception handling in Fig. 1.1 and 1.2. The automatic-signal bounded-queue is written using our framework. We use `monitor` modifier to indicate that the class is a monitor as in line 1. A `monitor` class provides mutually exclusive access to its member functions. For conditional synchronization, we use `waituntil` as in line 9. There are no `signal` or `signalAll` calls in the our approach. Note that, the `waituntil` statement can take any Boolean condition just like the `if` and `while` statements. Clearly, the automatic-signal monitor is much simpler than the explicit-signal monitor.

In this dissertation, we argue that automatic signaling is generally as

fast as explicit signaling (and even faster for some examples). The explicit signaling has to resort to `signalAll` in some examples; however, our automatic signaling never uses `signalAll`. Thus AutoSynch is considerably faster for synchronization problems that requires `signalAll`. The design principle underlying AutoSynch is to reduce the number of context switches and predicate evaluations.

**Context switch:** A context switch requires a certain amount of time to save and load registers and update various tables and lists. Reducing unnecessary context switches boosts the performance of the system. A `signalAll` call introduces unnecessary context switches; therefore, `signalAll` calls are never used in AutoSynch.

**Predicate evaluation:** In the automatic-signal mechanism, signaling a thread is the responsibility of the system. The number of predicate evaluations is crucial for efficiency in deciding which thread should be signaled. By analyzing the structure of the predicate, our system reduces the number of predicate evaluations.

There are three important novel concepts in AutoSynch that enable efficient automatic signaling — *closure of predicates, relay invariance*, and *predicate tagging*.

The technique of *closure* of a predicate $P$ is used to reduce the number of context switches for its evaluation. In the current systems, only the thread that

is waiting for the predicate $P$ can evaluate it. When the thread is signaled, it wakes up, acquires the lock to the monitor and then evaluates the predicate $P$. If the predicate $P$ is false, it goes back to wait resulting in an additional context switch. In AutoSynch system, the thread that is in the monitor evaluates the condition for the waiting thread and wakes it only if the condition is true. Since the predicate $P$ may use variables local to the thread waiting on it, AutoSynch system derives a closure predicate $P'$ of the predicate $P$, such that other threads can evaluate $P'$. The details of closure are in Section 2.2.

The idea of *relay invariance* is used to avoid `signalAll` calls in AutoSynch. The relay invariance ensures that if there is any thread whose waiting condition is true, then there exists at least one thread whose waiting condition is true and is signaled by the system. With this invariance, the `signalAll` call is unnecessary in our automatic-signal mechanism. This mechanism reduces the number of context switches by avoiding `signalAll` calls. The details of this approach are in Section 2.3.

The idea of *predicate tagging* is used to accelerate the process of deciding which thread to signal. All the waiting conditions are analyzed and tags are assigned to every predicate according to its semantics. To decide which thread should be signaled, we identify tags that are most likely to be true after examining the current state of the monitor. Then we only evaluate the predicates with those tags. The details of predicate tagging are in Section 2.4.

Our experimental results indicate that AutoSynch can significantly improve performance compared to other automatic-signal mechanisms [BH05].

In [BBF+95,BH05] the automatic-signal mechanism is 10-50 times slower than the explicit-signal mechanism; however, AutoSynch is only 2.6 times slower than the explicit-signal mechanism even in the worst case of our experiment results. Furthermore, AutoSynch is 26.9 times faster than the explicit-signal mechanism in the parameterized producer-consumer problem that relies on `signalAll` calls. Besides, the experimental results also show that AutoSynch is scalable; the performance of AutoSynch scales well even if the number of threads increases for many problems studied in the dissertation.

Although the experiment results show that AutoSynch is 2.6 times slower than the explicit-signal mechanism in the worst case, it is still desirable to have automatic signaling. First, automatic signaling simplifies the task of concurrent programming. In explicit-signal monitor, it is the responsibility of programmers to explicitly invoke a `signal` call on some condition variable for conditional synchronization. Using the wrong notification, and signaling a wrong condition variable are frequent sources of bugs. The idea is analogous to automatic garbage collection. Although garbage collection leads to decreased performance because of the overhead in deciding which memory to free, programmers avoid manual memory deallocation. As a consequence, memory leaks and certain bugs, such as dangling pointers and double free bugs, are reduced. Similarly, automatic-signal mechanism consumes computing resources in deciding which thread to be signaled; programmers avoid explicitly invoking `signal` calls. As a result, some bugs, such as using wrong notification and signaling a wrong condition variable, are eliminated. Secondly, in explicit-signal

monitor, the principle of separation of concerns is violated. Any method that changes the state of the monitor must be aware of all the conditions, which other threads could be waiting for, in other methods of the monitor. The intricate relation between threads for conditional synchronization breaks the modularity and encapsulation of programming. Finally, AutoSynch can provide rapid prototyping in developing programs and accelerating product time to market.

## 1.3 Asynchronous Monitor Method Executions

In this section, we present ActiveMonitor, a programming paradigm that provides significant programming ease in writing thread-safe programs as well as improves the runtime performance of these programs by exploiting asynchronous delegated executions on modern multi-core hardware. Most, if not all, programmers follow a standard recipe to implement shared memory parallel programs: they identify the critical sections in the serial implementation of the program, and make them thread-safe in the style of monitors [Hoa74]. Monitors provide dual abstractions: mutual exclusion and synchronization between threads. Their simplicity and elegance of use, and ready availability of mutexes/locks are two key factors behind such a wide adoption of this style. By enforcing serialized executions of critical sections, mutexes trivially guarantee the safety of data. Under high contention scenarios, however, such serialized executions become obvious performance bottleneck. In addition, mutexes force memory fencing due to which latency hiding techniques such as

caching, pre-fetching, and operation re-ordering cannot be exploited. Mutexes are synchronous: a thread invoking an acquire-lock operation must wait for the lock to become available and cannot perform any other useful work in the meanwhile. As a combined effect of all these factors, programs in traditional monitor-style fare poorly in terms of throughput and scalability on multi-core CPUs.

The design of monitors to enforce blocking executions was envisioned in 1970's when saving processor cycles of the single-core CPUs was a primary programming concern. In contrast, not only multi-core processors are now ubiquitous, but they are also significantly cheaper and faster. In order to exploit the multi-core resources, we enhance and allow a monitor object to exist as a thread — hence it becomes an *active* artifact of the program. With this change, method invocations on this monitor object can be delegated [OTY99]. In addition, we allow the monitor thread to execute critical sections *asynchronously*, so that calling threads can return to their local work without waiting for their completion.

Fig. 1.3 shows the actual usage of ActiveMonitor with `monitor`, and `asynchronous` keywords. The `put` method is defined as `asynchronous` since we do not need to wait for the completion of the method call. Hence, threads invoke `put` can continuously execute other tasks. However, `take()` method will be made synchronous by the framework as it returns a value and is not explicitly declared asynchronous. Threads invoke `take` may need its return value to achieve tasks.

```
1  monitor class BoundedQueue {
2    Object[] items;
3    int putPtr, takePtr, count;
4    asynchronous void put(Object item) {
5      waituntil(count < items.length);
6      items [putPtr++] = item;
7      putPtr = putPtr % item.length;
8      ++count;
9    }
10   Object take() {
11     waituntil(count > 0);
12     Object x = items[takePtr++];
13     takePtr = takePtr % item.length;
14     --count;
15     return x;
16   }
17 }
```

Figure 1.3: Bounded-Queue with ActiveMonitor

Our design and implementation integrates seamlessly with current constructs provided by most programming languages, and can thus benefit existing programs with only a handful of syntactic changes. The results of our experimental evaluation on five multi-threading problems show that ActiveMonitor outperforms, by a factor of two or more in some cases, traditional monitor based programs implemented using Java's ReentrantLock [Lea05], and delegation technique [OTY99] on most of these problems. In our current implementation of ActiveMonitor, support for recursive synchronous operations is not currently available, and use of thread dependent variables and functions is restricted. Note that this only disables the asynchronous executions provided by ActiveMonitor and the framework can still be used for such problems. We

discuss these two issues in 3.6.

## 1.4 Multi-Object Synchronization

Multi-core processors are now widely available but parallel programming on these processors is still challenging due to bugs resulting from concurrency and synchronization. The complex synchronization mechanisms and the nondeterministic nature of threads limit productivity of programmers. For example, synchronization on *global conditions* – conditions that span multiple objects – currently either requires complex code by the programmer, or use of a global lock. No current parallel programming paradigm provides simple constructs with efficient performance for multi-object synchronization. For example, transactional memory based systems support multi-object operations but do not support conditional waiting constructs [HMJH05, SR13]. The thread itself needs to recheck every time there is an update of the variables in the transaction. If there are multiple threads waiting on that condition, then each one of them will recheck the condition. Our focus is on efficient detection and signaling exactly one thread. In this dissertation, we propose and describe an implementation of simple constructs for global conditional synchronization in monitor-based systems to improve the productivity of programmers and the performance of the system.

Programmers often introduce deadlocks in parallel programming. Transactional memory [HM93, ST95] provides a simple and elegant mechanism for programmers to atomically execute a set of memory operations so that there

is no deadlock in transactional memory systems. However, there is no similar construct for monitor approaches. In this dissertation, we introduce `multisynch` constructs for avoiding deadlock. Fig. 1.4 shows the deadlock-free implementation for dining philosophers problem by using our `multisynch` construct with monitor objects as parameters. Programmers can access monitor objects in any order without deadlock. Our system automatically guarantees atomicity for the statement. In contrast, programmers need to maintain consistent locking order to avoid deadlock by using traditional monitors.

```
1 public void eat() {
2   multisynch(leftFork, rightFork) {
3     leftFork.pick();
4     rightFork.pick();
5     System.out.println("Philosopher is eating");
6     leftFork.put();
7     rightFork.put();
8   }
9 }
```

Figure 1.4: The Code Snippet of Dining Philosophers Problem

Many applications require certain action to be taken only if a condition that spans multiple objects is true. We call such a condition, a *global condition* or a *global predicate*. Suppose that there are two queues `Q1` and `Q2` in a system such that they are initially empty and a thread can continue its execution only when one of the queues becomes nonempty. Here, the condition `(!Q1.isEmpty() || !Q2.isEmpty())` is a global predicate. Waiting for such a global predicate to become true without continual evaluation is

hard in current systems. If a thread waits on a condition queue associated with `Q1`, then `Q2` may become nonempty and vice-versa. In this example, we would like the thread to be notified when either of the queues becomes nonempty. Since a thread can sleep either in the condition queue associated with `Q1` or with `Q2`, it is impossible to solve this problem using just local locks in current monitor-based programming systems. The current monitor systems would either require a global lock for both queues, or require that the Queue class contain a nonblocking method `isEmpty()`, and then check the conditions of both the queues continually. For this example, we support a construct `waituntil(!Q1.isEmpty() || !Q2.isEmpty())` which requires the system to wake the thread up whenever the global condition becomes true. We give an efficient implementation of this construct.

We extend AutoSynch with multi-object synchronization. Every method of a monitor is a critical section. If programmers need a critical section across multiple monitor objects, they can use the `multisynch` statement, which takes those monitor objects as parameters. Our system ensures that the operations in the statement are executed in a mutually exclusive fashion without any deadlock. If a thread has to wait (block) for a certain global condition to become true, programmers can still use the `waituntil` statement with the condition as an argument. The thread waits if the condition is false and our system will signal it automatically when the condition has become true.

Fig. 1.5 shows the bounded queue implementation that demonstrates the actual usage of `waituntil` statement with global conditions, and the

```
1 monitor class BoundedQueue {
2   public static void takeAndPut(BoundedQueue srcQ, BoundedQueue destQ) {
3     multisynch(srcQ, destQ) {
4       waituntil(!srcQ.isEmpty() && !destQ.isFull());
5       destQ.put(srcQ.take());
6     }
7   }
8 }
```

Figure 1.5: The Bounded Queue Example

`multisynch` statement. In this example, producers put an item into a shared queue, while consumers take an item out of the queue. We use `monitor` modifier to indicate that the class is a monitor as in line 1. A monitor class provides mutually exclusive access to its member methods. The `takeAndPut` method enables a thread to atomically take an item from `srcQ` and put the item in `destQ`. In this method, we use `multisynch` statement in line 4 so that all operations on both the queues in the scope of the `multisynch` statement are done under mutual exclusion. Furthermore, we need global conditional synchronization – a thread must wait when queue `srcQ` is empty or queue `destQ` is full. We use `waituntil` in line 5 for global conditional synchronization.

As another example, consider a pizza store with two types of threads: cooks and suppliers. The cooks loop forever, first waiting for ingredients, and then making a pizza. The cooks may require different ingredients to make different types of pizza. The suppliers also loop forever, producing ingredients when they are insufficient. Since traditional monitor approaches do not support global conditional synchronization, they would rely on a coarse-grained

lock and condition variables to achieve this goal. However, using a coarse-grained lock limits the parallelism since cooks requiring different ingredients would not be able to make their pizzas concurrently. By using our approach, every ingredient can be considered as a monitor object and there is no need for a coarse-grained lock. Fig. 1.6 demonstrates the code snippet for this problem using our constructs. A cook thread waits till it has enough quantity of each of the resources it needs. This is achieved by using the global predicate in `waituntil` statement. Each of the ingredients, cheese, tomato and pepperoni, is a different monitor object and the entire operation is done under `multisynch` to guarantee atomicity.

```
1 multisynch(cheese, tomato, pepperoni) {
2   waituntil(cheese.quantity()>= 6 &&
3       tomato.quantity()>= 3 && pepperoni.quantity()>= 5);
4   cheese.consume(6);
5   tomato.consume(3);
6   pepperoni.consume(5);
7 }
```

Figure 1.6: The Code Snippet of Pizza Store Problem

Thus, our multi-object synchronization monitor provides an alternative parallel programming paradigm to the traditional monitors and transactional memory systems. Our experimental results show that our approach is efficient as well as scalable in synchronization problems that involve global conditions. We believe that our research can complement current parallel programming paradigms and fill the gap between the traditional monitors and the transactional memory systems. Although our discussion on automatic notification has

been from the perspective of monitors, it is equally applicable to transactional memory [ST95, HLR10, SSAT+06]. Techniques implemented in multi-object synchronization can also be used for conditional synchronization in transactional memory.

## 1.5 Logical Compositionality

Our system addresses another important problem with current mechanisms for synchronization called the *compositionality problem* [HS08]. Continuing with the example of two queues, suppose that the programmer wants to delete an item $x$ from any of the nonempty queues `Q1` or `Q2`. Each of the queues is a monitor object and provides a `blocking` method call `take()` that returns an item from the queue. Since the programmer does not know in advance which queue is going to be nonempty, any method call `Q1.take()` or `Q2.take()` may result in thread blocking even though the other queue has an item available. An ad hoc way to implement this functionality is by using a global lock and a nonblocking implementation of `take`. In our system, we provide a construct called `OR` that executes exactly one of its operand task. For this example, the programmer can use the construct as `(x = Q1.take()) OR (x = Q2.take())`. This `OR` construct is a mechanism that takes multiple monitor methods as its argument and executes exactly one of them whenever the *enabling* condition of one of the monitor becomes true. In addition to `OR`, our system also provides `AND`, `selectone`, and `selectall`. Fig. 1.7 shows an example using `OR` and `selectone`. For `putInAQueue`, we use the `OR` construct

so that a producer is able to put an item in `Q1` or `Q2` depending on whichever queue is not full. A producer can put an item in any of the queues from an array of queues by using the `selectone` statement in `putInAnyQueue`.

```
1  monitor class BoundedQueue {
2    public static void putInAQueue(BoundedQueue Q1,
3        BoundedQueue Q2, Object item) {
4      Q1.put(item) OR Q2.put(item);
5    }
6    public static void putInAnyQueue(
7        BoundedQueue[] queues, Object item) {
8      selectone(int i = 0; i < buffs.length; ++i;
9          queues[i].put(item));
10   }
11 }
```

Figure 1.7: The Bounded Queue Example

To evaluate the performance of our composition operations, we implement both synchronous and asynchronous approaches for multicast channels communication. The results highlight the benefit of our synchronous composition operations since they are much faster than asynchronous approaches, global lock implementations, and transaction memory approaches.

## 1.6 Our Framework and Actual Usage

The framework for the implementation is shown if Fig. 1.8. It is composed of a preprocessor and a Java library. The preprocessor translates our proposed constructs into traditional Java code. Our developed techniques were implemented in the Java library, which is responsible for signaling an

appropriate thread for conditional synchronization and managing threads for asynchronous executions.

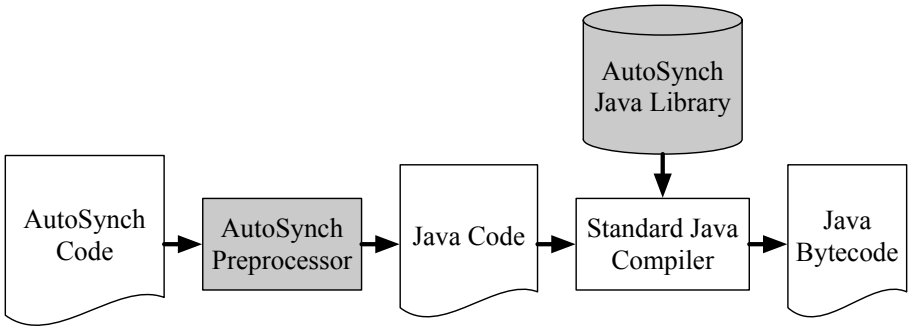


Figure 1.8: The framework of AutoSynch

Using our framework involves the following steps:

1. Programmers write a monitor based parallel program using our keywords, which includes: `monitor`, `waituntil`, `asynchronous`, `synchronous`, and `multisynch`. Note that, the `multisynch` statement involves multiple monitor objects. Furthermore, `waituntil` state can deal with global conditions within the scope of `multisynch` statements. Programmers can use additional operators `OR`, `AND`, `selectone`, and `selectall` for compositionality across multiple monitor objects. Our system automatically manages the use use of locks, and their acquisition/release so that the user is not required to explicitly program them. The user is also free from the responsibility of checking the predicate condition(s) and signaling appropriate threads. The framework observes the values of predicate conditions at runtime, and signals the appropriate threads automatically.

2. Programmers then runs our pre-processor to generate the program's equivalent Java code. The pre-processor injects code snippets to provide the corresponding functionality of framework keywords. The pre-processor also links invocations of our runtime library API in the generated code.

3. The program is then compiled as a standard Java program, and the binaries benefit from asynchronous executions of critical sections, and automatic signaling. If needed, the user can easily disable asynchronous executions at runtime by simply passing a flag.

Although we discuss the key implementation details of our framework and its prototype implementation in Java, our techniques are also applicable to other programming languages and models, such as pthread and C# [HWG03].

## 1.7 Overview

This dissertation is organized as follows. Chapter 2 presents design and implementation of our automatic signaling monitor. Chapter 3 demonstrates asynchronous executions of monitor tasks. In Chapter 4, we discusses multi-object synchronization. Our logical composition operands for monitor objects are shown in Chapter 5. Chapter 6 discuss the future work.

