Parameterized Optimization of Variable-Timeslice Process Scheduling

Robert Miller

May 2nd, 2016

Abstract

This paper discusses a proposed process scheduler which formulates the problem of fair scheduling as a nonlinear optimization problem. This formulation allows for a great deal of flexibility from the scheduler, including a significant potential for customization through parameterization or through outright modification of the objective function and associated constraints.

The proposed scheduler generates a vector of proposed timeslices for some user-defined period which can then be used with a variety of process-selection methodologies to choose the next process to be scheduled. Between the parameterization, extensibility, and variability of selection techniques, this model can be tweaked to fit the specific needs of a variety of systems or can be used to mimic the behaviors of other schedulers.

The MATLAB framework used to test an implementation of this scheduler can be found at `https://git.io/vwMBL`. Further details may be found at the project's web page, `http://bioniconion.github.io/pvt_sched/`.

Contents

1 Introduction **1**
1.1 Overview of Process Scheduling 1
1.2 Overview of Existing Schedulers 2
1.3 Proposal: PVT Scheduling 3

2 Optimization Formulation **4**

3 Experimentation **5**
3.1 Experimental Framework 5
3.2 Methodology 6
3.3 Results 7
3.3.1 Varying Priority & Simulation Modes 7
3.3.2 Varying Weight 9
3.3.3 Varying Timespan Length 9

4 Performance Analysis **10**
4.1 Quality of Results 10
4.1.1 Effect of Simulation Modes 10
4.1.2 Effect of Weights 10
4.1.3 Effect of Lengths 11
4.2 Comparison to Other Schedulers 11
4.3 Advantages & Disadvantages 11

5 Future Work **12**
5.1 aPVT: Approximated PVT Scheduling 12
5.2 oPVT: Optimized Parameterization of PVT 12
5.3 mPVT: Multiprocessing PVT Scheduling 12
5.4 Objective Function Extensions 13
5.5 Expanded Testing Framework 13

6 Conclusion **14**

1 Introduction

1.1 Overview of Process Scheduling

The overarching goal of process scheduling is quite simple: some number of processes need to utilize the limited resources of an available processor, and the task of the scheduler is to allocate those resources in some way to those processors. However, there are many inherent concerns and considerations inherent to this task that must be considered (and some compromise must be drawn between the varying factors) [FG]:

Throughput—Schedulers seek to maximize the number of processes being completed at any given point in time.

Latency—Schedulers seek to minimize the total elapsed time between the time that a process is created and the time that it is completed.

Response Time—Schedulers seek to minimize the delay between times that a process is scheduled (meaning that a process is able to effectively appear as if it is running in real time).

Starvation—Schedulers seek to ensure that every process is able to make progress at some point (though this makes no guarantee about exactly *when* progress will be made, and some processes may be deferred for extended amounts of time before ultimately being scheduled).

Fairness—Schedulers seek to allocate the available processing resources as fairly as possible, meaning that every process gets an equal share of processing time.

Overhead—Schedulers seek to run as efficiently as possible. On modern systems, scheduling occurs hundreds or thousands of times per seconds, meaning that algorithmic complexity of the scheduler can have a significant impact on overall system performance.

Although it is generally not possible for any scheduler to behave optimally with respect to each of these criteria, good schedulers are those which are able to perform well at many of the above tasks (although the exact set of tasks which are important for a 'good' scheduler is highly dependent on the domain in which that scheduler will be used; the priorities of a user-facing desktop system are naturally different from the priorities of a massive server).

1.2 Overview of Existing Schedulers

Many different schedulers already exist, both as theoretical models and as fully-functional software packages used with scheduling systems to govern the way that processes run on real computers. As it would be infeasible to write a meaningful comparison to every possible scheduler, I will limit my discussion to the following common types of schedulers:

First Come, First Served (FCFS)—Commonly also referred to as a First In/First Out (FIFO) scheduler, the FCFS model is quite simple: processes are allowed to run to completion in the order in which they arrived in the runqueue. FCFS tends to have extreme problems with throughput, latency, and response time, but has a strong guarantee about starvation (as long as all processes ultimately terminate). By its nature, FCFS is an O(1) scheduler—it must only select the next process from a queue when the time comes.

