Time-Function Scheduling: A General Approach To Controllable Resource Management

Liana L. Fong and Mark S. Squillante
IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598
{fong,mss}@watson.ibm.com

March 1995
Minor Revisions: August 1995

Abstract

In this paper we present a general approach to resource management called time-function scheduling (TFS). This scheduling method provides effective and flexible control over resource allocation that can be used to achieve different performance requirements for a large and diverse set of scheduling objectives. We demonstrate these capabilities by developing a prototype implementation of a relatively simple form of TFS and experimentally comparing it against other approaches that have appeared in the research literature. Our results show that TFS supports a level of resource allocation control that is not possible under the methods employed in many computing environments. We also show that TFS can achieve the same relative execution rates of the recently proposed lottery scheduling mechanism while considerably reducing the variability in response. Furthermore, TFS provides very fast and stable convergence to the desired resource allocation goals. These results are obtained with a simple, straightforward implementation of TFS that exhibits overheads comparable to lottery scheduling.

IBM Research Report RC 20155

1 Introduction

One of the fundamental functions of an operating system is the scheduling of limited computer resources among system and user tasks. The general goal of resource scheduling is to order the execution of such tasks to meet a set of system-level and user-level objectives.

Many diverse scheduling objectives have been proposed and used in various operating system designs. An important goal has always been to achieve different measures of performance for different types of jobs,* both in relative and absolute terms [8, 12, 34, 30, 24, 47, 35, 42, 36, 9, 46, 10, 23, 69, 14, 62, 22, 55]. Examples include minimizing the response time of interactive jobs, maximizing system throughput and processor utilization, and favoring I/O-bound jobs over compute-bound jobs. Other important scheduling objectives are to provide predictable (i.e., low variance) performance within and across job classes [8, 34, 24, 35, 42, 36, 9, 46, 10, 23, 26, 14, 55], to proportionally allocate resources (equally or differentially) among jobs and classes [8, 34, 30, 24, 47, 35, 36, 46, 23, 69, 26, 14, 11, 62, 22, 67], and to avoid indefinite postponement [8, 34, 35, 36, 10, 69, 26, 14, 22, 55]. Moreover, the need to satisfy real-time constraints continues to increase [39, 10, 25, 63, 38, 5, 6, 50, 54, 55], and maintaining a relatively low scheduler overhead has always been a key requirement [8, 34, 24, 10, 26, 14, 62, 22, 55].

Many of these objectives are in conflict, thus making resource scheduling a complex operating system design problem. As a simple example, note that executing jobs in a shortest-service-time-first order minimizes mean response time, but at the expense of an increase in response time variance. Given this complexity, most scheduling algorithms have been designed to meet only a subset of the above goals. While these policies are typically effective within their intended domains, they often have limited success, or do not perform well at all, in system environments with different requirements. This is a particularly important operating system design problem given the increasing use of computer systems to process various mixtures of jobs with diverse service needs, such as environments that integrate real-time and non-real-time applications [49], real-time systems with various timing constraints [50], and environments with different types of communication characteristics [51].

In addition to supporting multiple diverse scheduling objectives, a principal design challenge is to provide effective control over resource allocation to achieve the desired performance requirements for these simultaneous objectives. The scheduling of jobs with various timing constraints requires dynamic, predictable and flexible control mechanisms [50]. Scientific and engineering computing environments need the ability to ensure that system resources are allocated to meet installation-specific service rate goals [22]. Large time-sharing systems also require control over resource allocation decisions to achieve combinations of diverse performance objectives [35, 36, 46, 69]; e.g., providing a larger share of the resources to transaction processing jobs while not starving program development jobs under heavy loads. Unfortunately, most scheduling strategies do not support flexible parameterization of their algorithms to satisfy the varying performance requirements of different environments.

Over the past several years we have been exploring the use of time-based functions to provide effective and flexible control over resource allocation to achieve the above scheduling objectives and performance requirements. This paper describes a scheduling method, called *time-function scheduling* (TFS), that represents a generalization of our previous research activities. TFS schedules resources according to general functions of time that can be defined to achieve different performance

*Throughout this paper we use the term *job* quite generally, referring to any schedulable unit of work.

requirements for a large and diverse set of scheduling objectives. In the present paper, our primary focus is on computing resources and we simultaneously consider several different scheduling goals: relative per-class throughputs, low waiting time variance, avoiding indefinite postponement and support for real-time scheduling. It is important to note, however, that TFS can be applied more generally, both with respect to the set of scheduling objectives and the types of resources controlled. This is achieved by exploiting the theoretical foundation of our approach (see Appendix A), which is in stark contrast to the ad hoc and poorly understood methods employed in most existing resource allocation technologies.

We have developed a prototype implementation of TFS for a UNIX-based environment, with which we show the benefits of TFS under workloads that have very different characteristics. Specifically, we consider workloads of compute-bound jobs, as found in many scientific and engineering environments, and workloads of multi-user transaction processing in a commercial database environment. We use this experimental platform to compare TFS with *decay-usage scheduling*, a method employed in many computing environments [1, 37, 4], and *lottery scheduling*, an interesting approach recently proposed by Waldspurger and Weihl to control the proportional allocation of resources [67]. Our results demonstrate the ability of TFS to effectively and efficiently control the relative throughputs of different job classes while providing minimal variability in response. We also show that TFS supports a level of resource allocation control that is not possible with most resource scheduling policies. In the few instances where some control is provided, the ability to achieve various performance requirements can be quite difficult (if not impossible), the range of control is often very limited, and the control mechanisms are poorly understood in general [22]. Our numerous experiments with decay-usage scheduling confirm these observations.

Lottery scheduling is one of the few methods appearing in the research literature with objectives somewhat similar to TFS in that lottery supports control over the relative execution rates of computations. Our experimental results demonstrate that TFS achieves the same proportional execution rate goals while significantly reducing the waiting time variance, in some cases by as much as several orders of magnitude. We further demonstrate that TFS provides very fast and stable convergence to these resource allocation goals and low waiting time variances, due to the more deterministic control properties of TFS. It is important to note, however, that minimizing the variance of waiting times was not a design goal of lottery scheduling. In contrast, our work has attempted to include this aspect of the general scheduling problem, which is particularly important in commercial transaction processing environments and which has not been considered in most scheduling research studies. As previously observed [55], there has been very little work on scheduling to minimize variance while also satisfying other performance requirements. This has been one of the many objectives in our design of TFS.

The remainder of this paper is organized as follows. Section 2 presents time-function scheduling. We then describe our prototype implementation of TFS, as well as the prototype versions of decay-usage and lottery scheduling used in our experiments. Section 4 presents some of the results of our experiments, and in Section 5 we discuss some related work. Our concluding remarks are presented in Section 6, together with planned future work. Appendix A briefly describes some of the theoretical foundations of our TFS design.

2 Principles of Time-Function Scheduling

Time-Function scheduling (TFS) is based on the concept of dynamic priorities that change according to per-class functions of the time each job has spent waiting for the resource of interest. The effective and flexible control over resource allocation is supported by these general *time-functions*, which can encapsulate the performance requirements for various scheduling goals. A desired set of scheduling objectives is achieved by employing time-functions with properties that match these requirements. Various mechanisms can be used to dynamically adjust the per-class time-functions in response to changes in the system workload so that the desired objectives are continuously satisfied, as we briefly discuss in Appendix A. Similarly, diverse sets of time-functions can be employed during different periods of system operation to achieve the desired scheduling objectives over the corresponding time intervals.

It can be easily shown that the benefits given to one class of jobs are necessarily realized at the expense of other job classes [62]. We therefore primarily focus on relative performance requirements in this paper, and we specifically consider relative throughput goals among (most of) the job classes. We note, however, that TFS can be used to satisfy absolute performance requirements, if they are feasible, and to achieve objectives based on other performance measures such as mean waiting times, mean response times and/or proportional resource allocation (see Appendix A).

Our objective in this paper is to provide differential treatment among jobs of different classes while providing very predictable performance among the jobs within a class. This implies that jobs should be scheduled in a first-come first-served (FCFS) manner within each class, since the FCFS discipline generally minimizes the variance of waiting times [27]. The use of monotonically increasing time-functions ensures such an ordering, and also avoids indefinite postponement (provided the system is not saturated). While our general approach of using time-functions naturally accommodates the time varying value functions of time-driven real-time scheduling [25, 63, 5, 6], in the present paper we focus on support for the more common rate-monotonic scheduling algorithms [39, 38, 49, 50, 54].

2.1 Job Classes

All runnable jobs with similar characteristics and the same scheduling objectives are assigned to a particular job class, i.e., jobs that should receive similar treatment with respect to resource allocation are placed in the same class. A (unique) time-function is associated with each job class, thus yielding similar performance characteristics for the jobs that comprise the class. One can easily show that grouping such jobs in this manner while maintaining an FCFS ordering within the class will yield comparable first moment performance measures yet significantly better second moment performance measures than applying the same time-function policy to the jobs individually. This is also supported by our experimental results. The use of (a relatively small number of) job classes is fundamental to most large-scale commercial systems, and the requirement to control performance on a job class basis has been previously observed [35, 42, 36, 9, 46, 23, 69, 26]. Moreover, TFS provides a level of control and flexibility to meet specific performance objectives that eliminates the need for a large number of classes. This particular notion of job class is thus an important aspect of TFS.

