Stride Scheduling:
Deterministic Proportional-Share Resource Management

Carl A. Waldspurger * William E. Weihl *

Technical Memorandum MIT/LCS/TM-528
MIT Laboratory for Computer Science
Cambridge, MA 02139

June 22, 1995

Abstract

This paper presents *stride scheduling*, a deterministic scheduling technique that efficiently supports the same flexible resource management abstractions introduced by *lottery scheduling*. Compared to lottery scheduling, stride scheduling achieves significantly improved accuracy over relative throughput rates, with significantly lower response time variability. Stride scheduling implements proportional-share control over processor time and other resources by cross-applying elements of rate-based flow control algorithms designed for networks. We introduce new techniques to support dynamic changes and higher-level resource management abstractions. We also introduce a novel *hierarchical* stride scheduling algorithm that achieves better throughput accuracy and lower response time variability than prior schemes. Stride scheduling is evaluated using both simulations and prototypes implemented for the Linux kernel.

Keywords: dynamic scheduling, proportional-share resource allocation, rate-based service, service rate objectives

1 Introduction

Schedulers for multithreaded systems must multiplex scarce resources in order to service requests of varying importance. Accurate control over relative computation rates is required to achieve service rate objectives for users and applications. Such control is desirable across a broad spectrum of systems, including databases, media-based applications, and networks. Motivating examples include control over frame rates for competing video viewers, query rates for concurrent clients by databases and Web servers, and the consumption of shared resources by long-running computations.

Few general-purpose approaches have been proposed to support flexible, responsive control over service rates. We recently introduced *lottery scheduling*, a randomized resource allocation mechanism that provides efficient, responsive control over relative computation rates [Wal94]. Lottery scheduling implements *proportional-share* resource management – the resource consumption rates of active clients are proportional to the relative shares that they are allocated. Higher-level abstractions for flexible, modular resource management were also introduced with lottery scheduling, but they do not depend on the randomized implementation of proportional sharing.

In this paper we introduce *stride scheduling*, a deterministic scheduling technique that efficiently supports the same flexible resource management abstractions introduced by lottery scheduling. One contribution of our work is a cross-application and generalization of rate-based flow control algorithms designed for networks [Dem90, Zha91, ZhK91, Par93] to schedule other resources such as processor time. We present new techniques to support dynamic operations such as the modification of relative allocations and the transfer of resource rights between clients. We also introduce a novel *hierarchical* stride scheduling algorithm. Hierarchical stride

*E-mail: {carl, weihl}@lcs.mit.edu. World Wide Web: http://www.psg.lcs.mit.edu/. Prof. Weihl is currently supported by DEC while on sabbatical at DEC SRC. This research was also supported by ARPA under contract N00014-94-1-0985, by grants from AT&T and IBM, and by an equipment grant from DEC. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

scheduling is a recursive application of the basic technique that achieves better throughput accuracy and lower response time variability than previous schemes.

Simulation results demonstrate that, compared to lottery scheduling, stride scheduling achieves significantly improved accuracy over relative throughput rates, with significantly lower response time variability. In contrast to other deterministic schemes, stride scheduling efficiently supports operations that dynamically modify relative allocations and the number of clients competing for a resource. We have also implemented prototype stride schedulers for the Linux kernel, and found that they provide accurate control over both processor time and the relative network transmission rates of competing sockets.

In the next section, we present the core stride-scheduling mechanism. Section 3 describes extensions that support the resource management abstractions introduced with lottery scheduling. Section 4 introduces hierarchical stride scheduling. Simulation results with quantitative comparisons to lottery scheduling appear in Section 5. A discussion of our Linux prototypes and related implementation issues are presented in Section 6. In Section 7, we examine related work. Finally, we summarize our conclusions in Section 8.

2 Stride Scheduling

Stride scheduling is a deterministic allocation mechanism for time-shared resources. Resources are allocated in discrete time slices; we refer to the duration of a standard time slice as a *quantum*. Resource rights are represented by *tickets* – abstract, first-class objects that can be issued in different amounts and passed between clients.[1] Throughput rates for active clients are directly proportional to their ticket allocations. Thus, a client with twice as many tickets as another will receive twice as much of a resource in a given time interval. Client response times are inversely proportional to ticket allocations. Therefore a client with twice as many tickets as another will wait only half as long before acquiring a resource.

