DYON: Managing a New Scheduling Class to Improve System Performance in Multicore Systems*

Ramon Nou[1], Jacobo Giralt[1], and Toni Cortes[1,2]

[1] Barcelona Supercomputing Center (BSC), Spain, <name>.<surname> at bsc.es
[2] Technical University of Catalonia (UPC), Spain

Abstract. Due to the increase in the number of available cores in current systems, plenty of system software starts to use some of these cores to perform tasks that will help optimize the application behaviour. Unfortunately, current Onload mechanisms are too limited. On the one hand, there is no dynamic way to decide the number of cores that is taken from applications and given to these system helpers. And, on the other hand, the onload mechanisms do not offer enough control over when and where onloading tasks should to be executed. In this paper we propose a new Onload Framework that addresses these issues.
First, we propose DYON, a dynamic and adaptive method to control the amount of extra CPUs offered to the Onload Framework to generate benefits for the whole system.
And second, we propose a submission mechanism that given a task, executes it if there are idle resources or rejects it otherwise. This feature is useful to move the execution of small pieces of code out of the critical path (allowing parallel execution) when this is possible, or discard them and execute a code that will not rely on them.

1 Introduction

Currently, processors have an increasing number of cores used for CPU intensive workloads, but these cores are not exploited when they are idle. This extra capacity can be used, for instance, to reduce a kernel critical path by executing code in parallel, or to run additional services that provide additional benefits. Consider an I/O analyser as IOAnalyzer [12] (to improve the I/O scheduler) as an example of service (we will henceforth call them *Onloaded Services* in general). This service can reduce the I/O time of a job with some I/O phases up to 5 times [12] if the disk scheduler becomes optimal. This benefit affects the rest of the system, which can increase its performance or go to an IDLE state faster.

* This work was partially supported by the EU IST program as part of the IOLanes project (contract FP7-248615), by the Marie Curie Initial Training Network grant"SCALing by means of Ubiquitous Storage (SCALUS)", by the Spanish Ministry of Economy and Competitiveness under the TIN2012-34557 grant, and by the Catalan Government under the 2009-SGR-980 grant. We would also like to thank Neurocom for letting us use their TariffAdvisor application and Alberto Miranda for his help. Source available for kernel 2.6.32 under request at the IOLanes project [5]

If the code is not executed (idle resources are unavailable) the system will continue working, but could we get extra benefits with additional resources to those services? Yes, however the additional resources should be controlled dynamically. For example, an HPC job with a processing stage and an I/O stage will rarely obtain benefits if we change the I/O Scheduler. We can apply a dynamic CPU partition method modifying the two stages behaviour forcing non-idle resources to be offered only in the I/O stage. The proposed mechanism is called **DYON**.

An example of dynamic partition is the memory utilization between the page cache [4,9] and normal memory. The key is that the page cache reduces a possible access to disk and thus, having more page cache should be beneficial in most cases. Additionally, the kernel knows when the page cache size should be modified via *malloc* calls. The CPU has a different behaviour: we can not assign percentages of CPU directly (kernel divides cores using masks), and we neither know in advance CPU requirements nor the maximum amount that we have to offer to produce benefits to the system.

Onloaded Services need additional ways to be executed not found in current Linux systems: for example, parallelized critical path code needs to know immediately if it is going to be executed (using only idle resources), or processes doing I/O analysis should be executed using a low priority and stopped immediately if the resources are going to be used. In this paper we propose the **Onload Framework** that allows to execute those tasks and services only if there are idle resources. With our proposal, we immediately know if the onload code is being executed so we can then continue with a parallel workflow or move to a normal workflow if the execution is not possible. The Onloaded Framework also allows us to control the tasks priorities providing the needed tools to manage DYON.

Summarizing, the contributions of this article are the next ones:

1. An **Onload Framework** enhancing Linux *workqueues* with new capabilities, along with a new scheduling class (SCHED_ONLOAD).
2. **DYON**, a dynamic CPU distribution method that maximizes the system performance (CPU Cycles per I/O) modifying previous services priority.