Each job class is also assigned a run-queue. A *round robin* discipline is employed at all run-queues. That is, the processor is allocated to a job for a time quantum and the system preempts each job that exceeds its quantum, placing it at the end of the run-queue associated with its class to await another quantum of service. The time-function of each job is "reset" upon its return to the run-queue (either due to quantum expiration, preemption, a state change, etc), although our general approach is not limited to this assumption. The set of all run-queues is referred to as the dispatch queue.

We divide the job classes into three different groups: $\mathcal{H}$, $\mathcal{T}$ and $\mathcal{L}$. Jobs in $\mathcal{H}$ have absolute priority over those in $\mathcal{T}$, independent of time-function values. Thus, a job from $\mathcal{T}$ can be executed only if there are no runnable jobs from $\mathcal{H}$ present in the system. Similarly, jobs in $\mathcal{T}$ have absolute priority over those in $\mathcal{L}$. Within each group, scheduling priorities are defined by the time-functions associated with the job classes comprising the group. When selecting the next job to execute, the system schedules the job with the largest time-function value (at that time) from the first non-empty group of classes. We will number classes within each group starting at 1, under the convention that a smaller class number implies a higher priority class. Modifying the presented algorithms to reflect different priority conventions is straightforward.

More generally, TFS can be used with a wide variety of dispatch queueing structures, and a different quantum size can be employed for each job class. TFS can also be combined with any number of other classical scheduling techniques [8, 34, 24, 47, 7, 10, 62, 55]. In general, the number of groups of classes can be as small as one and as large as desired.

2.2 Time-Functions

The time-function $F_k(t)$ defines the priority of each job in class k, where t represents the amount of time the job has spent waiting since it was last placed in the run-queue associated with class k. In addition to the desired scheduling objectives and performance requirements, the resource usage behavior of the jobs should be factored into the definition of each time-function, thus also including resource usage in scheduling decisions.

The group of classes $\mathcal{H}$ is reserved for system tasks (e.g., the high and low kernel-level priorities in UNIX [1]), jobs with real-time constraints scheduled by rate-monotonic algorithms, and any other jobs that require absolute priority over those in $\mathcal{T}$.[†] These jobs are given the highest priority in the system to meet various timeliness constraints. A strict fixed priority scheme is employed among the jobs in $\mathcal{H}$. This is the approach often used to schedule system tasks [1, 37] and to schedule real-time tasks with rate-monotonic algorithms [49, 50, 54]. A fixed priority ordering can be achieved with constant time-functions, i.e., $F_k(t) = N_{\mathcal{H}} - k$ where $k \in \mathcal{H}$ and $N_{\mathcal{H}}$ denotes the number of classes in $\mathcal{H}$. The round robin management of each run-queue ensures the FCFS ordering of jobs even under constant time-functions. A similar scheduling strategy is employed for the group of classes $\mathcal{L}$, which is reserved for background tasks that should be executed only when no other jobs are runnable.

We use the group of classes $\mathcal{T}$ to control the relative throughputs among the jobs in these classes. A simple form of time-functions that provides this level of control includes those in which the priority of each job increases linearly with the amount of time it has spent waiting since it was last placed in the run-queue. We therefore use the time-functions $F_k(t) = \mathrm{s}_k t + \mathrm{o}_k$, where $k \in \mathcal{T}$, s_k is the slope and o_k is a constant offset. All of the slopes are positive, and our priority order convention implies that $\mathrm{s}_1 \geq \ldots \geq \mathrm{s}_{N_{\mathcal{T}}}$ ($N_{\mathcal{T}}$ denotes the number of classes in $\mathcal{T}$). Throughout this paper we will use $\mathrm{o}_k = k - 1$. When a context switch occurs and there are no runnable jobs in $\mathcal{H}$,

[†]A further partitioning within $\mathcal{H}$ among system, real-time and other tasks will be desirable in many systems. This can be accomplished in a manner similar to the division between high and low kernel-level priorities in UNIX [1].

the scheduling policy dispatches the job in $\mathcal{T}$ with the largest time-function value at that instant; ties are broken in favor of the higher priority class. The FCFS order on each run-queue ensures that the job with the longest waiting time is at the head of the queue. Since all of the time-functions in $\mathcal{T}$ are monotonically increasing, the scheduler need only consider the job at the head of each (non-empty) run-queue.

More generally, the time-function value of a job need not be *reset* upon each return to the dispatch queue. The resetting of a job's time-function (i.e., setting $t = 0$ in $F_k(t)$) can occur at any number of well-defined events. In addition to other job characteristics, this behavior should be factored into the definition of each time-function. The favoring of I/O-bound jobs can also be included in the definition/implementation of each time-function, thus improving system throughput and minimizing the possibility of the I/O device being a bottleneck [8].

2.3 Priority Calculations

Established methods can be used to find the highest priority runnable job in $\mathcal{H}$, such as maintaining a bit-mask to identify non-empty run-queues [37] or maintaining a hint to identify the probable location of the highest priority job [4]. A null bit-mask, or a null hint pointer, quickly indicates that there are no runnable jobs in $\mathcal{H}$, and the scheduler proceeds to look for runnable jobs in $\mathcal{T}$. Similar methods can be used to find the highest priority runnable job in $\mathcal{L}$, when there are no runnable jobs in $\mathcal{H}$ or $\mathcal{T}$.

Several approaches can be used to efficiently manage the run-queues in $\mathcal{T}$. When the number of job classes is small, a simple and relatively efficient approach consists of computing and comparing the time-function values of the jobs at the head of each run-queue, and scheduling the job with the largest value. Noting which run-queues are currently dominated by the other queues on each pass can be used as a minor optimization to reduce the search time for the next dispatch, as long as the overhead for doing so does not outweigh its potential benefits. The worst-case time complexity for this approach is $\mathcal{O}(N_{\mathcal{T}})$, although this is easily reduced to be on the order of the number of non-empty run-queues.

Another approach consists of discretizing the continuous time-functions of each class such that time-function values are only updated every Δ time units. Obviously, the value of Δ should be sufficiently larger than the time between dispatches. Combining this with the above approach decreases the time required for each pass through the run-queues in $\mathcal{T}$, since only one time-function would need to be computed on each dispatch; namely, the time-function value for the job that immediately follows the one selected on its run-queue (if any).

When the number of job classes is sufficiently large, this discretized version of time-functions can also be used to further reduce the time required for dispatching at the expense of slightly increasing the priority update operation. Specifically, the worst-case time complexity to dispatch the highest priority job in $\mathcal{T}$ can be reduced to $\mathcal{O}(\log N_{\mathcal{T}})$ by organizing the heads of the run-queues in some type of priority queue. Updating the time-function values of the jobs at the head of each run-queue, which is performed much less frequently than the dispatch operation (i.e., every Δ time units), requires the rebuilding of the priority queue. This has a worst-case time complexity of $\mathcal{O}(N_{\mathcal{T}} \log N_{\mathcal{T}})$. Note that the update operation does not necessarily reorder the priority queue completely, and thus one might be able to reduce this time complexity by taking these factors into account (the benefit of which depends upon the time-function parameters used in the system).

Given our notion and use of job classes (see Section 2.1), the number of job classes $N_{\mathcal{T}}$ are relatively small and (in particular) significantly less than the number of jobs, as has been our experience with commercial computing environments. Since the worst-case scheduling overhead under TFS is a function of the number of classes, as opposed to the number of jobs, the scheduling overhead will not increase with system load which is especially important in heavy load situations. This is in stark contrast to the approach used in UNIX, and its derivatives, where the decay function has a scheduling overhead on the order of the number of ready jobs in the system [1, 37]. A key point of our approach is that the properties of the employed time-functions should be exploited to obtain the most efficient implementation. Moreover, the value of Δ can be adjusted to achieved the desired level of system performance [45].

2.4 Priority Inversion

Priority inversion is the situation where a high priority job is blocked on a synchronization primitive that is held by a low priority job. This can be a serious problem that limits the ability of the scheduler to achieve certain performance requirements, since the low priority job can delay the high priority job for an indefinite period of time. TFS solves this problem by scheduling each job that holds a synchronizer according to the time-function associated with the highest priority job waiting for the synchronizer. The job returns to its original time-function once it releases the synchronizer, possibly with some penalty due to its favored treatment while holding the synchronizer. Similar techniques can be employed to thwart any countermeasures taken by users to inappropriately obtain preferential treatment.

Our approach is analogous to the priority inheritance protocols proposed in [53]. It is somewhat different, however, to the approach of ticket transfers in lottery scheduling [67], where the job holding a synchronizer receives resource rights equal to the sum of the rights owned by jobs blocked on the synchronizer. While this approach may have advantages with respect to proportional allocation, it has the potential disadvantage that a synchronizer on which a large number of low priority jobs are blocked can cause the low priority holder of the synchronizer to unnecessarily block the execution of independent higher priority jobs.

