Understanding scheduler performance : a feature-based approach

Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, Nicolas Palix

Understanding scheduler performance : a feature-based approach

Redha Gouicem*, Julien Sopena*, Julia Lawall*, Gilles Muller*, Baptiste Lepers^, Willy Zwaenepoel^, Jean-Pierre Lozi◊, Nicolas Palix$^{\Omega}$

* Sorbonne Université, LIP6, Inria, *first.last@lip6.fr*
^ University of Sydney, *first.last@sydney.edu.au*
◊ Oracle Labs, *jean-pierre.lozi@oracle.com*
$^{\Omega}$ Université Grenoble Alpes, *nicolas.palix@univ-grenoble-alpes.fr*

Résumé
The thread scheduler of an operating system is a performance-critical service for applications. However, general-purpose operating systems' schedulers do not offer the best performance for all applications, despite an increasing complexity in features and heuristics. Understanding the impact of individual features on the performance of applications is a difficult task because of these features' entanglement. We propose a feature-based model for scheduler and an experimental methodology in order to better understand the ins and outs of process scheduling.

1. Introduction

The thread scheduler is the operating system service allocating CPU time to threads in order to provide the best performance possible. However, the best performance cannot be achieved in the same way for all applications. For example, it is a widespread intuition that interactive applications usually require frequent short CPU time slices, while batch applications benefit more from having long time slices with no interruption. General-purpose operating systems such as Linux, FreeBSD or Windows usually provide a scheduling policy that is expected to perform well for most applications (i.e. CFS [1], ULE [9] and the Windows system scheduler [2], respectively). However, the development of other schedulers for Linux that perform better with specific use cases, such as BFS [5] or MuQSS [6] that target desktop systems, shows that it is no easy task. This claim has been recently confirmed by Bouron et al. [4] who have compared CFS and ULE by implementing the latter in Linux and comparing it with CFS on a large number of applications. They show that neither scheduler is always best : depending on the application, either CFS or ULE performs better.
However, the aforementioned scheduler comparisons share a common flaw : the difficulty to pinpoint which part of the studied schedulers makes one better than the other with a given application. In this paper, we propose a methodology to better understand the origin of the performance, or lack thereof, of a schedulers on specific applications. To do so, we model a process scheduler as a feature tree [8] in order to explore multiple variations of features, such as the choice of the thread to run on a core [1] or the load balancing algorithm. We have currently defined twenty-two features in our model, mainly inspired by CFS and ULE, and generate 5.832

1. In this paper, a core is a synonym for a hardware thread

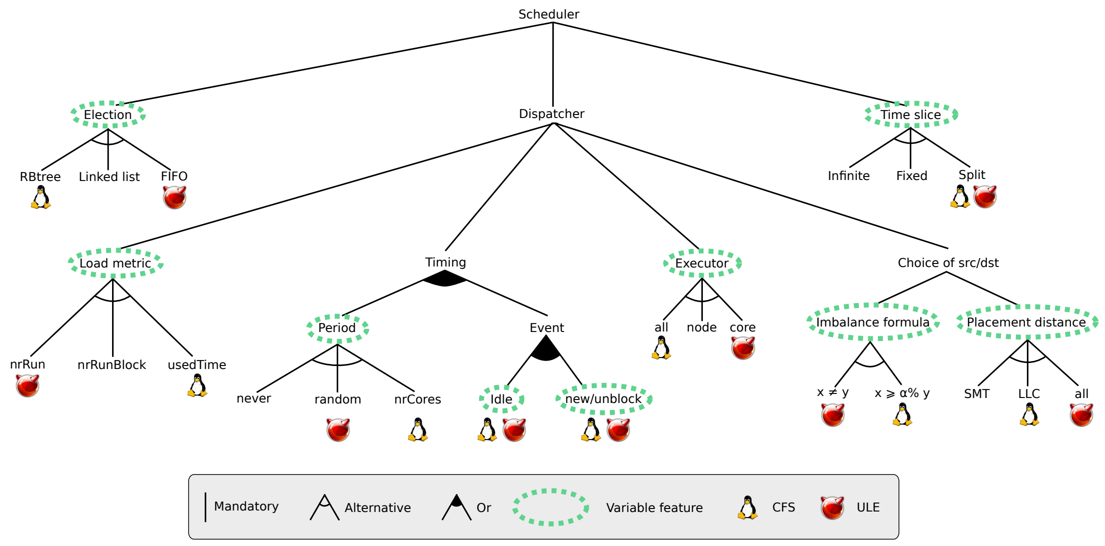


FIGURE 1 – Scheduler feature tree

schedulers by combining these selected features. We also propose a methodology to identify the good and bad features of a scheduler for a given application.
This paper is organised as follows. Section 2 presents our scheduler feature model. Section 3 presents our first preliminary results. Section 4 concludes.