2 Existing mechanisms overview

One of the current ways to execute code is the *workqueues* [10] mechanism, used by some components in the Linux kernel to defer tasks. The *workqueues* are implemented with a shared queue, being rarely used due the variability in workloads. Recently they has been extended with the inclusion of Concurrency Managed Work Queues [6], moving to a single pool of threads that serves all queues. A single thread pool implementation avoids contention among threads from each pool, but traditionally it has been difficult to find a solution for all components. For example, some modules have built mechanisms on top of work queues to handle ordering restrictions [8] and others simply use their own *workqueues* since it is the easiest way to defer work inside the kernel.

The execution under the *workqueues* mechanism does not allow a conditional execution of a parallel or sequential workflow depending on whether the task will

start executing or not (taking into account that these onloaded tasks will only use idle resources).

Another mechanism is the SCHED_IDLE scheduling class, the processes of such class are also enqueued and we do not know when they will be executed. Those mechanism defer tasks, but deferring is in some scenarios undesired as the execution can delay indefinitely not allowing a conditional execution of the code.

3 Onload Framework mechanism

Our framework complements the *workqueues*, with a potential to parallelize tasks and workflows and with a narrower target.

3.1 Design

The framework is designed for short kernel tasks and short/long off-kernel/user tasks. The former, requested by the kernel, lack the issues (deadlocks, delays [7]) that are problematic for the *workqueues* management. Off-kernel tasks, requested by a user level application, have similar requirements but they are preemptable because they can not be trusted (as may block the framework). The common and default behaviour of the framework is that it cannot preempt, so either it accepts a task to be run immediately or it rejects it. Conceptually this means either taking advantage of additional resources if they are present or doing it as it was planned if they are not. We are going to enumerate the different features and design principles:

Notifications - Tasks are executed asynchronously from the requester, so there must be a callback mechanism to notify the end of a task. This callback can be as simple as unlocking a semaphore. It is an optional feature, the tasks can implement a different mechanism in their code.

Parallelism - Some of the operations we plan to onload might take advantage of parallelism, and this means our framework needs to provide a way to express it. For that matter, a range of CPUs [3] can be requested at once. Using a range increases the possibility of success of the request, which is obviously critical since otherwise the user would have to poll downwards from its maximum degree of parallelism until the system can satisfy its requirements. At any moment, tasks can retrieve the value of a private sequential id for this parallel operation and the total number of threads collaborating.

Offloading - The framework also works in specialized resources (GPGPUs, SPUs, etc.) seen by the framework user as an extended CPU mask with both CPUs and these specialized cores. The user changes the target processor to execute the requested task only by modifying this mask.

Deadlines - A task can be queued for a short time (*deadline*) while there are not idle resources. The task will not use more than those idle resources on

[3] We use CPU as simplification, referring to the smallest processing unit that can run a task.

the machine, which will not happen using *workqueues*. This was implemented for tasks like deduplication: setting up a deadline can be seen as a way to express that the block to be deduplicated is not needed after the deadline.

NUMA selection - If we specify a memory address, the mechanism will try to use an idle core near the memory address automatically.

Interface - The interface to onload tasks is similar to that of *workqueues* but semantically richer and designed for parallel operations. Work requests are defined with the following list of parameters: *Function to onload, argument to pass to the function, callback, CPU mask, minimum and maximum number of resources requested, buffer address and a deadline value.*

3.2 Implementation

The Onload Framework is implemented as a pool of threads in the kernel, each bound to a core, from which only a subset will be available at a given moment depending on the system load. If one thread is ready, it means its associated CPU is idle so a task can be onloaded.

Internally, threads can be on two main states: ready and running. Availability of a thread/CPU is expressed through a bitmap that is updated atomically on every context switch. Checking whether there are idle resources to onload a task is a fast operation. Apart from request time, a task (using the deadline feature) can also be scheduled on two other points. Each time a CPU is about to idle, the queue is checked, then one of the pending tasks could be selected. Similarly, after an onloading task is finished, if its associated CPU run queue is still empty, another of the pending tasks can be dequeued to be run immediately.

