Thesis Proposal - Realizing Iterative Relaxed Scheduler in Kernel Space

Sreeram Sadasivam
M.Sc Distributed Software Systems
TU Darmstadt, Germany
sreeram.sadasivam@stud.tu-darmstadt.de

Overview

Concurrency bugs which are often resident in multi-threaded programs with shared memory designs are difficulty to find and reproduce. Deterministic multi-threading(DMT) is one such scheme indicated to resolve the above difficulty. But, DMT presents the challenge of having no scheduling constraints. However, currently there are no such techniques that allow to control the schedule of a multi-threaded program on a fine-grained level, i.e, on the level of single memory accesses. A design with a granularity of single memory accesses would help in enforcing the scheduling constraint.

Consider an example, there are two memory accesses MA_1 and MA_2 from two user level threads T_1 and T_2 respectively. The generated constraint for execution is MA_1 after MA_2. Let us assume, T_1 reaches MA_1 before T_2 executes MA_2. The scheduling design has to detect the memory access for MA_2 and MA_1 for enforcing the above constraint. With detection in place, the user level scheduler would enforce a conditional wait or busy wait on T_1 until T_2 completes its execution of MA_2. Moving the scheduling decision to kernel space would remove the above mentioned synchronization required for scheduling. Thus, improving the execution time of the user program.

In the existing design, we have a thread scheduler and a verification engine. The verification engine primarily focusses on instrumenting the user code and realizing memory accesses made by various user threads. The set of safe schedules are provided by the verification engine for the given user program. The generated execution pattern is later realized with the thread scheduler, when the user program is executed. The scheduler thread is realized in user space. It is realized in two design modes - Seperate thread and shared thread. Context switching of the scheduler thread is possible when executed in user space. Considering the example from above, the operating system scheduler might ignore the scheduling constraint set by the user level scheduler. Thus, executing T_1 on MA_1 before T_2 executes the MA_2.

Moving the scheduler task to the kernel space would help to realize the safe scheduling constraints set by the user. The proposed design can be realized by having the thread scheduler as a Loadable Kernel Module.

With the migration of scheduler module to the kernel space, there arises certain design changes and challenges. The execution of user program generates user threads. The kernel space needs to be aware of task IDs to which these threads would be mapped to. The design covers a mapping of these thread IDs to their respective task IDs via proc file system(proc fs). The trace file used as the scheduler input can be passed to the kernel space via a custom file created inside the proc fs. On occurence of an event(in this case a memory access of a global variable), the respective callbacks from the user program would trigger a system call to the kernel module. Such a design would faciliate towards a non-preemptive scheduler. By overcoming the additional synchronization overhead existent in user space design, we encounter the problem of invoking system calls for accessing kernel module. In a monolithic kernel architecture, most of the system calls are blocking synchronous calls to kernel space. Having too many system calls would increase the scheduler overhead on the program execution. One solution is to make system calls when there is an imminent context switch(expected thread switch in the provided safe schedule). The user space threads would assess the safe schedules or traces based on which the system calls for the kernel space scheduler would be made.

Such a design would be benchmarked on various thread conditional scenarios such reader-writer problems, Peterson solution, Lamport solution. These programs enforce the verification of correctness in multi-threaded environment. And also with Indexer and LastZero benchmarking programs. The evaluation is performed on the execution overhead exerted by the transition to the LKM module. The evaluation will also relate to the number of synchronizations taking place when using the system calls. And additional comparison of execution overhead generated by the rival designs PARROT, CORE-DET are also considered for the above mentioned benchmarking programs. The above comparison would also cover evaluations across instrumented and un-instrumented code. The evaluation is scaled from the use of thread count pertaining to the core count, to the use of scaled up version of thread count overshadowing the core count. Thus, creating a possibility of false sharing situations and various other potential execution overhead conditions.