Fixed Priority Preemptive Scheduling (FPPS)—Processes are stored in a runqueue sorted by priority, and the highest-priority process in the queue at the time of scheduling is the next one to run (which may mean that a running low-priority process will be swapped out in favor of a high-priority process). Much like FCFS, FPPS may have issues with throughput, latency, and response time, but may also face the issue of starvation with regards to low-priority processes:

if enough high-priority processes are enqueued, then the low-priority processes will never be allowed a chance to run. By maintaining a sorted list of processes—perhaps as a red-black tree—FPPS is able to run with extremely low overhead.

Round Robin (RR)—The goal of Round Robin scheduling is to maximize fairness by allotting every process a precisely equal amount of time to run in a given span. This is accomplished by cycling through a list of all running processes and granting each one some fixed timeslice worth of running time in turn. RR has significantly improved throughput, latency, and response time relative to FCFS and FPPS, as well as a guarantee of not starving processes and a reasonable semblance of fairness. However, RR scheduling makes no account of process priority; time-critical processes (i.e. streaming video) are given exactly as much time to run as processes which only require a limited amount of processing time (i.e. text editors).

Completely Fair Scheduling (CFS)—The CFS scheduler maintains a notion of the total time allotted to an individual process in an effort to balance out resource utilization. On top of this, individual processes are weighted by their user-assigned 'niceness', which governs their willingness to yield resources to other processes (and therefore determines the total percentage of system resources made available to them). Like RR scheduling, CFS has good behavior across the board—it has been chosen as the default Linux scheduler for this reason—but also requires a non-trivial amount of overhead to maintain state about running processes and to compute which process should be scheduled to run next [FG].

Optimization Formulations—In my research, I found multiple formulations of scheduling tasks as optimization problems, but each seemed to be focused on the task of assigning processes to machines (e.g. for usage in multiprocessing environments). One such formulation is that of 'job shop scheduling' [Bow59], which seeks to optimally schedule n processes on m machines. Although this is an interesting and valuable area of research, this type of scheduling is not what I intend to explore; my focus is on optimal single-machine scheduling techniques.

Each of these schedulers is suited to specific realms of scheduling, and none of them can be truly generally applied: FPPS may be useful for scientific computing (e.g. contexts in which many processes of varying priorities are running on the same system with no firm deadlines), but that same scheduling model would be totally impractical on a user-facing system. Additionally, each of these schedulers has a specific and concretely defined mode of operation: RR evenly allocates resources, for instance, and cannot be used in contexts which require prioritization of certain processes over others.

1.3 Proposal: PVT Scheduling

My proposal is to formulate the scheduling task as a nonlinear optimization problem which can be solved to determine the most fair allotment of system resources at any given time. One of the important goals of this formulation is

to make a parameterizable, customizable, and extensible framework for process scheduling which can be widely applied across a variety of contexts and use-cases. Due to this flexibility, I call the proposed scheduler the Parameterized Variable-Timeslice Scheduler (PVT).

Broadly speaking, the optimization formulation will solve for a vector of variable-length timeslices which represent the fairest allocation of resources to a list of processes. This information could then be used in a variety of ways to perform process scheduling; multiple methodologies will be discussed in Section 3 of this paper.

2 Optimization Formulation

The task of generating a timeslice vector has been formulated as the following optimization problem:

$$\begin{aligned} &\textit{minimize: } f(t) = w\sum_{i=1}^{n}\left[\frac{p_i(m-t_i)}{p_{max}}\right] + (1-w)\sum_{i=1}^{n}\left[\frac{s-l_i}{t_i}\right] \\ &\textit{subject to: } 0 < t_i < m \ \forall i = 1...n \\ &\qquad\qquad \sum_{i=1}^{n} t_i = m \end{aligned}$$

In the nonlinear program above, w and m are user-defined parameters: w controls the relative weighting of the two terms and m defines the overall timespan that the computed timeslices should cover; s and p_{max} define various aspects of system state (s is the current time measured in units since the epoch and p_{max} is the maximum priority among all processes); the vectors p and l contain information about process priorities and last scheduled times respectively. When solved, t will contain a set of optimal timeslices which will sum to the user-specified time value m. Additionally, the value of $f(t)$ corresponds to the amount of 'unfairness' inherent to the computed schedule.

