ActiveMonitor: Asynchronous Monitor Framework for Scalability and Multi-Object Synchronization*

Wei-Lun Hung[1], Himanshu Chauhan[2], and Vijay K. Garg[3]

1 **University of Texas, Austin, USA**
`wlhung@utexas.edu`
2 **University of Texas, Austin, USA**
`himanshu@utexas.edu`
3 **University of Texas, Austin, USA**
`garg@ece.utexas.edu`

Abstract

Monitor objects are used extensively for thread-safety 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. Their current design and implementation is also ill-suited for thread synchronization across multiple thread-safe objects. We present ActiveMonitor – a framework that allows multi-object synchronization without global locks, and 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 constructs.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases concurrent/parallel programming, monitors, concurrency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.29

1 Introduction

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 [22]. 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 to their fullest. 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. Mutex-based monitor implementations have another limitation: method invocations

* Supported in part by NSF Grants CNS-1346245, CNS-1115808, and Cullen Trust. A short version of this paper appeared as a brief announcement in DISC'2014.

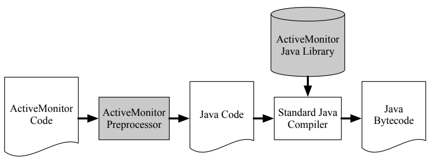

Figure 1 ActiveMonitor framework.

across multiple monitors cannot be combined easily. For example, given two thread-safe blocking queues, consider the problem of dequeueing an item from either of them. There is no easy solution to the problem of using mutex based synchronous monitors [19].

We present ActiveMonitor, a framework that provides significant programming ease in writing thread-safe programs, allows multi-object synchronization, as well as improves the runtime performance of these programs by exploiting asynchronous delegated executions on modern multi-core hardware. Extending our previous work AutoSynch [24], which provides waituntil keyword for automatic signaling and thread synchronization, ActiveMonitor framework enables asynchronous executions of critical sections, as well as method composition across monitor objects through simple constructs. Recall that monitors were 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 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 [35]. 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.

Using ActiveMonitor involves the following steps (Fig. 1 shows the framework overview):

(a) The programmer writes a monitor based parallel program using the ActiveMonitor keywords. These keywords are: monitor, waituntil, synchronous, asynchronous, and notthread-safe. He/she can use two additional operators OR and AND for compositionality across multiple monitor objects. ActiveMonitor 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.

(b) He/she then runs the ActiveMonitor 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 ActiveMonitor runtime library API in the generated code.

(c) 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.

ActiveMonitor enables operations that are not possible with traditional synchronous monitors. Solving the problem of removing an element from either of n blocking queues, where $n \geq 2$, is a challenging task with traditional monitors [19]. In ActiveMonitor it is just a matter of using the framework's OR construct: x = Q1.deqeue() OR Q2.dequeue() .... Similarly, the AND construct of the framework allows the programmer to aggregate results from multiple operations across different monitors. 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 (using Java[1]) 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 [30], and delegation technique [35] on most of these problems. In our current implementation of ActiveMonitor, 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 issues in § 9.

2 ActiveMonitor: Concepts & Design

In ActiveMonitor framework, each method of a monitor is a critical section – unless otherwise specified (using notthreadsafe keyword described shortly ahead). 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 [35] 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 [15, 10] 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 paper, 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. synchronous: keyword used in declaration of monitor methods. Such methods are made thread-safe but not delegated to the server (monitor thread) for execution.
4. 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 [12] instance which can be evaluated – a blocking call if the result is not yet available – to fetch the result.
5. notthreadsafe: this keyword in a method signature tells the framework to not generate thread-safe code for this method. incompatible with the previous two keywords: waituntil and asynchronous.
6. OR/AND: operators for logical composition of monitor methods. If a result is required from either of these operator calls, then the framework stipulates that all the operand

[1] Our technique is not limited to Java, and applies to any other modern programming language.

```
monitor class BoundedQueue<T> {
  T[] items;
  int putPtr, takePtr, count, size;
  BoundedQueue(int size) {
    this.size = size;
    items = new Object[size];
  }
  aysnchronous void put(T item) {
    waituntil(count < size);
    items [putPtr++] = item;
    putPtr = putPtr % size;
    ++count;
  }
  T take() {
    waituntil(count > 0);
    T x = (T)(items [takePtr++]);
    takePtr = takePtr % size;
    --count;
    return x;
  }
}
```

Figure 2 Bounded-Queue with ActiveMonitor.

method calls have the same return type. The order of operations is defined based on the evaluation of the pre-conditions (of operand monitor methods) at runtime.