2. Feature-based schedulers

The formalisation of process schedulers first requires a thorough study of multiple ones. Our study targeted the schedulers of general-purpose operating systems such as CFS and ULE. We expose multiple recurring features in all the studied schedulers. These features are implemented in various ways in order to achieve different performance goals. We choose the feature model described by Kang et al. [8] to represent as a formalism these recurring features. This model describes a system as a set of features and sub-features that can be mandatory, optional or alternatives (`xor`).
Figure 1 shows the resulting feature tree. Our study of existing process schedulers exposed two main features : the choice of which thread runs on a core (*election*) and for how long (*time slice*). In addition to these features, we must take into consideration the operating system's design at the scheduler level. In our case, we use the Linux kernel to implement our schedulers. On multicore systems, Linux instantiates one runqueue per core, thus raising the need for a mechanism that places threads on cores. The thread placement policy is determined by the *dispatcher*. This feature has four sub-features : *load metric*, *timing*, *executor* and *choice of source and destination*. For each recurring feature (dashed green circle), we propose multiple implementations. The features that are the closest to what Linux CFS or FreeBSD ULE implement are marked with the corresponding OS logo as well. The rest of this section describes these features in detail.

Election feature. This feature determines how the scheduler chooses the next thread to run on a core. On each core, threads are stored in a runqueue. We study two possible election mechanisms : choosing the thread that has run the least in the runqueue or the thread that arrived first in the runqueue. The first mechanism is implemented by sorting threads by ascending runtime in the runqueue, i.e. the thread with the lowest execution time has the highest priority. This allows the scheduler to be fair to interactive threads that need to be scheduled frequently for a short period of time. We implement two versions of this mechanism : one that uses a red-black tree [7] (called `rbtree`) and one that uses a circular doubly linked list (called `linked list`). Red-black trees provide insertion, deletion and pop operations with an $O(\log n)$ average complexity, but may be costly because of the rotations needed to keep the tree balanced. Sorted doubly-linked lists provide deletion and pop operations in constant time, but insertion is more costly ($O(n)$) because the list needs to be traversed to place the thread in the correct location and keep the list sorted. CFS implements this mechanism with a red-black tree. The second mechanism is implemented using a FIFO (First-In First-Out) circular doubly linked list. A FIFO provides insertion, deletion and pop operations in constant time. Since threads are not sorted by runtime but follow a first-in first-out pattern, latency sensitive applications may be delayed if there are a lot of threads in the runqueue. ULE implements a variant of this feature with two FIFO lists, one for interactive threads and one for batch threads.

Time slice feature. When a thread is given the right to run on a core, it owns this right for the duration of its time slice. The time slice feature determines how this duration is computed. We consider three alternatives for this feature. The `infinite` time slice alternative disables preemption. Context switches are driven by the thread voluntarily yielding the CPU or being blocked on a resource. This is present in batch scheduling policies. The `fixed` alternative allocates the same time slice for all threads. In our experiments, the fixed time slice is set to 10 ms. The `split` alternative allocates a time slice that depends on the number of threads present in the runqueue.

$$timeSlice = \begin{cases} X\ ms & \text{if } |threads| > T \\ \dfrac{Y}{|threads|}\ ms & \text{otherwise} \end{cases}$$

CFS and ULE both implement this feature with different values for X, Y and T (on our testing systems, $X = 3$, $Y = 24$ and $T = 8$ for CFS, $X = 4$, $Y = 25$ and $T = 6$ for ULE). Our implementations follows the definition used by CFS. If the core is overloaded (i.e. more than 8 threads in the runqueue), each thread is allocated a 3 ms time slice. Otherwise, each thread is allocated an equal share of a 24 ms period.