The first term of the objective function encodes the importance of process priorities in scheduling decisions. High-priority processes will be awarded greater percentages of the available timespan m in accordance with that process' priority.

The second term of the objective function encode liveness, or the amount of time that has elapsed since a process was most recently run. l contains the times at which each process finished running (meaning that $s-l_i$ will be 0 for the process which finished its timeslice just before scheduling occurred); the longer in the past the process was last run, the larger the denominator (the allotted timeslice) must be in order to keep the unfairness of the objective function low.

Although this formulation already provides a reasonably robust framework for process scheduling in a variety of contexts, adding additional terms or constraints could allow for a wider range of behaviors to be expressed. This is one

of the main strengths of PVT: its flexibility allows it to be employed in a variety of settings, and it can be parameterized or extended to mimic the behavior of other schedulers (more information is provided in Section 4 below).

3 Experimentation

3.1 Experimental Framework

To test PVT, an experimental framework was built with MATLAB to simulate a system with an arbitrary number of of running processes for an arbitrary duration (this testing framework can be found in the pvt_sched repository at `https://git.io/vwMBL`). The framework is comprised of two functions:

```
runSchedulerSimulation(numProcesses, simulationLength,
    varargin)

generateTimesliceVector(processes, currTime, weight,
    length)
```

The first of these functions runs a full simulation of PVT using user-specified parameters (or default values if none are provided) by calling the second function, which solved the optimization problem outlined above to generate a timeslice vector.

The options exposed to the user are the following:

OPTION	EFFECT
numProcesses	Number of processes in the simulation
simulationLength	Overall length of simulation
weight	Relative weighting of priority vs. liveness
length	Timescale of computed timeslice vector
priorityMode	Priority sampling mode (see below)
simulationMode	Simulation mode (see below)

The default value for weight is 0.5 and the default value for length is 100. numProcesses and simulationLength must be provided by the user and therefore have no default value.

The priorityMode parameter defines how the assigned priorities for the simulated processes can be generated, and can take any of the following values (default is 'random'):

OPTION	EFFECT
random	Priorities are assigned uniformly at random in (1,10)
uniform	All processes have a priority of 0
ascending	Processes are assigned ascending priorities 1 to n

The simulationMode parameter defines how the simulation interprets and uses the results of the optimization (e.g. how processes are scheduled based on the raw data in the timeslices vector). Practically speaking, this controls the number of timeslices that are taken from the timeslice vector before it is recalculated with another call to the generateTimesliceVector function. This parameter can take any of the following values (default is 'top'):

OPTION	EFFECT
top	Only the top timeslice is used
topThird	The top third of the timeslice vector is used
threshold	Timeslices above the average ($^{length}/_{numProcesses}$) are used

The runSchedulerSimulation returns a struct of processes with the following fields:

FIELD	MEANING
count	Total number of processes (equal to numProcesses)
lastRuns	Vector of most recent runtimes
priorities	Vector of process priorities
maxPriority	The maximum value in the priorities vector
numTimeSlices	Vector of number of times that processes were scheduled
runTimes	Vector of total allotted timeslices
numOptimizations	Total number of timeslice vectors computed

Running a simulation can be done by (for instance) the following function call:

```
procs = runSchedulerSimulation(5, 1000);
```

This will run the simulation of 5 processes for 1000 ticks, storing the results in the struct variable 'procs'.

3.2 Methodology

The experimental framework detailed above (with more details available as comments in the code itself, available in the repository at `https://git.io/vwMBL`) was used to perform a variety of tests on the PVT scheduler. In particular, tests were performed to judge the performance at default settings (meaning a weight of 0.5 and a timespan of 100 units per timeslice vector) for 5 processes over a duration of 1,000 units of overall time. Each of these tests was performed with all three of the priority sampling modes and all three of the simulation modes (as defined above).

