Review #2

Multiprocessor Support for Event-Driven Programs

N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazières, F. Kaashoek
USENIX 2003 Annual Technical Conference, General Track

Jean-Pierre Lozi

April 30, 2010

1 Problem

Many legacy applications have been designed without taking multithreading into account. Now that multiprocessor/multicore machines are fairly common, it would be useful to allow these applications to take advantage of hardware parallelism without having to rewrite large parts of the code. In the general case, this seems like an impossible task, since the usual tools used to manage parallelism, threads and locks, are complex to use, especially in the source code of a program that has been thought of as single-threaded from the beginning. A subclass of programs however, event-driven applications, appears to be a good candidate for easy parallelization. These programs' particularity resides in the fact that they register callbacks that are executed when certain events are triggered. Parallelizing the execution of some of these callbacks could be automated with an appropriate library, thus permitting hardware multiprocessing at a reduced cost.

2 Solution

The authors modified the event-based asynchronous programming library *libasync* in order to allow events to be processed in parallel. The new library, *libasync-smp*, allows the user to specify a color for each event to handle concurrency: two events of the same color are never executed in parallel.

It can be useful to split the execution of a callback in various blocks of various colors, since callbacks do not necessarily require the same level of isolation during the whole time of their execution. To this end, a new `cpucb()` function is provided. `cpucb()` allows scheduling the execution of a new callback with a new color as soon as the processor is idle, thus effectively permitting to create a new block using a new color. This splitting mechanism allows for more fine-grained optimizations.

libasync-smp has been optimized in two ways. First, it uses separate data structures (queues, lists) for each thread as much as possible in order to minimize the overhead caused by locks on these structures. Second, a *work stealing* mechanism allowing unused processors to steal jobs from other processors has been implemented in order to maximize the use of CPU resources.

3 Experiments

In order to evaluate the performance gain offered by the *libasync-smp* library, the authors tested it with 1) a simple web server of their creation that had no particularly processor-intensive operations and 2) the SFS file server whose callbacks were particularly computation-intensive due to encryption.

The experiments show that their solution was reasonably efficient in the first case (1.5x speedup on 4

cores[1], an improvement comparable to Apache's) and very efficient in the second case (2.5x speedup on 4 cores).

In each case, they showed that the performance gain was close to the upper bound (found by running N independant instances of the original program).

4 Benefits

The performance benefits are rather good given the small number of lines of code that had to be modified (90 out of 12,000 in the file server) to handle parallelism. Also, using *libasync-smp* is pretty intuitive: parallelization can be implemented incrementally (only one color at first and then more as independant callbacks are found).

5 Shortcomings

Applications for the *libasync-smp* library are rather limited since computation-intensive event-driven applications are not that common, especially those using *libasync*. Moreover, the color mechanism provided is easy to use but only allows expressing very simple concurrency relationships: it would be impossible to allow an infinite number of callbacks reading the same resource to be executed in parallel only when no other callback is writing the resource, for instance.

[1] This speedup appears to have been estimated relative to the performance of the modified application running on a single core, not the original application.