# Chapter 2

# Automatic-Signal Monitors

In this chapter, we demonstrate AutoSynch, which achieves efficiency in automatic signaling synchronization based on three novel ideas. We introduce an operation called *closure* that enables the predicate evaluation in every thread, thereby reducing context switches during the execution of the program. Secondly, AutoSynch avoids `signalAll` by using a property called *relay invariance* that guarantees that whenever possible there is always at least one thread whose condition is true which has been signaled. Finally, AutoSynch uses a technique called *predicate tagging* to efficiently determine a thread that should be signaled. To evaluate the efficiency of AutoSynch, we have implemented many different well-known synchronization problems such as the producers/-consumers problem, the readers/writers problems, and the dining philosophers problem. The results show that AutoSynch is almost as efficient as the explicit-signal monitor and even more efficient for some cases.

The following motivating example demonstrates the simplicity of automatic signaling monitors. Fig. 2.1 and 2.3 show the difference between the Java and the our proposed implementation for the parameterized producer-consumer problem, a variant producer-consumer problem (also known as the

```
1 class BoundedQueue {
2   Object[] items;
3   int putPtr, takePtr, count;
4   Lock mutex = new ReentrantLock();
5   Condition insufficientSpace = mutex.newCondition();
6   Condition insufficientItem = mutex.newCondition();
7   public BoundedQueue(int n) {
8     items = new Object[n];
9     putPtr = takePtr = count = 0;
10  }
11  public void put(Object[] objs) {
12    mutex.lock();
13    while (objs.length + count > items.length) {
14      insufficientSpace.await();
15    }
16    for (int i = 0; i < items.length; i++) {
17      items[putPtr++] = objs[i];
18      putPtr %= items.length;
19    }
20    count += objs.length;
21    insufficientItem.signalAll();
22    mutex.unlock();
23  }
24  public Object[] take(int num) {
25    mutex.lock();
26    while (count < num) {
27      insufficientItem.await();
28    }
29    Object[] ret = new Object[num];
30    for (int i = 0; i < num; i++) {
31      ret[i] = items[takePtr++];
32      takePtr %= items.length;
33    }
34    count -= num;
35    insufficientSpace.signalAll();
36    mutex.unlock();
37    return ret;
38  }
39 }
```

Figure 2.1: The parameterized bounded-queue using traditional Java

```
1 monitor class BoundedQueue {
2   Object[] items;
3   int putPtr, takePtr, count;
4   public BoundedQueue(int n) {
5     items = new Object[n];
6     putPtr = takePtr = count = 0;
7   }
8   public void put(Object[] objs) {
9     waituntil(count + objs.length <= items.length);
10    for (int i = 0; i < objs.length; i++) {
11      items[putPtr++] = objs[i];
12      putPtr %= items.length;
13    }
14    count += objs.length;
15  }
16  public Object[] take(int num) {
17    waituntil(count >= num);
18    Object[] ret = new Object[num];
19    for (int i = 0; i < num; i++) {
20      ret[i] = items[takePtr++];
21      takePtr %= items.length;
22    }
23    count -= num;
24    return ret;
25  }
26 }
```

Figure 2.2: The parameterized bounded-queue using our approach

bounded-buffer problem) [Dij65, Dij71]. In this problem, producers put items into a shared queue, while consumers take items out of the queue. The `put` function has a parameter `items`; the `take` function has a parameter, `num`, indicating the number of items taken. There are two requirements for synchronization. First, every operation on a shared variable, such as `buff`, should be done under mutual exclusion. Second, we need *conditional synchronization*; a producer must wait when the queue does not have sufficient space, and a consumer must wait when the queue has no sufficient items. The explicit-signal bounded-queue is written in Java. Programmers need to explicitly associate conditional predicates with condition variables and call `signal` (`signalAll`) or `await` statement manually. Note that, the `unlock` statement should be done in a `finally` block, `try` and `catch` blocks are also need for the `InterruptedException` that may be thrown by `await`. However, for simplicity, we avoid the exception handling in Fig. 2.1 and 2.3. The automatic-signal bounded-queue is written using our framework. We use `monitor` modifier to indicate that the class is a monitor as in line 1. An `monitor` class provides mutually exclusive access to its member functions. For conditional synchronization, we use `waituntil` as in line 9. There are no `signal` or `signalAll` calls in the our approach. Note that, the `waituntil` statement can take any Boolean condition just like the `if` and `while` statements. Clearly, the automatic-signal monitor is much simpler than the explicit-signal monitor. Furthermore, the experimental results indicates that our approach is faster than the traditional monitor implementation since our system avoids `signalAll` calls.

## 2.1 Background: Monitors

According to Buhr and Harji [BH05], monitors can be divided into two categories according to the different implementations of conditional synchronization.

**Explicit-signal monitor** In this type of monitor, condition variables, `signal` and `await` statements are used for synchronization. Programmers need to associate assertions with condition variables manually. A thread waits on some condition variable if its predicate is not true. When another thread detects that the state has changed and the predicate is true, it explicitly signals the appropriate condition variable.

**Automatic-signal (implicit-signal) monitor** This kind of monitor uses `waituntil` statements, such as line 9 in automatic-signal program in Fig. 2.3, instead of condition variables for synchronization. Programmers do not need to associate assertions with variables, but use `waituntil` statements directly. In monitor, a thread will wait as long as the condition of a `waituntil` statement is false, and execute the remaining tasks only after the condition becomes true. The responsibility of signaling a waiting thread is that of the system rather than of the programmers.

We note that the `signalAll` call is essential in explicit-signal mechanism when programmers do not know which thread should be signaled. In Fig. 2.1, a producer must wait if there is no space to put `num` items, while a

```
1 monitor class BoundedQueue {
2   Object[] items;
3   int putPtr, takePtr, count;
4   public BoundedQueue(int n) {
5     items = new Object[n];
6     putPtr = takePtr = count = 0;
7   }
8   public void put(Object[] objs) {
9     waituntil(count + objs.length <= items.length);
10    for (int i = 0; i < objs.length; i++) {
11      items[putPtr++] = objs[i];
12      putPtr %= items.length;
13    }
14    count += objs.length;
15  }
16  public Object[] take(int num) {
17    waituntil(count >= num);
18    Object[] ret = new Object[num];
19    for (int i = 0; i < num; i++) {
20      ret[i] = items[takePtr++];
21      takePtr %= items.length;
22    }
23    count -= num;
24    return ret;
25  }
26 }
```

Figure 2.3: The parameterized bounded-queue using AutoSynch

consumer has to wait when the buffer has insufficient items. Since producers and consumers can put and take different numbers of items every time, they may wait on different conditions to be met. Programmers do not know which producer or consumer should be signaled at runtime. Therefore, the `signalAll` call is used instead of `signal` calls in line 21 and 35. Although programmers can avoid using `signalAll` calls by writing complicated code that associates different conditions to different condition variables; the complicated code makes the maintenance of the program hard.

The `signalAll` call is expensive; it generally decreases the performance because it introduces redundant context switches, requiring computing time to save and load registers and update various tables and lists. Furthermore, `signalAll` calls cannot increase parallelism because threads are forbidden to access a monitor simultaneously. Although multiple threads are signaled at a time, only one thread is able to acquire the monitor. Other threads may need to go back to waiting state since another thread may change the status of the monitor.

## 2.2 Predicate Evaluation

In AutoSynch, it is the responsibility of the system to signal appropriate threads automatically. The predicate evaluation is crucial in deciding which thread should be signaled. We discuss how to preform predicate evaluations of `waituntil` statements.

A predicate $P(\vec{x}) : X \rightarrow \mathbb{B}$ is a Boolean condition, where $X$ is the space

spanned by the variables $\vec{x} = (x_1, \ldots, x_n)$. A variable of a monitor object is a shared variable if it is accessible by every thread that is accessing the monitor. The set of shared variables is denoted by $S$. The set of local variables, denoted by $L$, is accessible only by a thread calling a function in which the variables are declared.

Predicates can be used to describe the properties of conditions. In our approach, every condition of `waituntil` statement is represented by a predicate. We say a condition has been met if its representing predicate is true; otherwise, the predicate is false. Furthermore, we assume that every predicate, $P = \vee_{i=1}^{n} c_i$, is in disjunctive normal form (DNF), where $c_i$ is defined as the conjunction of a set of atomic Boolean expressions. For example, a predicate $(x = 1) \wedge (y = 6) \vee (z \neq 8)$ is in DNF, where $c_1 = (x = 1) \wedge (y = 6)$ and $c_2 = (z \neq 8)$. Note that, every Boolean formula can be converted into DNF using De Morgan's laws and distributive law.

Predicates can be divided into two categories based on the type of their variables [BH05].

**Definition 1** (Shared and complex predicate)**.** *Consider a predicate* $P(\vec{x}) : X \rightarrow \mathbb{B}$. *If* $X \subseteq S$, $P$ *is a shared predicate. Otherwise, it is a complex predicate.*

The automatic-signal monitor has an efficient implementation [Kes77] by limiting the predicate of a `waituntil` to a shared predicate; however, we do not limit the predicate of a `waituntil` statement to a shared predicate.

The reason is that this limitation will lead AutoSynch to be less attractive and practical since conditions including local variables cannot be represented in AutoSynch.

Evaluating a complex predicate in all the threads is not feasible because the accessibility of the local variables in the predicate is limited to the thread declaring them. To evaluate a complex predicate in all the threads, we treat local variables as constant values at runtime and define closure as follows.

**Definition 2** (Closure)**.** *Given a complex predicate* $P(\vec{x}, \vec{a}) : X \times A \rightarrow \mathbb{B}$*, where* $X \subseteq S$ *and* $A \subseteq L$*. The closure of* $P$ *at runtime* $\boldsymbol{t}$ *is the new shared predicate*

$$G_t(\vec{x}) = P(\vec{x}, \vec{a_t}),$$

*where* $\vec{a_t}$ *is the values of* $\vec{a}$ *at runtime* $\boldsymbol{t}$*.*

The closure can be applied to any complex predicate; a shared predicate can be derived from the closure. For example, in Fig. 2.3, the consumer $C$ wants to take 48 items at some instant of time. Applying the closure to the complex predicate `(count` $\geq$ `num)` in line 19, we derive the shared predicate `(count` $\geq$ `48)`.

**Definition 3** (Waituntil Period)**.** *Given a thread* $T$ *waiting on predicate* $P$*. The waituntil period indicates the duration that* $T$ *waits on* $P$*; that is, between the first time that* $T$ *evaluates* $P$ *as false, and the time that* $T$ *awakes up and evaluates* $P$ *as true.*

The following proposition shows that the complex predicate evaluation of `waituntil` statement in all threads can be achieved through the closure.

**Proposition 1.** *Consider a complex predicate $P(\vec{x}, \vec{a})$ in a **waituntil** statement. $P(\vec{x}, \vec{a})$ and its closure $P(\vec{x}, \vec{a_t})$ are semantically equivalent during the **waituntil** period, where $t$ is the time instant immediately before invoking the **waituntil** statement.*

*Proof.* Only the thread invoking the `waituntil` statement can access the local variables of the predicate; all other threads are unable to change the values of those local variables. Therefore, the value of $\vec{a}$ cannot change during the `waituntil` period. Since $\vec{a_t}$ is the value of $\vec{a}$ immediately before invoking the `waituntil` statement, $P(\vec{x}, \vec{a})$ and $P(\vec{x}, \vec{a_t})$ are semantic equivalent during the `waituntil` period. □

Proposition 1 enables the complex predicate evaluation of `waituntil` statement in all threads. Given a complex predicate in a `waituntil` statement, in the sequel we substitute all the local variables with their values immediately before invoking the statement. The predicate can now be evaluated in all other threads during the `waituntil` period.

## 2.3 Relay Invariance

As mentioned in Section 2.1, `signalAll` calls are sometimes unavoidable in the explicit-signal mechanism. In AutoSynch, `signalAll` calls are avoided by providing the *relay invariance*.

**Definition 4** (Active and inactive thread)**.** *Consider a thread that tries to access a monitor. If it is not waiting in a* `waituntil` *statement or has been signaled, then it is an* active *thread for the monitor. Otherwise, it is an* inactive *thread.*

**Definition 5** (Relay invariance)**.** *If there is a thread waiting for a predicate that is true, then there is at least one active thread; i.e., suppose $W_T$ is the set of waiting threads whose conditions have become true, $A_T$ is the set of active threads, then*

$$W_T \neq \phi \Rightarrow A_T \neq \phi$$

*holds at all time.*

AutoSynch uses the following mechanism for signaling.

*Relay signaling rule*: When a thread exits a monitor or goes into waiting state, it checks whether there is some thread waiting on a condition that has become true. If at least one such waiting thread exists, it signals that thread.

**Proposition 2.** *The relay signaling rule guarantees relay invariance.*

*Proof.* Suppose a thread $T$ is waiting on the predicate $P$ that is true. Since $T$ is waiting on $P$, $P$ must be false before $T$ went to waiting state. There must exist another active thread $R$ after $T$ such that $R$ changed the state of the monitor and made $P$ true. According to the rule, $R$ must signal $T$ or another thread waiting for a condition that is true before leaving the monitor or going

into waiting state. The thread signaled by $R$ then becomes active. Therefore, the relay invariance holds.

□

Our framework guarantees progress by providing *relay invariance*. The concept behind relay invariance is that, the privilege to enter the monitor is transmitted from one thread to another thread whose condition has become true. For example, in Fig. 2.3, the consumer $C$ tries to take 32 items; however, only 24 items are in the buffer at this moment. Then, $C$ waits for the predicate $P$: `(count` $\geq$ `32)` to be true. A producer, $D$, becomes active after $C$; $D$ puts 16 items into the buffer and then leaves the monitor. Before leaving, $D$ finds that $P$ is true and then signals $C$; therefore, $C$ becomes active again and takes 32 items of the buffer. Proposition 2 shows that the relay invariance holds in our automatic-signaling mechanism. Thus, `signalAll` calls are avoided in AutoSynch. Note that, although at most one thread is signaled at any time; the signaled thread is not guaranteed to get the lock. Some other thread trying to acquire the lock could also get the lock. The signaled thread may need to go back as a waiting thread, since the state of the monitor may have changed. However, this situation is rare in comparison with the `signalAll` call. The problem is now reduced to finding a thread waiting for a condition that is true.

## 2.4 Predicate Tag

In order to efficiently find an appropriate thread waiting for a predicate that is true, we analyze every waiting condition and assign different tags to every predicate according to its semantics. These tags help us prune predicates that are not true by examining the state of the monitor. The idea behind the predicate tag is that, local variables cannot be changed during the `waituntil` period; thus the values of local variables are used as keys when we evaluate predicates. First, we define two types of predicates according to their semantics.

**Definition 6** (Local and shared expression)**.** *Consider an expression* $E(\vec{x}) : X \to \mathbb{D}$*, where* $\mathbb{D}$ *represents one of the primitive data types in Java. If* $X \subseteq L$*, then* $E$ *is a local expression. Otherwise, if* $X \subseteq S$*,* $E$ *is a shared expression.*

We use $SE$ to denote a shared expression, and $LE$ to denote a local expression.

**Definition 7** (Equivalence predicate)**.** *A predicate* $P : (SE = LE)$ *is an equivalence predicate.*

**Definition 8** (Threshold predicate)**.** *A predicate* $P : (SE \ \boldsymbol{op} \ LE)$ *is a threshold predicate, where* $\boldsymbol{op} \in \{<, \leq, >, \geq\}$.

Note that, many predicates that are not equivalence or threshold predicates can be transformed into them. Consider the predicate $(x - a = y + b)$, where $x, y \in S$ and $a, b \in L$. This predicate is equivalent to $(x - y = a + b)$

which is an equivalence predicate. Thus, these two types of predicates can represent a wide range of conditions in synchronization problems.

Given an *Equivalence* or a *Threshold* predicate, we can apply the *closure* operation to derive a constant value on the right hand side of the predicate. In AutoSynch, there are three types of tags, *Equivalence*, $Threshold$, and *None*. Every *Equivalence* or $Threshold$ tag represents an equivalence predicate or a threshold predicate, respectively. If the predicate is neither equivalence nor threshold, it acquires the *None* tag. For example, consider the $Threshold$ predicate $x+b > 2y+a$ where $a$ and $b$ are local variables with values 11 and 2. We first use the closure to convert it to $(x - 2y > 9)$, which is represented by the tag $(Threshold,\ x-2y,\ 9,\ >)$. The formal definition of tag is as follows.

**Definition 9.** *A tag is a four-tuple* $(M,\ expr,\ key,\ op)$*, where*

- $M \in \{Equivalence,\ Threshold,\ None\}$*;*
- *expr is a shared expression if* $M \in \{Equivalence,\ Threshold\}$*; otherwise,* $expr = \perp$*;*
- *key is the value of a local expression after applying closure if* $M \in \{Equivalence,\ Threshold\}$*; otherwise,* $key = \perp$*;*
- $op \in \{<,\ \leq,\ >,\ \geq\}$ *if* $M = Threshold$*; otherwise,* $op = \perp$*.*

We say that a tag is true (false) if the predicate representing the tag is true (false).

### 2.4.1 Predicate Tagging

Tags are given to every predicate by the algorithm shown in Algorithm 1. A tag is assigned to every conjunction. The tags of conjunctions of a predicate constitute the set of tags of the predicate. When assigning a tag to a conjunction, the equivalence tag has the highest priority because the set of values that make an equivalence predicate true is smaller than the set of values that make a threshold predicate true. For example, the equivalence predicate $x = 8$ is true only when the value of $x$ is 8, whereas the threshold predicate $x > 3$ is true for a much larger set of values. Therefore, the *Equivalence* tags can help us prune predicates that are false more efficiently than other kinds of tags. If a conjunction does not include any equivalence predicate, then we check whether it includes any threshold predicate. If yes, then a *Threshold* tag is assigned to the conjunction; otherwise, the conjunction has a *None* tag.

**Algorithm 1** `predicateTagging(`$P$`)`

**Input:** A DNF predicate $P$
**Output:** Returns tags for $P$

```
 1: tags.empty()
 2: for each conjunction C of P do
 3:   if C contains an equivalence predicate se = le then
 4:     Tag t = null
 5:     t = (Equivalence, se, closure(le), null)
 6:   else if C contains a threshold predicate seople then
 7:     t = (Threshold, se, closure(le), op)
 8:   else
 9:     t = (None, null, null, null)
10:   tags.add(t)
```

Creating all tags for a conjunction is unnecessary. If a conjunction

includes multiple equivalence predicates or threshold predicates, only one arbitrary *Equivalence* tag or *Threshold* tag is assigned to the conjunction. If there are a large number of tags, then the performance may decrease because of the cost of maintaining tags. Assigning multiple tags to a conjunction cannot accelerate the searching process. For example, consider a conjunction $(x = 8) \wedge (y = 9)$. If only a tag $(Equivalence,\ x,\ 8,\ null)$ is assigned to the conjunction, we check the predicate when the tag is true. Adding another tag $(Equivalence,\ y,\ 9,\ null)$ cannot accelerate the searching process since we need to check both the tags. Note that multiple predicates with a shared conjunct may share a tag. For example, the predicates $(x = 5) \wedge (z \leq 4)$ and $(x = 5) \wedge (y \geq 4)$ would have a shared equivalence tag of $(x = 5)$.