After these initial experiments, another set of experiments was performed to assess the impact of adjustments to the weight and length variables. Weights were tested in increments of 0.2 with a constant length of 100 and lengths of 25, 50, 100, 250, 500, and 1,000 were tested with a constant weight of 0.5. For each of these tests, ascending priorities and top process sampling were used.

Although it is possible that varying both variables simultaneously may affect the results, the goal of these experiments was to assess the impact that each individual variable has on the observed scheduler performance.

For all experiments involving random factors (namely those using randomly assigned priorities), the experiment was repeated a total of five times. The results of each repetition of the experiment are reported separately, as the relationship between process priorities was important in producing the observed values (and thus no strong comparison can be made between multiple runs, even of the same experiment).

The process for generating each of the relevant process structs has been contained in a MATLAB script called runExperiment which can be found in the project repository.

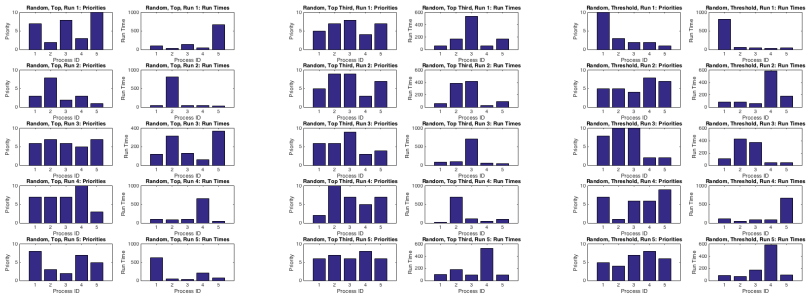
3.3 Results

Larger versions of all of the following figures can also be found at the project web page, `http://bioniconion.github.io/pvt_sched/`

This section merely provides the observed data gathered from running the experiments described in Section 3.2; analysis of the data obtained can be found in Section 4.

3.3.1 Varying Priority & Simulation Modes

The following bar graphs show the results of the experiments run with varying combinations of priority sampling modes and timeslice vector interpretation modes. In the case of the random priority results, the bar chart in the left half of each column contains the priorities for those processes and the chart on the right side holds the observed running times. For the ascending priority results, the processes are listed in order of ascending priority (from 1 to 5).