Defaults: ActiveMonitor makes all monitor methods thread-safe by default. Each method that returns void and updates monitor state is asynchronous by default unless otherwise declared. Each method that returns a type value (and not a void) is made synchronous unless explicitly declared asynchronous by the user. Each read-only method – determined by static analysis of the program in the pre-processing/compilation phase – is also made synchronous irrespective of its return type. By doing so, the framework is able to use read-locks for such methods to exploit the inherent read parallelism in the program. The bounded queue implementation in Fig. 2 shows the actual usage of monitor, and asynchronous keywords, as well as the waituntil statement. Note that take() method will be made synchronous by the framework as it returns a value and is not explicitly declared asynchronous. As shown in the design overview of Fig. 1, the framework has two main components: a pre-processor and a runtime Java library. The pre-processor translates ActiveMonitor code into Java code. In addition, it also identifies the critical sections that are eligible for asynchronous execution. For each such method (critical section), the pre-processor generates its equivalent *task*.

It then replaces invocation of these methods (by application threads on monitor object) by submission of tasks to the server of the monitor. The runtime 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 work AutoSynch [24]. We briefly discuss its steps, and refer the reader to [24] for details. 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, their generation and compositionality in the next section.

3 Monitor Tasks

In ActiveMonitor, a monitor task is defined as follows.

▶ **Definition 1** (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 8–13) of the bounded-buffer program of Fig. 2. For this monitor method, the equivalent monitor task t is defined by the code of lines 9–12. For t, the precondition P is (count $<$ buffer_size); 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 10 and 11 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.

3.1 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 [15, 10]. We use *futures* [12] 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 [35, 15, 10], 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. If the programmer understands the implications of out-of-program-order asynchronous executions, and wishes to exploit them then he/she can relax the program order execution by passing an argument to the runtime library. This change usually results in higher program throughputs. A detailed discussion on this topic can be found in our technical report [1].

4 Runtime Library

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 work AutoSynch [24] to enable functionality (a) for task based asynchronous executions and for multi-object synchronization through OR/AND operators. We summarize the key concepts here, and refer the interested reader to [24] for details.

4.1 Automatic Signaling

In current programming languages/libraries conditional synchronization through mutexes requires programmers to explicitly associate conditional predicates with condition variables and call *signal* (*signalAll*) or *await* statements manually. In contrast, ActiveMonitor framework manages conditional synchronization and thread signaling, and relieves the programmer of their explicit handling. The programmer only needs to use the waituntil clause. The idea of automatic signaling was initially explored by Hoare [22], but rejected in favor of condition variables due to efficiency considerations. Buhr et al. [3] claim that automatic monitors are 10 to 50 times slower than explicit signals. This is mainly due the sub-optimal implementation techniques that result in excessive predicate evaluations for conditions and subsequent context switches. In [24], we provide an efficient mechanism that improves the automatic signaling performance tremendously.

We use three concepts 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 waiting. This results in an additional context switch. In our 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, ActiveMonitor derives a closure predicate P' of the predicate P, such that other threads can evaluate P'.

The idea of *relay invariance* is used to avoid *signalAll* calls in ActiveMonitor. We ensure 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. With relay invariance, the privilege to enter the monitor is transmitted from one thread to another thread whose condition has become true. This mechanism guarantees progress, and reduces the number of context switches by avoiding *signalAll* calls.

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.

We extend these concepts to task based executions by allowing conditions within asynchronous tasks. As defined in Defn. 1, each task has a boolean predicate P. This predicate captures the pre-condition for the task's execution. Before executing any task, the server thread must verify that this condition is true. If not, the task is not executable and the server does not execute it. The runtime handling of conditional synchronization for OR/AND operators is described in § 5.

4.2 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 [21] 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 2.** *Rules 1, 2 and 3 guarantee executions equivalent to lock-based executions.*

5 Compositionality: Multi-object Synchronization

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. Consider two instances Q1 and Q2 of a blocking queue implementation, with dequeue method signature being deq(). As the queue is blocking, a call to deq() 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. In ActiveMonitor, the code is simply one statement: x = Q1.deq() OR x = Q2.deq(). Solving this problem using the traditional mutex based blocking queue implementations is extremely difficult [19]. An *ad hoc* solution is to use a global lock and a lock-free/wait-free implementation of deq. But this solution does not scale because a global lock inhibits parallelism. Even with transactional memory [19] the problem is not easy to solve. To the best of our knowledge, no transactional memory implementation provides explicit wait/notify construct on individual thread-safe objects to release the CPU. An implementation [38] to allow waiting in transactional memory requires continuous loop based busy-waiting on conditions. Implementations such as [9] propose global lock based solutions for waiting and thus curb parallelism. Not only ActiveMonitor's asynchronous execution approach provides an elegant solution, but it also allows parallelism. Similarly, the AND operator allows conjunction of operations across multiple monitor objects, such that these operations can be performed in parallel.