The throughput accuracy of a proportional-share scheduler can be characterized by measuring the difference between the specified and actual number of allocations that a client receives during a series of allocations. If a client has t tickets in a system with a total of T tickets, then its *specified* allocation after n_a consecutive allocations is $n_a \times t/T$. Due to quantization, it is typically impossible to achieve this ideal exactly. We define a client's *absolute error* as the absolute value of the difference between its specified and actual number of allocations. We define the pairwise *relative error* between clients c_i and c_j as the absolute error for the subsystem containing only c_i and c_j, where $T = t_i + t_j$, and n_a is the total number of allocations received by both clients.

While lottery scheduling offers probabilistic guarantees about throughput and response time, stride scheduling provides stronger deterministic guarantees. For lottery scheduling, after a series of n_a allocations, a client's expected relative error and expected absolute error are both $O(\sqrt{n_a})$. For stride scheduling, the relative error for any pair of clients is never greater than *one*, independent of n_a. However, for skewed ticket distributions it is still possible for a client to have $O(n_c)$ absolute error, where n_c is the number of clients. Nevertheless, stride scheduling is considerably more accurate than lottery scheduling, since its error does not grow with the number of allocations. In Section 4, we introduce a hierarchical variant of stride scheduling that provides a tighter $O(\lg n_c)$ bound on each client's absolute error.

This section first presents the basic stride-scheduling algorithm, and then introduces extensions that support dynamic client participation, dynamic modifications to ticket allocations, and nonuniform quanta.

2.1 Basic Algorithm

The core stride scheduling idea is to compute a representation of the time interval, or *stride*, that a client must wait between successive allocations. The client with the smallest stride will be scheduled most frequently. A client with half the stride of another will execute twice as quickly; a client with double the stride of another will execute twice as slowly. Strides are represented in virtual time units called *passes*, instead of units of real time such as seconds.

Three state variables are associated with each client: *tickets*, *stride*, and *pass*. The *tickets* field specifies the client's resource allocation, relative to other clients.

[1] In this paper we use the same terminology (*e.g.*, *tickets* and *currencies*) that we introduced for lottery scheduling [Wal94].

```
/* per-client state */
typedef struct {
  ...
  int tickets, stride, pass;
} *client_t;

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* initialize client with specified allocation */
void client_init(client_t c, queue_t q, int tickets)
{
  /* stride is inverse of tickets */
  c->tickets = tickets;
  c->stride = stride1 / tickets;
  c->pass = c->stride;

  /* join competition for resource */
  queue_insert(q, c);
}

/* proportional-share resource allocation */
void allocate(queue_t q)
{
  /* select client with minimum pass value */
  current = queue_remove_min(q);

  /* use resource for quantum */
  use_resource(current);

  /* compute next pass using stride */
  current->pass += current->stride;
  queue_insert(q, current);
}
```

Figure 1: **Basic Stride Scheduling Algorithm.** ANSI C code for scheduling a static set of clients. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

The *stride* field is inversely proportional to *tickets*, and represents the interval between selections, measured in passes. The *pass* field represents the virtual time index for the client's next selection.

Performing a resource allocation is very simple: the client with the minimum *pass* is selected, and its *pass* is advanced by its *stride*. If more than one client has the same minimum pass value, then any of them may be selected. A reasonable deterministic approach is to use a consistent ordering to break ties, such as one defined by unique client identifiers.

Figure 1 lists ANSI C code for the basic stride scheduling algorithm. For simplicity, we assume a static set of clients with fixed ticket assignments. The stride scheduling state for each client must be initialized via *client_init()* before any allocations are performed by *allocate()*. These restrictions will be relaxed in subsequent sections to permit more dynamic behavior.

To accurately represent *stride* as the reciprocal of *tickets*, a floating-point representation could be used. We present a more efficient alternative that uses a high-precision fixed-point integer representation. This is easily implemented by multiplying the inverted ticket value by a large integer constant. We will refer to this constant as $stride_1$, since it represents the stride corresponding to the minimum ticket allocation of one.[2]

The cost of performing an allocation depends on the data structure used to implement the client queue. A priority queue can be used to implement *queue_remove_min()* and other queue operations in $O(\lg n_c)$ time or better, where n_c is the number of clients [Cor90]. A skip list could also provide expected $O(\lg n_c)$ time queue operations with low constant overhead [Pug90]. For small n_c or heavily skewed ticket distributions, a simple sorted list is likely to be most efficient in practice.

