Toward Accurate Group Scheduling in Multi-core Environments

Kenta Ishiguro
Hosei University

Kenichi Yasukata
IIJ Research Laboratory

Toshio Hirotsu
Hosei University

The bandwidth control of a CPU scheduler is essential for the multi-tenancy of public clouds because resource isolation between virtual machines (VMs) relies on it. The Completely Fair Scheduler (CFS) used in Linux, one of the most widely deployed CPU scheduler implementations, achieves bandwidth control with its *group scheduling* extension. However, bandwidth control with group scheduling is challenging because CFS is responsible for accounting and managing multiple CPU cores in a work-conserving manner.

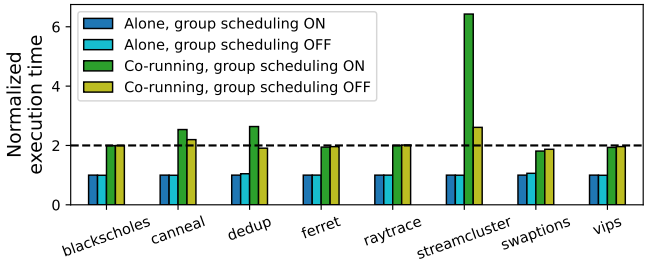
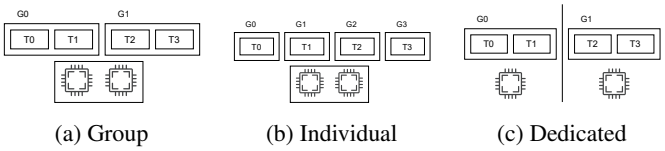


Figure 1: Normalized execution time of workloads run w/ and w/o group scheduling in `alone` and `co-running` with `swaptions`.

We observe that group scheduling of CFS sometimes degrades the application performance when multiple applications are consolidated. Fig. 1 shows the normalized execution time of PARSEC benchmarks run on an 8-vCPU VM hosted by an 8-pCPU physical machine (we call this VM *the primary VM*)[1]; in the cases labeled with `co-running`, we run, on the same physical machine, another 8-vCPU VM running `swaptions` that is one of the PARSEC workloads and engaged in a CPU-intensive task. In contrast, the `alone` case does not run this co-runner VM. In short, in the `co-running` case, the primary VM will have half CPU time; thus, the execution time of benchmark is *expected* to be 2× longer than the `alone` case. However, we observe that the primary VM slows down by 6× in `streamcluster`. Interestingly, without group scheduling, the slowdown of `streamcluster` goes down to 2.8×; this indicates that group scheduling incurs the root cause of this issue.

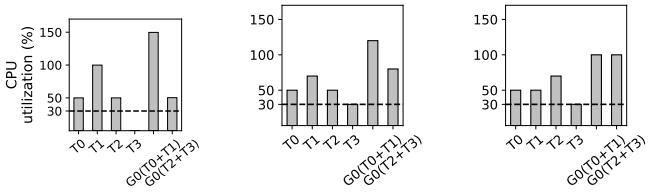
We conduct a detailed investigation using a synthetic workload. We reveal that the slowdown derives from an issue that significantly degrades CPU utilization of non-CPU-intensive workloads. Our synthetic workload runs two processes, `P0` and `P1`, on two CPU cores; `P0` runs two (POSIX) threads, `T0` and `T1`. `P1` also executes two threads, `T2` and `T3`. `T0`, `T1`, and `T2` execute a busy loop so that each consumes 100% CPU cycles

[1]Experiments in this paper are conduced on a machine containing a 2.80 GHz 8-pCPU Intel Xeon E-2378G processor with 64 GB of RAM, and we use Linux/KVM v5.15.64.



(a) Group (b) Individual (c) Dedicated

Figure 2: Group scheduling configurations



(a) Group (b) Individual (c) Dedicated

Figure 3: CPU utilization of the synthetic workload

at maximum, and `T3` periodically sleeps and utilizes 30% CPU cycles at maximum. In `group` (Fig. 2a) that corresponds to the setup of the PARSEC experiments, we put the threads (`T0` and `T1`) of `P0` into a group named `G0` and the threads (`T2` and `T3`) of `P1` in another group `G1`. We configure that `G0` and `G1` will have the same ratio of CPU time[2]. Surprisingly, the CPU utilization of `T3` is extremely low (0.3%), while the utilization is expected to be 30%. The CPU time supposed to be assigned for `T3` is consumed by `T1`, belonging to a different group from `T3`. Consequently, the total utilization of `G0` (`T0+T1`) is 149.7% and `G1` (`T2+T3`) is 50.3% (Fig. 3a) that are largely different from the intended ratio of 100% vs. 100%. This CPU time reduction imposes negative impacts on multi-threaded applications by causing, for instance, the straggler problem, a popular problem in distributed computing where the slowest worker delays the job completion. In another possible configuration, `individual`, all threads belong to different groups (Fig. 2b). While it is better than `group`, `individual` also cannot achieve the ideal ratio of 100% vs. 100%, and `G0` (`T0+T1`) consumes 120% and `G1` (`T2+T3`) utilizes 80% of CPU cycles (Fig. 3b). Another possibility is the `dedicated` configuration, where each group is associated with a dedicated CPU core (Fig. 2c). This configuration achieves the ideal bandwidth control, as shown in Fig 3c. However, the load balancing by CFS is theoretically disabled because each core is statically assigned to each group.

The experiments above demonstrate that CFS cannot achieve ideal group scheduling regardless of the best configuration efforts. In future work, we will explore a group scheduling mechanism that resolves all the issues simultaneously.

[2]using the `cpu.weight` option of cgroup.