Improving thread scheduling by thread grouping in heavily loaded many-core processor systems

Luka Milić* and Leonardo Jelenković**
* University of Zagreb, Faculty of Organization and Informatics, Varaždin, Croatia
**University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia
luka.milic@foi.hr, leonardo.jelenkovic@fer.hr

Abstract – Basic scheduling techniques currently employed in operating systems usually don't account for hierarchy of processors cores and caches resulting in suboptimal system efficiency, especially on heavy loaded systems. In this paper we explore possible improvements in scheduling for such heavy loaded many-core systems. Basic idea is to group threads of the same process as close as possible, preferably even on the same processor. We expect that doing so may improve efficiency of processor's cache usage, resulting in better overall performance. The idea is tested by adapting the Linux's simplest scheduler (in code) – a round-robin scheduler. Achieved results are presented in this paper.

I. INTRODUCTION

Today's operating systems schedulers mostly do not differentiate multiprocessor system where every processor is simple (one core) from many core system where many cores are coupled on single processor sharing its caches and connections to system busses. Virtual cores (hyper-threading) are also usually treated the same (as regular processor). That is not really a problem until there are not many cores/processors in the system. In the future that is very likely going to change as most computers working non-trivial jobs will have many cores [1].

Architectures of multi and many core systems are various. Most personal computer systems (and some smart phones and tablets) have a single multi core processor. Workstations and servers that require more computing power may have several multi core processors. In the future multi core processors (up to 8 cores) will probably be replaced by many core processors (from 8 cores and up). Computing power of many core processor systems is not relaying only on raw processor's core capabilities and number of cores but also on data acquisition. Cores in many core processor are usually arranged in local groups that share local cache memory. Although any core can fetch data from anywhere (local cache, non-local cache, main memory), delay such fetch operation request will be lesser if data is closer.

Thread scheduling algorithm should take thread's properties and inter-thread interactions into account for achieving more efficient use of processor's cache and thus improve overall system performance [2][3]. Usually, threads of the same process operate on shared data and have high level of interactions. We will call such threads as connected threads (co-threads). Although threads from different processes can also be connected (i.e. use same shared resources), that is rarely, more exception than a rule.

Scheduling algorithm should be aware of connected threads as well as processor's core architecture where those threads are to be placed.

In systems in which the average number of ready threads is not much greater than the number of processor cores, scheduling procedures should keep connected threads as close as possible, on cores that share same caches (L2/L3).

In heavier loaded systems (at least twice as much threads than cores) different way of scheduling may be appropriate. Considering that it is more difficult to achieve a synchronous work of interconnected threads on multiple cores, which may be the optimal scheduling for those jobs (and maybe generally), the alternative could be to collect such threads on the same core. The advantage of this approach is in efficient use of the cache of such cores because it is expected that such interconnected threads will use the shared data probably already present in cache. In this way we will lose potential for parallel execution, but on heavy loaded system probability for such parallelism might be very small, and efficient use of cache could prove more beneficial.

Since schedulers don't differentiate simple single core processor from one core of multi/many processor, when describing existing schedulers we use term processor as synonym for both. In same context, term multiprocessor is used for describing systems that have more processor cores, standalone ones or in multi core package.

For testing such hypothesis a rather simple scheduler `SCHED_RR` from the Linux operating system is used and adapted. Primary reason for choosing `SCHED_RR` even if its real-time scheduler over `SCHED_OTHER` default scheduler for general thread is in source code simplicity. `SCHED_OTHER` that implements Completely Fair Scheduling algorithm is far more complex and hard to adapt. If proposed adaptation don't give positive results for `SCHED_RR` (using threads with same priority, which de facto turns off any real-time properties of scheduler) there is no reason for going into adapting `SCHED_OTHER`.

In section II we first describe scheduler `SCHED_RR`. Section III describes our modification of `SCHED_RR` scheduler, and section IV presents simulation environment and achieved results.

II. LINUX SCHED_RR SCHEDULER

Linux, like most operating systems use different scheduling policies for different types of threads, particularly differentiate normal from real time threads.

A. Linux scheduling policies

Currently, for normal threads Linux supports scheduling policies `SCHED_OTHER`, `SCHED_BATCH` and `SCHED_IDLE`. Those scheduling policies are based on a completely fair distribution of the virtual processor time. This virtual time is a rather complex function of the fair thread priority, real time and the processor consumption by fair threads [4].

