Automatic-Signal Monitors with Multi-Object Synchronization

Wei-Lun Hung and Vijay K. Garg
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
wlhung@utexas.edu, garg@ece.utexas.edu

***Abstract*—Current monitor based systems have many disadvantages for multi object operations. They require the programmers to (1) manually determine the order of locking operations, (2) manually determine the points of execution where threads should signal other threads, (3) 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 paper, we propose new monitor based methods that provide automatic signaling for global conditions that span multiple objects.**

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. Furthermore, 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.

***Keywords*-automatic signal, implicit signal, monitor, synchronization, concurrency, parallel**

I. Introduction

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 [1], [2]. 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 paper, 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.

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.

To enable global conditional synchronization, our system provides `multisynch` construct with monitor objects for multi-object synchronization. Our system automatically guarantees atomicity for the `multisynch` statement. Furthermore, our `multisynch` constructs also avoid deadlock. Fig. 1 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 addresses another important problem with current mechanisms for synchronization called the *compositionality problem* [3]. 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

```
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: The Code Snippet of Dining Philosophers Problem

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 tasks. For this example, the programmer can use the construct as `(x = Q1.take()) OR (x = Q2.take())`. This `OR` construct is a blocking mechanism that takes multiple blocking methods as its argument and executes exactly one of them whenever the *enabling* condition of one of the monitor becomes true.

In our system, we introduce an automatic-signal monitor 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 condition to become true, programmers can use the `waituntil(P)` 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. Furthermore, the condition of a `waituntil(P)` statement can be global and involve multiple monitors. *AutoSynch* [4] has similar constructs for automatic signaling; however, it does not consider synchronization among multiple monitors. This paper focuses on multi-object synchronization.

Our framework provides the following constructs for writing monitor-based programs:

1) `multisynch`: a statement for multi-object synchronization. The statement is similar to Java's `synchronized` statement, but it can take an arbitrary number of monitors as argument.
2) `waituntil(P)`: a statement for conditional waits and notifications. The statement requires a Boolean predicate as an argument. The predicate can span multiple monitor objects. This statement can be in any monitor method or any `multisynch` statement.

```
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 static void takeAndPut(
15       BoundedQueue srcQ, BoundedQueue destQ) {
16     multisynch(srcQ, destQ) {
17       waituntil(!srcQ.isEmpty() && !destQ.isFull());
18       destQ.put(srcQ.take());
19     }
20   }
21   public static void putInAQueue(BoundedQueue Q1,
22       BoundedQueue Q2, Object item) {
23     Q1.put(item) OR Q2.put(item);
24   }
25   public static Object putInAnyQueue(
26       BoundedQueue[] queues, Object item) {
27     selectone(int i = 0; i < buffs.length; ++i;
28         queues[i].put(item));
29   }
30 }
```

Figure 2: The Bounded Queue Example

3) `OR/AND/selectone/selectall`: operators for logical composition of monitor guarded methods. The order of operations is defined based on the evaluation of the pre-conditions (of operand guarded monitor methods) at runtime.

Fig. 2 shows the bounded queue implementation that demonstrates the actual usage of `waituntil(P)` statement, the `multisynch` statement, and our composition constructs. In this example, producers put an item into a shared queue, while consumers take an item out of the queue. We do not show the `take` method since its implementation is similar to `put`. 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 16 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(P)` in line 17 for global conditional synchronization. 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.

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. 3 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(P)` 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 &&
4       pepperoni.quantity()>= 5);
5   cheese.consume(6);
6   tomato.consume(3);
7   pepperoni.consume(5);
8 }
```

Figure 3: 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 [5]–[7]. Techniques implemented in multi-object synchronization can also be used for conditional synchronization in transactional memory.

This paper is organized as follows. The related work is discussed in Section II. Section III discusses mutual exclusion among multiple objects. Section IV presents efficient algorithms for global condition notification. The logical composition operators are shown in Section V. The proposed methods are then evaluated with experiments in Section VI. Section VII gives the concluding remarks.