5.1 Implementing AND & OR 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, 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 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.enq(a) AND Q2.enq(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 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.

6 Implementation

We now describe implementation details that make ActiveMonitor practical in terms of use with real world applications, as well as scalable and faster. Recall that unlike other delegation/combining implementations [35, 15, 10], 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 [15, 10] 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 [30], 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 [30] for threads. Using the lower level API allows a more fine-grained control on execution of these threads.

6.1 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 producers. 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. The pseudocode of this algorithm can be found in the technical report at [1].

Our BoundedQueue is backed by a linked-list: the items are stored in the nodes of the linked-list. 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, 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 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

Table 1 Short description of problems evaluated.

Name	Short Desc.	CS Work [Type]	Details
PSSSP	Parallel single-source-shortest-path using Dijkstra's algorithm [7] using priority queue.	$\mathcal{O}(\log n)$ [Heavy]	(a) USA road network graphs (b) R-MAT Graphs [5]
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 (b) Write-heavy: 0% reads, 50% insert, 50% delete (c) Mixed: 70% reads, 20% insert, 10% delete
RR	Round-robin monitor access from [24].	$\mathcal{O}(1)$ [Light]	each thread accesses monitor in a predefined round-robin manner based on thread-id.

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.

For the single consumer multiple producer use-case, the throughput of our implementation is significantly higher than those of queue implementations from Java's util.concurrent package. The throughput comparison results on a saturation based micro-benchmark can be found in [1].

6.2 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.

7 Evaluation

We implement monitor based solutions to multiple concurrency problems using ActiveMonitor, ReentrantLocks from JDK7, and combining [10] – 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 (relatively small number of operations) 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 (large number of operations within CS) provide increased opportunity for exploiting asynchrony and parallelism. Table 1 presents a summary of problems used for our evaluation.

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 underlying 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 [8], and the remaining three graphs: R16, R128, and R512 are generated using the GTGraph [2] generator suite. The three synthetic graphs – R16, R128, and R512 – have 5×10^4 vertices each, and 1.6×10^6, 1.28×10^7, and 5.12×10^7 edges respectively.

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 [26], 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 [40] 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.

7.1 Results

Fig. 3 plots the throughput of the three PSSSP implementations in edges traversed per unit time format. Given that the three synthetic R-MAT [5] 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.

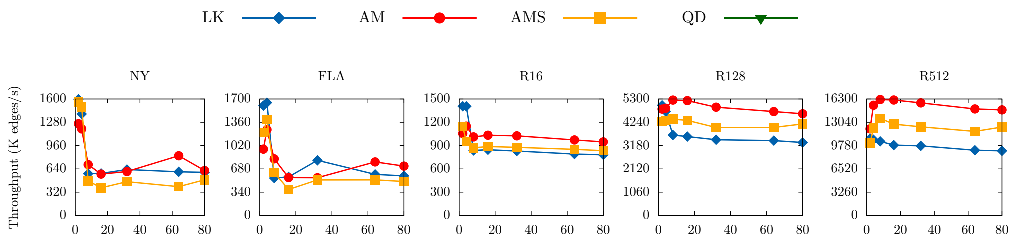

Figure 3 Throughput for PSSSP using priority queue (x-axis shows the number of threads)

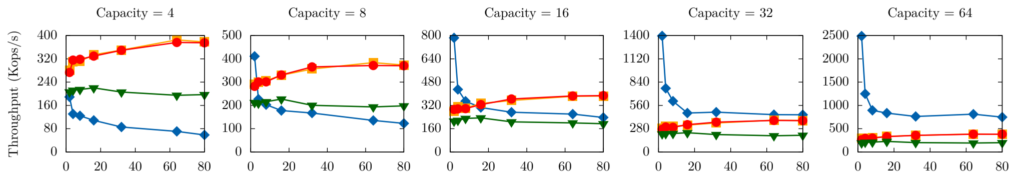

Figure 4 Throughput for Bounded FIFO Queue (x-axis shows the number of threads)

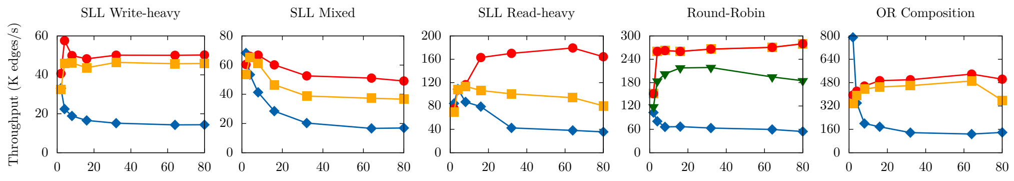

Figure 5 Throughput for SLL, RR, and OR problems (x-axis shows the number of threads).

