A Linux Kernel Scheduler Extension for Multi-Core Systems

Aleix Roca*, Vicenç Beltran*, Kevin Marquet†
*Barcelona Supercomputing Center
†Univ Lyon, INSA Lyon
E-mail: {arocanon, vbeltran}@bsc.es Kevin.Marquet@insa-lyon.fr

***Abstract*—Current runtime systems take care of getting the most of each system core by distributing work among the multiple CPUs of a machine but they are not aware of when one of their threads (workers) perform blocking calls (*e.g.* I/O operations). When such a blocking call happens, the processing core is stalled, leading to performance loss. In this project, we present two new and independent methods to minimize the effect of I/O operations: The first one is a Linux kernel extension denoted *User-Monitored Threads* (UMT) and the second one is a user-space library named *libsio2aio*. Our Linux kernel extension allows a user-space application to be notified of the blocking and unblocking of its threads, making it possible for a core to execute another worker thread while the other is blocked. The libsio2aio library intercepts the family of read/write system calls, interchanges them by its asynchronous version, and returns control back to the runtime while the I/O operation is being resolved. In both cases we use the Nanos6 runtime to test the new methods.**

Keywords*—Linux Kernel, Process Scheduler, I/O, High-performance computing.

I. Introduction

High performance computing applications usually execute in worker threads that are handled by a userland runtime system, itself executing on top of a general purpose operating system (OS). The main objective of the runtime system is to provide maximum performance by getting the most out of available hardware resources. On a multicore machine, this translates to distributing the work of applications among the machine's available cores and balance each core workload.

Runtime's balancing capabilities are subject to the underlying OS scheduler. When a thread performs a blocking I/O operation against the OS kernel, the core where the thread was running becomes idle until the operation finishes. This problem can lead to huge performance loss as some HPC or high-end server applications perform lots of I/O operations because they heavily deal with file and network requests.

One approach to address this issue is to make the runtime system aware of when blocking and unblocking events happen. In this way, it can chose to execute another worker thread while the first one is blocked. A general approach to detect any blocking operations (such as page faults) requires special kernel support, however, if we narrow the scope of blocking operations to the standard syscalls, a user-space library will suffice. There has been related work on the kernel side [1], [2] but is has been rejected due to its complexity. Instead, both our kernel and user space solutions main advantages are its simplicity.

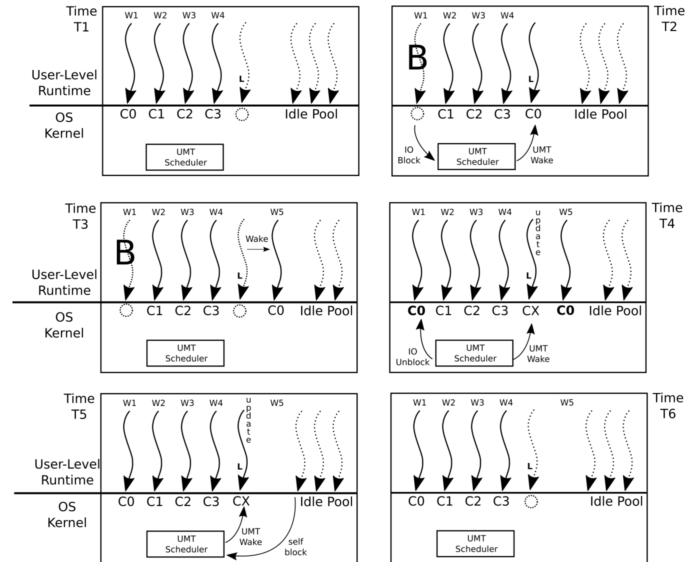

Fig. 1. UMT model overview example

II. UMT Overview

In UMT, the Linux kernel uses a communication channel to notify a user-space application of blocking and unblocking events among their threads. An overview of this functioning is given in Figure 1. The W_i are user-space runtime's worker and L denotes the user-space runtime's *Leader Thread* whose role is to monitor the communication channel. Basically:

- At time T1, four workers W1, W2, W3 and W4 are bound to CPU's C0, C1, C2 and C3 respectively. The Leader Thread is not bound to any CPU and is waiting for UMT events. A pool of idle workers remain blocked until they are needed.
- At time T2, the worker W1 blocks because of an I/O operation and the Leader Thread is notified of the event.
- At time T3, the Leader Thread wakes an idle worker from the pool and waits again for more events. (When W5 wakes, it would also generate an unblock event which is omitted for simplicity). Worker W5 is now running on a CPU; without the proposed mechanism, it would have been idle.
- At time T4, W1 is unblocked after the I/O operation finishes. An unblocking event is generated and the

Leader Thread wakes up. Because there is not any free CPU at the moment, the Leader Thread waits until it momentously preempts another worker. Once it does so, it reads the UMT events and registers that multiple workers (W1 and W5) are running on the same CPU (C0).

