**Motivation and Approach:** Monitors are a prevalent programming technique for thread synchronization in shared-memory parallel programs. The current design of monitors uses the *wait/notification* mechanism that blocks threads from executing without exclusive access to critical sections. We explore the idea of allowing non-blocking executions of monitor methods to improve the collective worker thread throughput and cache-locality in multi-threaded programs.

Our proposed framework, called *ActiveMonitor*, uses the concept of *futures* to provide non-blocking monitors by creating: (i) an *executor* for every monitor object (similar to remote-core-locking [3]), and (ii) *tasks* — equivalent to monitor methods — that are submitted to the executors. Our framework handles these steps automatically. The framework allows the programmer to use the keyword ‘*nonblocking*’ in signatures of monitor methods to make their execution non-blocking. Non-blocking methods return a *future* reference, which can be used to retrieve the result of method invocation. We re-interpret linearizability in this context, and enforce two rules to guarantee correctness: (a) all the tasks submitted to one monitor executor are processed in FIFO order. (b) tasks corresponding to a worker thread’s invocations of methods on different monitors are processed in program order (of the worker thread). See [4] for details.

**Evaluation:** We present the performance evaluation of our approach for two monitor-based problems in Java. In our benchmark, worker threads collectively perform 512000 operations in total on shared data protected by monitors. We vary the number of workers from 2 to 24 on a 24-way machine, and measure the time required for all the workers to complete their operations.

1. **Bounded-Buffer Problem:** Every producer’s *put* invocation is non-blocking, and every consumer’s *take* is blocking. Items are plain objects. We also compare runtimes of Java’s *ArrayBlockingQueue* based implementation (denoted by ABQ). We collect runtimes by varying: (a) number of workers for a fixed buffer-size (=4). (b) buffer-size for fixed number of producers/consumers (=16 each). (c) limit on non-blocking tasks allowed for fixed buffer-size (=4), and 16 producers and consumers each. Fig. 1 shows the results of these three experiments. Across all results, we use these legends for implementation techniques: LK: Java Reentrant locks, AS: *AutoSynch* [5], AM: *ActiveMonitor* (this paper).

2. **Sorted Linked-List Problem:** Worker threads insert or remove, with equal
probability, random integer values on a pre-populated linked-list of integers that is sorted in non-decreasing order. Both insert and remove operations are non-blocking. Each worker thread also performs some local operations outside the critical section (CS) between successive updates to the list. We collect the runtimes by varying: (a) number of workers, keeping local operations outside CS/worker fixed at 250. (b) number of workers as well as number of local operations outside CS. The results of these two experiments are shown in Fig. 2.

See [4] for extended evaluation on other monitor problems, details of CPU and memory consumption, and comparison with other implementation techniques.

References