Whenever a new work request arrives at the framework, first its number of CPU threads requirement (for example, use only core #4) is checked. If there are idle cores to onload, the framework moves their associated threads to the running list and then they wake up with a description of the task to perform. This is a pointer to a function with three parameters: a void pointer and an id plus the number of threads allocated for the task. While this task is running, others can be accepted as long as there are resources to fulfill them all. Scheduling decisions are simple at this point, but could be more elaborated if the optional deadline functionality becomes more important in the future.

Once a task is finished, its associated callback is invoked and then the current CPU load is checked. If its runqueue is still empty, the framework inspects if there is another onload task waiting (to maximize CPU utilization). If it is not, the thread is put back to the pool and the CPU marked as unavailable.

The scheduling is handled through a new scheduling policy which is set with the POSIX function *sched_setscheduler()*. Current Linux versions implement five different policies: FIFO, round robin, other (standard), batch and idle. Our framework adds an extra one, SCHED_ONLOAD, whose effect is those processes will only be scheduled whenever there are idle cores (similar to SCHED_IDLE but with lower priority and different preemption scheme). For our IOAnalyzer example means that the processing and analysis threads are of the SCHED_ONLOAD class so their execution is subject to the existence of idle cores.

4 DYON - Dynamic Mechanism

As we have explained in Section 1, for certain types of services it is appropriate to offer additional resources to increase the system performance. For example, if we offer some extra CPU to IOAnalyzer we can get an extra gain surpassing the loss of performance. With the default policy, we will not get this extra gain resulting in a slower working system. Summarizing, **DYON** offers more or less performance to some threads that are aiding the system to improve the overall system performance.

4.1 Selection Mechanism

This dynamic system does not have a monotonic relationship between the CPU partition (Onloaded Services and the rest of the system) and the performance obtained. Therefore, we do not know if increasing or decreasing the processing power offered is going to affect the performance. For this reason, we need to test all the possibilities.

The method used to select the most beneficial partition is iterative and guided by past performance. This method is based on the Armed Bandit [3], which chooses the next partition based on the near-past observed performance. As it is probability based, sometimes non-favourable partitions will be used. Those non-favourable partitions may become favourable at any time, recalibrating the system.

To summarize, the global workflow is the next one: each t seconds we obtain $P(\text{performance})$ for the current partition n_i. The new performance is merged with previous performance values for the same partition. Once we have the system values stored, we update the different probabilities (gradually) to favor the best partition and select, using the afore-mentioned probabilities, the new partition that will be used. We do not do any filtering as noise is considered to be inside the system to optimize.

To get a finite number of partitions, we divide the CPUs in a number, n, of parts. For example, 5 parts will divide the system processing power in : 0%, 25%, 50%, 75% and 100%. A 25% means that a 25% of the CPUs will be dedicated to Onloaded Services. Furthermore, 0% will select the default SCHED_ONLOAD policy, meaning that only idle resources will be used.

4.2 Metrics chosen to improve

To guide our dynamic system, we are going to use an I/O oriented metric extended to cover CPU Only scenarios. The metric, CPIO [1] (Cycles per I/O, lower is better), is defined as the next formula:

$$\frac{(user\ cycles + system\ cycles) * CPUFreq * \#Cores}{system\ I/O\ blocks\ read + written}$$

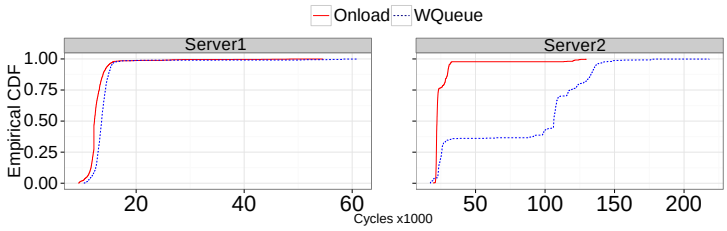


Fig. 1. ECDF for CPU Cycles to create-execute-finish a task on different servers.