- At time T5, after the W5 worker finishes executing tasks, it checks the Leader Thread registers and realizes that there is an oversubscription problem affecting its current CPU. To fix the problem, the worker self surrenders and returns to the pool of idle workers. This generates another event that wakes up the Leader Thread and updates the register of events.
- At time T6, the oversubscription problem has ended and the four workers are running normally.

A. UMT design: Linux kernel-side

Our proposal for the UMT kernel support includes two new system calls to initiate and manage UMT and the infrastructure for the notification channel between kernel- and user-space based on the standard *eventfd*[1] (EFD) file descriptors.

When calling *um_mode_enable* the Linux Kernel initializes an EFD for each CPU on the system and stores them in the context of the calling process. This process' threads start being monitored as soon as each of them allows it by calling the *ctlschedumfd* syscall. The main idea is that each of these EFD keeps a per-CPU count of how many monitored threads are in the ready state. The actual Linux kernel instrumentation of the EFD writing points has been placed into a wrapper around the main context switch entry point called `__schedule()`.

B. UMT design: User-space runtime design

In order to validate the proposal, we have adapted the *Nanos6* runtime[3] of the *OmpSs-2*[4] task-based programming model to work with our kernel extension.

Nanos6 consists of a set of workers threads whose objective is to run tasks and a special management thread called Leader thread. The Leader thread first calls *um_mode_enable* to initialize the UMT kernel structures and then monitors all the per-CPU EFDs using a standard epoll system call. Each worker thread first calls *ctlschedumfd* once to enable monitoring and then start executing tasks. When one of the monitored threads produces an event (it block or unlocks), the Leader Thread wakes up from the epoll sleep and reads the EFD. If the count of ready threads on the CPU that has triggered the event is zero and there are still tasks to execute, the Leader Thread retrieves an idle thread from a pool and gives it a task to execute on the idle CPU.

If the previously blocked worker wakes up while the new worker is running, both threads will have to compete for the CPU. However, this oversubscription problem only prevails for a limited amount of time. Workers have been provided with a oversubscription protection mechanism that consists on checking the counter of ready threads of its CPU after finising executing a task. If the count is greater than one, workers self surrender to allow other workers to run freely on the CPU.

[1]An eventfd is a simplified pipe that was designed as a lightweight inter-process synchronization mechanism. Internally, an eventfd holds a 64 bit counter that can be written to increment its internal value or read to clear and return it.

III. LIBSIO2AIO OVERVIEW

The libsio2aio user-space library defines wrappers for the `pread()`, `pwrite()`, `preadv()` and `pwritev()` syscalls (all Linux Kernel native AIO supported syscalls) which call the asynchronous version of the intercepted syscall. After submitting the request, it checks whether it has immediately completed or not. If it is the case, the wrapper returns immediately as well. Otherwise, it pauses the execution of the current tasks (not the thread) and transfers control to the runtime. The runtime is then able to execute other tasks in the current CPU while the I/O operation is being resolved. The runtime periodically checks whether any AIO request has completed and if it is the case, the task that submitted the AIO request is unblocked. Unblocked tasks execution are later resumed by Workers.

IV. EXPERIMENTATION

We have tested UMT using a synthetic benchmark. The benchmark simply maps a region of memory using *mmap* and creates a set of independent tasks whose purpose is to write and sync random mapped data. As a result we have achieved a speedup of x10. We are currently testing libsio2aio and we have not yet been able to find an appropiate benchmark that benefits from its advantages.

V. CONCLUSION

Finally, we conclude that both UMT and libsio2aio have two main effects: on the one hand, they provide a mechanism to queue more I/O operations which approaches the real I/O rate to the one specified by the manufacturer of the storage device. On the other hand, blocked processes no longer obstruct the core and useful computations can be done while I/O petitions are being served. In the case of UMT, the oversubscription problem limits performance but as results show, it is not always a problem. Future work will focus on finding more I/O intensive applications to test both presented approaches.

REFERENCES

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 53–79, 1992.

[2] V. Danjean, R. Namyst, and R. D. Russell, "Linux kernel activations to support multithreading," in *In Proc. 18th IASTED International Conference on Applied Informatics (AI 2000*. Citeseer, 2000.

[3] BSC, "Nanos6 runtime," https://github.com/bsc-pm/nanos6, 2018.

[4] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé, "Improving the integration of task nesting and dependencies in openmp," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 809–818.

Aleix Roca is a Linux Kernel passionate. He admires the leading Linux developers technical skills and their tremendous effort to manage the world's biggest open source community. Aleix studied computer engineering at UPC-FIB. After obtaining his degree he enrolled the Master in Innovations and Research in Informatics specialized on High Performance Computing at UPC-FIB, where he obtained the *Severo-Ochoa MSc scholarship*. During his master studies he joined the *Barcelona Supercomputing Center* where he developed his final master thesis on the Linux kernel and programming models. Currently he has started a PhD at BSC where he continues his research on the Linux Kernel and HPC.