As another example, consider the predicate $p = ((x < 5) \wedge (y = 3)) \vee ((x > 5) \wedge (foo2())) \vee foo1()$, where $x$ and $y$ are shared variables, and, $foo1()$ and $foo2()$ are boolean functions. The predicate $p$ has three tags, $(Equivalence,\ y,\ 3,\ null)$ for the clause $(x < 5) \wedge (y = 3)$, $(Threshold,\ x, 5, >)$ for the clause $(x > 5) \wedge (foo2())$, and *None* tag for $foo1()$.

### 2.4.2 Tag Signaling

Signaling mechanism is based on tags in AutoSynch. Since the equivalence tag is more efficient in pruning the search space than the threshold tag, the predicates with equivalence are checked prior to the predicates with other tags. If no true predicate is found after checking *Equivalence* tags and *Threshold* tags, our algorithm does the exhaustive search for the predicates with a

*None* tag.

**Equivalence tag signaling:** Observe that, an equivalence predicate becomes true only when its shared expression equals the specific value of its local expression after applying *closure*. For distinct equivalence tags related to the same shared expression, at most one tag can be true at a time because the value of its local expression is deterministic and unique at any time. By observing the value of its local expression, the appropriate tag can be identified. For example, suppose there are three *Equivalence* tags for predicates $x = 3$, $x = 6$, and $x = 8$. We examine $x$ and find that its value is 8. Then we know that only the third predicate $x = 8$ is true. Based on this observation, for each unique shared expression of an equivalence tag, we create a hash table, where the value of the local expression is used as the key. By using this hash table and evaluating the shared expression at runtime, we can find a tag that is true in $O(1)$ time if there is any. Then we check the predicates having the tag.

**Threshold tag signaling:** Threshold tag signaling exploits monotonicity of the predicate to reduce the complexity of evaluating predicates. For example, suppose there are two predicates, $x > 5$ and $x > 3$. We know that if $x > 3$ is false, then $x > 5$ cannot be true. Hence, we only need to check the predicate with the smallest local expression value for $>$ and $\geq$ operations. Similarly, the predicate $x > 3$ cannot be true when $x \geq 3$ is false; i.e., we only need to

check the predicate $x \geq 3$. We use a min-heap data structure for storing the threshold tags related to a same shared expression with $\boldsymbol{op} \in \{>, \geq\}$. If two predicates have the same local expression value but different operations, then the predicate with $\geq$ is considered to have a smaller value than the predicate with $>$ in the min-heap. Dually, the max-heap is used for threshold tags with $\boldsymbol{op} \in \{<, \leq\}$.

The signaling mechanism for *Threshold* tag is shown if Algorithm 2. In general, the tag in the root of a heap is checked. If the tag is false, all the descendant nodes are also false. Otherwise, all predicates having the tag need to be checked for finding a true predicate. To maintain the correctness, if no predicate is true, the tag is removed from the heap temporarily. Then the tag in the position of the new heap root is checked again until a true predicate is found or a false tag is found. Those tags removed temporarily are reinserted in the heap. The reason to remove the tags is that the descendants of the tags may also be true since the tags are true. So we also need to check the descendant tags. For example, consider the predicates $P_1 : (x \geq 5) \wedge (y \neq 1)$ and $P_2 : (x > 7)$. $P_1$ has the tag $Q_1 : (Threshold,\ x,\ 5,\ \geq)$ and $P_2$ has the tag $Q_2 : (Threshold,\ x,\ 7,\ >)$. $Q_1$ is the root and $Q_2$ is its descendant. Suppose at some time instant $x = 3$, then $Q_1$ is false; thus, there is no need to check $Q_2$. Now, suppose $x = 9$ and $y = 1$, then $Q_1$ is true. We check all predicates that have tag $Q_1$. Since $P_1$ is false, no predicate having tag $Q_1$ is true. Then $Q_1$ is removed form the heap temporarily. We find the new root $Q_2$ is true and $P_2$ that has tag $Q_2$ is also true. We signal a thread waiting for $P_2$ and

then add $Q_1$ back to the heap.

**Algorithm 2** `thresholdTagSignaling()`

**Input:** A DNF predicate $P$ and the `heap` of its threshold `tags`

```
 1: backup.empty()
 2: Tag t = heap.peek()                    ▷ retrieve but not remove the root
 3: while t is true do
 4:    for each predicate P with t do
 5:       if P is true then
 6:          signal a thread waiting on P
 7:          for each b ∈ backup do
 8:             backup.add(b)
 9:          return
10:    backup.add(heap.poll())              ▷ retrive and remove the root
11:    t = heap.peek()
12: for each b ∈ backup do
13:    heap.add(b)
```

Suppose there are $n$ *Threshold* tags for a shared expression with different keys, and these tags are assigned to $m$ predicates. The time complexity for maintaining the heap is $O(n \log(n))$ However, the performance is generally much better because we only need to check the predicates of the tags in the root position in the most cases. The time complexity for finding the root is $O(1)$. In the worst case, we need to check all predicates; thus, the time complexity is $O(n \log(n) + m)$. However, this situation is rare. Furthermore, this algorithm is optimized for evaluating threshold predicates by sacrificing performance in tag management.

## 2.5 Evaluation

We present the experimental setup and its results to evaluate the performance of AutoSynch in this section. We compare the performance of different signaling mechanisms in three sets of classical conditional synchronization problems. The first set of problems relies on only shared predicates for synchronization. Next, we explore the performance for problems using complex predicates. Finally, we evaluate the problems in which `signalAll` calls are required in the explicit-signal mechanism.

### 2.5.1 Experimental environment

All of the experiments were conducted on a machine with 16 Intel(R) Xeon(R) X5560 Quad Core CPUs (2.80 GHz) and 64 GBs memory running Linux 2.6.18.

We conducted two types of experiments. The first is a *saturation* test [BH05], in which only monitor accessing functions performed. That is, no extra work is performed in the monitor or out of the monitor. The other set of experiments simulate different workloads of the monitors [BBF+95]. For each monitor operation, there is a fixed time to perform other operations out of the monitor. For every experiment, we ran the program 25 times, and removed the best and the worst results. Then we compared the average runtime for different signaling mechanisms.

We do not report memory usage due to space limitations. Although some additional data structures are created in our framework, the additional

memory consumption is insignificant. The reason is that, whenever a predicate has no waiting thread, it is put in an inactive list for reuse. If the size of the inactive list exceeds a predefined threshold, e.g. $2n$ ($n$ is the number of threads), then we remove the oldest predicate and the conditional variable from the list and the table. Moreover, the size of active predicates is always less than $n$.

### 2.5.2 Signaling mechanisms

Four implementations using different signaling mechanisms have been compared.

**Explicit-signal** Using the original Java explicit-signal mechanism.

**Baseline** Using the automatic-signal mechanism relying on only one condition variable. It calls `signalAll` to wake every waiting thread. Then each thread that wakes up re-evaluates its own predicate after re-acquiring the monitor.

**AutoSynch-T** Using the approach described in this chapter but excluding predicate tagging.

**AutoSynch** Using the approach described in this chapter.

### 2.5.3 Test problems

Seven conditional synchronization problems are implemented for evaluating our approach.

#### 2.5.3.1 Shared predicate synchronization problems

**Bounded-buffer [Dij65, Dij71]** This is the traditional bounded-buffer problem. Every producer waits if the buffer is full, while every consumer waits if the buffer is empty.

**$H_2O$ problem [And99]** This is the simulation of water generation. Every $H$ atom waits if there is no $O$ atom or another $H$ atom. Every $O$ atom waits if the number of $H$ atoms is less than 2. The code snippets are shown in Fig. A.1.

#### 2.5.3.2 Complex predicate synchronization problems

**Round-Robin Access Pattern** Every test thread accesses the monitor in round-robin order. The code snippets are shown in Fig. A.2.

**Readers/Writers [CHP71]** We use the approach given in [BH05], where a ticket is used to maintain the accessing order of readers and writers. Every reader and writer gets a ticket number indicating its arrival order. Readers and writers wait on the monitor for their turn. The code snippets are shown in Fig. A.3.

**Dining philosophers [Dij71]** This problem requires coordination among philosophers sitting around a table and is described in [Dij71].

#### 2.5.3.3 Synchronization problems requiring `signalAll` in explicit

**Parameterized bounded-buffer [Dij65, Dij71]** The parameterized bounded-buffer problem shown in Fig. 2.3.

### 2.5.4 Experimental results

Fig. 2.4 and 2.5 plot the results for the bounded-buffer and the $H_2O$ problem. The y-axis shows the runtime in seconds. The x-axis represents the number of simulating threads. Note that, in the $H_2O$ problem, only one thread simulates an $O$ atom. The x-axis represents the number of threads simulating $H$ atoms. As expected, the baseline is much slower than other three signaling mechanisms, which have similar performance in the both problems. This phenomenon can be explained as follows. There is only a constant number of shared predicates in `waituntil` statements for automatic-signal mechanisms. For example, in the bounded-buffer problem, there are two `waituntil` statements with global predicates, $count > 0$ (not empty condition) and $count < buff.length$ (not full condition). Therefore, the complexity for signaling a thread in AutoSynch and AutoSynch-T is also constant. Hence, both AutoSynch and AutoSynch-T are as efficient as the explicit-signal mechanism. These experiments illustrate that the automatic-signal mechanisms are as efficient as the explicit-signal mechanisms for synchronization problems relying on only shared predicates.

Fig. 2.6, 2.7 2.8 present the experimental results for the round-robin access pattern, the readers/writers problem, and the dining philosophers prob-

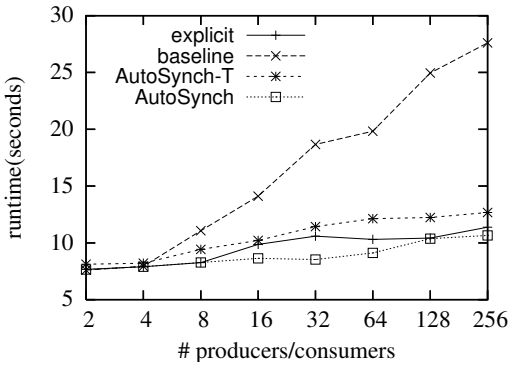


Figure 2.4: The results of bounded-buffer problem

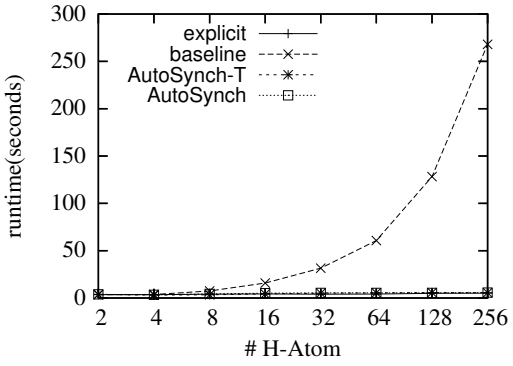


Figure 2.5: The results of $H_2O$ problem

lem. The result of the baseline is not plotted in these figures since its performance is extremely inefficient in comparison with other mechanisms. In this set of experiments, the explicit-signal mechanism has an advantage since it can explicitly signal the next thread to enter the monitor. For example, in the round-robin access pattern, an array of condition variables is used for associating the id of each thread and its condition variable. Each thread waits on its condition variable until its turn. When a thread leaves the monitor, it signals the condition variable of the next thread. As can be seen, the

performance of explicit-signal mechanism is steady as the number of threads increases in the round-robin access pattern and the reader/writers problem. In AutoSynch-T, its runtime increases significantly as the number of threads increase. For AutoSynch, the performance is between 1.2 to 2.6 times slower than the explicit-signal mechanism for the round-robin access pattern. However, the performance of AutoSynch does not decrease as the number of threads increases. Note that, in the readers/writers problem, the AutoSynch-T is more efficient than AutoSynch when the number of threads is small. The reason is that AutoSynch sacrifices performance for maintaining predicate tags. The benefit of predicate tagging increases as the number of threads increases. Another interesting point is that the performance of the explicit signal mechanism does not outperform implicit signal mechanisms much in the dining philosophers problem. The reason is that a philosopher only competes with two other philosophers sitting near him even when the number of philosophers increases.

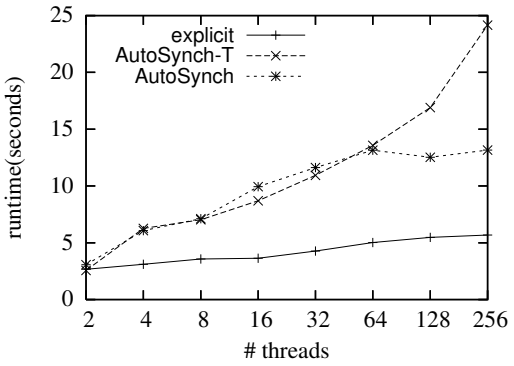


Figure 2.6: The results of round-robin access pattern

Table 2.1 presents the CPU usage (profiled by YourKit [you]) for the

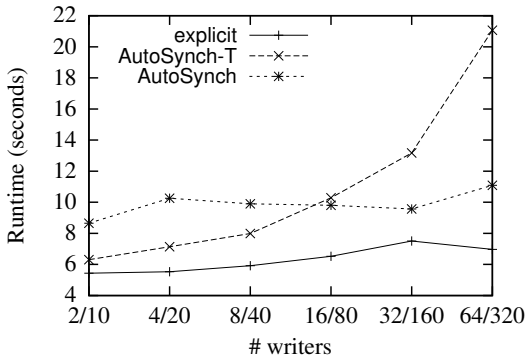


Figure 2.7: The results of readers/writers problem

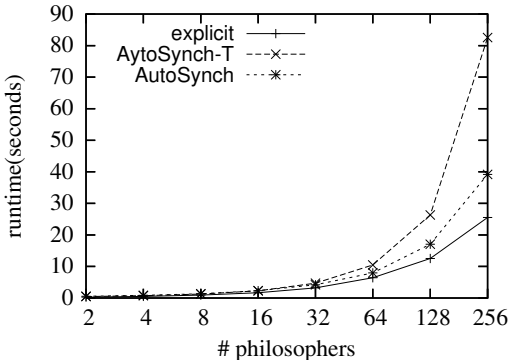


Figure 2.8: The results of dining philosophers problem

round-robin access pattern with 128 threads. The *relay signal* is the process of deciding which thread should be signaled in both AutoSynch and AutoSynch-T. *Tag Mger* is the computation for maintaining predicate tags in AutoSynch. As can be seen, the predicate tagging significantly improves the process for finding a predicate that is true. The CPU time of *relaySingal* process is reduced 95% with a slightly increased cost in tag management.

In Fig. 2.9, we compare the results of the parameterized bounded-buffer in which `signalAll` calls are required in the explicit-signal mechanism. In

| | await | | lock | | relay signal | |
|---|---|---|---|---|---|---|
| | T | % | T | % | T | % |
| explicit | 21365 | 99.7% | 28 | 0.15% | $NA$ | $NA$ |
| AutoSynch-T | 410377 | 98.5% | 3140 | 0.7% | 2108 | 0.5% |
| AutoSynch | 96754 | 98.8% | 812 | 0.8% | 112 | 0.1% |

| | Tag Mger | | others | | total |
|---|---|---|---|---|---|
| | T | % | T | % | T |
| explicit | $NA$ | $NA$ | 28 | 0.15% | 21433 |
| AutoSynch-T | $NA$ | $NA$ | 1033 | 0.2% | 416658 |
| AutoSynch | 124 | 0.1% | 148 | 0.02% | 97950 |

Table 2.1: The CPU usage for the round robin access pattern

this experiment, there is one producer, which randomly puts 1 to 128 items every time. The y-axis indicates the number of consumers. Every consumer randomly takes 1 to 128 items every time. As can be seen, the performance of the explicit-signal mechanism decreases as the number of consumers increases. AutoSynch outperforms the explicit-signal mechanism by 26.9 times when the number of threads is 256. This can be explained by Fig. 2.10 that depicts the number of contexts switches. The number of context switches increases in the explicit-signal mechanism in which the number of context switches is around 2.7 million when the number of threads is 256. However, the numbers of context switches are stable in *AutoSynch* even when the number of threads increase. It has around 5440 context switches when the number of threads is 256. This experiment demonstrates that the number of context switches can be dramatically reduced and the performance can be increased in AutoSynch for the problems requiring `signalAll` calls in the explicit-signal mechanism.

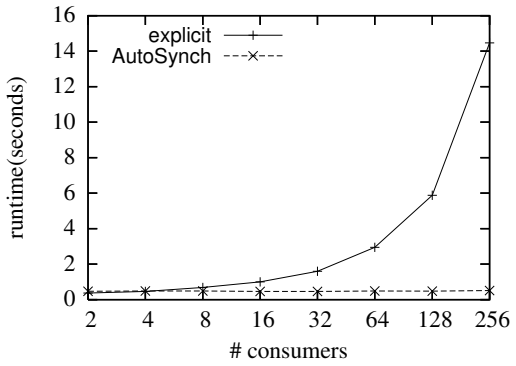


Figure 2.9: The results of the parameterized bounded-buffer problem

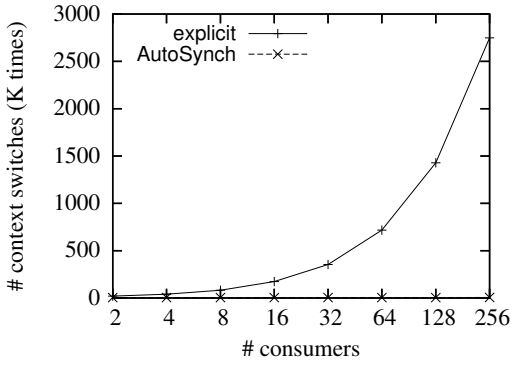


Figure 2.10: The number of context switches of the parametrized bounded-buffer problem

Fig. 2.11 and 2.12 present the run time ratio of our approaches and the *explicit* approach for the round-robin access pattern with 256 threads and the readers/writers problem with 64 writers and 320 readers. The y-axis indicates the runtime ratio and the x-axis shows the delay time, the amount of time in which the threads perform operations out of the monitor between every two monitor operations, in microseconds. As expected, the performance difference decreases as the duration of delay time increases. AutoSynch is two times

slower than the explicit approach with no delay time (saturation test) for round-robin access pattern. However, when the duration of delay time is 5000 microseconds, AutoSynch is only 7.7% slower than the explicit approach. Note that, even AutoSynch-T performs well when the duration of delay time increases. The similar observation can be seen for the readers/writers problem in Fig. 2.12. The results suggest that our approach could be more useful for practical problems that perform more monitor unrelated operations.