Figure 2 illustrates an example of stride scheduling. Three clients, A, B, and C, are competing for a time-shared resource with a $3:2:1$ ticket ratio. For simplicity, a convenient $stride_1 = 6$ is used instead of a large number, yielding respective strides of 2, 3, and 6. The pass value of each client is plotted as a function of time. For each quantum, the client with the minimum pass value is selected, and its pass is advanced by its stride. Ties are

[2]Appendix A discusses the representation of strides in more detail.

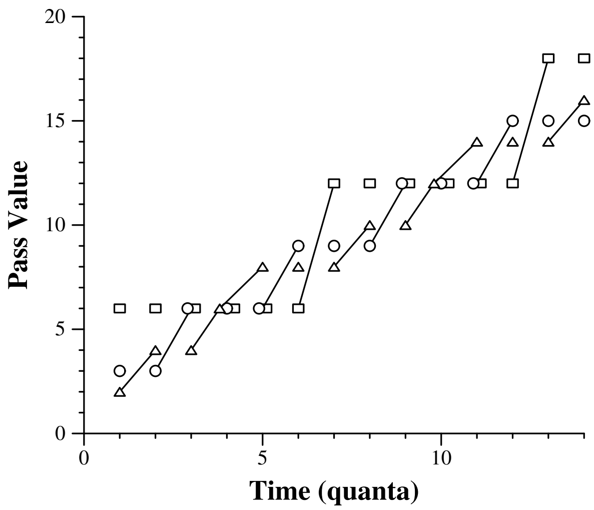

Figure 2: **Stride Scheduling Example.** Clients A (triangles), B (circles), and C (squares) have a 3 : 2 : 1 ticket ratio. In this example, *stride*$_1$ = 6, yielding respective strides of 2, 3, and 6. For each quantum, the client with the minimum pass value is selected, and its pass is advanced by its stride.

broken using the arbitrary but consistent client ordering A, B, C.

2.2 Dynamic Client Participation

The algorithm presented in Figure 1 does not support dynamic changes in the number of clients competing for a resource. When clients are allowed to join and leave at any time, their state must be appropriately modified. Figure 3 extends the basic algorithm to efficiently handle dynamic changes.

A key extension is the addition of global variables that maintain aggregate information about the set of active clients. The *global_tickets* variable contains the total ticket sum for all active clients. The *global_pass* variable maintains the "current" pass for the scheduler. The *global_pass* advances at the rate of *global_stride* per quantum, where *global_stride* = *stride*$_1$ / *global_tickets*. Conceptually, the *global_pass* continuously advances at a smooth rate. This is implemented by invoking the *global_pass_update()* routine whenever the *global_pass* value is needed.[3]

A state variable is also associated with each client to store the remaining portion of its stride when a dynamic change occurs. The *remain* field represents the number of passes that are left before a client's next selection. When a client leaves the system, *remain* is computed as the difference between the client's *pass* and the *global_pass*. When a client rejoins the system, its *pass* value is recomputed by adding its *remain* value to the *global_pass*.

This mechanism handles situations involving either positive or negative error between the specified and actual number of allocations. If *remain* $<$ *stride*, then the client is effectively given credit when it rejoins for having previously waited for part of its stride without receiving a quantum. If *remain* $>$ *stride*, then the client is effectively penalized when it rejoins for having previously received a quantum without waiting for its entire stride.[4]

This approach makes an implicit assumption that a partial quantum now is equivalent to a partial quantum later. In general, this is a reasonable assumption, and resembles the treatment of nonuniform quanta that will be presented Section 2.4. However, it may not be appropriate if the total number of tickets competing for a resource varies significantly between the time that a client leaves and rejoins the system.

The time complexity for both the *client_leave()* and *client_join()* operations is $O(\lg n_c)$, where n_c is the number of clients. These operations are efficient because the stride scheduling state associated with distinct clients is completely independent; a change to one client does not require updates to any other clients. The $O(\lg n_c)$ cost results from the need to perform queue manipulations.

2.3 Dynamic Ticket Modifications

Additional support is needed to dynamically modify client ticket allocations. Figure 4 illustrates a dynamic allocation change, and Figure 5 lists ANSI C code for

[3] Due to the use of a fixed-point integer representation for strides, small quantization errors may accumulate slowly, causing *global_pass* to drift away from client pass values over a long period of time. This is unlikely to be a practical problem, since client pass values are recomputed using *global_pass* each time they leave and rejoin the system. However, this problem can be avoided by very infrequently resetting *global_pass* to the minimum pass value for the set of active clients.