3 Prototype Implementation

We have implemented a prototype of the basic TFS system described in Section 2 on an IBM RS/6000 Model 530 running AIX version 3.2.5. In addition to the standard UNIX decay-usage scheduling algorithm [1, 37], a prototype implementation of lottery scheduling [67] is included in this experimental platform for comparison in the next section.

To support both TFS and lottery with a minimal number of changes and to provide a fair comparison among the policies considered, we modified some of the scheduling and dispatching functions provided on this platform. This included turning off the usage recalculation at the end of each 10 millisecond time quantum and the decay recalculation of all jobs every second. The initial priority of each job serves as its class assignment, and we used the existing code base to handle jobs in $\mathcal{H}$ and $\mathcal{L}$. Since the usage and decay recalculations are eliminated, this yields a fixed-priority scheduling scheme for these groups of job classes.

In the remainder of this section, we describe the additional changes made to handle jobs in $\mathcal{T}$ under TFS, lottery and decay-usage.

3.1 Time-Function Scheduling

The time-function for each job class in $\mathcal{T}$ is defined by a slope and a constant offset (see Section 2). The time when each job was most recently placed on the run-queue is recorded in its process control block, denoted by *arrival_time* below, where time is recorded in units of 10 milliseconds (i.e., $\Delta = 10ms$). To select for execution the job in $\mathcal{T}$ with the highest time-function value, the dispatcher uses select_job_in_T() as follows.

```
select_job_in_T()
{
   high_tf = -1;  high_job = NULL;
   for ( each non-empty run_queue in T ) {
      job_tf = calc_job_tf(run_queue->head);
      if ( high_tf < job_tf ) {
         high_tf = job_tf;
         high_job = run_queue->head;
      }
   }
   return( high_job );
}

calc_job_tf(j)
{
   return( (current_time - j->arrival_time) * class_slope + class_offset );
}
```

3.2 Lottery Scheduling

Under lottery scheduling, resource rights are represented by lottery tickets and the resource is granted to the job with the "winning" ticket. As previously noted in Section 2.1 and confirmed by our experiments, a policy that schedules jobs with similar performance characteristics/objectives as a single class under an FCFS ordering within the class yields the same average job performance as applying the corresponding scheduling policy to these jobs independently, but with significant reductions in variance. Our prototype therefore implements a slightly modified version of lottery scheduling in which tickets are assigned to job classes instead of individual jobs. The resource is then granted to the job at the head of the run-queue associated with the class holding the winning ticket. An FCFS ordering is maintained at each run-queue. Of course, our approach is equivalent to the original lottery definition in [67] when there is one job per class.

Each job class in $\mathcal{T}$ is scheduled according to this lottery algorithm. To select the next job in $\mathcal{T}$ for execution, the dispatcher uses select_job_in_T() as follows, where the per-class run-queues are organized in decreasing ticket count order.

```
select_job_in_T()
{
   ticket_count = 0;  high_job = NULL;
   winning_ticket = random_number_generator(total_tickets);
   for ( each non-empty run_queue in T ) {
```

```
        ticket_count += class_tickets;
        if ( winning_ticket < ticket_count ) {
            high_job = run_queue->head;
            break;
        }
    }
    return( high_job );
}
```

The function random_number_generator() returns a number between 0 and the number of tickets in the current lottery. Our implementation of this function is based on the random number generator described in [48].

3.3 Decay-Usage Scheduling

The standard UNIX decay-usage policy is used to schedule jobs in $\mathcal{T}$. Each job waits on a run-queue associated with its dispatch_pri value, where a smaller dispatch_pri value implies higher priority. The dispatcher simply searches for the first non-empty run-queue in $\mathcal{T}$, and selects for execution the job at the head of this queue.

The job running at the end of each 10 millisecond time quantum has its dispatch_pri value adjusted according to the following usage calculation.

```
cpu_usage++;
dispatch_pri = usage_rate * cpu_usage + base_pri;
```

In addition, all runnable jobs have their dispatch_pri value adjusted every second according to the following decay calculation.

```
cpu_usage /= decay_rate;
dispatch_pri = usage_rate * cpu_usage + base_pri;
```

Note that both calculations cause the jobs involved to be moved to the run-queue associated with their updated dispatch_pri value, thus having important performance implications with respect to cache effects. The base_pri value can be adjusted somewhat via so-called nice values.

4 Experimental Evaluation

In this section we present some of the experimental results obtained from our prototype implementations of Section 3 to evaluate the benefits of TFS. Numerous experiments have been performed on this platform under two workloads with very different characteristics. The first is based on the Dhrystone benchmark [68] and it is used to represent compute-bound workloads, as found in many scientific and engineering environments. Our second workload is the TPC-C transaction processing benchmark [64] which emulates interactive users accessing a database in commercial environments. We compare the control over processor allocation provided by TFS with that obtained under decay-usage and lottery scheduling. The performance measures used for these comparisons include throughput, mean waiting time, waiting time variance and scheduling overhead. All of the scheduling policies considered prevent indefinite postponement.

Our first set of experimental results considers the abilities of each of the scheduling policies to achieve specific relative throughput requirements, under a compute-bound workload represented by two classes of Dhrystone execution that spans the entire measurement interval. In Figure 1 we plot the ratio of class 1 throughput to class 2 throughput as a function of the desired throughput ratio objectives, ranging from 2 to 10, over a measurement interval of 500 seconds. We observe that both TFS and lottery scheduling accurately achieve the desired performance goals. TFS provides somewhat more accurate relative throughput measures, but the differences are quite small. On the other hand, we find that the desired control over resource allocation is not possible under decay-usage scheduling. The measurement plotted for decay-usage was the closest we could come to achieving a 2:1 throughput ratio by adjusting nice values. Somewhat better results could have been obtained by adjusting the usage and decay rates of the scheduling algorithm (see Section 3.3), but even this would have provided limited control [22]. The inability of decay-usage to control processor allocation decisions to achieve specific performance requirements was consistently observed across all of our experiments, as is illustrated in each of the subsequent figures. We simply note this point here and do not continue to repeat it for each set of results that follow.

We next consider the per-class waiting times measured over this same set of experiments. In Figure 2 we plot the mean waiting times for both job classes under the different scheduling policies as a function of the relative throughput objective. Figure 3 provides the corresponding per-class variance of waiting times. We observe that TFS and lottery provide similar mean waiting time performance. Moreover, the average per-class response times (i.e., waiting time plus service time) under both policies are proportional to the throughput ratio objectives, because the mean service times for each class are the same (i.e., one time quantum). There are, however, significant differences between TFS and lottery scheduling with respect to the variance of waiting times. For the specific case of a 2:1 execution rate goal, the waiting time variance under lottery for class 1 and class 2 is 2.6 and 65.4 times larger, respectively, than that provided by TFS. As the throughput ratio objective increases, the waiting time variance for class 1 under lottery scheduling tends toward that of TFS, while the variance for class 2 becomes considerably worse. The 10:1 allocation goal under lottery yields a waiting time variance that is 1.13 times larger for class 1 and 743.83 times larger for class 2, in comparison with TFS. These large waiting time variances are caused by the randomness inherent in the lottery scheduling policy. To help explain this, we note that there is a relatively large likelihood of one class having a series of consecutive "wins" exceeding its allocation objective, which is then offset by a series of consecutive "wins" for the other class to achieve the desired proportional execution rate on average.[‡] As a specific example, one can easily show that class 1 will receive more than 2 consecutive processor allocations 30% of the time under lottery scheduling with a 2:1 throughput goal [16]. This causes the waiting time variance for these classes to be larger than that provided by TFS, which stays much closer to the allocation objectives in its processor scheduling decisions. TFS provides the same level of control over the relative execution rates in a more deterministic manner, thus maintaining a very predictable (i.e., low variance) response.

Our next set of results considers how quickly the scheduling policies converge toward the various performance objectives, under the above compute-bound workload. In Figures 4 and 5 we plot the measured throughput ratios and per-class waiting time variance, respectively, as a function of the

[‡]Note that this problem exists even if a "perfect" random number generator is used, and it is not a result of the particular random number generator used in our prototype. We ran experiments with different seed values, but this had a relatively small effect on our results. The results presented in this section are the ones that exhibited the smallest waiting time variance for lottery scheduling.

measurement interval length for a 2:1 throughput goal. More detailed plots of both measures for 10 to 100 second intervals are provided in Figures 6 and 7, respectively. Results corresponding to Figures 4 and 5 for an 8:1 throughput ratio objective are plotted in Figures 8 and 9. We observe that TFS converges very rapidly to the desired throughput ratios, and also provides a less variable achievement of these goals. The same holds true for the variance of per-class waiting times, where TFS again provides more stable and less variable results. We further observe that the time required under lottery scheduling to converge toward the throughput ratio objectives, as well as the more stable waiting time variances, increases with the throughput ratio goal; e.g., compare Figure 8 with 4, and Figure 9 with 5. This effect was also observed in [67].