(a) Top (b) Top Third (c) Threshold

Figure 1: Results for Randomly Assigned Priorities

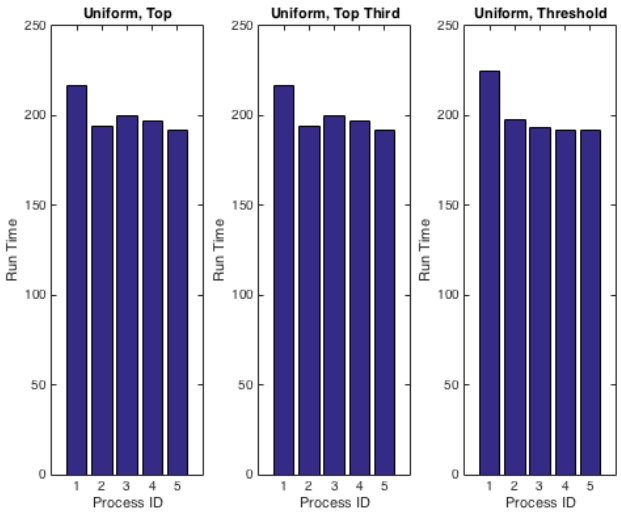


Figure 2: Results for Uniform Priorities

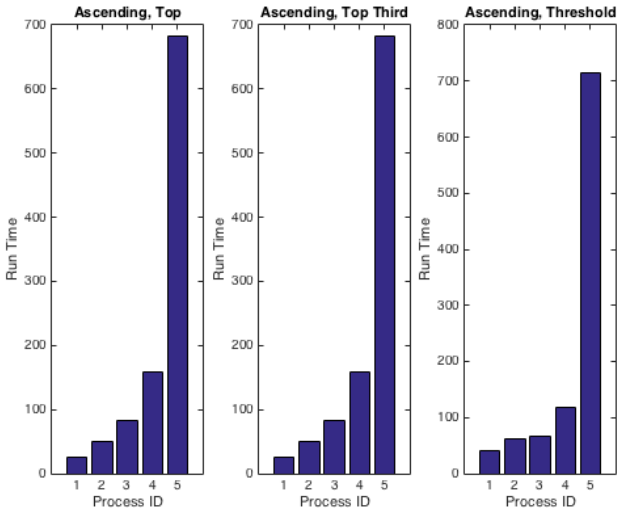


Figure 3: Results for Ascending Priorities

3.3.2 Varying Weight

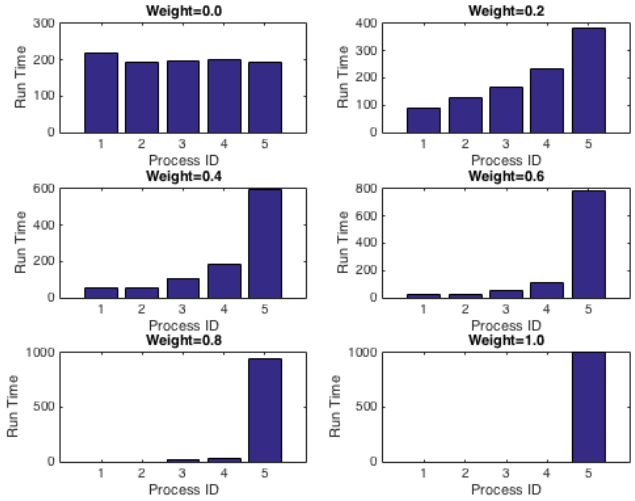


Figure 4: Results for Varying Weights

3.3.3 Varying Timespan Length

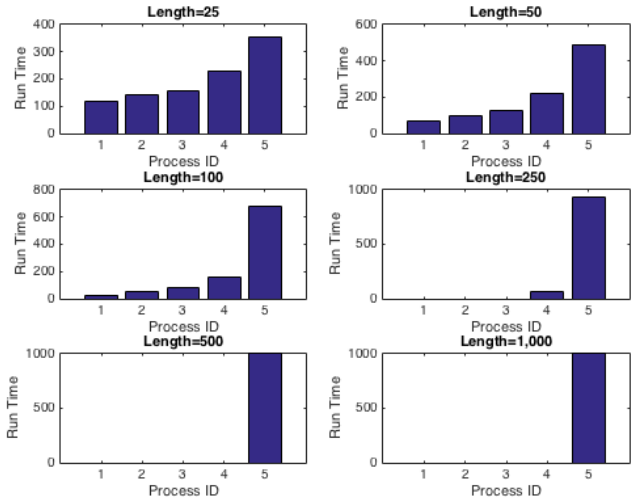


Figure 5: Results for Varying Timespan Lengths

4 Performance Analysis

4.1 Quality of Results

4.1.1 Effect of Simulation Modes

One of the interesting things to note in Figures 2 and 3 is that the different methods of selecting processes to schedule from the timeslice vector (e.g. taking only the top value as opposed to taking multiple values) had only a limited effect on the actual performance of the scheduler: the results in each of the three sub-graphs are almost entirely consistent with one another, meaning that the various scheduling models can effectively be used interchangeably. However, each of the different modes had a quantifiable effect on scheduler running time: the number of calls to the generateTimeSliceVector function was recorded for each of the simulations, yielding the following results:

Priority Type	**Top**	**Top Third**	**Threshold**
Uniform	32	32	12
Ascending	27	27	14

Top Third mode was unable to gain any benefit relative to Top (as $5/3$ floors to 1), but the Threshold mode saw a marked performance increase relative to the other two modes. Although running in Threshold mode means accepting slightly less mathematically perfect results with regards to fairness, the performance gains are significant enough to make it a serious consideration for usage in situations when precise accuracy isn't a requirement.