We change it with $(user\ cycles + system\ cycles)^{-1}$ (lower is better) to cover applications without I/O. Thanks to CPIO we can compare two system snapshots and decide if an action produces a better working global system without the need of application metrics that are not always available.

5 Evaluation

We will evaluate the two components presented, i) the Onload Framework with the Workqueues in a common scenario (creation-execution-finish of a task) and ii) DYON using IOAnalyzer as Onload Service.

5.1 Onload Framework - Task Creation-Execution-Finish comparison

If we exclude checking the CPU availability, onloading a task could be similar to queueing the task to be run by the global workqueue. The workqueue needs to be empty and restricted to schedule on a different CPU than the requester. We use 2 servers: Server 1 has 4 cores and Server 2 has 8 cores. Figure 1 shows the range of # of CPU cycles to create, execute and finish a task (1000 runs). Our proposal is more stable(less variation) when we have a longer number of cores as the restriction of running the task on a different core results on high variability for the workqueue mechanism, evidencing it is not the appropriate way to onload a task in the Linux kernel given a processor restriction. This comes from the fact that they do not take processor availability into account. Other features of the Onload Framework are not directly comparable.

5.2 DYON

The system under test is an Intel Core 2 Quad CPU Q9300 @ 2.50GHz with 4Gb of RAM with a ST31500341AS HDD (Server 1 in previous experiment). The onloaded service selected for the evaluation is an I/O workflow analyzer to select the I/O scheduler (IOAnalyzer [12]) for the current workload automatically. As the service is highly optimized, we modified it in order to increase its CPU

consumption. We will use the following applications and benchmarks to build the evaluation scenario creating a 3-phases workload, composed as explained in Table 1, generating an heterogeneous workload. Each application has its own performance metric that will be used to evaluate the system.

TariffAdvisor (TA) is a real parallel application [11], it uses CPU to make fast rating for Telecom operators and does intensive writes to store the results. The metric studied is the *reports per second* obtained (higher better).

Flexible I/O (FIO) [2] is a benchmark doing sequential I/O to 10 files in parallel. The performance metric is the *runtime* (lower better).

CPU is a basic CPU consumer, it tries to use a 100% of each CPU for the time specified. The metric to improve is the number of operations (higher better). Offering CPU power to the Onload Framework, reduces its performance.

Table 1. Workload phases description.

Phases composition	Description
TA + CPU	An equilibrated value is the best as we are running one CPU intensive parallel process and one I/O intensive process.
TA + FIO + CPU	We have 2 I/O processes and 1 CPU consumer. The best global performance will be obtained improving the I/O.
CPUc	In this phase CPU is a direct target to optimize. Hence, the best option is to reduce the CPU devoted to onload services to 0% to lower the CPIO metric.

We will evaluate three scenarios using the IOAnalyzer service to optimize the I/O Scheduler of the TA and FIO (able to offer a 50% of improvement for each application): i) The static partition of CPUs, ii) The dynamic partition using CPIO metric or application metrics and iii) the selection mechanism reactivity.

i. Static partition. For this experiment, we partition the CPU from 0% to 100% in 10% steps, so 90% means that 90% of extra non-idle resources are available to IOAnalyzer. For each of the partitions we are running the 3-phases workload defined in Table 1 obtaining a normalized mean performance value for the entire workload.

On Figure 2 (left side) we show each static partition performance as a boxplot. We can observe how with a 50% static partition the median performance (box horizontal line) is higher than any other partition. On the other side, a 80% static partition is a 10% less efficient than the optimal. The default partition (0%) which only offers idle resources to IOAnalyzer, also obtains less performance than the optimal.

If we analyse each workload phase, the optimal partition changes (50% for TA+CPU, 70% for TA+FIO+CPU and 0% for CPU) as the effect of IOAnalyzer over the applications depends on the level of I/O (1, 2 or 0 applications

generating I/O respectively). This produces an heterogeneous workload, being a perfect target for a dynamic method as **DYON**.