We note that similar trends were observed for workloads consisting of more than 2 job classes, as well as those consisting of multiple jobs per class. Figures 10 and 11 plot the measured throughput ratios and per-class waiting time variance, respectively, as a function of the measurement interval length for three job classes and a 3:2:1 throughput objective. Once again, TFS achieves the desired throughput and waiting time variance measures in a very stable manner and rapidly converges to these values. As previously noted (see Section 2.1), conducting lotteries on a per class basis, and using an FCFS ordering within each class, reduces the per-class waiting time variance over that obtained by conducting lotteries on a per job basis, while achieving the same proportional execution rate goals. This was demonstrated in a number of experiments, although we omit these results here in the interest of space.

Our next set of results compares the overhead under TFS with that of lottery scheduling. In Figure 12 we plot the total number of dispatches under both policies as a function of the measurement interval length. These results demonstrate that our simple, straightforward implementation of TFS exhibits overheads comparable to lottery scheduling. We note that this result was consistently observed across all of our experiments.

In addition to the experimental results presented thus far for compute-bound workloads based on the Dhrystone benchmark, we conducted a number of similar experiments for a more interactive multi-user workload. This workload is based on the TPC-C transaction processing benchmark which emulates interactive users accessing a database in commercial environments. The experiments consisted of two groups of application users each submitting transaction requests to a separate database. Upon receipt of user requests, the database agents performed the necessary transactions to the corresponding database. Each group contains an equal number of users and agents, and each group is placed in a separate job class. For each set of experiments, multiple runs were performed and each run yielded similar measures. These experimental results confirm our observations made above based on the compute-bound workload. We simply present here some of the results obtained with the TPC-C benchmark.

In Tables 1 and 2 we summarize the measurement results for two independent runs with a 2:1 allocation ratio. We observe that TFS and lottery both fall somewhat short of achieving the desired transaction completion ratios, although TFS comes closer to achieving these objectives than lottery scheduling. This can be explained by noting that our experiments were performed under moderate TPC-C utilizations, where an independent set of runs with a fixed priority scheduler could only achieve a transaction completion ratio near 3 (see Table 2). Thus, TFS and lottery would both need larger input parameters (i.e., larger class 1 slopes under TFS and larger class 1 ticket counts under lottery) to achieve a 2:1 allocation ratio under the moderate TPC-C utilization considered. This can be achieved dynamically under TFS by employing the methods briefly discussed in Appendix A. Based on these results, we also note that increasing the utilization would bring the results closer to the desired transaction completion ratios of both TFS and lottery under the original per-class slopes and ticket counts, respectively (see equation (2) in Appendix A).

5 Related Work

Our time-function scheduling work arises out of previous research on *delay cost scheduling* [18, 19, 43, 45, 56, 44, 58, 59, 57, 60]. Delay cost scheduling, due to Franaszek and Nelson [18, 19], is based on the notion that the system is charged a cost for the delay in completing each job and that the goal of the scheduler is to minimize the total cost incurred by the system. The cost incurred for a given job of class k over a specific time period is calculated by integrating the cost function associated with class k over the given time interval, and the total cost incurred by the system is simply the sum of the costs incurred for each job. This basic notion of scheduling based on a cost of delay measure has also been proposed in [21].

A number of studies have focused more narrowly on providing proportional sharing of the processor among jobs. Lottery scheduling is an important example, as are the various instances of fair share schedulers [23, 26, 14] under which the processor (and other resources) is shared fairly among jobs and/or users. An interesting alternative to lottery scheduling is proposed by Stoica and Abdel-Wahab [61] where a deterministic scheme is used to allocate time-slices to each job in proportion to its shares. Another interesting proportional allocation scheme is developed by Dewey, Glynn, Hough and Rao [11], which takes an approach somewhat analogous to TFS but with respect to linear functions of processor usage.

Another class of related and interesting scheduling work is based on notions from the field of economics [13, 17, 40, 41, 66]. This approach typically consists of employing market-based mechanisms to allocate resources among jobs, where jobs use an abstraction of monetary funds to compete for resources in an attempt to individually maximize their profit. These scheduling schemes are somewhat similar to TFS in the sense that both employ various forms of cost or price mechanisms. Our TFS approach, however, eliminates the stability and overhead problems typically associated with these market-based schemes.

TFS provides more general and flexible control over resource allocation than the approaches described above, thus making it possible to effectively achieve different performance requirements for a large and diverse set of scheduling objectives.

6 Conclusions and Future Work

In this paper we have described the design, implementation and performance of a general approach to resource management called time-function scheduling (TFS). This scheduling method provides effective and flexible control over resource allocation that can be used to achieve different performance requirements for a large and diverse set of scheduling objectives. We demonstrated these capabilities by experimentally comparing a relatively simple form of TFS with decay-usage and lottery scheduling.

Our results clearly demonstrate the ability of TFS to achieve desired performance requirements across diverse system workloads. TFS is shown to provide an effective, flexible and efficient level of control over resource allocation that is not possible with most scheduling policies, including the common decay-usage approach. Our results demonstrate the ability to achieve a wide range of relative throughputs for multiple job classes while providing predictable (i.e., low variance) response. In particular, we show that TFS can achieve the same relative execution rates of lottery scheduling while significantly reducing the variability in response. Moreover, TFS provides very fast and stable convergence to these resource allocation goals. These performance benefits are obtained with a simple, straightforward implementation of TFS that exhibits overheads comparable to lottery scheduling.

We are currently extending this research in several directions. We believe that TFS can be used to satisfy relative response time measures in general, as well as more complex instances of resource allocation control, by exploiting known theoretical properties in the construction of time-functions. Although not considered in this paper, our general approach can be used with functions of measures other than waiting time, which is the subject of current and future work. The TFS algorithms can also be augmented with adaptive feedback mechanisms to dynamically adjust the time-function employed to achieve more fine-tuned control. Appendix A discusses these issues in more detail. We believe that the TFS methodology is quite general, and that TFS may prove to be an effective approach to manage various system resources in addition to processors, such as I/O, memory and communication bandwidth. All of these features are currently being incorporated into our prototype implementation of TFS for further experimentation.

We are also extending our experimental platform to include other application workloads. In particular, we are currently pursuing experiments with real-time applications that are scheduled by the rate-monotonic support provided in our TFS prototype. Incorporating time-driven real-time scheduling [25, 63, 5, 6] in our TFS prototype together with non-real-time jobs is the subject of future work.

Finally, we believe that combining the benefits of TFS and lottery scheduling may provide a very effective and flexible control mechanism over resource allocation decisions. As a specific example, note that lottery scheduling provides a nice mechanism for ordering the execution of tasks that are based on probabilistic algorithms, e.g., Monte-Carlo methods [67]. By capturing the dynamic properties of each task in its ticket allotments, lottery scheduling can naturally provide an appropriate ordering among these tasks, as shown in [67]. One possible approach of combining both scheduling mechanisms would be to use TFS to control resource allocation across classes, while using a lottery scheme to determine which task within the class (or which thread within an application) gets dispatched. We plan to use our experimental platform to further investigate the benefits of combining TFS and lottery scheduling in this manner (and others) as part of our future work.

Acknowledgements

We thank Brent Hailpern, Roger Hough and David Yao for fruitful discussions regarding this research and for helpful comments on earlier drafts of the paper. We also thank Maria Butrico and Bill Kish for helping us with various aspects of our system prototype environment, Nayeem Islam for providing us with the Dhrystone benchmark, and Ajay Mohindra and Lilian Romero for helping us with the TPC-C benchmark.

A General Time-Function Control of Performance

Our design of time-function scheduling has a strong theoretical foundation that supports general and robust control over resource allocation. This includes a number of control formulas that can be encapsulated in time-functions to achieve different performance requirements for a large and diverse set of scheduling objectives. Additional levels of control over resource allocation can be obtained via adaptive mechanisms that exploit theoretical results to properly adjust these time-functions in response to changes in the system load, the workload mix, the scheduling objectives, etc. In this appendix we briefly discuss the theoretical foundation of our TFS design and its use to control various relative and absolute performance measures beyond the throughput ratios considered in Section 2.

A number of theoretical studies have addressed the problem of scheduling to satisfy mean waiting time (or response time) objectives in rather general settings [21, 42, 9, 18, 51, 20, 52, 19, 3, 2]. These results together with the considerable analysis of linear time-dependent priority schemes [28, 29, 31, 32, 33, 65, 15, 43] can be used to construct time-functions that achieve a diverse set of performance requirements. We simply point out a few of the more interesting results, and refer the interested reader to these papers for more details.

The average waiting time of each class in a K-class system under linear time-dependent priorities can be explicitly written for general service time assumptions. Specifically, Kleinrock [28] has shown that the expected waiting time for class k jobs in this system is given by

$$\overline{W}_k \;=\; \frac{W_0/(1-\rho) \;-\; \sum_{i=k+1}^{K} \rho_i \overline{W}_i (1 - s_i/s_k)}{1 \;-\; \sum_{i=1}^{k-1} \rho_i (1 - s_k/s_i)}, \quad 1 \le k \le K. \tag{1}$$

In addition, Nelson [43] derived a heavy traffic relation among the per-class waiting times, showing that the expected waiting times satisfy

$$\lim_{\rho \to 1} \frac{\overline{W}_k}{\overline{W}_j} \;=\; \frac{s_j}{s_k}, \quad 1 \le k \le K,\; 1 \le j \le K. \tag{2}$$

Note that in both cases the mean waiting times depend only upon the ratios of the slopes as opposed to their actual values.