4.1.2 Effect of Weights

Figure 4 shows that nudging the weight value even just slightly can have a significant impact on the way that the scheduler performs. On the lower extreme, priority is totally ignored and processes are allowed to run for virtually identical periods (the slight variation is a function of the lastRun value; processes which are not scheduled in a given round are 'rewarded' with a slightly longer timeslice in following rounds). On the opposite extreme, fairness is ignored in favor of priorities—the highest-priority task is allowed to run to completion (a feature not included in this testing framework; see Section 5.5 for more details.

However, the more interesting results are those obtained when the weight is somewhere between 0 and 1. Figure 4 shows that the scheduler performance interpolates smoothly between these two extremes, increasingly favoring high-priority tasks as the weight increases. This gives users a significant tool for adjusting the performance of their system: in concert with user-assignable process priorities, users can tightly control the manner in which the scheduler will operate.

4.1.3 Effect of Lengths

As can be seen in Figure 5, varying the lengths of the timespans causes the results to skew increasingly in favor of higher-priority processes—these processes will be the first to be assigned timeslices, and due to the longer duration of each assigned timeslice (scaled by the parameter provided), they will be allowed to run for longer before another scheduling call is made.

However, there is also a clear advantage to allowing for longer timespans: efficiency. The following table shows the number of calls to the generateTimesliceVector function for each timespan length value:

Timespan	25	50	100	250	500	1000
Function Calls	114	57	27	9	3	1

Although the results of the 25- or 50-unit schedules are arguably the most fair, there is a significant trade-off with regards to how many times the expensive optimization solver must be invoked. Much like the discussion in Section 4.1.1, this provides another 'lever' for optimization between mathematical precision (e.g. minimizing unfairness) and computational performance.

4.2 Comparison to Other Schedulers

Comparing PVT to other scheduling solutions is made difficult by the imprecise definition of how a PVT scheduler functions: because the behavior of PVT is so dependent on the parameters passed to it, any performance comparisons made to other schedulers would be dependent on the precise parameters used. However, one of the interesting capabilities of PVT is the way that it can be used to mimic the behaviors of other scheduling solutions. For example, providing a weight of 0 will cause PVT to behave exactly like FPPS (prioritizing high-priority processes to the point that low-priority processes may be deferred indefinitely), while a weight of 1 behaves identically to RR (treating all processes equally and alternating between each of them in sequence).

Especially with further research and extensions to PVT, this capability to mimic other scheduling algorithms could be leveraged as a valuable feature of PVT, especially due to its ability to be re-parameterized in real time to switch efficiently between multiple scheduling techniques based on the current needs of the system (a similar concept is presented in Section 5.2).

4.3 Advantages & Disadvantages

As has been discussed above, the primary advantage of PVT scheduling is its high degree of customization through parameterization. The exact behavior of the scheduler can be controlled with a significant degree of granularity, enabling users to truly customize it to behave according to their needs—and due to its nature, parameters could be changed in real time to accommodate any changes in needs (Section 5.2 discusses this idea in greater detail).

However, this power and freedom comes at a significant cost: overhead, one of the fundamental scheduling concerns laid out in Section 1.1, is a serious concern with the current implementation of the PVT scheduler. Section 5 contains multiple ideas on how this factor might be mitigated through further research and development, but performance remains the single greatest limiting factor for PVT.

5 Future Work

Due to the highly configurable nature of PVT, it lends itself quite naturally to further research and experimentation. As such, this list of potential future topics is (by necessity) incomplete and represents only some of the more compelling avenues for further development.

5.1 aPVT: Approximated PVT Scheduling

As was noted above, the primary limiting factor of PVT is the amount of overhead associated with solving a constrained optimization problem for every call to the scheduler. Although some of the simulation modes presented in this paper helped to mitigate the impact of this overhead, that overhead remains a major concern—and therefore is a compelling hook for further research.

The fundamental intuition to this branch of research is that a mathematically exact solution is unnecessary in this context: as long as processes are scheduled more or less fairly according to the principles outlined in Section 1.1, the scheduler is doing its job. As such, the constrained optimization problem presented in this paper could be relaxed to an unconstrained problem (perhaps through Lagrangian relaxation [Ber99], though other means of relaxation would merit examination). Although unconstrained optimization still has a nontrivial overhead associated with it, many algorithms exist in that realm which could be leveraged to improve results.