Load metric feature.. This feature is used to represent the amount of work available on a core. This metric allows the dispatcher to compare cores and decide if threads should be migrated from one core to another. We consider three alternative metrics : `nrRun`, `nrRunBlock` and `usedTime`. `nrRun` measures the load of a core as the number of threads in the runqueue (i.e. runnable threads), as implemented in ULE. `nrRunBlock` takes into account the number of runnable threads as well as the number of threads that blocked last on this core. This allows the scheduler to keep track of threads that blocked on a core and will eventually wake up on it. This can be useful when blocked threads wake up immediately since it prevents the scheduler from migrating a distant thread to balance a load that will immediately be balanced anyway. However, threads that stay blocked for a long time will weight on the load and may prevent balancing even if there is no runnable thread in the runqueue. `usedTime` defines the load of a thread as the proportion of time the thread spent runnable ($runnableTime$) regarding its allocated time slice ($timeSlice$). This alternative also takes previous loads into account to smooth the load over time : 80% of the load corresponds to the previous load while 20% depends on $runnableTime$ and $timeSlice$. This is a simplified version of CFS's implementation of decaying load average. Therefore, the load of a thread at time t ($load_t$) is computed as follows :

$$load_t = 0.8\ load_{t-1} + 0.2\ \frac{runnableTime}{timeSlice}$$

Timing feature.. This feature is used to choose at which moment threads are migrated among cores. We divide this feature into two sub-features : *period* and *event*. The *period* feature determines if a load balancing operation should be triggered periodically and, if so, the period

between those operations. We implement three alternatives for this feature. The `never` alternative disables periodic load balancing altogether. The `random` alternative determines the time of the next balancing event by randomly picking a value in the $[0.5, 1.5]$ seconds range at each periodic load balancing event. This is ULE's implementation of this feature. The `nrCores` alternative determines the period of load balancing depending on the number of cores on the machine : on an n-core machine, a load balancing event is triggered every n milliseconds. CFS implements this feature in this way.
The *event* feature triggers migrations when a core has no more threads to run (*idle* event) or when a thread is created or wakes up from a blocking operation like a synchronous IO operation (*new/unblock* event). When an idle event occurs, the scheduler can either reduce the power consumption of this core by entering a lower power state or perform idle balancing to try to keep it busy. In the latter case, when becoming idle, a load balancing operation is triggered on this core to allow it to find pending work on another core. This feature determines if idle balancing is enabled or disabled. As for the new/unblock event, the scheduler can migrate the thread concerned at this moment. CFS and ULE both balance threads during these events. The choice of the core where the thread that triggered this event is determined by the *placement distance* feature presented later on.

Executor feature.. The periodic load balancing algorithm distributes work among cores. In order to do that, load balancing events are triggered periodically on the cores of the machine. The executor feature determines the cores that perform load balancing operations. The `all` alternative allows each core to perform periodic load balancing operations to balance itself with another core of its own choosing. All cores can do this in parallel but may take conflicting migration decisions and fail. CFS implements this alternative. The `node` alternative allows only a single core per NUMA node to perform load balancing operations. This core performs a load balancing operation for each core in the node. On a n-node machine, n load balancing operations can take place in parallel, thus minimising the probability of conflicts. The `core` alternative allows only one core to perform load balancing for all the cores of the machine. ULE implements this alternative.

Choice of source/destination feature.. This feature determines how the source and destination core of a migration are determined. For migrations due to the load balancing algorithm (either periodic or after an idle event), the destination is the core for which the load balancing is executed. The source, however, is determined by the *imbalance formula* sub-feature that defines if two cores need to be balanced. We implement two alternatives for this formula. The $x \neq y$ formula considers two cores unbalanced if their loads are different. The $x > 120\%y$ formula considers two cores unbalanced if there is at least a 20% difference between their loads. For both alternatives, the chosen source is the most loaded core among those which satisfy the formula.
For migrations due to a new or an unblock event, the source is the current position of the thread (or of the thread's parent for new threads), and the destination is determined by the *placement distance* sub-feature. In most cases, it is beneficial to keep a thread close to its most recently used core because the data it was using may still be available in this core's hardware caches. However, this can lead to a load imbalance between cores if threads are always kept on the same subset of cores. The placement distance feature determines how far away from its previous location a thread can be migrated. We implement three alternatives : a thread can only be placed on a SMT sibling (`SMT`), a core in its last level cache domain (`LLC`), or any core on the entire machine (`all`). The core selected is the least loaded core among the cores that respect the placement distance.