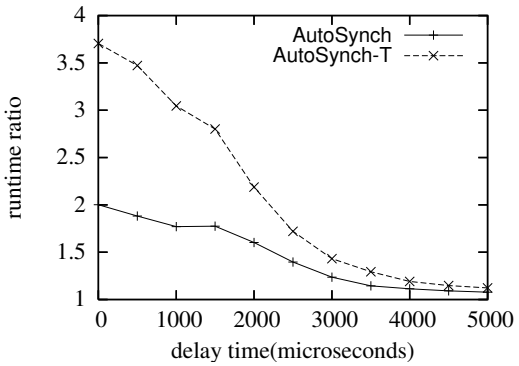


Figure 2.11: The runtime ratio of round-robin

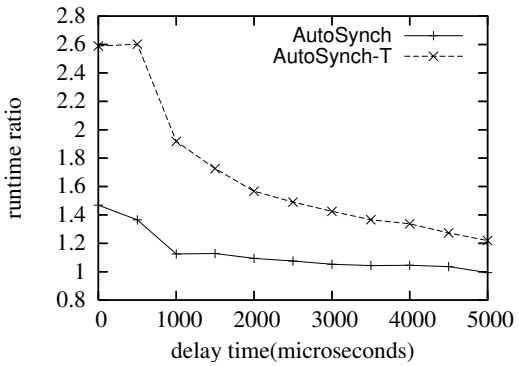


Figure 2.12: The runtime ratio of ticket readers/writers

## 2.6 Summary

In this chapter, we have proposed AutoSynch framework that supports automatic-signal mechanism with AutoSynch class and `waituntil` statement. AutoSynch uses the *closure* operation to enable the complex predicate evaluation in every thread. It also provides *relay invariance* that some thread waiting for a condition has met is always signaled to avoid `signalAll` calls. AutoSynch also uses predicates tag to accelerate the process in deciding which thread should be signaled.

# Chapter 3

# Asynchronous Monitor Method Executions

Monitor objects are used extensively for mutual exclusion and synchronization in shared memory parallel programs. They provide ease of use, and enable straightforward correctness analysis. However, they inhibit parallelism by enforcing serial executions of critical sections, and thus the performance of parallel programs with monitors scales poorly with number of processes. In this chapter, we present ActiveMonitor — a framework that improves parallelism by exploiting asynchronous execution of critical sections. We evaluate the performance of Java based implementation of ActiveMonitor on micro-benchmarks involving light and heavy critical sections, as well as on single-source-shortest-path problem in directed graphs. Our results show that on most of these problems, ActiveMonitor based programs outperform programs implemented using Java's reentrant-lock and condition variable constructs.

## 3.1 Concept and Design

In ActiveMonitor, we use the term *worker* to denote an application thread/process. A monitor object can be instantiated as a thread/process

based on the availability of system resources. This thread is called a *server*, and invocation of critical sections of monitor by workers are delegated to it. Delegation [OTY99] is a technique in which critical sections of a monitor are not executed directly by workers invoking the method, but are processed by the *server* thread on behalf of workers. The workers *announce* their execution requests — in the form of *tasks* — to the server by adding the requests (task objects) to a shared storage that is owned by the monitor. Combining [HIST10, FK12] is a version of delegation in which the role of server is assumed by the worker that succeeds in acquiring the lock to the critical section. This thread becomes the *combiner*, and in addition to its own request, serves requests announced by other threads for a period of time before releasing the lock and allowing some other thread to become the combiner. Throughout this dissertation, we use the term *server* in both delegation and combining contexts. A critical section is *asynchronous* (or non-blocking) if the worker can return to executing its own local program from the critical section before its completion. Otherwise the critical section is synchronous (or blocking).

ActiveMonitor provides the following constructs for writing monitor based programs:

1. `monitor`: keyword that declares a class as a monitor, and frees the user from explicit lock instantiations, and their acquisition/release to make the critical sections thread-safe.

2. `waituntil`: a statement for conditional waits and notifications. The statement requires a boolean predicate as an argument.

3. `asynchronous`: keyword used in declaration of monitor methods. Such methods are delegated to the server (monitor thread) , and the worker thread returns to its own local execution before completing the method. If the worker requires the result of the computation, it receives a future [Hal85] instance which can be evaluated — a blocking call if the result is not yet available — to fetch the result. Fig. 3.1 shows the usage of `asynchronous` as in line 4.

```
1  monitor class BoundedQueue {
2    Object[] items;
3    int putPtr, takePtr, count;
4    asynchronous void put(Object item) {
5      waituntil(count < items.length);
6      items [putPtr++] = item;
7      putPtr = putPtr % size;
8      ++count;
9    }
10   Object take() {
11     waituntil(count > 0);
12     Object x = items[takePtr++];
13     takePtr = takePtr % size;
14     --count;
15     return x;
16   }
17 }
```

Figure 3.1: Bounded-Queue with ActiveMonitor

It then replaces invocation of these methods (by application threads on monitor object) by submission of tasks to the server of the monitor. The run-

time library has two sub-components: condition manager and task executer. The condition manager is responsible for observing the state of the monitor object for conditional waits and signaling an appropriate thread whenever its precondition becomes true. The task executer component manages the submission and completion of monitor tasks and also handles their asynchronous executions.

Our pre-processor uses a set of parsing rules that identify the ActiveMonitor keywords, and is an extension of the pre-processor in our previous chapter, AutoSynch. We briefly discuss its steps. For a source class that is declared `monitor`, the pre-processor ensures that each method of the class is protected using the re-entrant lock by inserting lock acquisition and release statements at the beginning and end of method code. It then parses the method code for `waituntil` statements, and for each such statement it creates a new condition in the monitor class. For every condition, the notification criteria is the boolean predicate provided as the argument to its corresponding `waituntil` statement. Then it analyzes the method to decide whether or not it should be delegated. If the method is declared `asynchronous` or does not returns a value and updates the shared data, the pre-processor generates an equivalent task for delegation. We discuss monitor tasks in the next section.

## 3.2 Monitor Tasks

In ActiveMonitor, a monitor task is defined as follows.

**Definition 10.** *Monitor Task: A monitor task t consists of a boolean predicate $P$ and a set of statements $S$. At runtime, if the precondition defined by $P$ is true then t is 'executable' and statements in $S$ can be executed to complete t. Otherwise, t is 'unexecutable'.*

For a task $t$, its set of statements $S$ can be empty. The pre-condition $P$ — passed as an argument to `waituntil` statement — can either be absent altogether or may not appear as the first statement in the monitor method. When a monitor method has no precondition, the pre-processor creates a task with its precondition as tautology, indicating that the task can be executed at any time. If a monitor method does not start with a `waituntil` statement but has some such statement in between, then the precondition of the first derived task is a tautology. Consider the `put` method (lines $4-9$) of the bounded-buffer program of Fig. 3.1. For this monitor method, the equivalent monitor task $t$ is defined by the code of lines $5-8$. For $t$, the precondition $P$ is `(count < items.length)`; and it checks if the buffer has any space to insert the item. If this condition is false, the `waituntil` construct ensures that any thread trying to complete this task has to wait until the buffer has some space to insert the items. Lines $6-8$ together form the set of statements $S$. The method is explicitly declared `asynchronous`, so the generated task is submitted for an asynchronous execution to the monitor thread.

### 3.2.1 Asynchronous Execution of Tasks

After an equivalent task $t$ for a method $m$ has been generated, all the invocations of $m$ by workers are executed with combining technique [HIST10, FK12]. We use *futures* [Hal85] for asynchronous (non-blocking) executions of critical sections. For each asynchronous method call the pre-processing phase injects submission of a task to the server (monitor thread) . A *future* reference is returned to the worker as a pointer to the computation. Whenever the server finishes the execution of a task, it updates its corresponding future reference with the result of the computation. If the worker needs the result of the computation it *evaluates* the future. Evaluation of a future is a blocking method: if the computation has not finished then the caller must wait until its completion. Note that unlike the schemes of [OTY99, HIST10, FK12], neither the server nor the worker threads perform busy-wait/spinning in ActiveMonitor. Thus, we do not waste any processing cycles and yield the CPU when there are no tasks to execute. Hence, ActiveMonitor provides a much more practical implementation for delegated executions.

To guarantee program order, ActiveMonitor framework stipulates that each worker can only submit one asynchronous task at a time. The task executor sub-component of the runtime library handles this by storing a map of ids of worker threads and their corresponding task submissions. Whenever a worker tries to submit an asynchronous task, it first checks the map to verify if there is some previous asynchronous task stored against its id that is not yet finished. The worker is forced to wait — by evaluating the future — for

the completion of that task before being allowed to submit the new task.

## 3.3 Runtime Library for Asynchronous Execution of Tasks

The runtime library of ActiveMonitor provides two key functionalities: (a) automatic signaling of threads under conditional waiting, and (b) delegation and asynchronous executions of critical sections. We extend our previous chapter, AutoSynch [HG13] to enable functionality (a) for task based asynchronous executions.

### 3.3.1 Execution of Monitor Tasks

ActiveMonitor runtime library executes monitor tasks using the following rules.

**Rule 1** (Mutex Invariant)**.** *If some thread $t$ is executing a task $m$ of monitor $M$, then no other thread can execute any task $m'$ of $M$ concurrently.*

This rule maintains the mutual exclusion of critical sections of a monitor. We require two additional rules to guarantee execution of tasks in program order. Let $proc(t)$ denote the worker thread that submits the task $t$ to a monitor. Let $sub(t)$ and $exe(t)$ respectively indicate the timestamps when $t$ is submitted to the monitor, and when the server thread starts executing $t$.

**Rule 2.** *For a pair of tasks $s$ and $t$ submitted to a monitor $M$, if $proc(s) = proc(t)$, then*
$sub(s) < sub(t) \Rightarrow exe(s) < exe(t)$.

This rule ensures that a server (monitor thread) executes every worker's tasks in the program order of worker.

**Rule 3.** *Let $m_1$, $m_2$ be two successive method invocations by a worker thread on two different monitors $M_1$ and $M_2$ in the user program, and let $t_1$, $t_2$ be their corresponding task submissions at runtime. Then, $t_1$ must be completed before $t_2$'s submission.*

This rule enforces the constraint on a thread's successive invocations of methods on different monitor objects. Blocking method invocations in between these two calls are acceptable.

The notions of method *invocation* and *response* used to define linearizability [HHWW90] need a different interpretation under asynchronous executions. In short, *invocation* now corresponds to submission of the equivalent task to monitor thread, and *response* corresponds to this task's completion. Observe that the legal sequential history we get may not preserve the order of invocation of operations, but only the thread order. With this interpretation, we can easily validate the following result.

**Lemma 1.** *Rules 1, 2 and 3 guarantee executions equivalent to lock-based executions.*

*Proof.* We show that for any execution in our model there exists an equivalent lock-based execution. Since all tasks of any monitor object are executed by a single thread due to Rule 1, mutual exclusion is preserved just as in any

lock-based execution. We only need to show that the order of execution of the tasks corresponds to a schedule in which worker threads execute the tasks.

It is sufficient to show that all tasks submitted by a single worker thread execute in the order of submissions. Let $s$ and $t$ be two consecutive tasks submitted by the thread. If they are submitted for the same monitor, then the Rule 2 preserves the order. If $s$ is a blocking task, then by definition of blocking task, $t$ cannot submitted before $s$ is completed. Hence, execution of $s$ precedes execution of $t$. If $s$ is a non-blocking task and is on a different monitor object from $t$, then due to Rule 3 we wait for $s$ to finish before submission of $t$.

□

### 3.3.2 Implementation

We now describe implementation details that make ActiveMonitor scalable and faster, as well as practical in terms of use with real world applications. Recall that unlike other delegation/combining implementations [OTY99, HIST10, FK12], threads do not perform busy-wait in ActiveMonitor. To enable conditional wait and yielding the CPU, our implementation uses a read/write lock for executing updates on each server thread. This ensures: (a) reads do not return stale values, and (b) servers/workers can release the CPU and go into waiting state whenever required as per runtime conditions. We employ a modified version of combining [HIST10, FK12] for executing critical section updates. When submitting a task to a monitor, a worker thread

checks if the server of the monitor is in waiting state. If so, the worker acquires the lock — becomes the *combiner* — and executes a predefined number (five in our implementation) of tasks before releasing the lock. Observe that the actual acquisitions of the write-lock are mostly uncontended under this approach. Uncontended lock acquisitions are known to be relatively inexpensive, and thus threads does not incur significant performance penalty in doing so. For asynchronous tasks, we use a lightweight version of future objects that are shared between only one worker thread and the server. Only the server can update the state of these objects. Instead of using the default ones provided by the Java concurrent library [Lea05], we create these objects using only a few volatile variables. Instead of using the default wait/notify mechanism provided by Java, we use the lower level API of `park` and `unpark` [Lea05] for threads. Using the lower level API allows a more fine-grained control on execution of these threads.

### 3.3.3 Storage of Tasks: Single Consumer Optimal Bounded Queue

Although asynchronous executions generally benefit the application performance, a large number of asynchronous tasks in the system lead to degraded performance due to higher number of cache misses. To prevent this, ActiveMonitor maintains a bounded FIFO queue for each server in which the workers enqueue their tasks. Given that ActiveMonitor instantiates only one server thread (if any) per monitor object, this bounded-queue is a special case of the producer-consumer problem with only one consumer and multiple pro-

ducers. Only the server consumes the items (tasks) from this queue, and all the workers produce the items. For this use-case, we developed an optimized algorithm for a thread-safe bounded FIFO queue that minimizes the synchronization costs for the consumer.

Only insertions in the queue require guarded execution under a lock to ensure correctness while multiple threads concurrently attempt to insert items. Only a single thread performs removal of items (through the `take` method), and thus we do not require a lock to protect concurrent removals. However, maintaining the correct count of actual number of items in the queue is essential. This is done using the atomic integer `count`. We adopt a 'stealing' strategy in which the consumer locally caches the number of available items, using the `takeCount` variable, in a look-ahead manner and reads and updates the atomic integer `count` only when needed. Hence, the number of upadates to the atomic integer `count` is kept low, which in turn reduces the cache-coherence traffic, and improves the throughput and scalability.

Whenever there is no task (in its bounded-queue) for the server to execute, it is forced to go into wait. The server performs this wait outside the queue using a condition variable that it owns. The automatic signaling mechanism of the runtime library ensures that it is signaled and wakes up from the wait if a new executable task is enqueued in the queue.

The pseudocode for `put` and `take` methods of our bounded-queue is shown in Figure 3.2.

```
count: atomic integer
capacity: integer // capacity of the bounded queue
putlock: mutexes for put operations
takeCount: integer // stores value of items that can be taken without locking
notFull: condition variable
// items are stored in a linked-list
void put(T e) {
    node = new Node<T>(e)
    putlock.lock()  // lock guarded
    while (count.get() == capacity) notFull.await()
    enqueue(node) // linked-list add tail
    lcount = count.getAndIncrement()
    if (lcount + 1 < capacity) notFull.signal()
    putlock.unlock()
}
// Called only from take
void signalNotFull() {
    putlock.lock()
    notFull.signal()
    putlock.unlock()
}
T take() {
    if (takeCount > 0) {
        --takeCount
        return dequeue() // linked-list remove head
    }
    takeCount = count.get()
    if (takeCount == 0) {
        signalNotFull()
        return null
    }
    T x = dequeue() // remove head from linked-list
    lcount = count.getAndAdd(-takeCount)
    if (lcount == takeCount) signalNotFull()
    --takeCount
    return x
}
```

Figure 3.2: BoundedQueue for single consumer and multiple producers

### 3.3.4 Monitor Thread Management

If we spawn a new thread for every monitor object, the performance of programs with relatively large number of monitors could suffer. ActiveMonitor allows the programmer to manually control this number, as well as itself controls the number of monitor threads based on the system hardware resources. The programmer can indicate an upper bound on the number of monitor threads when starting the application. The ActiveMonitor runtime library uses this limit in restricting the number of monitor threads spawned. If this limit is reached, no other monitor threads are created, and invocations of asynchronous methods on remaining monitors (that are not instantiated as threads) also follow the conventional synchronous (blocking) execution.

Irrespective of the user provided upper bound on server threads, the runtime library only instantiates a thread for a monitor if there is sufficient hardware available. The runtime library monitors the system environment information: CPU usage (for example from `/proc/stat` on Unix), and the size of wait-queues of monitor objects, to decide whether or not monitors should be executing as threads. If the CPU usage is high, our framework switches to traditional locking.

## 3.4 Evaluation

We implement monitor based solutions to multiple concurrency problems using ActiveMonitor, ReentrantLocks from JDK7, and combining [FK12] — that does not perform continuous busy-waits — by executing ActiveMonitor

in only synchronous mode. We evaluate the performance of these implementations on light and heavy critical sections. Light critical sections do not involve much work within them, and favor traditional lock-based monitors as the overhead of maintaining additional information for delegated executions outweighs their benefits. On the other hand, heavy critical sections provide increased opportunity for exploiting asynchrony and parallelism. Table 3.1 presents a summary of problems used for our evaluation.

| Name | Short Desc. | CS Work [Type] | Details |
|---|---|---|---|
| PSSSP | Parallel single-source-shortest-path using Dijkstra's algorithm [Dij59] using priority queue. | $\mathcal{O}(\log n)$ [Heavy] | (a) USA road network graphs<br>(b) R-MAT Graphs [CZF04] |
| BQ | Bounded FIFO queue of plain Java objects. | $\mathcal{O}(1)$ [Light] | Capacity varied from 4 to 64; number of enqueuers is equal to the number of dequeuers. |
| SLL | Linked-list of integers; entries are kept sorted in non-decreasing order. | $\mathcal{O}(n)$ [Heavy] | (a) Read-heavy: 90% reads, 9% insert, 1% delete<br>(b) Write-heavy: 0% reads, 50% insert, 50% delete<br>(c) Mixed: 70% reads, 20% insert, 10% delete |
| RR | Round-robin monitor access from [HG13]. | $\mathcal{O}(1)$ [Light] | each thread accesses monitor in a predefined round-robin manner based on thread-id. |

Table 3.1: Short description of problems evaluated. Critical section (CS) is light/heavy if the total number of operations performed inside it are small-/large.

All the experiments are conducted on a 40-core Intel Xeon machine that consists of four sockets of Xeon E7-4850 10-core (20 hyper-threads), running at 2 GHz with 32 KB L1, 256 KB L2, and 24 MB LLC, respectively. Compilation and execution both are performed with Oracle Java 1.7 (64-bit VM). Across all results, we denote the implementations with the following notation: LK: implementation using Java's ReentrantLock, AM: ActiveMonitor with asynchronous executions, and AMS: ActiveMonitor running with only synchronous delegations.

For PSSSP problem, a thread-safe priority queue is used as an under-

lying data structure. ActiveMonitor solution of this problem uses the monitor-based implementation of an unbounded blocking priority queue from Java's concurrency package java.util.concurrent, and only modifies it to make the put method asynchronous. We evaluate the time taken to compute the shortest paths to all vertices from a randomly selected source vertex. We use five large sized directed graphs. Two of these graphs, FLA and NY, are USA road-network graphs of Florida, and New York obtained from [dim], and the remaining three graphs, R16, R128, and R512 are generated using the GT-Graph [BM06] generator suite.