These expressions can then be inverted, using an invertible mapping [15, 44], to obtain the slopes that satisfy a set of waiting time objectives. That is, given a characterization of the system workload (including the resource usage behavior of the jobs) and per-class average waiting time goals, these formulas can be used to construct and adapt the per-class time-functions employed to achieve the desired performance objectives. Relative (steady state) waiting time objectives among the various classes are directly satisfied with this approach. When absolute waiting times are specified, this approach will achieve the performance requirements if they are feasible and will otherwise come as close to these absolute objectives as possible under the circumstances. Adaptive feedback mechanisms can also be used together with these control formulas to adjust time-function parameters as needed [69, 20, 3, 2].

Our combined use of fixed priorities and linear time-functions in Section 2 has also been studied and corresponding sets of results are available [29, 31]. Moreover, similar results have been obtained for (symmetric) multiprocessor systems with a global set of run-queues that are scheduled according to linear time-dependent priority functions [15]. These general results can be used in the above manner to provide the desired control over resource allocation in the corresponding system environments.

References

[1] M. J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[2] P. Bhattacharya, L. Georgiadis, and P. Tsoucas. Problems of adaptive optimization in multi-class M/GI/1 queues with feedback. *Mathematics of Operations Research*, May 1995.

[3] P. Bhattacharya, L. Georgiadis, P. Tsoucas, and I. Viniotis. Adaptive lexicographic optimization in multi-class M/GI/1 queues. *Mathematics of Operations Research*, 18(3):705–740, 1993.

[4] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, pages 35–43, May 1990.

[5] K. Chen. A study on the timeliness property in real-time systems. *The Journal of Real-Time Systems*, 3:247–273, 1991.

[6] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the Alpha real-time distributed kernel. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 127–146, April 1992.

[7] E. G. Coffman, Jr. and P. J. Denning. *Operating Systems Theory*. Prentice Hall, 1973.

[8] E. G. Coffman, Jr. and L. Kleinrock. Computer scheduling methods and their countermeasures. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 32, pages 11–21, April 1968.

[9] E. G. Coffman, Jr. and I. H. Mitrani. A characterization of waiting-time performance achievable by single-server queues. *Operations Research*, 28(3):810–821, 1980.

[10] H. M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, 1984.

[11] P. H. Dewey, W. J. Glynn, R. E. Hough, and M. R. Rao. Method and apparatus for dynamic work reassignment among asymmetric, coupled processors. U. S. Patent 5,301,324. Filed November 1992., April 1994.

[12] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[13] K. E. Drexler and M. S. Miller. Incentive engineering for computational resource management. In B. A. Huberman, editor, *The Ecology of Computation*, 1988.

[14] R. B. Essick. An event-based fair share scheduler. In *Proceedings of the Winter 1990 USENIX Conference*, pages 147–161, January 1990.

[15] A. Federgruen and H. Groenevelt. M/G/c queueing systems with multiple customer classes: Characterization and control of achievable performance under nonpreemptive priority rules. *Management Science*, 34(9):1121–1138, September 1988.

[16] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume I. John Wiley and Sons, Third edition, 1968.

[17] D. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic algorithms for load balancing in distributed computer systems. In *Proceedings of the International Conference on Distributed Computer Systems*, June 1988.

[18] P. A. Franaszek and R. D. Nelson. Delay cost scheduling for timesharing systems. Technical Report RC 13777, IBM Research Division, June 1988.

[19] P. A. Franaszek and R. D. Nelson. Properties of delay cost scheduling in timesharing systems. Technical Report RC 13232, IBM Research Division, February 1990.

[20] L. Georgiadis, C. Nikolaou, and I. Viniotis. A family of job scheduling strategies for average response time constraints. In *Proceedings of the Second Annual Allerton Conference on Communications, Control, and Computing*, pages 810–819, September 1989.

[21] M. Greenberger. The priority problem and computer time sharing. *Management Science*, 12(11):888–906, July 1966.

[22] J. L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[23] G. J. Henry. The fair share scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, October 1984.

[24] C. A. R. Hoare and R. H. Perrott, editors. *Operating Systems Techniques.* Academic Press, 1972.

[25] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 112–122, 1985.

[26] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[27] J. F. C. Kingman. The effect of queue discipline on waiting time variance. In *Proceedings of the Cambridge Philosophical Society*, volume 58, pages 163–164, 1962.

[28] L. Kleinrock. A delay dependent queue discipline. *Naval Research Logistics Quarterly*, 11:329–341, 1964.

[29] L. Kleinrock. Queueing with strict and lag priority mixtures. In *Proceedings of the Fourth International Conference on Operational Research*, pages 46–67, 1966.

[30] L. Kleinrock. A continuum of time-sharing scheduling algorithms. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 36, pages 453–458, May 1970.

[31] L. Kleinrock and R. P. Finkelstein. Time dependent priority queues. *Operations Research*, 15:104–116, 1967.

[32] G. P. Klimov. Time sharing service systems I. *Theory of Probability and Its Applications*, 19(3):532–551, 1974.

[33] G. P. Klimov. Time sharing service systems II. *Theory of Probability and Its Applications*, 23(2):314–321, 1978.

[34] B. W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 11(5):347–360, May 1968.

[35] J. Larmouth. Scheduling for a share of the machine. *Software – Practice and Experience*, 5:29–49, 1975.

[36] J. Larmouth. Scheduling for immediate turnaround. *Software – Practice and Experience*, 8:559–578, 1978.

[37] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implmentaion of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[38] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotone scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[39] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[40] T. W. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard. Enterprise: A market-like task scheduler for distributed computing environments. In B. A. Huberman, editor, *The Ecology of Computation*, 1988.

[41] M. S. Miller and K. E. Drexler. Markets and computation: Agoric open systems. In B. A. Huberman, editor, *The Ecology of Computation*, 1988.

[42] I. Mitrani and J. H. Hine. Complete parameterized families of job scheduling strategies. *Acta Informatica*, 8:61–73, 1977.

[43] R. D. Nelson. Heavy traffic response times for a priority queue with linear priorities. *Operations Research*, 38(3):560–563, 1990.

[44] R. D. Nelson. Invertible mapping of waiting times in a M/G/1 queue with linear priorities. Unpublished Report, June 1993.

[45] R. D. Nelson and M. S. Squillante. On the use of update intervals for efficient delay cost scheduling implemenations. Unpublished Study, August 1991.

[46] J. P. Newbury. Immediate turnaround – an elusive goal. *Software – Practice and Experience*, 12:897–906, 1982.

[47] E. I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, 1972.

[48] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[49] K. K. Ramakrishnan, L. Vaitzblit, C. Gray, U. Vahalia, D. Ting, P. Tzelnic, S. Glaser, and W. Duso. Operating system support for a video-on-demand file service. In *Proceedings of the Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 225–236, November 1993.

[50] K. Ramamritham and J. A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, January 1994.

[51] K. W. Ross and B. Chen. Optimal scheduling of interactive and noninteractive traffic in telecommunication systems. *IEEE Transactions on Automatic Control*, 33(3):261–267, March 1988.

[52] K. W. Ross and D. D. Yao. Optimal dynamic scheduling in Jackson networks. *IEEE Transactions on Automatic Control*, 34(1):47–53, January 1989.

[53] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[54] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[55] A. Silberschatz and P. B. Galvin. *Operating System Concepts.* Addison-Wesley, Fourth edition, 1994.

[56] M. S. Squillante. Task dispatching data structures and algorithms for delay cost scheduling. Technical Report RN 92-4-13 (IBM Confidential), IBM Research Division, April 1992.

[57] M. S. Squillante and L. L. Fong. An algorithmic approach for efficient delay cost scheduling implemenations. Technical Report RN 94-11-21 (IBM Confidential), IBM Research Division, November 1994.

[58] M. S. Squillante, L. L. Fong, S. Liu, and S. K. Ryan. A control study of delay cost scheduling: Part I. Technical Report RC 19765 (IBM Confidential), IBM Research Division, September 1994.

[59] M. S. Squillante, L. L. Fong, S. Liu, and S. K. Ryan. A control study of delay cost scheduling: Part II. Technical Report RC 19771 (IBM Confidential), IBM Research Division, September 1994.

[60] M. S. Squillante, L. L. Fong, S. Liu, and S. K. Ryan. A control study of delay cost scheduling: Part III. Technical Report RC 19891 (IBM Confidential), IBM Research Division, December 1994.

[61] I. Stoica and H. Abdel-Wahab. A new approach to implement proportional share resource allocation. Technical Report 95-05, Department of Computer Science, Old Dominion University, 1995.

[62] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[63] H. Tokuda, J. W. Wendorf, and H.-Y. Wang. Implementation of a time-driven scheduler for real-time operating systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 271–280, 1987.

[64] Transaction Processing Performance Council. *TPC Benchmark C – Standard Specificantion, Revision 2.0*, October 1993.

[65] P. P. Varaiya, J. C. Walrand, and C. Buyukkoc. Extensions of the multiarmed bandit problem: The discounted case. *IEEE Transactions on Automatic Control*, 30:426–439, 1985.