Real time scheduling policies `SCHED_FIFO` and `SCHED_RR` are priority based schedulers. Thread with highest priority is always scheduled first. Difference between `SCHED_FIFO` and `SCHED_RR` policies are in scheduling of more than one highest priority threads.

For example if we have ready threads A and B with same highest priority, `SCHED_FIFO` will pick one that become ready first and execute it until it finished or block itself voluntarily. `SCHED_RR` will alternate threads A and B on processor, giving each a time interval before switching to next. On multiprocessor system (or multi/many core) both threads will be running in parallel, each thread on its processor.

On multiprocessors, real time thread scheduling is based on a system-wide strict priority scheduling. Since we use this mechanism to adapt scheduling, system-wide strict priority scheduling is detailed in next subsection.

B. Push-pull algorithm

Strict priority scheduling in a multiprocessor system with *N* processors must ensure that *N* highest priority threads are always chosen as active on those processors (one thread per processor) [5].

The priorities of real time threads are in range from 1 to 99, where higher number represent higher priority. For each processor there is separate data structure containing 99 thread queues, one thread queue per priority. Every ready real time thread is placed into one thread queue, one that matches thread's priority. Active thread for each processor is chosen from its ready queues (thread from highest non-empty queue), and then removed from that queue and marked as active.

System-wide strict real-time priority scheduling is implemented by using *push-pull algorithm* on overloaded thread queues. A real time thread queue is considered to be *overloaded* if it holds at least one another thread (besides currently active) which could be migrated to another processor according to that thread's processor affinity mask.

Push part of the algorithm is performed on processor which uses overloaded thread queues. Its purpose is to push overloaded threads to other processors that will immediately execute them. Such processors must be acceptable by such threads (they must be in thread affinity mask) and currently running lesser priority threads.

More precisely, push algorithm is activated on particular processors only after events:

1. active real time thread is changed (replaced with higher priority thread or its priority is lowered);
2. a new real time thread is created on current processor;
3. a real time thread is waked on current processor.

The push algorithm splits into three algorithms: *pushing*, *finding* and *searching*.

The *pushing* algorithm is activated when a queue on a given (current) processor is overloaded. Highest-priority thread from overloaded queue is then passed to *finding* algorithm which might find a processor where migrate such thread.

The *finding* algorithm use *search* algorithm to find a processor on which the thread will be migrated. Algorithm is repeat up to three times because of scheduling data structures locking semantics [6].

The *searching* algorithm looks processors that are in observed thread's processor affinity map. From that set of processors algorithm search for a lowest priority subset of processors, i.e. processors that currently run threads with currently lowest priority in system. If there are more processors in this subset, choose processor on which thread was run last time. If there is not such processor in subset, then choose one of the closest to processor on which thread was run last time (by scheduling domains).

Similarly to push algorithm, *pull* algorithm is activated when a thread on single processor is finished or blocked or its priority is lowered. Pull algorithm looks at ready queues of other processors (overloaded queues) and if it finds a real time thread with higher priority than the highest priority thread in his queues, it pulls such thread to this processor where such thread will immediately become active. Only threads that can be migrated to current processor are observed. If there are more such threads, only one with highest priority is chosen and pulled.

III. ADAPTING SCHED_RR

The goal of adaptations was to try to group threads of the same process (here called co-threads). The solution was implemented by four ideas, i.e. changes, for push-pull algorithm.

Push algorithm, i.e. its searching algorithm is modified as follows.

1. First try to find a processor on which are threads that belong to the same process (co-threads) but only if there are none on current processor. If such processor is found push thread to its thread queue.
2. If processor where thread was last executed is in thread's available processor subset push thread there.
3. Search for processor closest (by scheduling domains) to processor with co-threads (processor

that is not thread's available processor subset). If one is found push thread there.

4. Search for processor closest (by scheduling domains) to processor where thread was last executed.
5. Return any processor from thread's available processor subset.

In the pull algorithm, pull is additionally performed in situations when remote thread has the same priority as top priority local one if this remote thread has co-threads here and not on remote processor.

The scheduling of real-time threads is not actually changed with this adaptations – system-wide strict real-time priority scheduling is preserved.

Adaptations in Linux source code was done by adding a new scheduling policy `SCHED_RR2` based on `SCHED_RR`. Since changes to `SCHED_RR` are minimal, adaptations (additions) are preformed in same code.

Files included in adaptations (modified files) are:

- `include/linux/sched.h`
- `kernel/sched/sched.h`
- `kernel/sched/core.c`
- `kernel/sched/cpupri.c`
- `kernel/sched/rt.c`.