For all other problems we collect the throughput of operations over a 2 second period with varying number of workers. For BQ problem, the items in queue are randomly generated strings, with enqueue operation being asynchronous and dequeue being synchronous. For SLL problem, we pre-populated the data structure with 1000 entries to simulate steady state behavior. For all the operations, the operand values are chosen uniformly at random between 0 and 2000. This guarantees that on average, half of the operations are successful and the structure size does not grow too large. Insertions and deletions in the list are asynchronous and searches are synchronous. For RR, all accesses to the critical section are synchronous. BQ and RR problems require threads to perform conditional waiting. For these two problems, we also compare the performance of ActiveMonitor with that of Queue Delegation Locking [SDSC14], denoted by QD notation, by adding conditional waiting to QD. The purpose of this comparison is to establish that our approach of using automatic signaling

with asynchronous executions can out-perform QD's approach of asynchronous delegation under lock-unavailability. In addition, we also compute throughput of performing OR implementations. For logical-or operations, we also tried to evaluate the performance of a transactional memory implementation [?] but this implementation resulted in runtime errors and could not execute the statements.

We perform multiple warm-up runs to negate just-in-time compilation related performance variations. In addition, all threads perform a fixed number of warm-up operations before starting the time measurements. For all the experiments, we collect runtimes for 7 runs, and report the mean value of 5 runs after discarding the highest and lowest values.

### 3.4.1 Results

Fig. 3.3 plots the throughput of the three PSSSP implementations in edges traversed per unit time format. Given that the three synthetic R-MAT [CZF04] generated graphs are relatively dense in comparison to the road network graphs NY and FLA, the throughput values for all the implementations are higher for these graphs. AM outperforms both of LK and AMS. Specifically, on R512 graph — one with the highest density — AM is much faster than the other two. Given that the same implementation of priority queue is used as the underlying data structure for all three implementations, and the only difference is in terms of asynchronous inserts, these results validate our claim that AM approach is much more beneficial for heavy critical

sections.

Fig. 3.4 plots throughput of operations for different capacities of bounded queues for three implementation techniques. For smaller buffer sizes, in the range of 4 to 16 AM significantly outperforms LK implementation. This result highlights the benefits of asynchronous executions because LK is much slower in comparison to AM, as well as AMS due to high contention on locks. For larger capacities of 32 and 64, LK implementations perform better than AM because the availability of sufficient storage space allows worker threads to repeatedly acquire critical sections without being blocked out, and LK benefits from Java's policy of non-fairness in lock acquisitions. In contrast, AM and AMS provide *almost* 'fair' executions for workers. However, in doing so, they end up performing more work in cases blocking due to unavailability of space occurs rarely.

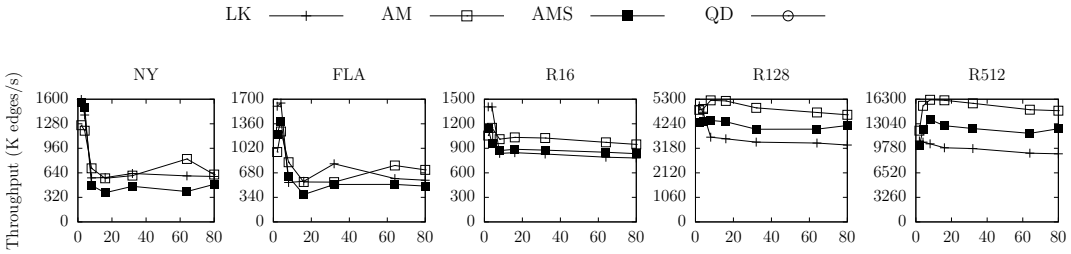


Figure 3.3: Throughput for PSSSP using priority queue (x-axis shows the number of threads)

Fig. 3.5 shows the operations throughput for the SLL and RR. In all runs on these problems, (AM) clearly and significantly outperforms the read-

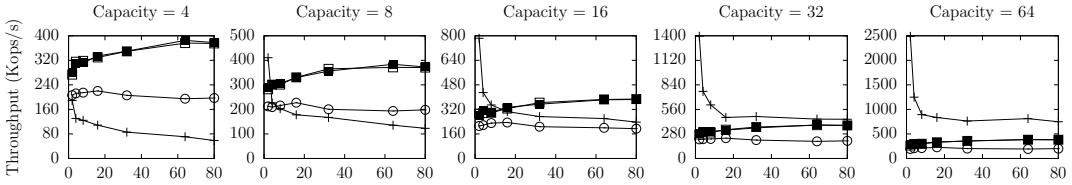


Figure 3.4: Throughput for Bounded FIFO Queue (x-axis shows the number of threads)

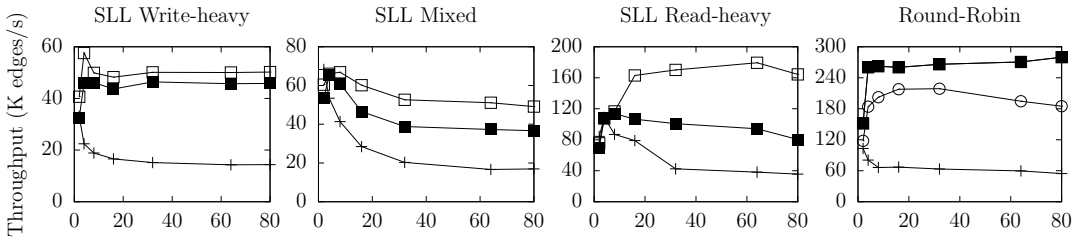


Figure 3.5: Throughput for SLL, and RR (x-axis shows the number of threads)

write reentrant lock based monitor (LK), as well as delegation technique of AMS. Note that RR problem does not involve any asynchronous operation, and thus AM and AMS runs are exactly the same. Given that the critical section involved in SLL problem is heavy, the performance gap highlights the benefits of asynchronous monitors for such cases. Surprisingly, AM (as well as AMS) is $\sim 3-4\times$ faster than LK on RR problem too. This is because the RR problem setup simulates a critical section in similar to that of BQ problem with capacity one. Hence, LK implementation spends a lot of its execution time in waiting for lock acquisitions, whereas AM and AMS benefit from lower contention.

On all the problems with conditional waits, AM significantly outperforms QD in terms of throughput. Hence, extending QD to incorporate conditional waiting is not sufficient to match our approach. Our techniques for efficient conditional synchronization with automatic signaling provide significant benefits in comparison to QD.

## 3.5 Related Work

Our idea of having monitor objects execute as independent threads is influenced by Hoare's proposed communicating sequential processes (CSP) [HH78] mechanism in which all objects are *active*, of long ago. However, CSP does not have the notion of shared memory, and every object is a process. In contrast, our focus is solely on shared memory parallel programs on multi-core machines.

We use *futures* [Hal85,Lea05] to realize the idea of non-blocking/asynchronous executions. Kogan et al. [KH14] explore a similar approach in making use of *futures* for non-blocking executions. However, we explore changes to the general paradigm of monitors, whereas [KH14] only focuses on three data structures: stacks, queues, and linked-lists, none of them requiring conditional waiting. In addition, [KH14] uses data structure specific local elimination/combining, and allows read/fetch operations on these data structures to be asynchronous whereas we do not — our assumption being that in almost all the cases, a programmer needs the result of read/fetch immediately so that she can use it in the subsequent program logic. Hence, our approach spans a more generic

level of monitors, and does not rely on knowledge of internal functionality of critical section it protects. Some theoretical results that establish the bounds on improvements in cache locality by the use of futures have been established in [HL14]. These results are not directly related to monitor based executions, but lead the direction in terms of use of futures for improving the performance of multi-threaded programs.

Existing implementations of the combining technique [OTY99, FK12, HIST10] perform busy waits for task completions and do not yield the CPU; additionally they also do not provide any mechanisms for conditional waits — these issues together make them more or less impractical for use in real world applications. Remote Core Locking (RCL) [LDT+12] addresses such issues by allowing conditional waits, and uses a dedicated core for executing critical section, but does not incorporate asynchronous executions. Recently, works such as [PcRS14, CDH+13] have performed extensive experimental analysis in identifying the performance gains/losses with asynchronous message-passing like executions over synchronous shared memory ones. [PcRS14] provides various insights for effective implementations that perform well using hardware message passing support on shared memory machines. This work minimizes the remote-memory-references (RMRs) during executions, and quantifies the performance gains for asynchronous executions, but assumes that the method data fits in a single cache-line. In addition, it does not consider the conditional wait based monitor implementations. Similarly, [CDH+13] studies the pros and cons of message passing based executions on performance of shared memory

parallel programs. This work highlights that different approaches perform best under different circumstances, and that the communication overhead of message passing can often outweigh its benefits, and discusses ways in which this balance may shift in the future. Queue Delegation Locking (QDL) [SDSC14], uses the approach of combining to provide a locking library implementation in C++. However, QDL does not provide a mechanism for synchronization between threads, and waiting, based on conditions.

Transactional memory [HM93, ST95] is a well-known research effort that proposes modified syntax for ease of writing multi-threaded programs. However, constructs for conditional waiting under transactional memory are limited [SR13, LM11, DS09]. Hence, writing many conditional synchronization based multi-threaded programs is rather difficult. Also, unlike transactional memory, our approach merely transfers the responsibility of data manipulation to monitor threads and does not require any complicated rollback mechanism for resolving conflicting updates on the shared data. x10 [CGS+05] programming language focuses on providing features that have an overlap with both transactional memory and our work. However, there are significant differences in the support and usage of these constructs. The support for conditional waiting is present syntactically, but as stated in [CGS+05] is deprecated for runtime execution.

Lock-free algorithmic techniques using atomic hardware instructions such as *compare-and-swap* have gained momentum for implementing scalable thread-safe data structures [Har01, MM02, HSY04, FHS04, Her88, KP11, KP12,

TBKP12, NM14]. In addition, [HHL+06, IS14] have explored alternate implementation techniques that combine/eliminate complementary operations for increasing parallelism in data structures. However, the difficulty involved in designing lock-free/wait-free algorithms, and operation eliminating data structures is well known. At present, it is not clear how lock-free techniques can be used to implement critical sections that involve many operations spanning across multiple shared objects. The absence of any wait-notify mechanism in lock-free techniques is another hurdle for their use in many real world programs.

## 3.6 Discussion

Our current implementation has the following two limitations. First, in our current implementation, thread dependent variables and functions within a monitor method cannot be used directly in the `Runnable` or `Callable` object that is used in task generation by our approach. The reason is that the tasks are executed by the monitor thread and not by the worker thread. For example, suppose there is a monitor method that invokes `Thread.currentThread()`, if we directly add this statement to the generated `Runnable` object (in the task), then this method's invocation at runtime will return the reference to the monitor thread when it is executed. However, it is obvious that the intent of this call inside the monitor method was to refer to the worker thread. Second, our current pre-processing implementation does not support synchronous recursive methods on monitors. The reason is that the number of the method invoca-

tions to be made at the runtime is non-deterministic. Therefore, we cannot know how many tasks we need to create at pre-processing time. In addition, since the method is blocking/synchronous, the monitor thread will get blocked when it recurs. We plan to address these two issues in the future as Section 6.3.

## 3.7 Summary

We have shown that our proposed scheme of asynchronous executions in monitors provides significant improvement over traditional lock-based monitors. At present, writing parallel programs that provide high throughput and scalability is an arduous task for most programmers. The main challenge is a lack of simple programming language constructs that guarantee thread-safety while exploiting parallelism of executions and availability of hardware in a seamless and portable manner. Our proposed design of asynchronous monitors is a step in the direction of providing such constructs. The current version of our implementation consumes some additional processing resources. However, we believe that with further research efforts in this direction, and further optimizations in our implementation, our proposed technique can lead to significant improvements in programmability as well as performance of shared memory parallel programs.

# Chapter 4

# Multi-Object Synchronization

Current monitor based systems have many disadvantages for multi object operations. They require the programmers to manually determine the order of locking operations, and use global locks or perform busy waiting for operations that depend upon a condition that spans multiple objects. Transactional memory systems eliminate the need for explicit locks, but do not support conditional synchronization. They also require the ability to rollback transactions. In this chapter, we propose new monitor based methods that provide automatic signaling for global conditions that span multiple objects. First, we introduce the `multisynch` construct for multi-object mutual exclusion which lets the system determine the order of locking multiple objects. Second, our system provides automatic notification for global conditions. Assuming that the global condition is a boolean expression of local predicates, our method allows efficient monitoring of the conditions without any need for global locks. Third, our system solves the compositionally problem of monitor systems without requiring global locks. We have implemented our constructs on top of Java and have evaluated their overhead. Our results show that on most of the multi-object problems, not only our code is simpler but also faster

than Java's reentrant-lock as well as the Deuce transactional memory system.

## 4.1 Multi-Object Mutual Exclusion

It is the responsibility of our system to ensure mutual exclusion for multiple monitor objects of `multisynch` statements. Programmers use a `multisynch` statement to specify which monitors should be synchronized. The parameters of `multisynch` can be a sequence of an arbitrary number of monitor objects. If an array of monitor objects is a parameter of a `multisynch` statement, the system ensures mutual exclusion for all elements of the array. Note that the order of parameters does not matter. The system decides how to acquire locks of monitors at runtime automatically.

Assuming that all threads acquire locks only using `multisynch` statement and that there are no nested `multisynch` statements, the system ensures that there is no deadlock due to inconsistent locking order. Deadlocks occur when two (or multiple) threads acquire locks on the same monitors but in different order. One well-known way to prevent deadlock is to ensure that all threads acquire locks in a consistent order in the entire system [GPB+06]. However, it is not always obvious for programmers to identify inconsistent lock ordering. Our system minimizes the risks of deadlocks by removing the burden of ensuring consistent lock ordering from the programmer. It acquires locks according to the order of unique ids of all monitors.

To implement a `multisynch` statement, our system acquires locks only when it is required. Whenever a thread needs to access a monitor object, the

system checks if it has acquired the lock for the object. If not, the thread acquires its lock and locks of other monitors that have smaller ids and are listed in the `multisynch` statement. These locks are acquired in the increasing order of ids. The system releases all locks at the end of the `multisynch` statement. Fig. 4.1 shows an example. Suppose `obj1`, `obj2`, and `obj3` are monitors, where `obj1.id` $<$ `obj2.id` $<$ `obj3.id`. In line 2, the program wants to access `obj2`, our system automatically acquires locks of `obj1` and `obj2` because `obj1` has smaller id than `obj2`. It acquires the lock of `obj3` at line 4. We note here that all these techniques are well known; the main contribution of this dissertation is in mechanisms for detecting global conditions on these objects which requires implementation of `multisynch`.

```
1 multisynch(obj1, obj2, obj3) {
2   obj2.access();
3   obj1.access();
4   obj3.access();
5 }
```

Figure 4.1: An example of the `multisynch` statement

The `multisynch` statement requires programmers to specify the monitor objects, which is a disadvantage in comparison with transactional memory systems, which require only `atomic` blocks. However, it is difficult to provide conditional synchronization using transactional memory constructs. Our system provides not only mutual exclusion but also conditional synchronization. In addition, transaction may be aborted and re-executed automatically. There-

fore, irreversible operations cannot use transactional memory. Our system is applicable even for such operations.

## 4.2 Efficient Automatic Notification of Global Conditions

We first show that techniques developed for automatic signaling for local conditions (such as in AutoSynch) cannot be simply extended for global conditions. In AutoSynch, when a thread exits a monitor or goes into waiting state, it checks whether there is some thread waiting on a condition that has become true. If at least one such waiting thread exists, it signals that thread. The predicate evaluation is crucial in deciding which thread should be signaled. To avoid unnecessary context switches, AutoSynch computes *closure* of the predicate with respect to local variables of waiting threads so that any thread can evaluate the predicate. Since these variables do not change while the thread is waiting, the closure of the predicate is exact. Although this technique works on conditions based on a single monitor, it does not work for global conditions in Java without assuming global locks.

In the Java memory model, every thread can be considered as running on a different CPU. Because CPUs hold registers that cannot be directly accessed by other CPUs, one thread does not know about values being manipulated by another thread in such a model. Hence, the evaluation of a global predicate by the thread $T$ holding the lock on one monitor object can be wrong because $T$ may not observe some concurrent updates of the predicate by other

threads. For example, suppose that a thread $T_1$ is waiting for the predicate `(!Q2.isEmpty() && !Q3.isEmpty())` to become true. Then, two threads $T_2$ and $T_3$ concurrently execute `Q2.put(x)` and `Q3.put(y)`, respectively. Before leaving monitor `Q2`, $T_2$ evaluates the global predicate as false because $T_2$ cannot observe that `!Q3.isEmpty()` has become true since the update occurs only on the register of $T_3$ and $T_2$ does not acquire lock of `Q3` before evaluation. Therefore, $T_2$ does not signal $T_1$. Similarly, $T_3$ does not signal $T_1$. In this case, $T_1$ is still waiting while the predicate `(!Q2.isEmpty() && !Q3.isEmpty())` has become true. A global predicate can be evaluated correctly only if a thread acquires locks for all monitors related to the predicate. However, acquiring all locks of its monitors is expensive because other threads that want to access those monitors are forced to wait. A wrong predicate evaluation, on the other hand, may introduce a deadlock because the system may miss signaling a thread waiting on a global condition that has become true. To ensure correctness, our system must provide the following no-missed-signal property.

**Definition 11** (No-Missed-Signal Property)**.** *If threads wait on a global condition that has become true, then at least one thread waiting on the condition is signaled.*

Note that, no-missed-signal property is similar to the relay invariance in Definition 5; however, relay invariance deals with only local conditions but not global conditions.

We do not require all threads to be signaled, just one. Whenever that thread exits the monitor, it will wake up another thread so long as the global

condition stays true. We also note that since Java treats signals as *hints*, it is okay from the correctness perspective for the system to send a signal even if the global predicate is false. The thread that wakes up would reevaluate the global condition. Hence, one naive strategy is that threads waiting on global conditions are always signaled. However, this naive approach decreases overall performance because it introduces redundant context switches and limits parallelism. The notified threads may need to go back to waiting state since their conditions are still false. Furthermore, other threads cannot access monitors because notified threads acquire monitors related to their predicates. Missing signals introduces deadlocks while false signals decrease overall performance. In this section, we discuss two approaches to efficiently detect global predicates that avoid missed signals and reduce false signals.

### 4.2.1 Preliminaries

A global predicate is a global Boolean condition involving a set of monitor objects. For example, the condition `(!Q1.isEmpty() || !Q2.isEmpty()) || (Q1.size() > Q2.size())` is a global condition involving two monitor objects, `Q1` and `Q2`. We call predicate `(!Q1.isEmpty())` and predicate `(!Q2.isEmpty())` local because they involve only one monitor object. The condition `(Q1.size() > Q2.size())` is a complex predicate because it involves both `Q1` and `Q2`. We first discuss global predicates containing only local predicates. The case of global predicates involving complex predicates is discussed in Sec. 4.2.4.

A global predicate can be represented by $P : X \rightarrow \mathbb{B}$, involving a set

of monitor objects, $M = \{M_1, \ldots, M_n\}$, where $X$ is the space spanned by the variables $\vec{x} = (x_1, \ldots, x_m)$. Note that, $X = \cup_{i=1}^{n} X_i$, where $X_i$ indicates the set of variables related to $M_i$. Each variable represents an atomic local Boolean expression. For example, the queue `Q1` and its condition `Q1.isEmpty()` can be represented as $M_1$ and a variable $x \in X_1$, and the queue `Q2` and its condition `Q2.isEmpty()` can be expressed as $M_2$ and a variable $y \in X_2$. For any global state of the system, $G$, the predicate $P$ is evaluated based on the values of all local predicates in $G$. We assume that all predicates in the `waituntil` statement are read-only and free from side effects. Any evaluation of those predicates does not update any variable or change the state $G$.