[4] Several interesting alternatives could also be implemented. For example, a client could be given credit for some or all of the passes that elapse while it is inactive.

```c
/* per-client state */
typedef struct {
  ...
  int tickets, stride, pass, remain;
} *client_t;

/* quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* global aggregate tickets, stride, pass */
int global_tickets, global_stride, global_pass;

/* update global pass based on elapsed real time */
void global_pass_update(void)
{
  static int last_update = 0;
  int elapsed;

  /* compute elapsed time, advance last_update */
  elapsed = time() - last_update;
  last_update += elapsed;

  /* advance global pass by quantum-adjusted stride */
  global_pass +=
    (global_stride * elapsed) / quantum;
}

/* update global tickets and stride to reflect change */
void global_tickets_update(int delta)
{
  global_tickets += delta;
  global_stride = stride1 / global_tickets;
}

/* initialize client with specified allocation */
void client_init(client_t c, int tickets)
{
  /* stride is inverse of tickets, whole stride remains */
  c->tickets = tickets;
  c->stride = stride1 / tickets;
  c->remain = c->stride;
}

/* join competition for resource */
void client_join(client_t c, queue_t q)
{
  /* compute pass for next allocation */
  global_pass_update();
  c->pass = global_pass + c->remain;

  /* add to queue */
  global_tickets_update(c->tickets);
  queue_insert(q, c);
}

/* leave competition for resource */
void client_leave(client_t c, queue_t q)
{
  /* compute remainder of current stride */
  global_pass_update();
  c->remain = c->pass - global_pass;

  /* remove from queue */
  global_tickets_update(-c->tickets);
  queue_remove(q, c);
}

/* proportional-share resource allocation */
void allocate(queue_t q)
{
  int elapsed;

  /* select client with minimum pass value */
  current = queue_remove_min(q);

  /* use resource, measuring elapsed real time */
  elapsed = use_resource(current);

  /* compute next pass using quantum-adjusted stride */
  current->pass +=
    (current->stride * elapsed) / quantum;
  queue_insert(q, current);
}
```

Figure 3: **Dynamic Stride Scheduling Algorithm.** ANSI C code for stride scheduling operations, including support for joining, leaving, and nonuniform quanta. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

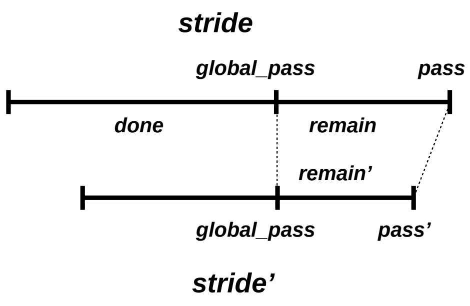

Figure 4: **Allocation Change.** Modifying a client's allocation from *tickets* to *tickets*$'$ requires only a constant-time recomputation of its *stride* and *pass*. The new *stride*$'$ is inversely proportional to *tickets*$'$. The new *pass*$'$ is determined by scaling *remain*, the remaining portion of the the current *stride*, by *stride*$'$ / *stride*.

```
/* dynamically modify client ticket allocation */
void client_modify(client_t c, queue_t q, int tickets)
{
  int remain, stride;

  /* leave queue for resource */
  client_leave(c, q);

  /* compute new stride */
  stride = stride1 / tickets;

  /* scale remaining passes to reflect change in stride */
  remain = (c->remain * stride) / c->stride;

  /* update client state */
  c->tickets = tickets;
  c->stride = stride;
  c->remain = remain;

  /* rejoin queue for resource */
  client_join(c, q);
}
```

Figure 5: **Dynamic Ticket Modification.** ANSI C code for dynamic modifications to client ticket allocations. Queue manipulations can be performed in $O(\lg n_c)$ time by using an appropriate data structure.

dynamically changing a client's ticket allocation. When a client's allocation is dynamically changed from *tickets* to *tickets*$'$, its *stride* and *pass* values must be recomputed. The new *stride*$'$ is computed as usual, inversely proportional to *tickets*$'$. To compute the new *pass*$'$, the remaining portion of the client's current *stride*, denoted by *remain*, is adjusted to reflect the new *stride*$'$. This is accomplished by scaling *remain* by *stride*$'$ / *stride*. In Figure 4, the client's ticket allocation is increased, so *pass* is decreased, compressing the time remaining until the client is next selected. If its allocation had decreased, then *pass* would have increased, expanding the time remaining until the client is next selected.

The *client_modify()* operation requires $O(\lg n_c)$ time, where n_c is the number of clients. As with dynamic changes to the number of clients, ticket allocation changes are efficient because the stride scheduling state associated with distinct clients is completely independent; the dominant cost is due to queue manipulations.