[66] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[67] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, November 1994.

[68] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[69] C. M. Woodside. Controllability of computer performance tradeoffs obtained using controlled-share queue schedulers. *IEEE Transactions on Software Engineering*, SE-12(10):1041–1048, October 1986.

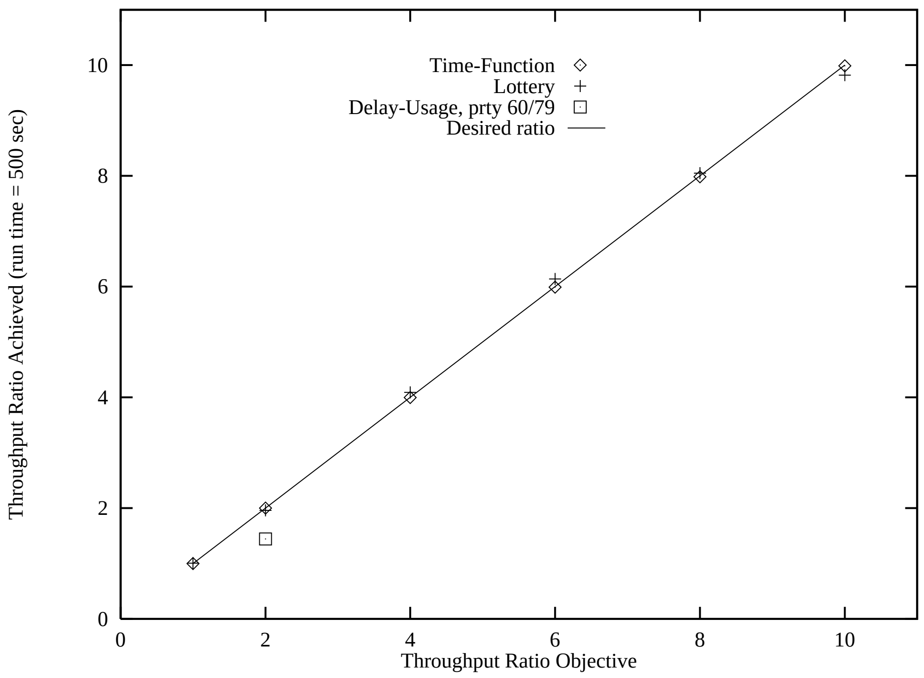

Figure 1: Throughput Ratios Achieved as a Function of the Throughput Ratio Objective

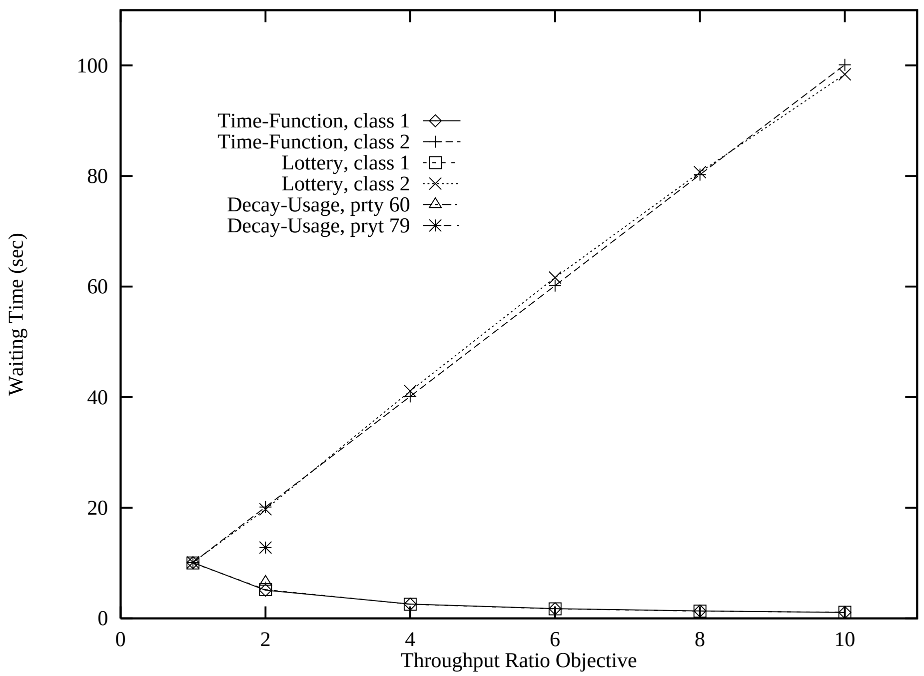

Figure 2: Mean Waiting Times for each Class as a Function of the Throughput Ratio Objective

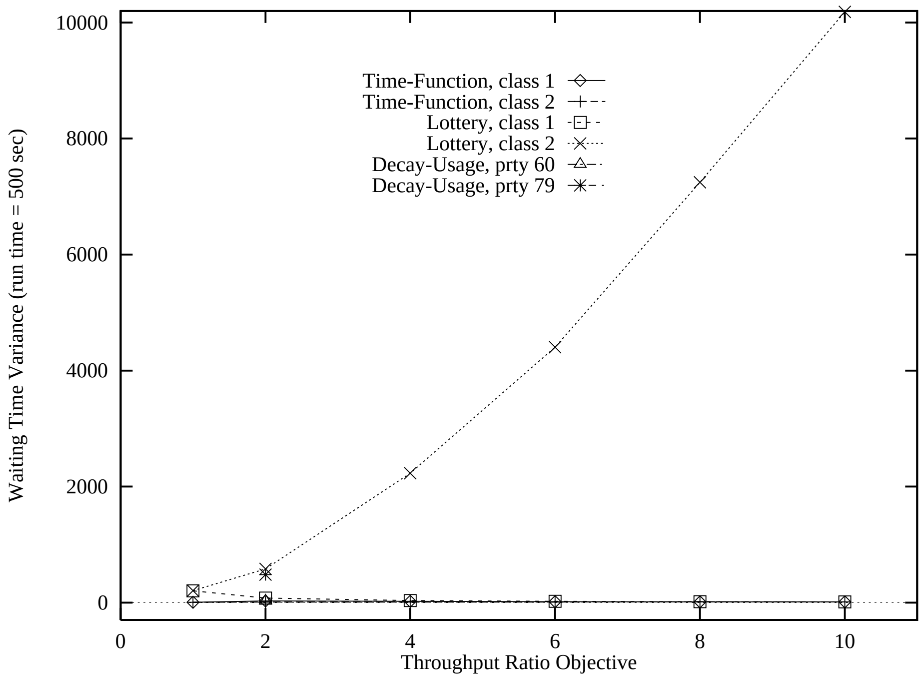

Figure 3: Variance of Waiting Times for each Class as a Function of the Throughput Ratio Objective