In kernel code, modifications regarding `SCHED_RR2` scheduling (in respect to vanilla `SCHED_RR`) are protected with a macro. To use `SCHED_RR2` macro must be set. Otherwise, when macro is not set vanilla `SCHED_RR` is used (all modifications for `SCHED_RR2` are not compiled). Experiments were performed on both kernel: one compiled with defined macro (using proposed modifications, i.e. `SCHED_RR2` scheduler) and one without macro (vanilla `SCHED_RR` scheduler).

IV. Experimental results

The processor on which experiments are performed has Intel's processor of *Sandy Bridge* architecture, namely the model *i7-2670QM*. This processor has four hyper-threaded cores (8 logical cores) from which every physical core has 32 kB L1 cache and 256 kB L2 cache. All cores share 6144 kB L3 cache. The processor itself runs on a frequency of 2.2 GHz. The kernel version used in experiments was Linux 3.3.0-rc7 [7].

Test program constructed for evaluation of adapted scheduler try to mimic general multithreaded program whose threads usually operate on shared data. Program first spawns given number of processes. Each spawned process further creates given number of threads producing large number of threads that simulate heavy loaded system. Since our test system has only four physical cores, size of shared data is kept small to amplify cache utilization in achieved improvements.

Test program was executed on unmodified version of kernel using `SCHED_RR` policy, and on modified kernel using adapted `SCHED_RR2` policy. Each run was timed to 10 seconds when progress was printed (number of predefined operations on shared data). Tests are performed many times and average results are presented in Table I.

TABLE I. Experimental results

Test No.	*Processes per program*	*Threads per process*	*Shared data size*	*Efficiency Improvement*
1	7	1	small	0.37%
2	11	5	small	16.02%
3	9	3	small	14.29%
4	9	3	medium	12.15%
5	9	3	larger	9.93%

Results are surprisingly good. Of course, this is synthetic test and maybe not a real measure for real application, but improvements of 10% and more are encouraging. As expected, when there are small number of threads (test no. 1) our adaptation didn't change performance (0.37% may be counted as statistical error). But when the number of threads is significantly greater than number of processors (all other tests) expected improvements did happen and surpass our expectations. Those improvements are greater with more threads and smaller shared data. As shared data grows, less cache utilization is achieved, lesser improvement is obtained.

Once again we must say that used scheduler is adaptation of real time scheduler and we compare obtained results by original real time scheduler. Real time scheduler is not concerned with fairness and maximal resource utilization. Therefore, we expect that if same ideas of grouping were implemented in scheduler for normal threads (policy `SCHED_OTHER`), improvements, if any, will be much lower since that scheduler uses more sophisticated load balancing algorithms. However, on heavily loaded many core systems we still expect that improvements in efficiency with proposed thread grouping should grow as number of cores grows.

V. Conlusion

This paper presents an idea to improve scheduler for many core systems when such systems are heavy loaded with lots threads. Idea is to group threads of same processes on same processor and improve efficiency of processor's local caches. In simulated environment adaptation to simple round robin scheduler (using `SCHED_RR` on Linux) show significant improvements up to 15%, which encourage further research in this direction. Further research should be oriented to adapting more complex schedulers, e.g. ones that are used for scheduling of normal threads. Furthermore, test program could also be improved to better mimic variety of real multithreaded applications.

REFERENCES

[1] T. Zangerl, “Operating system scheduling on multi-core architectures,” University of Innsbruck, Seminary thesis, June 2008.

[2] T. A. Anderson, M. Rajagoplan, B. T. Lewis, “Thread scheduling for multi-core platforms”, Technical report, Programming Systems Lab, Intel Corporation, 2007, http://static.usenix.org/event/hotos07/tech/full_papers/rajagopalan/rajagopalan.pdf.

[3] A.Fedorova, M. Seltzer and M. D. Smith, “Cache-fair thread scheduling for multicore processors”, Technical report, Harvard University, 2006, http://www.eecs.harvard.edu/~fedorova/papers/cache-fair.pdf.

[4] C. S. Pabla, “Completely Fair Scheduler”, Linux Journal, vol. 184, August 2009.

[5] A. Garg, “Real-time linux kernel scheduler,” Linux Journal, vol. 184, August 2009.

[6] R. Love, “Kernel locking techniques”, Linux Journal, 2002.

[7] Intel Corporation, “Intel Core i7-2670QM Processor”, Technical documentation, 2011.