5.2 oPVT: Optimized Parameterization of PVT

One of the things that comes with parameterization is the opportunity to optimize over those parameters—the weights and timespans used to compute the timeslice vector have a tangible effect on the performance of the scheduler; this performance impact could potentially be quantified such that another optimization problem could be formulated to improve the performance of PVT in real time.

5.3 mPVT: Multiprocessing PVT Scheduling

As written, PVT can only handle a single runqueue at a time, meaning that it can only effectively handle scheduling of a single core at a time. However, as noted in the introduction, some optimization formulations already exist for

scheduling processes in multiprocessing contexts. Expanding PVT to work across multiple processing cores would entail adding additional information about the number of cores available, the current core on which a process is run, and the cost of migrating a process from one core to another (among other costs), but could potential result in a globally optimally schedule for multiprocessing systems.

5.4 Objective Function Extensions

As mentioned above, one of the fundamental strengths of the PVT formulation of process scheduling is the extensibility of its scheduling model. With some amount of reformulation, the objective function could be rewritten to either include additional terms by default (based on any further needs identified in the scheduler) or could take a lambda function as an additional parameter such that users could define their own terms or the scheduler to use. Either (or both) of these techniques could serve to extend the parameterization options of the scheduler, furthering the ultimate goal of creating a general-purpose scheduling framework.

By way of example, additional terms could be added to emulate a fixed-timeslice scheduler by applying a penalty to any timeslice vectors which don't conform to some user-specified timeslice length. This could allow PVT to simulate a variety of fixed-timeslice schedulers more closely, which may be useful either for research purposes or in contexts in which fixed-length timeslices are preferable (e.g. on systems with relatively low-resolution timers).

5.5 Expanded Testing Framework

The current testing framework (as described above) has no notion of process completion or of processes entering/exiting the system; the tests are built around the simplifying assumption that all tasks start simultaneously and continue infinitely. An expanded testing framework could be built which more accurately simulates the true conditions under which computers operate, with processes entering and leaving the queue at effectively arbitrary times.

Additionally, a multiprocessing expansion of the testing framework (with per-core runqueues) could facilitate a multiprocessing extension to PVT, which would in itself by an intriguing avenue for future research (see above).

Finally, different policies for interpreting the timeslices vector could facilitate better performance. The three options currently available cover a reasonable range, but further research could devise alternative methods which could deliver better performance and/or efficiency.

6 Conclusion

As the results above show, PVT is able to perform well as a scheduler, offering a fair, customizable system for performing scheduling tasks which can be tweaked to be suitable in a variety of settings. PVT can be made to perform similarly to the expected behavior of algorithms such as FPPS and RR, but with the additional benefits of further customization (and of blending between these two extremes).

Ultimately, though, PVT is unlikely to be useful as a replacement for CFS (or other schedulers) as the implementation of choice for running inside the kernel to determine process scheduling in real time on users' systems. The overhead associated with solving optimization problems is simply too great for it to be a particularly feasible means of process scheduling without a significant amount of further development in optimizing the running time (such as the aPVT formulation proposed in Section 5.1).

However, PVT still has utility as a highly customizable scheduler which might be feasible for use in contexts which do not require the same degree of granularity necessary on user-facing devices (e.g. PVT may still be useful for scheduling tasks on a job server, for instance). This customizable nature may also be beneficial in research contexts: rather than implementing multiple scheduling algorithms to simulate their behaviors, extensions to PVT could effectively simulate the behaviors of those other algorithms.

References

[Ber99] Dimitri P. Bertsekas. *Nonlinear Programming*. Second. Belmont, Massachusetts: Athena Scientific, 1999.

[Bow59] Edward H. Bowman. "The Schedule-Sequencing Problem". In: *Operations Research* 7.5 (1959), pp. 621–624.

[FG] David Ferry and Chris Gill. "Scheduling of Non-Real-Time Tasks in Linux". URL: `http://www.cse.wustl.edu/~ferryd/courses/cse522/`.