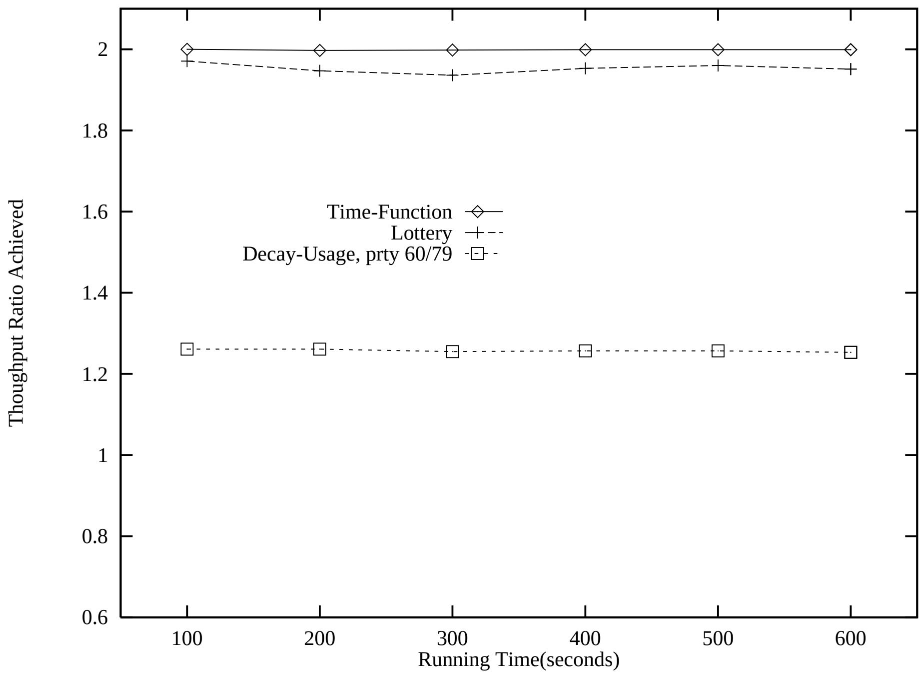

Figure 4: Throughput Ratios Achieved for a 2:1 Throughput Objective

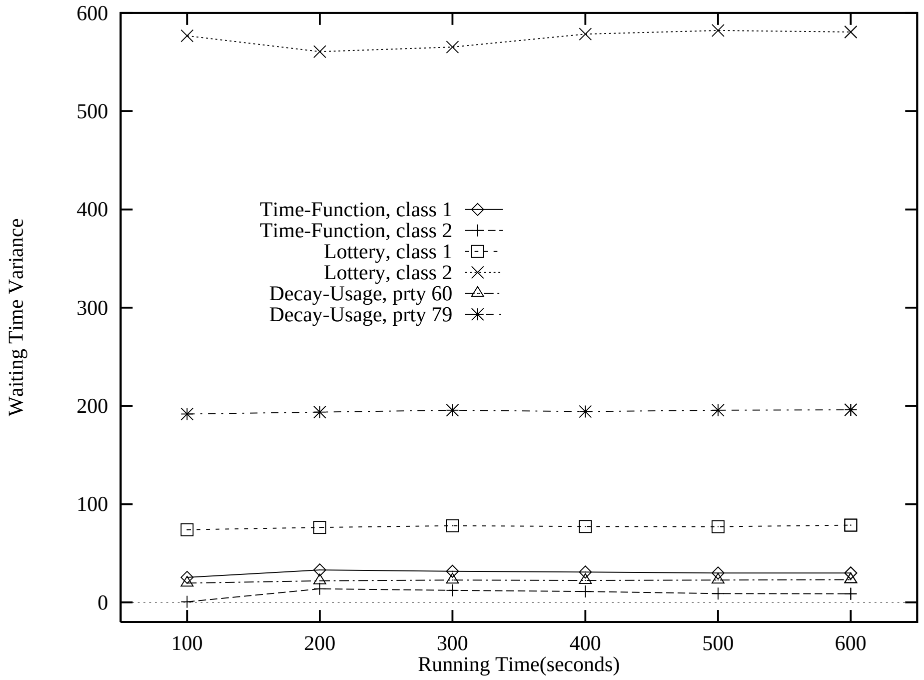

Figure 5: Variance of Waiting Times for a 2:1 Throughput Objective

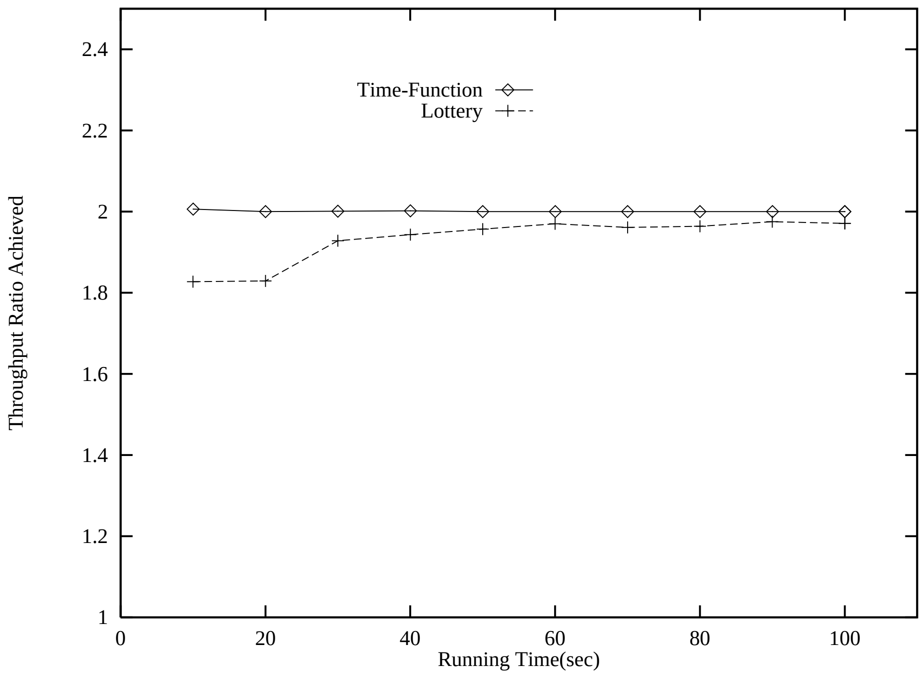

Figure 6: Throughput Ratios Achieved for a 2:1 Throughput Objective

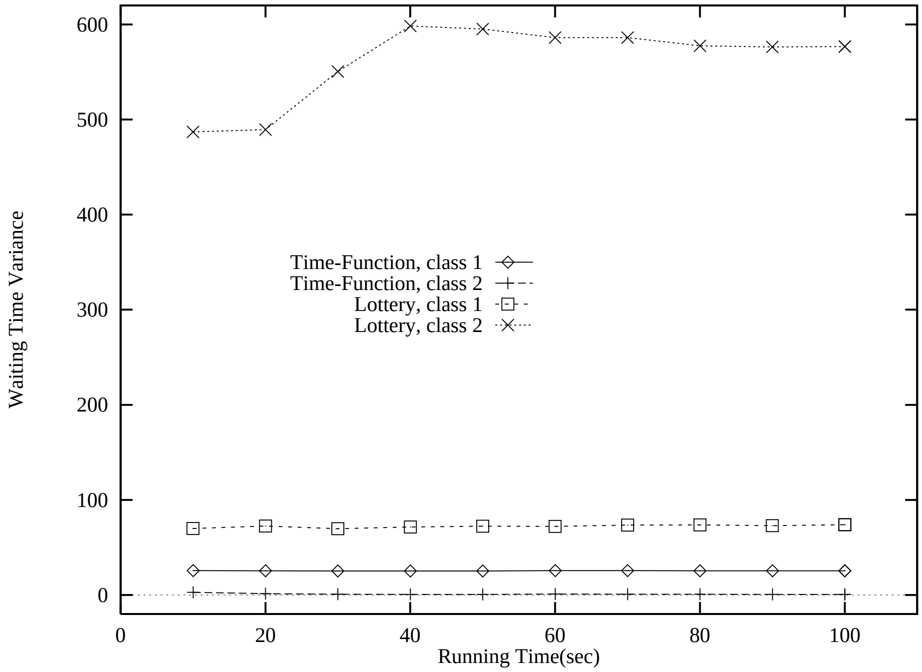

Figure 7: Variance of Waiting Times for a 2:1 Throughput Objective

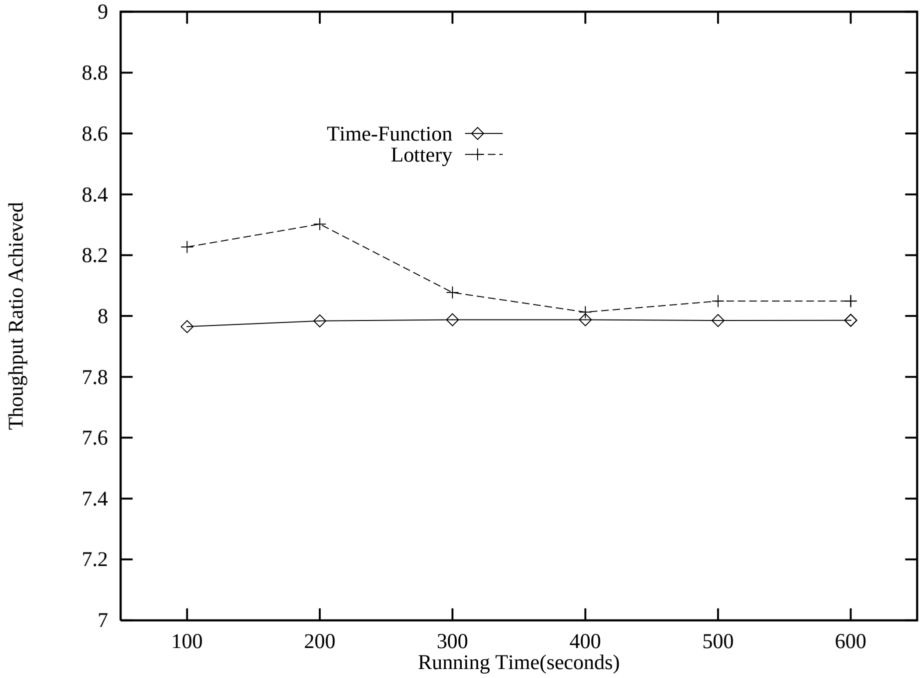

Figure 8: Throughput Ratios Achieved for an 8:1 Throughput Objective

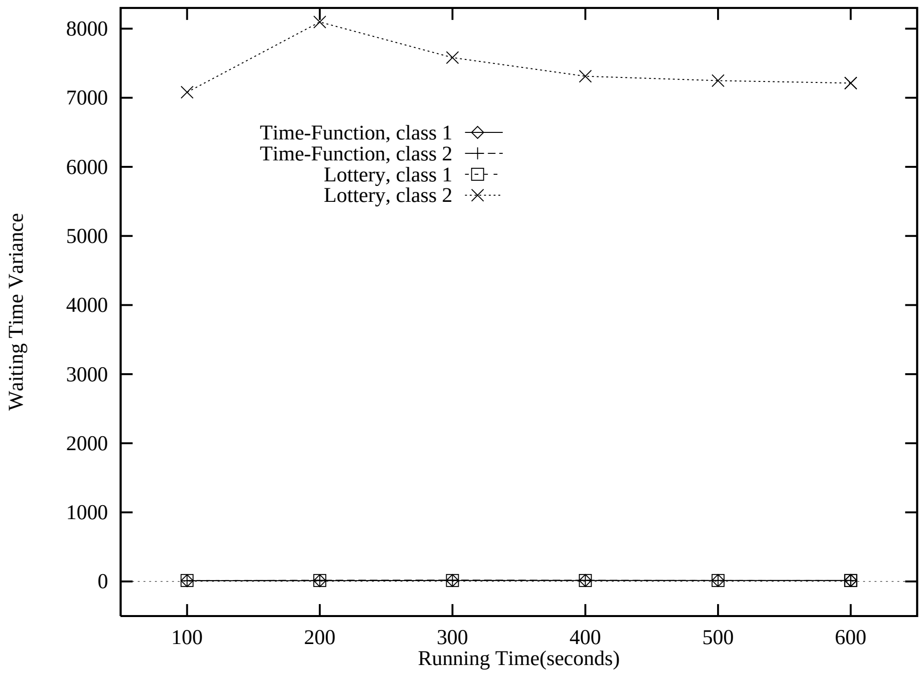

Figure 9: Variance of Waiting Times for an 8:1 Throughput Objective

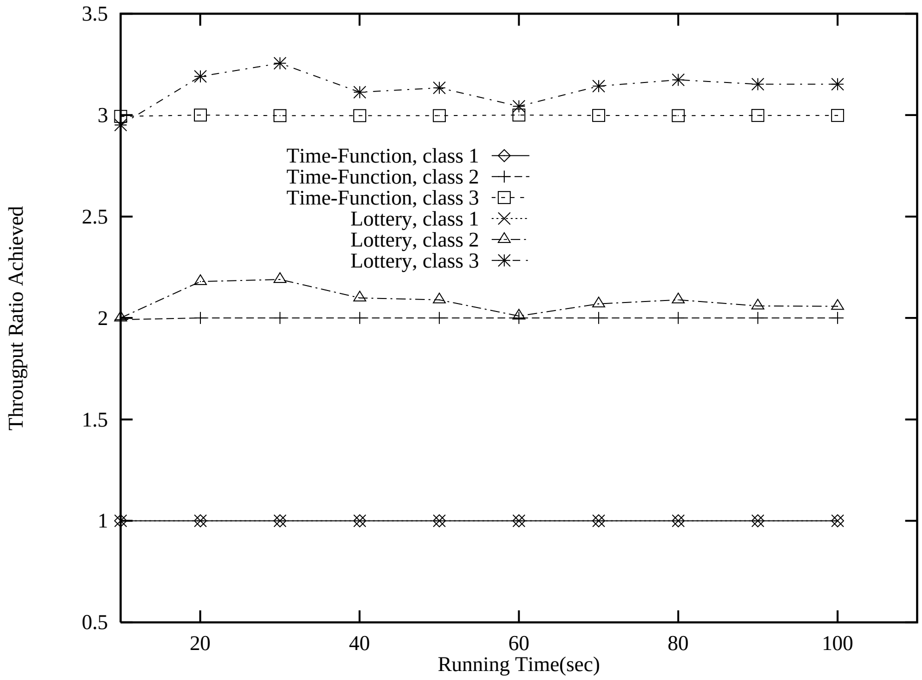

Figure 10: Throughput Ratios Achieved for a 3:2:1 Throughput Objective

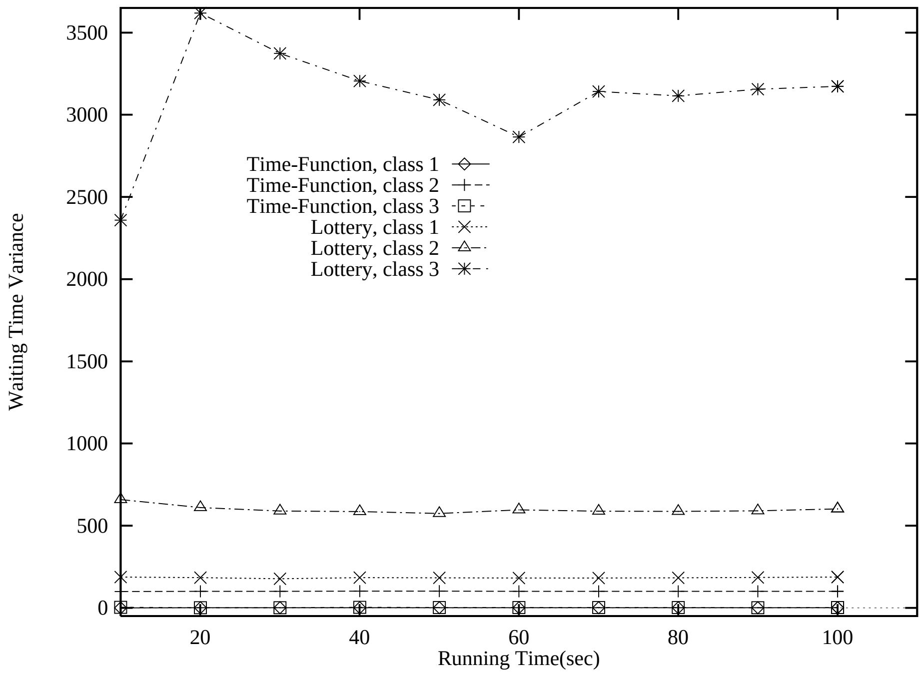

Figure 11: Variance of Waiting Times for a 3:2:1 Throughput Objective

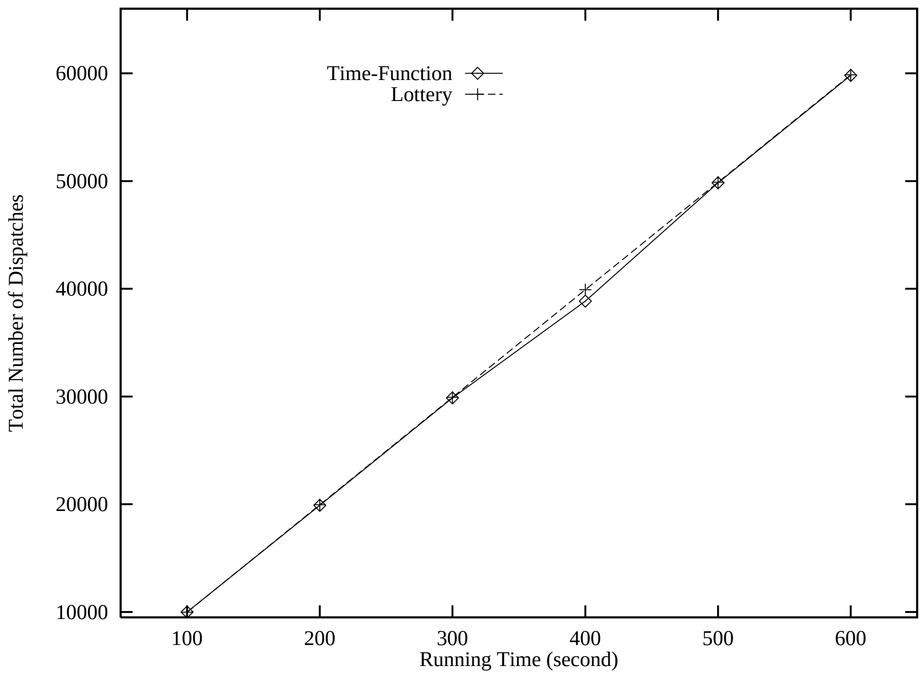

Figure 12: Total Number of Dispatches Measured across all Classes