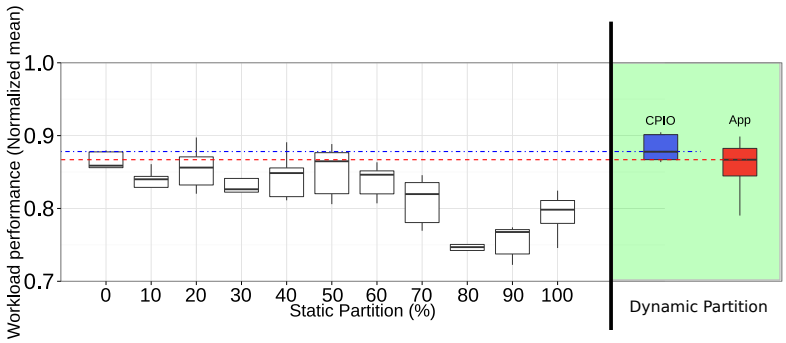


Fig. 2. Static partition (left side) showing the mean application performance on Y-Axis of the three phases described on Table 1. Results with DYON using CPIO or application metrics are on the right side.

ii. Dynamic partition. Using the previous Fig.2 (right side), we show the performance obtained running DYON guided by CPIO metric (CPIO, blue) and Application metrics (App, red).

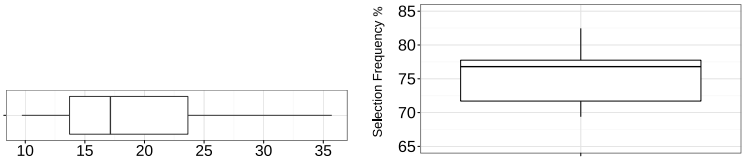
With any of the two metrics, we are able to obtain more performance than the optimal static partition (50%) as we are on an heterogeneous workload. In the case of CPIO we obtain clearly a 1.5% more, but the benefits with Application metrics are lower.

This may be surprising at first: On the one hand, CPIO optimizes the global system behaviour but it does not know anything about the jobs performance. On the other hand, application metrics potentially can offer better results as the output is more direct if they are of enough quality. Precisely, quality is the problem on our scenario: for example, TA application metrics only report how many records per second we have in the last period: It does not include any relation with the CPU used for that processing, i.e., effectiveness reducing the information offered to **DYON** in comparison with CPIO.

If we analyse each of the phases separately (not shown), we obtain more performance with **DYON** than any static partition in the (TA+FIO+CPU) phase because it is heterogeneous. However, CPUc phase obtains 10% less performance than the maximum attainable: As this workload does not benefit from IOAnalyzer, any resource taken from it will decrease its performance. As **DYON** needs to try different partitions to find the best one, these tries give resources to IOAnalyzer, but result in no benefit to the CPU application.

iii. Selection mechanism and system reactivity. There are two considerations in order to have a working selection mechanism as the one we are proposing:

Number of partitions. Having a lot of partitions reduces the capacity of the algorithm to react. For example, with 10 partitions and only 1 of them optimal (not a very common scenario) we need in median 17 tries or iterations to find and select the optimal value in a stable way (two consecutive tries) as we can see on Fig. 3a. On the same scenario, we have more than a 75% of the selections going to the optimal partition (Fig. 3b). The other 25% will be devoted to near optimal partitions and occasionally testing other ones. However, as the performance of the other partitions is bad the maximum performance will be 75% of the optimal.



(a) Iterations to Optimal (b) Optimal value frequency

Fig. 3. Selection mechanism and System Reactivity.

Time between checks. Between each partition change, the selection mechanism waits t seconds. For the previous experiments we selected a high t value (15 seconds) to accentuate bad selections. However, when we were designing the experiment, we found that the test time must be selected accordingly: with a short test time (to be able to run more repetitions, in a reasonable time) the dynamic mechanism will not be able to detect/learn about how the system behaves. Additionally, changes below t inside a workload will not be detected directly.