Fig. 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 these cases where blocking due to unavailability of space occurs rarely. In the technical report version [1] of this paper, we analyze the performance benefit of asynchronous delegation by dropping the program order constraint and conducting the same experiment. In the new setting, AM performance further improves, and outperforms LK even on capacity of 32 (see Fig. 7 in [1]). These results highlight that when asynchronous executions are allowed to be out of program order, the overall throughput of the program can improve significantly.

Fig. 5 shows the operations throughput for the SLL and RR, and OR composition problems. In all the runs on these problems, (AM) significantly outperforms the read-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 ∼3–4× faster than LK on RR problem too. This is because the RR problem setup simulates a critical section that is similar to 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.

8 Related Work

Our idea of having monitor objects execute as independent threads is influenced by Hoare's proposed communicating sequential processes (CSP) [23] 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* [12, 30] to realize the idea of non-blocking/asynchronous executions. Kogan et al. [27] explore a similar approach in making use of *futures* for non-blocking executions. However, we explore changes to the general paradigm of monitors, whereas [27] only focuses on three data structures: stacks, queues, and linked-lists, none of them requiring conditional waiting. In addition, [27] 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 [17]. 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 [35, 10, 15] 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) [31] 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 [36, 4] have performed extensive experimental analysis in identifying the performance gains/losses with asynchronous message-passing like executions over synchronous shared memory ones. [36] 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, [4] 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) [26], 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 [18, 37] 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 [38, 32, 9]. 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 [6] 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 [6] 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 [13, 33, 16, 11, 20, 28, 29, 39, 34]. In addition, [14, 25] 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.

9 Discussion & Conclusion

Despite providing programming ease and performance benefits, our framework's current implementation has some limitations. We discuss them below.

Thread Dependent Variables and 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. This is because 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.

Concluding Remarks: 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. We believe, however, that with further research efforts in this direction our proposed technique can lead to significant improvements in programmability as well as performance of shared memory parallel programs.

References

1 ActiveMonitor: Technical Report Version. `http://pdsl.ece.utexas.edu/TechReports/2016/opodis_tr.pdf`.

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

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

4 Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra J. Marathe, and Mark Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *OPODIS*, pages 83–97, 2013.

5 Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

6 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. *Acm Sigplan Notices*, 40(10):519–538, 2005.

7 E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959. `doi:10.1007/BF01386390`.

8 9th DIMACS Implementation Challenge – Shortest Paths. `http://www.dis.uniroma1.it/challenge9/download.shtml`.

9 P. Dudnik and M. Swift. Condition variables and transactional memory: Problem or opportunity? In *The 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

10 Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the combining synchronization technique. *ACM SIGPLAN Notices*, 47(8):257–266, 2012.

11 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*, PODC'04, pages 80–87, 2004.

12 Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985.

13 Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.

14 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, WilliamN III Scherer, and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems*, pages 3–16. Springer Berlin Heidelberg, 2006.

15 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. In *SPAA*, pages 355–364, 2010.

16 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*, SPAA'04, pages 206–215, 2004.

17 Maurice Herlihy and Zhiyu Liu. Well-structured futures and cache locality. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'14, Orlando, FL, USA, February 15-19, 2014*, pages 155–166, 2014.

18 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA'93, pages 289–300, 1993.

19 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., 2008.

20 Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC'88, pages 276–290, 1988.

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

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

23 C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

24 Wei-Lun Hung and Vijay K. Garg. AutoSynch: An Automatic-signal Monitor Based on Predicate Tagging. In *PLDI*, pages 253–262, 2013.

25 Joseph Izraelevitz and Michael L. Scott. Brief announcement: a generic construction for nonblocking dual containers. In *ACM Symposium on Principles of Distributed Computing, PODC'14*, pages 53–55, 2014.

26 David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par*, 2014.

27 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. ACM, 2014.

28 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*, PPoPP'11, pages 223–234, 2011.

29 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*, PPoPP'12, pages 141–150, 2012.

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

31 Jean-Pierre Lozi et al. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX Annual Technical Conference*, pages 65–76, 2012.

32 V. Luchangco and V. J. Marathe. Revisiting condition variables and transactions. In *The 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

33 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, 2002.

34 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. ACM, 2014.

35 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.

36 Darko Petrovic, Thomas Ropars, and André Schiper. Leveraging hardware message passing for efficient thread synchronization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'14, Orlando, FL, USA, February 15-19, 2014*, pages 143–154, 2014.

37 Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'95, pages 204–213, 1995.

38 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*, 2013.

39 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*, PPoPP'12, pages 309–310, 2012.

40 TMWare – TMJava. `http://tmware.org/`.