Active Monitors

Javier Cortes (jc82563), Kyle Polansky (kpp446)

Abstract

Java monitors are commonly used for their simplistic mutual exclusion and thread safety properties when accessing shared resources. However, monitors block and execute critical sections serially, which means that they do not scale well in programs with many threads and lots of critical sections. To improve performance in these programs with heavy monitor use, we have created ActiveMonitor which allows asynchronous execution of critical sections. This is achieved by using a dedicated monitor thread instead of global synchronization. Testing on code with a high density of critical sections, our implementation of ActiveMonitor shows performance improvements over Java's monitor implementation.

Introduction

Java Monitor is a high level object that easily allows programmers to execute code in a mutually exclusive and synchronized manner [1]. This is a quick and easy way to ensure data consistency in critical sections, but they are not always the most efficient. For example, since there are many context switches between threads using blocking monitors, hardware prefetching and caching optimizations are often obsolete [2]. For example, if two separate threads execute on the same critical section, they each need to cache the global resources locally, leading to twice the work. Another example is that monitors are blocking and do not execute asynchronously. Finally, consider composite operations such as performing an OR

operation on two separate Monitor objects. This is a complex operation that cannot easily be realized due to Monitor's blocking.

ActiveMonitor aims to fix these issues. ActiveMonitor runs on a separate thread in the system, hence the active portion of the name. This thread is responsible for all incoming monitor requests, and will immediately return a future object. Incoming requests will be placed in a queue and until they are able to be executed. Worker threads are spawned to complete the critical section. The result is passed back to the ActiveMonitor, and the ActiveMonitor will return a future back to the calling code, allowing the calling thread to immediately continue unlock the blocking nature of Monitors [3].

ActiveMonitor is also responsible for re-ordering and grouping critical section requests for the same monitor objects on the same worker threads. The reordering ensures synchronization on each thread, and grouping by monitor lock object takes advantage of hardware caching on worker threads. To solve the issue of composite critical sections, ActiveMonitor runs completion logic when the worker threads return. For example, completing the first predicate in an OR request will dequeue the other task and immediately return the Future to the caller.

Description of Project

Our implementation of ActiveMonitor creates a background thread that listens for critical section requests. On request, the task will be placed in a queue. The queue will be executed in order, but can be reordered to take advantage of hardware caching and combinational requests. The reordering will follow the following rules: (1) only one worker thread can execute a task. (2) all tasks from a process must be submitted to worker threads in the order that they were submitted. (3) Only one worker may be executing a task per unique lock object at a time.

(1) Ensures that the tasks are only completed once as the calling thread has intended. (2) Ensures that tasks are completed in the order that they are received so serialization and starvation do not occur. (3) Ensures that the tasks satisfy mutual exclusion. The use a a queue ensures that all tasks will eventually get processed.

Design Alternatives

Executing code in parallel is complex and does not have a best case fits all answer. Our ActiveMonitors with Futures require the programmer to have a basic understanding of basic concurrent programming concepts. In [4], automatic signalling is used to automatically signal when variables are released from a critical section. While this is much easier for the programmer instead of needing to work with Future objects, it comes with a heavy performance overhead of either keeping track of all waited variables and their corresponding threads, or notifying all threads when a variable is changed. Additionally, [4] uses a preprocessor to convert constructs into Java code. This again means less lines of code for the programmer, but requires an extra step during compilation.

We have also experimented with using more than one ActiveMonitor thread and having them communicate with each other similar to [5]. However, this caused much overhead with shared resources, and the worker threads were often found to be the performance bottleneck. In the cases where ActiveMonitor queue operation logic was a bottleneck, coordinating shared resources between threads created an even bigger performance overhead.

Performance Results

ActiveMonitor is designed to improve performance in highly parallel code with heavy critical sections. ActiveMonitor will trivially inhibit performance compared to Java Monitor in

single processor systems, as the thread context switches are a lot of overhead compared a global monitor lock that is only ever accessed one at a time. In the case of a single ActiveMonitor worker thread, performance is also decreased. There is still the bottleneck of one thread in a critical section at a time similar to Java Monitor, but with the overhead of the monitor thread as well.

With more worker threads, performance is increased as multiple critical sections can execute simultaneously, obviously given that they are from different caller threads and are on different monitor objects.

In programs with a lower number of critical sections, the performance comparison is negligible. This is because often only one critical section is running at a time, and hence the parallelization is not taken advantage of. This is largely based on the algorithm being used. In cases where the algorithm takes advantage of asynchronous critical sections, small performance gains can be observed.

Adding a very high number of worker threads, we see diminishing returns. The number of processors in a system limits the number of threads running at once without context switching. Additionally, with our test cases, we do not have a large number of unique monitor objects to run on different threads concurrently. They must wait for a thread to finish to satisfy mutual exclusion. This means that even with a high number of worker threads, many are left idle due to the above restraints.

Conclusions

To conclude, it can be seen that our implementation of ActiveMonitor can achieve performance increases in certain situations with high thread count and high number of critical sections. In certain situations such as a low density of critical sections or a low number of threads, it can

also lead to performance degradation. Since writing efficient parallel code can often be a difficult task, our library is one more simple tool that programmers can use to speed up their applications.

References

[1] Synchronizing Threads with Java Monitors. http://www.csc.villanova.edu/~mdamian/threads/javamonitors.html.

[2] Daniel Sorin, Mark Hill, David Wood. Relaxed Memory Consistency. *A Primer on Memory Consistency and Cache Coherence*. Pages 76-79.

[3] Ray Toal. Mutual Exclusion. http://cs.lmu.edu/~ray/notes/mutualexclusion/.

[4] ActiveMonitor: Technical Report Version. http://pdsl.ece.utexas.edu/TechReports/2016/opodis_tr.pdf.

[5] C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666-667, 1978.

Appendix A - Educational Material

In addition to Java Monitors presented in chapter 3, ActiveMonitor can be used to get increased performance in applications with many threads and a heavy use of critical sections. An ActiveMonitor runs in a separate thread and listens for all critical section tasks that the program threads submit. The ActiveMonitor will immediatly return a Future object, allowing the calling program to run asynchronously. This is a major advantage over normal monitors that block when reaching a critical section. This means less time waiting for critical section blocks, and more time spent executing asynchronous code.

ActiveMonitors can also lead to performance degradation. In programs with infrequent critical sections, the overhead of a monitor thread and worker threads is slower than executing the critical section using a traditional monitor. As this is a high level programming construct, it will not be as fast as the lower level methods such as Semaphore.

Implementing ActiveMonitor is very simple. You simply create a new ActiveMonitor class, and pass in the method and parameters that you want to call. The class will automatically spawn worker threads ensuring that no two threads are ever working on the same Object, which is used as the monitor lock object. The ActiveMonitor will return a Future object, and the calling code can evaluate that Future object like normal.

Example of ActiveMonitor Usage:

```
Public class Test {
      ActiveMonitor mon = new ActiveMonitor();
      Foo foo = new Foo(nextName());
      //Call ActiveMonitor on foo object with method 'bar' arguments A,B
      Future resFuture = mon.execute(foo,"bar",A,B);
      //Do some other non-related work while waiting for Future
      resFuture.get() //Get result
}
private static class Foo{
        public int bar(Integer a, Integer b)
        {
            //Implementation not shown
        }
}
```

It is left up to the user to determine locations where ActiveMonitor can be used to improve efficiency.

ActiveMonitor Library Download: https://github.com/KylePolansky/EE360P