Transaction Type	Time-Function Ugrp1 Tx	Time-Function Ugrp2 Tx	Time-Function Tx ratio	Lottery Ugrp1 Tx	Lottery Ugrp2 Tx	Lottery Tx ratio	Decay-Usage Ugrp1 Tx	Decay-Usage Ugrp2 Tx	Decay-Usage Tx ratio
New Order	1244	730	1.70	1255	772	1.62	991	958	1.03
Payment	1253	719	1.74	1201	772	1.56	1092	899	1.21
Order status	109	73	1.49	98	83	1.18	91	88	1.03
Delivery	96	64	1.50	102	67	1.59	89	77	1.16
Stock Level	110	57	1.93	108	87	1.23	102	100	1.02

Table 1: Experimental Results for 2 TPC-C User-Groups under TFS 2:1, Lottery 2:1, and Decay-Usage with Same Initial Priorities

Transaction Type	Time-Function Ugrp1 Tx	Time-Function Ugrp2 Tx	Time-Function Tx ratio	Lottery Ugrp1 Tx	Lottery Ugrp2 Tx	Lottery Tx ratio	Fixed-Priority Ugrp1 Tx	Fixed-Priority Ugrp2 Tx	Fixed-Priority Tx ratio
New Order	1322	734	1.80	1200	826	1.45	1543	501	3.08
Payment	1302	692	1.88	1153	801	1.44	1428	479	2.98
Order status	109	61	1.78	115	81	1.41	141	36	3.91
Delivery	117	70	1.67	94	65	1.44	137	49	2.80
Stock Level	122	48	2.54	118	57	2.07	146	40	3.65

Table 2: Experimental Results for 2 TPC-C User-Groups under TFS 2:1, Lottery 2:1, and Fixed Priorities