II. 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 parallel programs. In *AutoSynch* [4], 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 [8], [9], 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 conditional synchronization easily. The C++ transactional memory constructs specification proposal [10](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. In transactional memory implementations [1], [2], 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. Implementations such as [11] use global lock based solutions for waiting and thus limit parallelism. Using transaction-friendly condition variables is proposed in [12], [13], 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.

III. Multi-Object Mutual Exclusion

It is the responsibility of our system to ensure mutual exclusion without deadlock 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 in any arbitrary order as shown in line 1 of Fig. 3. If an array of monitor objects is a parameter of a `multisynch` statement, the system ensures mutual exclusion for all elements of the array.

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 [14]. 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.

We note here that all these techniques are well known and C++ 11 also provides `std::lock()` for multiple mutex objects [15]; the main contribution of this paper is in mechanisms for detecting global conditions on these objects that requires implementation of `multisynch`.

IV. 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 the 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 1 (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.*

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.

A. 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 `(!Q1.isEmpty())` and `(!Q2.isEmpty())` local predicates 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 that are Boolean expression of local predicates. The case of global predicates involving complex predicates is discussed in Sec. IV-D.

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(P)` statement are read-only and free from side effects. Any evaluation of those predicates does not update any variable or change the state G.

B. Atomic-Variable Approach

A thread cannot evaluate global predicates correctly without acquiring global locks because it cannot observe all con-

current 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. Java provides atomic varibles in the package `java.util.concurrent.atomic`. 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 1 shows our atomic-variable approach maintains the no-missed-signal property.

Proposition 1. *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. ■

C. 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 2 (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 1 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 1, 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 1 computeCritical(P, G)

Input: A global predicate P, the current state G such that P is false in G
Ensure: 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 2. *Algorithm 1 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 (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 1, 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 (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 shows an example. The symbol • 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 $z \in X_4$.

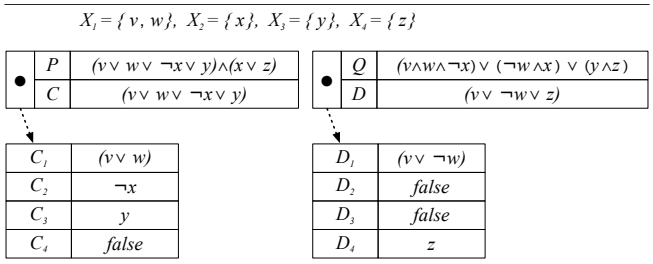

Figure 4: The Critical-Clause Example

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. T looks up the table of $P1$ and evaluates C_1 to decide whether threads waiting on P should be signaled. This signaling rule is shown in Algorithm 2.

Algorithm 2 signalGlobalCondition(M_i)

Input: A monitor M_i
Ensure: 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 3. *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 1. Now, since P is true, $\vee_{i=1}^{n} C_i$ must be true by Def. 2. 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. ■

D. 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.

V. Logical Composition Operations

In addition to the automatic notification of global conditions, our system supports composition of monitor methods. We support two forms of composition: `OR` and `AND`. These composition operators are applicable to *guarded* monitor methods as defined next.

Definition 3 (Guarded Monitor Method)**.** *A member method of a monitor object is called* guarded *if any* `waituntil(P)` *statement in the method is at the beginning of the method. The Boolean predicate P for* `waituntil(P)` *statement is called the* pre-condition *of the method.*

A monitor method with no `waituntil(P)` 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 `select-one` 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(P)` 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(P)` statement is false in a method call, our system cannot abort it and rollback. Note that, this restriction does not limit the applicability of our system. Our global condition synchronization still works for a monitor method has a `waituntil(P)` in the middle. This restriction is only for composition operators.