### 4.2.2 Atomic-Variable Approach

A thread cannot evaluate global predicates correctly without acquiring global locks because it cannot observe all concurrent monitor objects updates by different threads. To evaluate global predicates precisely, we exploit *atomic boolean variables*, which have `set` and `get` methods where a `set` call has a happens-before relationship with any subsequent `get` call on the same variable. Generally speaking, for any global predicate $P$, we can derive a predicate $\hat{P}$, in which every local boolean expression of $P$ is represented by an atomic boolean variable $\hat{x}$. If the boolean expression is true, then we set $\hat{x}$ to be true; otherwise, we set $\hat{x}$ to be false. For example, the global predicate $P =$ `(!Q1.isEmpty() || !Q2.isEmpty()) && (!Q3.isFull() || !Q4.isFull())` has a corresponding $\hat{P} = (\hat{w} \vee \hat{x}) \wedge (\hat{y} \vee \hat{z})$. Our system can

decide if threads waiting on $P$ should be signaled based on the evaluation of $\hat{P}$. Any thread $T$ that acquires monitor $M_i$ needs to update $\hat{P}$ before releasing $M_i$ by setting the values of atomic boolean variables related to $M_i$. After $T$ updates the variables, it releases monitor $M_i$, evaluates $\hat{P}$, and decides whether to signal threads waiting on $P$. For example, consider the global predicate `(!Q1.isEmpty() && !Q2.isEmpty()) || !Q3.isFull()`. It has a corresponding $\hat{P} = (\hat{x} \wedge \hat{y}) \vee \hat{z}$, where every variable is set as false by a thread waiting on the condition. Suppose $T_1$ accesses `Q1` and determines that `!Q1.isEmpty()` is true. Before $T_1$ releases `Q1`, it updates $\hat{P}$ by setting $\hat{x}$ as true. $\hat{P}$ is still false since $\hat{y}$ is false. $T_1$ does not signal any thread waiting on $P$. Thread $T_2$ then accesses `Q2` and finds that `!Q2.isEmpty()` has become true. $T_2$ updates $\hat{P}$ by setting $\hat{y}$ as true and signals a thread waiting on $P$ since $\hat{P}$ has become true.

Proposition 3 shows our atomic-variable approach maintains the no-missed-signal property.

**Proposition 3.** *Our atomic-variable approach provides no-missed-signal property.*

*Proof.* Suppose there are some threads waiting on a global predicate $P$ that has become true. $\hat{P}$ consists of only atomic variables that establish a happens-before relationship with any subsequent `get` call on those variables. Without loss of generality, suppose thread $T$ is the last thread that updates $\hat{P}$ and makes $\hat{P}$ true. $T$ can evaluate $\hat{P}$ correctly by using `get` calls on atomic variables of

$\hat{P}$. In our approach, $T$ signals a thread waiting on $P$. □

### 4.2.3 Critical-Clause Approach

The atomic-variable approach attempts to accurately evaluate global predicates. In this section, we discuss another approach that approximately evaluates local predicates to decide if threads waiting on global predicates should be signaled.

In order to efficiently detect global predicates that have become true, threads waiting on global predicates must analyze their predicates and keep records before they go to a waiting state. These records are used to accelerate the process of detecting which global predicate has become true. The idea behind the records is that a global predicate is false because some of its clauses are false. The global predicate can become true only if those clauses become true. We call these clauses critical. Our system observes critical clauses and signals threads waiting on global conditions only when their critical clauses become true. Critical clauses are defined next.

**Definition 12** (Critical Clause)**.** *Given a Boolean predicate $P : X \to \mathbb{B}$, and a state $G$ such that $P$ is false in $G$, we say $C$ is a critical clause for $P$ with respect to $G$ if and only if the following three properties are satisfied.*

1. *$C$ is also false in $G$.*
2. *For any state $H$, if $C$ is false in $H$, then $P$ is also false in $H$. That is, $P \Rightarrow C$.*

*3. $C$ is a pure disjunction of local predicates.*

Informally, these properties mean that notifications based on $C$ starting from state $G$ will provide no-missed-signal property. Since $C$ is a pure disjunction of local predicates, it can be evaluated locally by all the involved monitors. We call each of the local predicate in the critical clause, *a local critical clause.*

As a simple example, consider the predicate $P$ equal to `!Q1.isEmpty() && !Q2.isFull()`. Suppose $P$ is false in some state $G$. This means that either `Q1.isEmpty()` or `Q2.isFull()`. If `Q1` is empty, then the critical clause $C$ for $P$ is `Q1.isEmpty()`. The critical clause $C$ satisfies all three conditions: (1) $C$ is false in $G$, (2) so long as $C$ stays false, $P$ will stay false, and (3) it is a pure disjunction of local predicates. Therefore, instead of detecting $P$, the system simply detects and signals when $C$ becomes true.

We now describe Algorithm 3 that computes a critical clause $C$ for a general global predicate $P$ that is false under the state $G$. The algorithm is recursive and assumes that the global predicate can be viewed as an expression tree with local predicates as the leaves of the tree and the *or* and *and* operators as the internal nodes of the tree. Because the negation of a local predicate is also local, a boolean expression can therefore be written as an expression made of just disjunctions and conjunctions. Every boolean expression of local predicates can be put in this form by pushing the negation operator to the innermost level by using De Morgan's laws.

In Algorithm 3, lines 1-2 take care of the base case. If $P$ is a local predicate, then it also acts as its critical clause. Lines 3-5 take care of the case when $P$ is a conjunction of $P_1 \dots P_m$. Since $P$ is false, one of the conjuncts, say $P_i$, must be false and the algorithm returns `computeCritical(`$P_i, G$`)`. Finally, lines 6-7 take care of the case when $P$ is a disjunction of $P_1 \dots P_m$. In this case, the algorithm returns the disjunction of each of `computeCritical(`$P_i, G$`)`.

**Algorithm 3** `computeCritical(`$P$`, `$G$`)`

**Input:** A global predicate $P$, the current state $G$ such that $P$ is false in $G$
**Output:** Returns a critical clause $C$

1: **if** $P$ is local to a monitor $M_i$ **then**
2: **return** $P$
3: **if** $P = \wedge_{i=1}^{m} P_i$ **then**
4: $\exists P_i$, such that $P_i$ is $false$ under $G$
5: **return** `computeCritical(`$P_i$`, `$G$`)`
6: **if** $P = \vee_{i=1}^{m} P_i$ **then**
7: **return** $\vee_{i=1}^{m}$`computeCritical(`$P_i$`, `$G$`)`

**Proposition 4.** *Algorithm 3 returns a critical clause for $P$ with respect to $G$.*

*Proof.* We use induction on the depth of the expression tree for $P$.

**Base case:** $P$ is local to a monitor $M_i$: $C$ equals $P$ from the algorithm and therefore properties 1 and 2 are trivially true. Since $P$ is a local predicate, 3 is also true.

**Induction case for conjunction:** $P = \wedge_{i=1}^{m} P_i$, $P$ is false in $G$. Let $C =$ `computeCritical(`$P_i$`, `$G$`)` such that $P_i$ is false in $G$. We show properties 1, 2, and 3 are satisfied.

1. Since $P_i$ is false in $G$ and $P_i$ has fewer operators, from induction we get that `computeCritical(`$P_i$`, G)` is also false in $G$.

2. Now suppose that $C$ is false in $H$. Again, by induction, $C$ is false in $H$ implies $P_i$ is false in $H$. Therefore, $P$ is also false in $H$.

3. Since $P_i$ has fewer operators than $P$, `computeCritical (`$P_i$`, G)` is a pure disjunction of local predicates by induction.

**Induction case for disjunction:** $P = \vee_{i=1}^{m} P_i$, $P$ is false in $G$. Let $C = \vee_{i=1}^{m}$ `computeCritical(`$P_i$`, G)`.

1. We show that $C$ is false in $G$. Since $P$ is false in $G$, all $P_i$ are false. Since $P_i$ is false and $P_i$ has fewer operators than $P$, `computeCritical(`$P_i$`, G)` is also false for all $i$. Hence, their disjunction $C$ is also false.

2. Now suppose that $C$ is false in state $H$. Since $C$ is a disjunction, it implies that $\forall i$, `computeCritical(`$P_i$`, G)` is false in $H$. From induction, we get that $\forall i, P_i$ is false in $H$. From the definition of $P$, we get that $P$ is false in $H$.

3. $\forall i$, `computeCritical(`$P_i$`, G)` is a pure disjunction of local predicates. Therefore, $C = \vee_{i=1}^{m}$ `computeCritical (`$P_i$`, G)` is also a pure disjunction of local predicates.

□

For example, consider the predicate $P = (v \vee w \vee \neg x \vee y) \wedge (x \vee z)$ in Fig. 4.2 (which is in conjunctive normal form). Then, `computeCritical(`$P1$`,` $G$`)` returns one of the disjunctive clauses that is false in $G$. Assume that $(v \vee w \vee \neg x \vee y)$ is false. Based on line 6 in Algorithm 3, we conclude that the clause $(v \vee w \vee \neg x \vee y)$ is critical. Its set of local critical clauses are $C_1 = v \vee w$, $C_2 = \neg x$, and $C_3 = y$.

Consider the predicate $Q = (v \wedge w \wedge \neg x) \vee (\neg w \wedge x) \vee (y \wedge z)$ in Fig. 4.2 (which is in disjunctive normal form). Then `computeCritical(`$P_2$`,` $G$`)` returns a disjunctive clause with one literal from each of the conjunctive clause such that the literal is false in $G$. For example, if we find $v$ is false in the first minterm, $\neg w$ is false in the second minterm, and $z$ is false in the third minterm, then we derive the critical clause $v \vee \neg w \vee z$. Its set of local predicates for the critical clauses are: $D_1 = v \vee \neg w$, $D_2 = false$, $D_3 = false$ and $D_4 = z$.

Our system maintains the global predicates, condition variables, and their critical-clause tables. Fig. 4.2 demonstrates an example. The symbol $\bullet$ indicates a condition variable. There are two predicates $P$ and $Q$ in the system, where $v, w \in X_1$, $x \in X_2$, $y \in X_3$, and $z \in X_4$.

Every monitor $M_i$ keeps a list of all related global conditions. Any thread $T$ that acquires the monitor needs to check if there is any related global condition that has become true before it releases $M_i$. For example, consider the thread $T$ that acquires monitor $M_1$. Before releasing $M_1$, $T$ checks if it needs to signal threads waiting on $P1$ in Fig. 4.2. $T$ looks up the table of $P1$

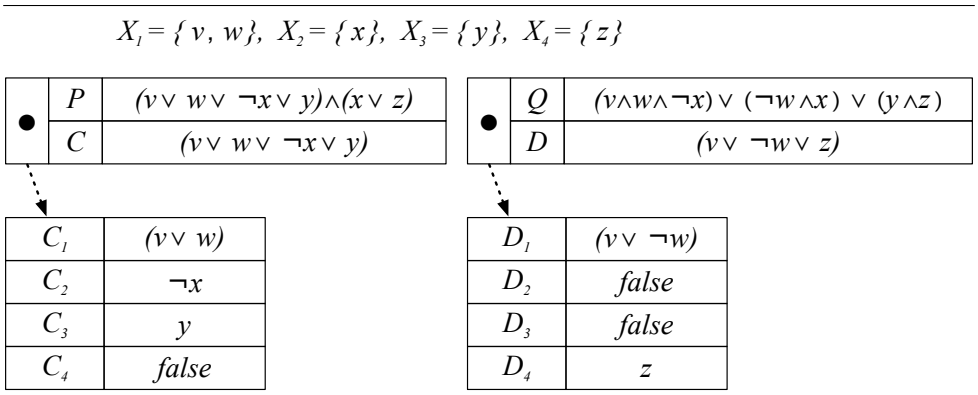


Figure 4.2: The Critical-Clause Example

and evaluates $C_1$ to decide whether threads waiting on $P$ should be signaled. This signaling rule is shown in Algorithm 4.

**Algorithm 4** `signalGlobalCondition(`$M_i$`)`

**Input:** A monitor $M_i$
**Output:** Signal threads waiting on global conditions with true $C_i$

```
1: for each global predicate P related to M_i do
2:    if table.get(M_i) is true then
3:       signal a thread t waiting on P
```

**Proposition 5.** *Our critical-clause approach provides no-missed-signal property.*

*Proof.* Suppose the thread $T$ is the last thread to wait on a global predicate $P$ that has become true. Since $T$ went to a waiting state, $P$ must be false at that point of time. Hence $T$ derived $C = \vee_{i=1}^{n} C_i$ a critical clause where each $C_i$ is local to $M_i$ using Algorithm 3. Now, since $P$ is true, $\vee_{i=1}^{n} C_i$ must be

true by Def. 12. There is one $C_i$ that is true. Hence, there must exist another thread $R$ after $T$ such that $R$ changed the state of monitor $M_i$ and made $C_i$ true. $R$ signals a thread waiting on $P$ according to our signaling rule. □

### 4.2.4 Global Conditions with Complex Predicates

Our approach cannot handle complex predicates because threads cannot correctly evaluate complex predicates by acquiring a lock for only one monitor object. For example, the predicate `Q1.size() > Q2.size()` cannot be evaluated unless both monitor locks of `Q1` and `Q2` are acquired. However, if we conservatively assume the complex condition to be true whenever one of its related monitor is updated, our approaches can still satisfy the no-missed-signal property at the risk of false signals. The threads waiting on the global condition will be signaled after all other non-complex conditions are met. The notified threads can correctly evaluate the complex predicate by acquiring all locks.

## 4.3 Evaluation

In this section, we evaluate the performance of our prototype implementation on two sets of problems. The first set of problems relies on `multisynch` statements but not on global conditions. The second set considers problems that use global predicates.

All the experiments are conducted on a machine with four Intel Xeon E7-4850 10-core CPUs (total 80 hyper-threads), running at 2 GHz with 32

KB L1, 256 KB L2, and 24 MB LLC, respectively. Compilation and execution both are performed with Oracle Java 1.7 (64-bit VM).

For every experiment, we ran the program 17 times, and report the mean value of 15 runs after discarding the highest and lowest values.

### 4.3.1 Evaluation for `multisynch` Statements

We compare the performance of our `multisynch` implementation with fine-grained locking and software transactional memory. Each implementation is denoted as following notation: `FL`: implementation using fine-grained locking with Java's `ReentrantLock`, `TM`: implementation using DeuceSTM transactional memory [AKZ10, KSF10], and `MS`: implementation using our `multisynch` statement.

#### 4.3.1.1 Examples Using `multisynch` Statements

**Dining Philosopher** This problem requires coordination among philosophers sitting around a table and is described in [Dij71]. Each philosopher alternatively eats and think. This is a saturation test so that there no extra operation performed in thinking or eating. In the fine-grained approach, odd philosophers pick their left-hand forks first and even philosophers pick their right-hand forks first for eating so that there is no deadlock in the system. For the transactional memory implementation, we use a boolean variable to indicate a fork. A philosopher needs to atomically set the left and right fork as true before eating and unset them after eating. In our `multisynch` approach,

every fork is a monitor object. A philosopher uses our `multisynch` statement with its two forks as arguments to eat.

**genome+** The genome is an application in the STAMP benchmark [MCKO08]. It takes a large number of DNA segments and matches them to rebuild the original genome. The genome+ uses the recommended configurations and data sets in the original paper [MCKO08].

#### 4.3.1.2 Results

Fig. 4.3 plots throughput of operations for the dining philosopher problem for three implementation techniques. Both FL and MS are around 3 – 14× faster than TM. This is because the test is saturation and keeps accessing critical data. The TM implementation suffers from frequent conflicts. Our MS implementation is slightly slower than FL implementation in the most cases and better than FL in some cases. The results indicate our approach is scalable and comparable to the fine-grained implementation.

Fig. 4.4 plots runtime for genome+ benchmark. Our MS implementation and FL implementation outperform TM around 4 – 15×. Note that, the runtime of TM implementation surges as the number of threads increases. This phenomenon indicates TM implementation performs poorly under high-contention. The performances of FL and MS are almost identical. Furthermore, MS implementation is stable and scalable since the runtime is steady as the number of threads increases. The results indicate that our approach can be as

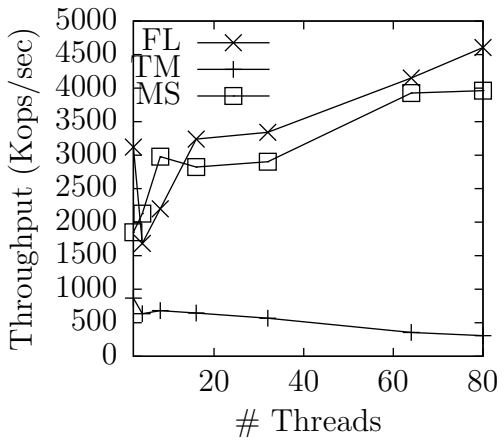


Figure 4.3: Throughput for the Dining Philosopher Problem

efficient as fine-grained lock approach.

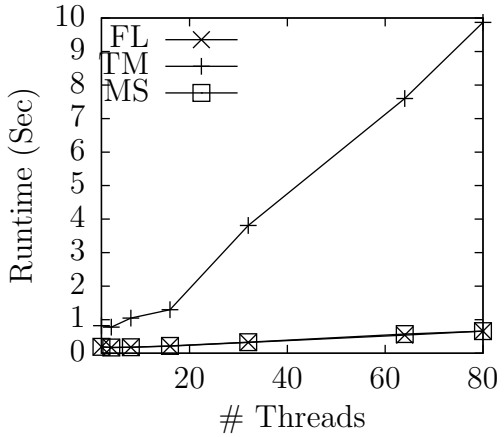


Figure 4.4: Runtime for genome+

### 4.3.2 Evaluation for Global Condition Problems

In this section, we study the throughputs of global condition problems among different implementations. We denote the implementations with the fol-

lowing notation: GL: using coarse-grained locking with Java's `ReentrantLock`, TM: using DeuceSTM transactional memory [AKZ10, KSF10], AS: using an automatic signaling approach in which threads waiting on a global condition are always signaled by a thread releasing a monitor related to the condition, AV: using our atomic-variable approach, and CC: using our critical-clause approach.

#### 4.3.2.1 Applications and Examples

We show some global conditional synchronization problems across multiple objects. We focus on applications that involve global conditions or composition operations of monitors. Traditional monitor may solve these problems by using a coarse-grained lock.

**Pizza Store Problem** This problem is described in Chapter 1. Consider a pizza store with two types of threads: cooks and suppliers. The cooks loop forever, first waiting for ingredients, and then making a pizza. The cooks may require different ingredients to make different types of pizza. The suppliers also loop forever, producing ingredients when they are insufficient. Fig. 1.6 demonstrates the code snippet for this problem using our constructs. A cook thread waits till it has enough quantity of each of the resources it needs. This is achieved by using the global predicate in `waituntil` statement. Each of the ingredients, cheese, tomato and pepperoni, is a different monitor object and the entire operation is done under `multisynch` to guarantee atomicity.