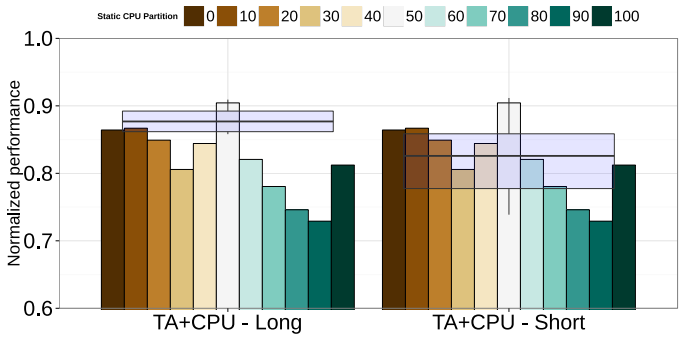


Fig. 4. Effect of the test length on the dynamic mechanism.

Reducing test time below that value will produce unexpected results (mainly not being able to find the optimal value). To show this described effect, we repeated **TA+CPU** phase using an 8 times shorter workload than in the evaluation increasing the variability and reducing the performance obtained (Figure 4, compares the each static partition performance for the phase, as 11 vertical bars, with the performance obtained with **DYON** as a blue boxplot). This is produced because all the partitions and its associated performance has not been explored.

6 Conclusions

The Onload Framework complements Linux *workqueues* to support our Onloaded Services, background-like services and tasks aiding the system, and offer new workflow capabilities to parallelize code based on the resources available.

We show that if we allow to use more CPU to those Onloaded Services we can obtain extra benefits. To control the quantity of CPU offered to those services dynamically we created **DYON**, a dynamic CPUs partition system. **DYON** is guided by CPIO metric, providing a way to compare two system snapshots and decide if the the new CPU partition is providing more or less performance.

We have seen on the evaluation as CPIO metrics offers more performance than direct application metrics as CPIO captures the whole system behaviour. With **DYON** we obtain more performance than any static selection.

References

1. Akram, S., Marazakis, M., Bilas, A.: NUMA Implications for Storage I/O Throughput in Modern Servers. 3rd Workshop on Computer Architecture and Operating System co-design (CAOS'12) (2012)
2. Axboe, J.: fio Flexible IO Tester. http://git.kernel.dk/?p=fio.git (2012)
3. Berry, D.A., Fristedt, B.: Bandit problems: Sequential allocation of experiments. Monographs on Statistics and Applied Probability, 1985 (1985)
4. Bovet, D.P., Cesati, M.: Understanding the linux kernel. O'Reilly (2003)
5. FORTH, UPM, BSC, IBM, INTEL, NEUROCOM: IOLanes - Advancing the Scalability and Performance of I/O Subsystems in Multicore Platforms. `http://www.iolanes.eu` (2010)
6. Heo, T.: RFC of Concurrency Managed Workqueues patch. `http://lwn.net/Articles/355347/` (2012)
7. Heo, T.: RFC of Concurrency Managed Workqueues patch take 2. `http://lwn.net/Articles/367289/` (2012)
8. Klassert, S.: RFC of padata patch. `http://www.mail-archive.com/linux-crypto@vger.kernel.org/msg03329.html` (2012)
9. Love, R.: Linux System Programming: Talking Directly to the Kernel and C Library. O'Reilly Media, Inc. (2007)
10. Molnar, I.: Inclusion of Work Queues on Linux. `http://lwn.net/Articles/11247/` (2012)
11. Neurocom: TariffAdvisor / TariffSuite. `http://www.tariffsuite.com` (2012)
12. Nou, R., Giralt, J., Cortes, T.: Automatic i/o scheduler selection through online workload analysis. In: Apduhan, B.O., Hsu, C.H., Dohi, T., Ishida, K., Yang, L.T., Ma, J. (eds.) UIC/ATC. pp. 431–438. IEEE Computer Society (2012)