A. Implementation

In our system, 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 3 and 4, where the set of operands (guarded monitor methods) is denoted as O. The `executeOneOperand` is a nonblocking method that 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 3 `executeOneOperand(`O`)`

Input: A set of operands O
Ensure: 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 5 shows the implementation for `selectone` and `OR` operators. Our system invokes the `executeOneOperand` method in the speculative phase. If there is an executable operand, our system executes it and returns. Otherwise, it goes to the synchronization

Algorithm 4 `createExecutablePredicate(`O`)`

Input: A set of operands O
Ensure: Return P indicating some operand is executable
1: $P := false$
2: **for each** operand $o \in O$ **do**
3: $P := P \vee o$.pre_condition
4: **return** P

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 5 `orComposition(`O`)`

Input: A set of operands O ▷ Speculative Phase
1: **if** `executeOneOperand(`O`)` $\neq null$ **then**
2: **return** ▷ Synchronized Phase
3: $P :=$ `createExecutablePredicate(`O`)`
4: `lockOperands(`O`)`
5: **waituntil(**P**)**
6: `executeOneOperand(`O`)`
7: `unlockOperands(`O`)`

Algorithm 6 `andComposition(`O`)`

Input: A set of operands O
1: **repeat** ▷ Speculative Phase
2: $o :=$`executeOneOperand(`O`)`
3: $O := O - o$
4: **until** $O = \emptyset$ or $o = null$
5: **if** $O = \emptyset$ **then return** ▷ Synchronized Phase
6: **repeat**
7: $P :=$ `createExecutablePredicate(`O`)`
8: `lockOperands(`O`)`
9: **waituntil(**P**)**
10: **for each** $o \in O$ such that o.pre_condition is true **do**
11: execute o
12: o`.unlock()`
13: $O := O - o$
14: `unlockOperands(`O`)`
15: **until** $O = \emptyset$

The implementations for our `AND` and `selectall` are shown in Algorithm 6. It is similar to the implementation of `OR` and `selectone`.

VI. Evaluation

In this section, we study the throughputs of global condition problems among different implementations. We denote the implementations with the following notation: GL: using coarse-grained locking with Java's `ReentrantLock`, TM: using DeuceSTM transactional memory [16], [17], 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. Table I summarized the notations we used.

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.

A. Application and Examples

We show 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.

The first application is the atomic `takeAndPut` method as shown in Fig. 2, in which a thread atomically takes an item from `srcQ` and puts the item in `destQ`. Since `srcQ` and `destQ` are different monitor objects, this problem involves the global condition.

The second problem is the piazza store problem as shown in Fig. 3, where cooks may require different ingredients to make different types of pizza so that the global predicate is needed to solve this problem.