2.4 Nonuniform Quanta

With the basic stride scheduling algorithm presented in Figure 1, a client that does not consume its entire allocated quantum would receive less than its entitled share of a resource. Similarly, it may be possible for a client's usage to exceed a standard quantum in some situations. For example, under a non-preemptive scheduler, client run lengths can vary considerably.

Fortunately, fractional and variable-size quanta can easily be accommodated. When a client consumes a fraction f of its allocated time quantum, its *pass* should be advanced by $f \times$ *stride* instead of *stride*. If $f < 1$, then the client's *pass* will be increased less, and it will be scheduled sooner. If $f > 1$, then the client's *pass* will be increased more, and it will be scheduled later. The extended code listed in Figure 3 supports nonuniform quanta by effectively computing f as the *elapsed* resource usage time divided by a standard *quantum* in the same time units.

Another extension would permit clients to specify the quantum size that they require.[5] This could be implemented by associating an additional *quantum*$_c$ field with each client, and scaling each client's stride field by

[5] An alternative would be to allow a client to specify its scheduling period. Since a client's period and quantum are related by its relative resource share, specifying one quantity yields the other.

quantum$_c$ / *quantum*. Deviations from a client's specified quantum would still be handled as described above, with f redefined as the *elapsed* resource usage divided by the client-specific *quantum*$_c$.

3 Flexible Resource Management

Since stride scheduling enables low-overhead dynamic modifications, it can efficiently support the flexible resource management abstractions introduced with lottery scheduling [Wal94]. In this section, we explain how ticket transfers, ticket inflation, and ticket currencies can be implemented on top of a stride-based substrate for proportional sharing.

3.1 Ticket Transfers

A *ticket transfer* is an explicit transfer of tickets from one client to another. Ticket transfers are particularly useful when one client blocks waiting for another. For example, during a synchronous RPC, a client can loan its resource rights to the server computing on its behalf. A transfer of t tickets between clients A and B essentially consists of two dynamic ticket modifications. Using the code presented in Figure 5, these modifications are implemented by invoking *client_modify(A, q, A.tickets – t)* and *client_modify(B, q, B.tickets + t)*. When A transfers tickets to B, A's stride and pass will increase, while B's stride and pass will decrease.

A slight complication arises in the case of a complete ticket transfer; *i.e.*, when A transfers its entire ticket allocation to B. In this case, A's adjusted ticket value is zero, leading to an adjusted stride of infinity (division by zero). To circumvent this problem, we record the fraction of A's stride that is remaining at the time of the transfer, and then adjust that remaining fraction when A once again obtains tickets. This can easily be implemented by computing A's *remain* value at the time of the transfer, and deferring the computation of its stride and pass values until A receives a non-zero ticket allocation (perhaps via a return transfer from B).

3.2 Ticket Inflation

An alternative to explicit ticket transfers is *ticket inflation*, in which a client can escalate its resource rights by creating more tickets. Ticket inflation (or deflation) simply consists of a dynamic ticket modification for a client. Ticket inflation causes a client's stride and pass to decrease; deflation causes its stride and pass to increase.

Ticket inflation is useful among mutually trusting clients, since it permits resource rights to be reallocated without explicitly reshuffling tickets among clients. However, ticket inflation is also dangerous, since any client can monopolize a resource simply by creating a large number of tickets. In order to avoid the dangers of inflation while still exploiting its advantages, we introduced a *currency* abstraction for lottery scheduling [Wal94] that is loosely borrowed from economics.

3.3 Ticket Currencies

A *ticket currency* defines a resource management abstraction barrier that contains the effects of ticket inflation in a modular way. Tickets are denominated in currencies, allowing resource rights to be expressed in units that are local to each group of mutually trusting clients. Each currency is backed, or *funded*, by tickets that are denominated in more primitive currencies. Currency relationships may form an arbitrary acyclic graph, such as a hierarchy of currencies. The effects of inflation are locally contained by effectively maintaining an *exchange rate* between each local currency and a common *base* currency that is conserved. The currency abstraction is useful for flexibly naming, sharing, and protecting resource rights.

The currency abstraction introduced for lottery scheduling can also be used with stride scheduling. One implementation technique is to always immediately convert ticket values denominated in arbitrary currencies into units of the common base currency. Any changes to the value of a currency would then require dynamic modifications to all clients holding tickets denominated in that currency, or one derived from it.[6] Thus, the scope of any changes in currency values is limited to exactly those clients which are affected. Since currencies are used to group and isolate logical sets of clients, the impact of currency fluctuations will typically be very localized.