**Distributed Discrete-Event Simulation** Discrete event simulation [Mis86] is helpful for studying and analyzing realistic complex systems. We show our system can be used in distributed discrete-event simulation. Consider a message-passing system consisting of multiple processes. A process has a set of incoming neighbors, which send events that are ready to be performed to the process. Each event has a time stamp and the process has to perform events in increasing time order. For each neighbor, the process has a queue to keep its events in increasing time order. Whenever the process wants to perform an event, it needs to ensure that the event has the smallest time stamp among all of queues. Fig. 4.5 shows the code snippet for above example. The variable `queues`, an array of queues, indicates the event queues of neighbors. Each queue is a monitor object. The function `extractFirstEvent` examines the first event for each queue and return the event with the smallest time stamp. Traditional monitor approaches would need to use a coarse-grained lock for this application.

```
1 multisynch(queues) {
2   waituntil(!queues[0].isEmpty() && ...
3       && !queues[n - 1].isEmpty());
4   e = extractFirstEvent(queues);
5 }
```

Figure 4.5: The code snippet of Distributed Discrete-Event Simulation

#### 4.3.2.2 Results

Fig. 4.6 demonstrates the results for the threads that atomically take an item from a queue and put that item in another queue. There are 80 queues with 2048 buffer size. Every thread randomly selects a source queue and a destination queue every time. As can be seen, all three automatic signaling approaches outperform coarse-grained approach. The reason is that the coarse-grained approach limits parallelism since every thread needs to acquire the same coarse-grained lock to perform operations. Transactional memory approach is inefficient since it does not have efficient constructs for conditional synchronization problems. Note that, the AS approach is more efficient than AV and CC. The reason is that the buffer size is huge in this experimental setup so that the global synchronization condition is true in the most of the cases. Therefore, the AS approach does not introduce many false signals. Furthermore, AS does not have any overhead on predicate evaluations for signaling threads.

Fig. 4.7 plots the results for the pizza store problem in which we have 15 ingredients and 15 different types of pizza. Each cook thread randomly makes one type of pizza at any given time. As can be seen, both atomic-variable and critical-clause approaches significantly outperform the coarse-grained approach in all runs. The reason is that cooks can concurrently make different types of pizzas that have no identical ingredient; however, coarse-grained lock approach limits parallelism since every cook needs to acquire the same coarse-grained lock to make a pizza. Note that, the AS approach is extremely slow in

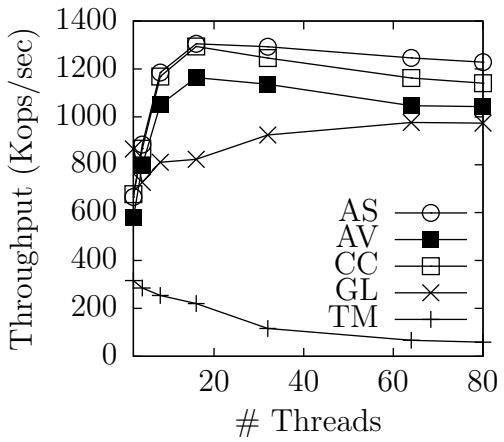


Figure 4.6: Throughput for Atomic Take and Put

comparison with AV and CC. This phenomenon can be explained by Fig. 4.8 that depicts the number of false evaluations of threads waiting on global conditions. The AS approach requires around $2-7\times$ higher number of evaluations than AV and CC. In the AS approach, a thread releasing a monitor related to a global condition always unconditionally signals a cook waiting on the condition. Therefore, this approach has a large number of false signals. Note that, CC has a slightly higher number of false evaluations than AV while CC slightly outperforms AV. This can be explained by that AV has higher overhead to maintain and evaluate predicate than CC.

Fig. 4.9 shows throughput of distributed discrete-event simulation. In the simulation, we consider a process thread with variant numbers of incoming neighbors, mimicked by threads that generate tasks with increasing time order. In this experiment, the coarse-grained approach performs better than AV and CC for smaller number of threads (less than 32 threads). The reason is that

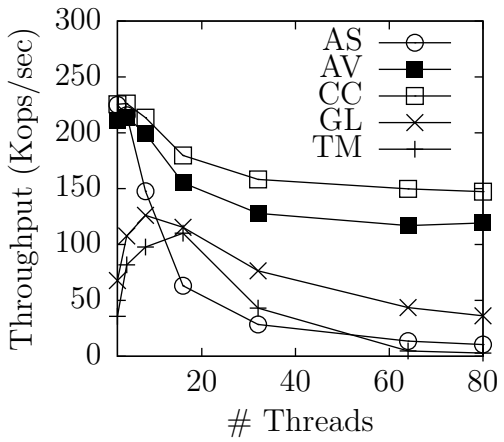


Figure 4.7: Throughput for the Pizza Store Problem

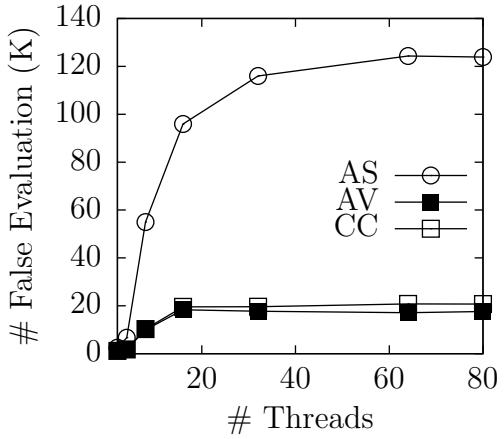


Figure 4.8: False Evaluation for the Pizza Store Problem

it requires the process thread to waiting on a global condition that involves all monitor objects, so that the process thread need to require all monitors before executing a task, which does not provide better parallelism than a single coarse-grained lock. For the number of threads is high (more than 32 threads), AV and CC outperforms coarse-grained lock approach. This can be

explained by high contention on the single coarse-grained lock.

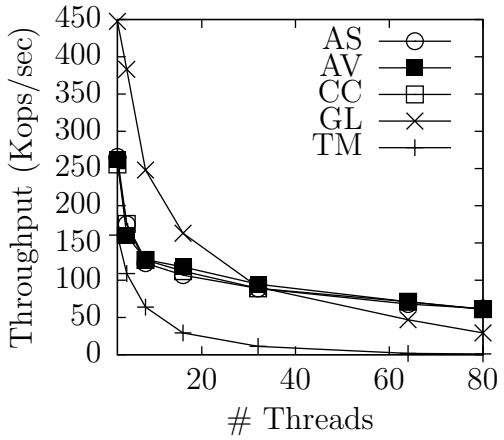


Figure 4.9: Throughput for the Discrete-Event Simulation

## 4.4 Related Work

Java and C++ use conditional variables with explicit notification for conditional synchronization. Programmers need to explicitly use `signal` or `signalAll` to wake a thread waiting on a condition variable. Using the wrong notification (`signal` versus `signalAll`), or forgetting to do the notification are frequent sources of bugs in Java parallel programs. In AutoSynch [HG13], there is no notion of condition variables and it is the responsibility of the system to automatically signal appropriate threads. However, *AutoSynch*, and indeed all traditional monitor approaches [Hoa74, Han75], can deal only with conditions local to a single monitor object. They cannot handle complex conditional synchronizations that involve multiple monitor objects.

Transactional memory based systems also cannot handle global condi-

tional synchronization easily. The thread itself needs to recheck every time there is an update of the variables in the transaction. The C++ transactional memory constructs specification proposal [LW14](Section 7.11) points out that there is still no solution to support conditional synchronization in transactional memory because no monitor can be passed to the condition variable in an atomic block. Transactional memory implementations [HMJH05, SR13] would have to check the global predicate every time and then abort and retry the transaction when the condition is false. Implementations such as [DS09] use global lock based solutions for waiting and thus limit parallelism. Using transaction-friendly condition variables is proposed in [WLS14, WS16], in which programmers need to declare additional condition variables and explicitly wait/signal on those variables. This approach, however, brings back all the hazards of explicit signaling.

## 4.5 Summary

Writing parallel programs that provide high performance and scalability is still a challenging task for most programmers. The main reason is that there is no simple parallel programming paradigm that guarantees multi-object mutual exclusion as well as simple conditional synchronization and compositionality. Our proposed design of multi-object monitors is a step in the direction of providing such constructs. We have shown that our proposed framework of multi-object synchronization monitors provides significant improvement over traditional lock-based monitors. We believe that with further research efforts

in this direction, and further optimizations in our implementation, our proposed technique can lead to significant improvements in programmability as well as performance of shared memory parallel programs.

# Chapter 5

# Logical Compositionality

In addition to the automatic notification of global conditions, our system also aims to solve the compositionality problem [HS08] which is best understood with the following producer-consumer example. Consider two instances `Q1` and `Q2` of a blocking queue implementation, with dequeue method signature being `take()`. As the queue is blocking, a call to `take()` will block the calling thread if the queue is empty. Consider the problem of dequeueing from either of these instances, and storing the returned item into a variable `x`. If both queues are empty, then we should block until an item is available in either one. Solving this problem using the traditional monitor based blocking queue implementations is extremely difficult [HS08]. An ad hoc solution is to use a global lock and a lock-free/wait-free implementation of `take()`. But this solution does not scale because a global lock inhibits parallelism. Even with transactional memory [HS08] the problem is not easy to solve. The thread itself needs to recheck every time there is an update of the variables in the transaction. If there are multiple threads waiting on that condition, then each one of them will recheck the condition.In our framework, the code is simply one statement: `x = Q1.take() OR x = Q2.take()`.

To deal with composition, our system supports four operations: `OR`, `AND`, `selectone`, and `selectall`. In this chapter, we first introduce guarded monitor methods, which can be used together with our composition operations. Next, we show both synchronous and asynchronous implementations for our logical compositionality. The results indicate that our synchronous composition operations are extremely efficient in comparison with asynchronous approach, transactional memory, and global lock approach.

## 5.1 Guarded Monitor Methods

Our composition operators are applicable to *guarded* monitor methods as defined next.

**Definition 13** (Guarded Monitor Method)**.** *A member method of a monitor object is called* guarded *if any* `waituntil` *statement in the method is at the beginning of the method. The boolean predicate $P$ for* `waituntil` *statement is called the* pre-condition *of the method.*

A monitor method with no `waituntil` statement is also considered as a guarded method in which the pre-condition $P$ is *true*.

Both `OR` and `AND` have two operands. The `OR` operation executes either of the two operand while the `AND` operation executes the two operands in any order. The order of operations is defined based on the evaluation of the pre-conditions (of operand monitor methods) at runtime. If a result is required from either of these operator calls, then programmers can assign

the results of the operand methods to variables, e.g., `(x = Q1.take()) OR (x = Q2.take())`, `Q1.put(item) AND Q2.put(item)`. The `selectone` and `selectall` can be considered as the generalized constructs for `OR` and `AND`, respectively. Both constructs have four arguments, initialization expression, termination expression, increment expression, and the guarded function. The first three arguments are identical to the for-loop, providing a way to iterate over a collection of monitor objects instead of just two.

We restrict operands of our composition operators to guarded monitor methods because allowing `waituntil` statements in the middle makes it impossible to guarantee atomicity without rollbacks. Since our implementation is lock based, a method call cannot be rolled back (as in transactional memory implementations). If a middle `waituntil` statement is false in a method call, our system cannot abort it and rollback. However, this restriction does not limit the applicability of our system. If a monitor method has a `waituntil(P)` in the middle, we can split the method into two guarded monitor methods such that the second method begins with the `waituntil` statement. Furthermore, our global condition synchronization allows `waituntil` in the middle. This restriction is only for composition operators.

## 5.2 Synchronous Execution of Compositional Operations

For synchronous implementation, there are two phases for each composition operation, the speculative phase and the synchronized phase. In the speculative phase, our system tries to iterate over a set of operands and check

if they can be executed until the composition operation is completed or no operand is executable. If the operation is not completed, we go to the synchronized phase. In this phase, we need to acquire the locks of all operands. Those locks are acquired according to their ids just as in the `multisynch` statement. The details of our implementations are shown next. Note that, `OR` and `selectone` have the same implementation while `AND` and `selectall` have the same implementation.

We use two helper methods as shown in Algorithm 5 and 6, where the set of operands (guarded monitor methods) is denoted as $O$. The nonblocking method `executeOneOperand` iteratively checks and executes if there is some executable operand. The `createExecutablePredicate` method generates the disjunction of the set of pre-conditions of operands. If the generated global predicate is true, then one of the operands has become executable.

**Algorithm 5** `executeOneOperand(`$O$`)`

**Input:** A set of operands $O$
**Output:** Execute an executable operand and return it or return null

```
1: ret := null
2: for each operand o ∈ O do
3:    if o.tryLock()  then
4:       if  o.pre_condition is true then
5:          execute o
6:          ret := o
7:       o.unLock()
8: return ret
```

Algorithm 7 shows the implementation for `selectone` and `OR` operators. Our system invokes the `executeOneOperand` method in the speculative

**Algorithm 6** `createExecutablePredicate(`$O$`)`

**Input:** A set of operands $O$
**Output:** Return $P$ indicating some operand is executable

```
1: P := false
2: for each operand o ∈ O do
3:     P := P ∨ o.pre_condition
4: return P
```

phase. If there is an executable operand, our system executes it and returns. Otherwise, it goes to the synchronization phase. We derive a global predicate $P$ by invoking the `createExecutablePredicate` method. Then we acquire all locks of operands and wait on the global predicate. Once the predicate becomes true, we can find an executable operand and execute it.

**Algorithm 7** `orComposition(`$O$`)`

**Input:** A set of operands $O$ ▷ Speculative Phase

```
1: if executeOneOperand(O) ≠ null then
2:     return                                  ▷ Synchronized Phase
3: P := createExecutablePredicate(O)
4: lockOperands(O)
5: waituntil(P)
6: executeOneOperand(O)
7: unlockOperands(O)
```

The implementations for our `AND` and `selectall` are shown in Algorithm 8. It is similar to the implementation of `OR` and `selectone`.

**Algorithm 8** `andComposition(`$O$`)`

**Input:** A set of operands $O$

```
 1: repeat                                              ▷ Speculative Phase
 2:    o :=executeOneOperand(O)
 3:    O := O − o
 4: until O = ∅ or o = null
 5: if O = ∅ then return                                ▷ Synchronized Phase
 6: repeat
 7:    P := createExecutablePredicate(O)
 8:    lockOperands(O)
 9:    waituntil(P)
10:    for each o ∈ O such that o.pre_condition is true do
11:       execute o
12:       o.unlock()
13:       O := O − o
14:    unlockOperands(O)
15: until O = ∅
```

## 5.3 Asynchronous Execution of Compositional Operations

We extend ActiveMonitor as described in Chapter 3 to provide asynchronous compositional operations. Guarded monitor methods can be converted to equivalent monitor tasks as defined in Defn. 10. Monitor tasks are compositional in nature. Suppose a monitor method declares $n$ in the form of `waituntil` $(P_i)$ $S_i$, where $1 \leq i \leq n$, to enforce that the set of statements $S_i$ must be executed iff predicate $P_i$ is true. To execute this method, ActiveMonitor generates $n$ tasks such that each task $t_i$ has a precondition $P_i$ and a corresponding set of statements $S_i$. More importantly, with monitors allowed to be 'active' as threads, ActiveMonitor enables compositionality of

blocking operations across different monitor objects. The following section demonstrates our implementation in details.

### 5.3.1 Implementing Composition Operators in ActiveMonitor

For both of these operators, ActiveMonitor stipulates that the operands — monitor method calls — must be on different monitor objects. This is needed to guarantee program order under conditional synchronization across monitors. The pre-processor raises a parsing error if this constraint is not met. If the constraint is met, the pre-processor generates the equivalent task for each operand conjunct/disjunct clause, and stores them as a collection within a container object that is directly mapped to the operator. Note that if there are multiple statements with same operator usage (`selectone` and `selectall` statments), all of them are treated as independent, and a container object is generated for each of them. The operand calls are then replaced by the submission of tasks to the corresponding monitors.

The runtime library delegates the tasks to their respective target servers (monitor threads) for execution. It also observes all the preconditions of these tasks and ensures that they are executed whenever these conditions are met. For `AND` (`selectall`) operator, the worker that called the operator is forced to wait for the completion of all the tasks. This is achieved by forcing the worker to evaluate the future reference returned by each task submission. Once all the futures have been evaluated, the result of the operator is stored in the designated storage if needed. For example, consider the statement: `Q1.put(a)`

`AND Q2.put(b)`; where `Q1` and `Q2` are two bounded-queues. Then the framework generates two tasks `t1` and `t2`, and submits them to the server threads of `Q1` and `Q2`. It then registers the returned future references with the worker thread that called the statement, and forces it to evaluate both the futures such that the worker remains blocked until both `a` and `b` are enqueued in `Q1` and `Q2` respectively.

For statements with `OR` (`selectone`) operator, the container object that holds the tasks — that are equivalent to the constituent disjunct clauses of `OR`— also maintains an atomic flag called *taken*. This flag is initially set to false. To execute the composition statement, the runtime first parks the calling worker thread, and submits the tasks stored in the container object to their respective server (monitor). Recall that the *relay invariance* of our automatic signaling ensures that whenever the pre-condition of some task of the `OR` is met, its server thread is signaled. To guarantee that only one clause (equivalent task) of the `OR` statement is executed, the server thread performs a compare-and-swap (CAS) operation on the *taken* flag of the container object. If and only if the server's CAS operation succeeds, ie. the value of the flag was false and this server set it to true, the server proceeds to execute the task submitted to it. Since only one thread can succeed in atomically setting the flag, we are guaranteed that only one of the tasks will be executed. Every other server thread that executes the CAS and fails can discard its task for the `OR` statement.

## 5.4 Evaluation

In this section, we study the throughputs of compositionality problems among different implementations. We denote the implementations with the following notation: GL: using coarse-grained locking with Javas ReentrantLock, TM: using DeuceSTM transactional memory [AKZ10,KSF10]. Three different automatic signaling approaches are implemented with our synchronous compositionality mechanism as described in Section 5.2, where AS: using an automatic signaling approach in which threads waiting on a global condition are always signaled by a thread releasing a monitor related to the condition, AV: using our atomic-variable approach, CC: using our critical-clause approach. Our asynchronous approach as described in Section 5.3 is denoted as AM.