The third problem is multicast channels communication problem. Consider a web service, a 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 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;
4       msg = queues[i].take());
5   handleMessage(msg);
6 }
```

Figure 5: The Code Snippet of Multicast Channels Communication

B. Results

Fig. 6 demonstrates the results for the threads that atomically take an item from a queue and put that item in another

notation	GL	TM	AS	AV	CC
implementation	coarse-grained locking	transactional memory	always signaling	atomic-variable	critical-clause

Table I: Notations for different implementations

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.

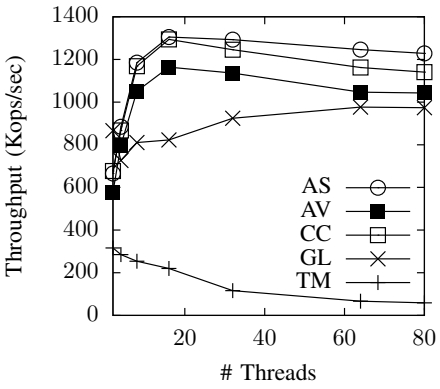

Figure 6: Throughput for Atomic Take and Put

Fig. 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 comparison with AV and CC. This phenomenon can be explained by Fig. 8 that depicts the number of false evaluations of threads waiting on global conditions. The AS approach requires around 2 – 7× 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.

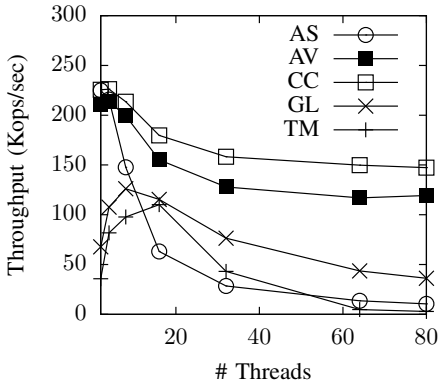

Figure 7: Throughput for the Pizza Store Problem

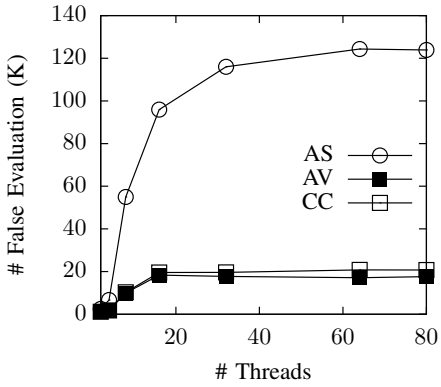

Figure 8: False Evaluation for the Pizza Store Problem

Fig. 9 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. AV, CC, and AS implementations significantly outperform coarse-grained lock. This result highlights the benefit of composition operations because the coarse-grained lock approach is much slower than to AV, CC, and 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.

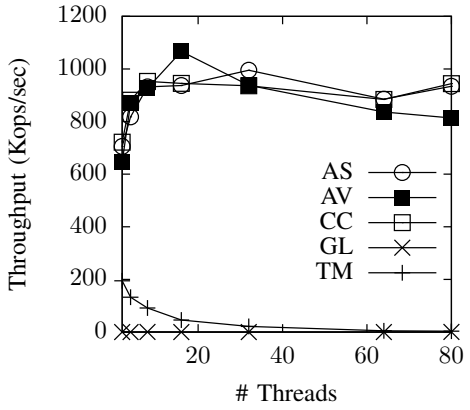

Figure 9: Throughput for Multicast Channels Communication

VII. Conclusions

Writing parallel programs that provide high performance and scalability is a challenging task for most programmers. One of the 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.

Acknowledgment

We thank Himanshu Chauhan and Yen-Jung Chang for many insightful comments to improve this paper.

References

[1] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy, "Composable memory transactions." *PPOPP*, pp. 48–60, 2005.

[2] A. Skyrme and N. Rodriguez, "From Locks to Transactional Memory: Lessons Learned from Porting a Real-world Application," in *The 8th ACM SIGPLAN Workshop on Transactional Computing*, Mar. 2013, pp. 1–9.

[3] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[4] W.-L. Hung and V. K. Garg, "AutoSynch: an automatic-signal monitor based on predicate tagging," in *PLDI '13: Proceedings of the 2013 ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 253–262.

[5] N. Shavit and D. Touitou, "Software transactional memory," in *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. ACM Request Permissions, Aug. 1995.

[6] T. Harris, J. R. Larus, and R. Rajwar, *Transactional Memory*. Morgan & Claypool, 2010.

[7] B. Saha, B. Saha, A.-R. Adl-Tabatabai, A.-R. Adl-Tabatabai, R. L. Hudson, R. L. Hudson, C. C. Minh, C. C. Minh, B. Hertzberg, and B. 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, Mar. 2006.

[8] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[9] P. B. Hansen, "The Programming Language Concurrent Pascal," *IEEE Trans. Softw. Eng.*, vol. 1, no. 1, pp. 199–207, 1975.

[10] V. Luchangco and M. Wong. (2014, Feb.) Transactional Memory Support for C++. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3919.pdf

[11] P. Dudnik and M. M. Swift, "Condition Variables and Transactional Memory: Problem or Opportunity?" in *The 4th ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2009, pp. 1–10.

[12] C. Wang, Y. Liu, and M. F. Spear, "Transaction-friendly condition variables." *SPAA*, pp. 198–207, 2014.

[13] C. Wang and M. Spear, "Practical Condition Synchronization for Transactional Memory," in *Proceedings of the Eleventh European Conference on Computer Systems*. New York, NY, USA: ACM, 2016, pp. 32:1–32:16.

[14] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java concurrency in practice*. Addison-Wesley Professional, 2006.

[15] C. S. Committee. (2011) Working draft, standard for programming language C++ . [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

[16] Y. Afek, G. Korland, and A. Zilberstein, "Lowering STM Overhead with Static Analysis." *LCPC*, pp. 31–45, 2010.

[17] 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.