		Execution time (s)		
Application	**Threads**	**CFS**	**CFS + Pin**	**Best generated scheduler**
blackscholes	24	$39.8 \pm 0.41\%$	$39.7 \pm 0.54\%$	$39.4 \pm 0.21\%$ (-1.0% w.r.t. CFS)
bodytrack	16	$43.1 \pm 1.75\%$	$43.3 \pm 2.59\%$	$42.56 \pm 0.84\%$ (-1.3% w.r.t. CFS)
canneal	12	$60.6 \pm 0.53\%$	$59.6 \pm 0.53\%$	$59.4 \pm 1.90\%$ (-2.0% w.r.t. CFS)
facesim	16	$168.9 \pm 0.71\%$	$169.9 \pm 0.80\%$	$155.0 \pm 0.01\%$ (-8.2% w.r.t. CFS)
ferret	12	$42.2 \pm 0.94\%$	$42.9 \pm 1.01\%$	$44.3 \pm 0.67\%$ ($+5.0\%$ w.r.t. CFS)
fluidanimate	16	$89.2 \pm 2.43\%$	$83.4 \pm 1.40\%$	$85.1 \pm 3.54\%$ (-4.6% w.r.t. CFS)
streamcluster	12	$108.8 \pm 1.73\%$	$96.8 \pm 3.20\%$	$96.8 \pm 0.44\%$ (-11.0% w.r.t. CFS)

TABLE 1 – PARSEC performance with different schedulers (lower is better)

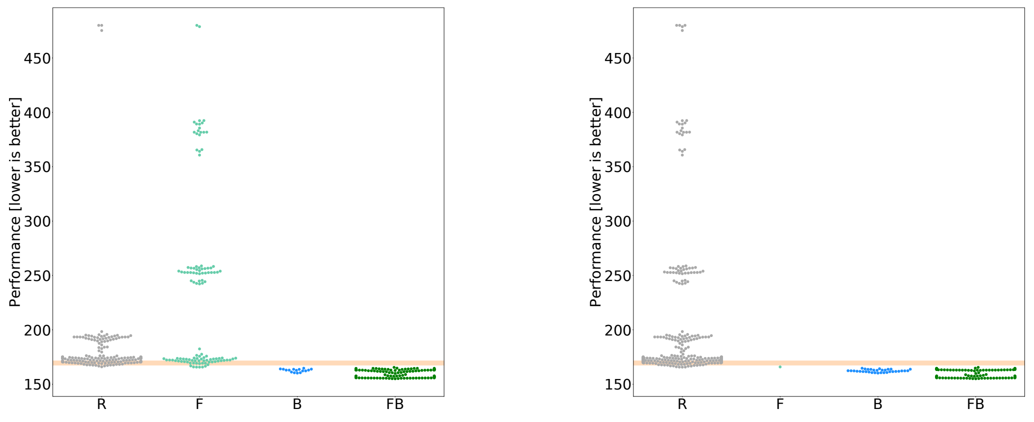
3. Preliminary results

From the feature tree described in Section 2, we can generate 5.832 schedulers by combining the features we identified. The idea is to test all these schedulers on a set of applications and measure their performance. To do so, we developed a new API in the Linux kernel to allow scheduler hotplugging and generate our schedulers with this API. In this paper, we only present preliminary results with 486 generated schedulers (some features were fixed). We then analyse these experimental results to figure out which features have a positive impact on application performance.

Experimental setup. We run our experiments on a two-socket hyperthreaded Intel Xeon E5645 (12 cores, 24 threads) and 64 GiB of RAM. All machines run a Debian 8 operating system, a Linux 4.19 kernel (patched to allow our schedulers to run) and all our benchmarks and dependencies installed. Each application is always run on the same machine to avoid discrepancies due to hardware differences. We run all our generated schedulers, as well as CFS and CFS with thread pinning as baselines, on each application 10 times. CFS with thread pinning disables thread migration and places threads sequentially on the machine (first thread on core 0, second thread on core 1, and so on). These preliminary results only present the evaluation for a set of seven PARSEC [3] applications : `blackscholes`, `bodytrack`, `canneal`, `facesim`, `ferret`, `fluidanimate` and `streamcluster`.