### 5.4.1 Application: Multicast Channels Communication

A web server needs to handle numerous requests from clients. Suppose the server uses a queue for each client to keep its requests. The server has to fulfill clients' requests as efficiently as possible but does not care about the order of requests. Fig. 5.1 demonstrates the implementation by using our constructs. We can use our composition construct `selectone` to `take` a message among `queues`, indicating the request queues of clients. Another way to implement it is by using concurrent queues; however, the server needs to busy wait and check if there is any message in queues with this implementation. If we want to avoid busy waiting, we need to use a coarse-grained lock and conditional variables.

```
1 while (true) {
2   Message msg;
3   selectone(int i = 0; i < queues.length; ++i; msg = queues[i].take());
4   handleMessage(msg);
5 }
```

Figure 5.1: The Code Snippet of Multicast Channels Communication

### 5.4.2 Results

Fig. 5.2 demonstrates the throughput for multicast channels communication. We consider a server thread with varying number of clients, simulated by threads that generate requests. The goal of this experiment is to evaluate the performance of our composition operations. Our synchronous approaches, AV, CC, and AS, significantly outperform coarse-grained lock. However, the asynchronous implementation AM is not as efficient as synchronous implementations. AM gains benefits only when the number of threads is low. This can be explained by the fact that the overhead of creating tasks and monitor threads offsets the parallelism of asynchronous executions. This result highlights the benefit of synchronous composition operations because the coarse-grained lock approach and asynchronous approach are much slower in comparison to AV, CC, as well as the AS approach. The reason is that our composition operations are nonblocking whenever there is some executable operand. Note that, the software transactional memory approach performs better than the coarse-grained lock when the number of threads is low (less than 32). However, it is still extremely inefficient in comparison with our approaches.

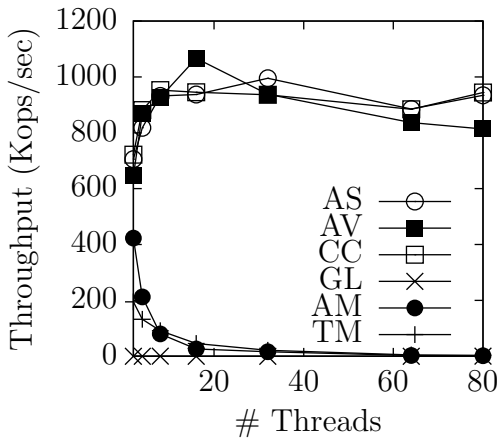


Figure 5.2: Throughput for Multicast Channels Communication

## 5.5 Summary

In this chapter, we tackle the compositionality problem of traditional memory. Our system provides four composition operations: `OR`, `AND`, `selectone`, and `selectall` for guarded monitor methods so that programmers can use such methods together without additional efforts. We discuss both synchronous and asynchronous approaches for our composition operations. The experimental results indicate that our synchronous approach not only simplifies compositionality problem but also gains performance.

# Chapter 6

# Future Work

This chapter discusses the future work for our parallel programming paradigms and framework.

## 6.1 Monitors with Read/Write Lock

In our current framework, we assume that every monitor method execution is mutually exclusive. However, multiple read-only monitor methods should be executed simultaneously for further parallelism. We plan to provide the key word `readonly` to indicate that a monitor method is read-only and to allow parallel execution on read-only methods. To achieve this, we plan to adopt the read/write lock. For each monitor, we create a read/write lock for it. If a thread invokes a read-only method, the thread tries to acquire the read lock of the monitor and then executes the operations of the method by itself. Otherwise, if a thread invokes a monitor method updating the shared data, then the thread needs to acquire the write lock or to create corresponding monitor tasks and submits to its monitor thread as described in Chapter 3. For each monitor thread try to execute monitor tasks, the thread needs to acquire the write lock of the monitor.

## 6.2 Asynchronous Monitor with Fairness and Priority

In Chapter 3, we focus on the performance of the monitor but not the flexibility of programs. In this section, we discuss that providing priority and fairness policy for monitor methods can give programmers more choices when developing software systems.

We first define different policies and then discuss the flexibility that our system can gain with those policies.

**Definition 14** (Safe Policy)**.** *If a task $T$ of a monitor $M$ is executable and there is no other executable task of $M$, then $T$ is executed.*

Safe policy guarantees that executable tasks are eventually executed if there is no other executable task. The exactly execution order of tasks depends on the runtime situation. Our asynchronous monitor discussed in Chapter 3 provides this safe policy.

The formal definition of our fairness policy is as follows.

**Definition 15** (Fairness Policy)**.** *The order must satisfy safe policy. In addition, for all task $t$ of a monitor $M$, $t$ is executed if there is no other task $s$ of $M$, such that $s$ is executable and $sub(s) < sub(t)$.*

Fairness policy guarantees that the executed task is the earliest submitted executable task, so that it avoids starvation and prevents accessing out of date information.

For a task $t$ of a priority method call, we use $priority(t)$ to denote the priority of task $t$. Our system guarantees that higher priority executable task is always been executed earlier than others. The following defines our priority policy.

**Definition 16** (Priority Policy)**.** *The order must satisfy safe policy. In addition, for all task $t$ of a monitor $M$, $t$ is executed if there is no other task $s$ of $M$, such that $s$ is executable and $priority(s) > priority(t)$.*

Now we show a motivating example that programmers can gain benefit from our framework with these three policies in Fig. 6.1, a readers/writers monitor example. In the example, the monitor can be fairness, writer preference, and reader preference without modifying the code but by choosing different policies as described in Proposition 6 and 7. Thus, programmers are able to gain more flexibility when design their programs with these three policies. Furthermore, programmers can even implement only one program for different purposes by choosing different policies. We proposed that programmers can use annotations to choose polices. First, programmers does not need to add any annotation for safe policy because safe policy is the default policy in our system. For fairness policy, programmers add `@FairnessPolicy` annotation to the monitor class. For example, programmers can add `@FairnessPolicy` in front of line 1 in Fig. 6.1 to implement a fairness readers/writers monitor. Finally, programmer can use `@Priority` annotation with a number to indicates the priorities of methods. For example, to implement a writer preference readers/writers monitor, programmers can add `@Priority(2)` annotation to both

```
1  public monitor class ReadersWritersMonitor {
2    Thread waitingWriter;
3    boolean isWriting;
4    int rcnt;
5    public ReadersWritersMonitor() {
6      waitingWriter = null;
7      isWriting = false;
8      rcnt = 0;
9    }
10   public void startRead() {
11     waituntil(waitingWriter != null && !isWriting);
12     rcnt++;
13   }
14   public nonblocking void endRead() {
15     rcnt--;
16   }
17   public void startWrite() {
18     waituntil(waitingWriter != null);
19     waitingWriter = Thread.currentThread();
20     waituntil(rcnt == 0 && isWriting == false &&
21             waitingWriter == Thread.currentThread());
22     waitingWriter = null;
23     isWriting = true;
24   }
25   public nonblocking void endWrite() {
26     isWriting = false;
27   }
28 }
```

Figure 6.1: The examples of readers/writers monitor with priority annotation

`startWrite()` and `endWrite()`; and add `@Priority(1)` to both `startRead()` and `endRead()` to indicate that writers have high priority.

**Proposition 6.** *The readers/writers shown in Fig. 6.1 is fairness using the fairness policy.*

**Proposition 7.** *The readers/writers shown in Fig. 6.1 is reader preference*

*when using the priority policy that all startRead() calls have higher priority than the startWrite() calls; it is writer preference when using the priority policy that all startWrite() calls have higher priority than startRead() calls.*

Safe policy maximizes throughput while fairness policy deals with starvation and staleness/freshness issues. Furthermore, priority policy provides programmers more choices. In our system, programmers should be able to decide a priority for every monitor method call by specifying the policy through annotations for monitor classes and method calls.

Chapter 3 describes only the implementation of safe policy. To implement fairness and priority policy, we may rely on the concurrent first-in-first-out (FIFO) queue and the concurrent priority queue to store submitted tasks. Then the monitor thread takes tasks from the queue for executing.

## 6.3 Enhancing Support of Asynchronous Monitor

Our approach of creating an independent thread for a monitor object and coupling this with asynchronous executions of monitor methods is aimed at improving the performance of multi-threaded programs by increasing parallelism. Understandably, the benefits provided by this approach can be tangible only if the lifespan of such monitor objects is long enough that the improved cache locality and parallelism introduced by our approach overcomes the additional resource and time costs involved in creation of threads and overheads associated with delegation based critical section executions. Hence, our ap-

proach is not beneficial for applications that use short-lived monitor objects. We highlight three categories of current limitations of our prototype implementation. We plan to overcome these limitations in our future work.

### 6.3.1 Exception Handling

For an asynchronous method invocation, after submitting its corresponding task to the executor, the invoker does not need to wait for the completion of the task. The task is executed in parallel by a monitor thread. Thus, if an exception occurs during its execution, the thread that submitted it must be notified of this exception. Our framework must have an exception handler that keeps a log of every exception and provides different mechanisms for programmers to handle exceptions in the asynchronous method. The users may choose to ignore the exceptions or they can specify a maximum number of times a task may be considered for automatic re-tries. Furthermore, our system should also provides a hook so that the programmer can write their custom exception handler.

### 6.3.2 Thread Dependent Variables/Functions

In our current implementation, thread dependent variables and functions within a monitor method cannot be used directly in the `Runnable` or `Callable` object that is used in task generation by our approach. The reason is that the tasks are executed by the monitor thread and not by the worker thread. For example, suppose there is a monitor method that invokes

`Thread.currentThread()`, if we directly add this statement to the generated `Runnable` object (in the task), then this method's invocation at runtime will return the reference to the monitor thread when it is executed. However, it is obvious that the intent of this call inside the monitor method was to refer to the worker thread. To handle this situation, currently, we require the programmer to perform reference copy and storage and storage in thread-local variables. For read operations of thread dependent variables and functions, the worker thread would need to evaluate them outside the monitor, and store the result with final variables. These final variables can be accessed by the runnable and callable objects. An additional constraint/limitation applies for the case of write operation on thread dependent variables. For write operations, if the monitor method is non-blocking then the results can be stored as intermediate data. The worker thread then writes these results back to its local variable after the task is executed.

### 6.3.3 Blocking recursive method

Our current pre-processor does not support blocking recursive method on monitors. This is because the number of the method invocations to be made at the runtime is non-deterministic. Thus, we cannot know how many tasks we need to create at pre-processing time. In addition, since the method is blocking, the monitor thread will get blocked when it recurs.

# Appendix

## A.1 The $H_2O$ Problem

```
1  public monitor class H2OBarrier {
2      int numAvailableO = 0;
3      int numAvailableH = 0;
4      int numWaitingO = 0;
5      int numWaitingH = 0;
6      public void OReady() {
7          ++numWaitingO;
8          waituntil((numAvailableO > 0) || (numWaitingH >= 2));
9          if (numAvailableO == 0) {
10             numWaitingH -= 2;
11             numAvailableH += 2;
12             numWaitingO -= 1;
13         } else {
14             numAvailableO -= 1;
15         }
16     }
17     public void HReady() {
18         ++numWaitingH;
19         waituntil((numAvailableH > 0) || (numWaitingO >= 1 && numWaitingH >= 2));
20         if (numAvailableH == 0) {
21             numWaitingH -= 2;
22             numAvailableH += 1;
23             numWaitingO -= 1;
24             numAvailableO += 1;
25         } else {
26             numAvailableH -= 1;
27         }
28     }
29 }
```

Figure A.1: The $H_2O$ our framework

## A.2 Round-Robin Access Pattern

```
1  public monitor class RoundRobinMonitor {
2      private int numProc;
3      private int currId;
4
5      public RoundRobinMonitor(int numProc) {
6          this.numProc = numProc;
7          currId = 0;
8      }
9
10     public void access(int myId) {
11         waituntil(currId == myId);
12         ++currId;
13         currId %= numProc;
14     }
15 }
```

Figure A.2: The round robin access pattern using our framework

## A.3 Ticket Readers/Writers Monitor Example

```
1 public monitor class ReadersWritersMonitor {
2   int rcnt;
3   int tickets, serving;
4   public ReadersWritersMonitor() {
5     rcnt = 0;
6     tickets = serving = 0;
7   }
8   public void startRead() {
9     int ticket = tickets;
10    tickets++;
11    await(ticket == serving);
12    rcnt++;
13    serving++;
14    return;
15  }
16  public void endRead() {
17    rcnt--;
18  }
19  public void startWrite() {
20    int ticket = tickets;
21    tickets++;
22    await(ticket == serving && rcnt == 0);
23  }
24  public void endWrite() {
25    serving++;
26  }
27 }
```

Figure A.3: The ticket readers/writers monitor using our framework

## A.4 Sleeping Barber Problem

```
1  public monitor class BarberShopMonitor {
2    public void cutHair() {
3      waituntil(numFreeSeat < maxSeat);
4      ++numFreeSeat;
5      ++numAvailableBarber;
6    }
7    public boolean waitToCut() {
8      if (numFreeSeat == 0) {
9        return false;
10     }
11     --numFreeSeat;
12     waituntil(numAvailableBarber > 0);
13     --numAvailableBarber;
14     return true;
15   }
16 }
```

Figure A.4: The sleeping barber problem using our framework

# Bibliography


[AKZ10] Yehuda Afek, Guy Korland, and Arie Zilberstein. Lowering STM Overhead with Static Analysis. *LCPC*, pages 31–45, 2010.

[And99] Greg R Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[BBF+95] Peter A. Buhr, Peter A. Buhr, Michel Fortier, Michel Fortier, Michael H Coffin, and Michael H Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.

[BH05] Peter A. Buhr and Ashif S. Harji. Implicit-Signal Monitors. *ACM Transactions on Programming Languages and Systems*, 27(6):1270–1343, November 2005.

[BM06] David A Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite. *Atlanta, GA, February*, 2006.

[But97] David R Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[CDH+13] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message Passing or

Shared Memory: Evaluating the Delegation Abstraction for Multicores. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems: 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings*, pages 83–97. Springer International Publishing, Cham, 2013.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40(10):519–538, 2005.

[CHP71] P J Courtois, F Heymans, and D L Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[CZF04] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SIAM International Conference on Data Mining*, pages 442–446, February 2004.

[Dij59] E W Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1(1):269–271, 1959.

[Dij65] Edsger Wybe Dijkstra. Cooperating Sequential Processes, Technical Report EWD-123. Technical report, 1965.

[Dij68] Edsger W Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[Dij71] Edsger Wybe Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[dim] 9th DIMACS Implementation Challenge - Shortest Paths. Technical report.

[DS09] Polina Dudnik and Michael M Swift. Condition Variables and Transactional Memory: Problem or Opportunity? In *The 4th ACM SIGPLAN Workshop on Transactional Computing*, pages 1–10, February 2009.

[FHS04] Faith Fich, Danny Hendler, and Nir Shavit. On the Inherent Weakness of Conditional Synchronization Primitives. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, pages 80–87, New York, NY, USA, 2004. ACM.

[FK12] Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 257–266, New York, NY, USA, 2012. ACM.

[GJS+14] James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java concurrency in practice*. Addison-Wesley Professional, 2006.

[Hal85] Robert H Halstead, Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Han75] Per Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Trans. Softw. Eng.*, 1(1):199–207, 1975.

[Har01] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[Her88] Maurice P Herlihy. Impossibility and Universality Results for Wait-free Synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 276–290, New York, NY, USA, 1988. ACM.

[HG13] Wei-Lun Hung and Vijay K Garg. AutoSynch: an automatic-signal monitor based on predicate tagging. In *PLDI '13: Pro-*

*ceedings of the 2013 ACM SIGPLAN conference on Programming language design and implementation*, pages 253–262, 2013.

[HH78] C A R Hoare and C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), August 1978.

[HHL+06] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A Lazy Concurrent List-based Set Algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, pages 3–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[HHWW90] Maurice P Herlihy, Maurice P Herlihy, Jeannette M Wing, and Jeannette M Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[HIST10] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364, New York, NY, USA, 2010. ACM.

[HL14] Maurice Herlihy and Zhiyu Liu. Well-structured Futures and Cache Locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 155–166, New York, NY, USA, 2014. ACM.

[HLR10] Tim Harris, James R Larus, and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2010.

[HM93] Maurice Herlihy and J Eliot B Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *ISCA*, pages 289–300, 1993.

[HMJH05] Tim Harris, Simon Marlow, Simon L Peyton Jones, and Maurice Herlihy. Composable memory transactions. *PPOPP*, pages 48–60, 2005.

[Hoa74] C A R Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-free Stack Algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 206–215, New York, NY, USA, 2004. ACM.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[IS14] Joseph Izraelevitz and Michael L Scott. Brief Announcement: A Generic Construction for Nonblocking Dual Containers. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 53–55, New York, NY, USA, 2014. ACM.

[Kes77] J L W Kessels. An Alternative to Event Queues for Synchronization in Monitors. *Communications of the ACM*, 20(7):500–503, 1977.

[KH14] Alex Kogan and Maurice Herlihy. The Future(s) of Shared Data Structures. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 30–39, New York, NY, USA, 2014. ACM.

[KP11] Alex Kogan and Erez Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 223–234, New York, NY, USA, 2011. ACM.

[KP12] Alex Kogan and Erez Petrank. A Methodology for Creating Fast Wait-free Data Structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 141–150, New York, NY, USA, 2012. ACM.

[KSF10] G Korland, N Shavit, and P Felber. Noninvasive concurrency with Java STM. In *MultiProg 2010: Third Workshop on Programmability Issues for Multi-Core Computers*, 2010.

[LDT+12] Jean-Pierre Lozi, Florian David, Ga e l Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[Lea05] Doug Lea. The Java.Util.Concurrent Synchronizer Framework. *Sci. Comput. Program.*, 58(3):293–309, 2005.

[LM11] V Luchangco and V J Marathe. Revisiting Condition Variables and Transactions. In *The 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

[LW14] Victor Luchangco and Michael Wong. Transactional Memory Support for C++, February 2014.

[MCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. *IISWC*, pages 35–46, 2008.

[Mis86] Jayadev Misra. Distributed Discrete-event Simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[MM02] Maged M Michael and Maged M Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel*

*Algorithms and Architectures*, pages 73–82, New York, NY, USA, 2002. ACM.

[NM14] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 317–328, New York, NY, USA, 2014. ACM.

[OTY99] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA'99). World Scientific*, 1999.

[PcRS14] Darko Petrovi c, Thomas Ropars, and André Schiper. Leveraging Hardware Message Passing for Efficient Thread Synchronization. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 143–154, New York, NY, USA, 2014. ACM.

[SDSC14] Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors. *Delegation Locking Libraries for Improved Performance of Multithreaded Programs*, Cham, 2014. Springer International Publishing.

[SR13] Alexandre Skyrme and Noemi Rodriguez. From Locks to Transactional Memory: Lessons Learned from Porting a Real-world

Application. In *The 8th ACM SIGPLAN Workshop on Transactional Computing*, pages 1–9, March 2013.

[SSAT+06] Bratin Saha, Bratin Saha, Ali-Reza Adl-Tabatabai, Ali-Reza Adl-Tabatabai, Richard L Hudson, Richard L Hudson, Chi Cao Minh, Chi Cao Minh, Benjamin Hertzberg, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Request Permissions, March 2006.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. ACM Request Permissions, August 1995.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., February 2000.

[TBKP12] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free Linked-lists. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 309–310, New York, NY, USA, 2012. ACM.

[WLS14] Chao Wang, Yujie Liu, and Michael F Spear. Transaction-friendly condition variables. *SPAA*, pages 198–207, 2014.

[WS16] Chao Wang and Michael Spear. Practical Condition Synchronization for Transactional Memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 32:1–32:16, New York, NY, USA, 2016. ACM.

[you] Yourkit Java Profiler.