Performance analysis. Table 1 presents the results of our benchmarks. The performance exhibited in the *best generated scheduler* column corresponds to the best scheduler generated from our model for the given application, excluding schedulers with too large a standard deviation. Out of the seven tested applications, we are able to generate a scheduler that outperforms CFS for six of them. This confirms the result of Bouron et al. [4] and scheduler developers that currently, there is no scheduler that achieves the best performance for all applications. This also exhibits the fact that simple schedulers like the ones we generated can outperform an industry-level scheduler such as CFS. Compared to CFS with pinned threads, our best generated scheduler performs similarly. This is an expected result since these applications use less threads than available cores on the machine. This means that pinning a thread per core should be optimal in terms of thread placement. However, for `facesim`, we are able to generate a scheduler that outperforms CFS with pinned threads by 8.8%. This shows that local scheduling is also important regarding performance.

Feature analysis. Now that we have a large variety of schedulers, we want to understand which feature makes a scheduler good for a given application. There is no straightforward way to know if a feature is always good, always bad or has no impact on the performance of



(a) Round 1 (b) Round 2

FIGURE 2 – Feature analysis for `facesim`

an application. Moreover, features can be correlated : two features can be beneficial when used together but harmful if used separately. In this paper, we present a simple methodology to classify features as good, bad or neutral for an application, regardless of the previously mentioned correlations.
First, we count the number of occurrences of each feature among the best performing schedulers. These best schedulers are those at most 10% worst than the best scheduler (known as B). If a feature appears in more than 80% of B, it is marked as good, and if it appears in less than 10%, it is marked as bad. Schedulers that have all the good features and no bad feature are in the F set. The schedulers in F and B are in the FB set. Figure 2a shows these new sets of scheduler for `facesim` (to ease comprehension, points in multiple sets are only shown in the rightmost one). To minimise the number of false positives (schedulers in F), we mark as bad the features that are over-represented in F and underrepresented in FB. Figure 2b shows the new sets of schedulers built after this second round. With this new decomposition, we have only 1% of false positives, and 76% of the schedulers in B implement the features we selected as good and bad.

4. Conclusion

In this paper, we present a new feature-based model for process schedulers in order to study the impact of individual features on the performance of applications. We also provide some first promising results in the identification of good features for an application thanks to a new API that allows hotplugging schedulers in the Linux kernel. Future work will consist of applying our methodology to a larger set of applications and try to drive the benchmarks in order to test less schedulers without losing precision in the identification of good features.

Bibliographie

1. CFS scheduler design. – `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`. Accessed : 24-12-2018.
2. Windows Scheduler. – `https://docs.microsoft.com/en-us/windows/desktop/procthread/scheduling`. Accessed : 24-12-2018.

3. Bienia (C.). – *Benchmarking Modern Multiprocessors*. – Thèse de PhD, Princeton University, January 2011.
4. Bouron (J.), Chevalley (S.), Lepers (B.), Zwaenepoel (W.), Gouicem (R.), Lawall (J.), Muller (G.) et Sopena (J.). – The Battle of the Schedulers : FreeBSD ULE vs. Linux CFS. – In *USENIX Annual Technical Conference*, pp. 85–96. USENIX Association, 2018.
5. Con Kolivas. – FAQs about BFS. – `http://ck.kolivas.org/patches/bfs/bfs-faq.txt`. Accessed : 03-01-2019.
6. Con Kolivas. – MuQSS - The Multiple Queue Skiplist Scheduler. – `http://ck.kolivas.org/patches/muqss/sched-MuQSS.txt`. Accessed : 03-01-2019.
7. Guibas (L. J.) et Sedgewick (R.). – A dichromatic framework for balanced trees. – In *FOCS*, pp. 8–21. IEEE Computer Society, 1978.
8. Kang, Kyo C and Cohen, Sholom G and Hess, James A and Novak, William E and Peterson, A Spencer. – *Feature-oriented domain analysis (FODA) feasibility study*. – Rapport technique, Carnegie-Mellon University, Pittsburgh PA, Software Engineering Institute, 1990.
9. Roberson (J.). – ULE : a modern scheduler for FreeBSD. 2003.