[6] An important exception is that changes to the number of tickets in the base currency do not require any modifications. This is because all stride scheduling state is computed from ticket values expressed in base units, and the state associated with distinct clients is independent.

4 Hierarchical Stride Scheduling

Stride scheduling guarantees that the *relative* throughput error for any pair of clients never exceeds a single quantum. However, depending on the distribution of tickets to clients, a large $O(n_c)$ *absolute* throughput error is still possible, where n_c is the number of clients.

For example, consider a set of 101 clients with a $100:1:\ldots:1$ ticket allocation. A schedule that minimizes absolute error and response time variability would alternate the 100-ticket client with each of the single-ticket clients. However, the standard stride algorithm schedules the clients in order, with the 100-ticket client receiving 100 quanta before any other client receives a single quantum. Thus, after 100 allocations, the intended allocation for the 100-ticket client is 50, while its actual allocation is 100, yielding a large absolute error of 50. This behavior is also exhibited by similar rate-based flow control algorithms for networks [Dem90, Zha91, ZhK91, Par93].

In this section we describe a novel hierarchical variant of stride scheduling that limits the absolute throughput error of any client to $O(\lg n_c)$ quanta. For the 101-client example described above, hierarchical stride scheduler simulations produced a maximum absolute error of only 4.5. Our algorithm also significantly reduces response time variability by aggregating clients to improve interleaving. Since it is common for systems to consist of a small number of high-throughput clients together with a large number of low-throughput clients, hierarchical stride scheduling represents a practical improvement over previous work.

4.1 Basic Algorithm

Hierarchical stride scheduling is essentially a recursive application of the basic stride scheduling algorithm. Individual clients are combined into groups with larger aggregate ticket allocations, and correspondingly smaller strides. An allocation is performed by invoking the normal stride scheduling algorithm first among groups, and then among individual clients within groups.

Although many different groupings are possible, we consider a balanced binary tree of groups. Each leaf node represents an individual client. Each internal node represents the group of clients (leaf nodes) that it covers, and contains their aggregate tickets, stride, and pass

```
/* binary tree node */
typedef struct node {
  ...
  struct node *left, *right, *parent;
  int tickets, stride, pass;
} *node_t;

/* quantum in real time units (e.g. 1M cycles) */
const int quantum = (1 << 20);

/* large integer stride constant (e.g. 1M) */
const int stride1 = (1 << 20);

/* current resource owner */
client_t current;

/* proportional-share resource allocation */
void allocate(node_t root)
{
  int elapsed;
  node_t n;

  /* traverse root-to-leaf path following min pass */
  for (n = root; !node_is_leaf(n); )
    if (n->left == NULL ||
        n->right->pass < n->left->pass)
      n = n->right;
    else
      n = n->left;

  /* use resource, measuring elapsed real time */
  current = n;
  elapsed = use_resource(current);

  /* update pass for each ancestor using its stride */
  for (n = current; n != NULL; n = n->parent)
    n->pass += (n->stride * elapsed) / quantum;
}
```

Figure 6: **Hierarchical Stride Scheduling Algorithm.** ANSI C code for hierachical stride scheduling with a static set of clients. The main data structure is a binary tree of nodes. Each node represents either a client (leaf) or a group (internal node) that summarizes aggregate information.

values. Thus, for an internal node, *tickets* is the total ticket sum for all of the clients that it covers, and *stride* = *stride*$_1$ / *tickets*. The *pass* value for an internal node is updated whenever the pass value for any of the clients that it covers is modified.

Figure 6 presents ANSI C code for the basic hierarchical stride scheduling algorithm. Each node has the normal tickets, stride, and pass scheduling state, as well as the usual tree links to its parent, left child, and right child. An allocation is performed by tracing a path from the root of the tree to a leaf, choosing the child with the smaller pass value at each level. Once the selected client has finished using the resource, its pass value is updated to reflect its usage. The client update is identical to that used in the dynamic stride algorithm that supports nonuniform quanta, listed in Figure 3. However, the hierarchical scheduler requires additional updates to each of the client's ancestors, following the leaf-to-root path formed by successive parent links.

Each client allocation can be viewed as a series of pairwise allocations among groups of clients at each level in the tree. The maximum error for each pairwise allocation is 1, and in the worst case, error can accumulate at each level. Thus, the maximum absolute error for the overall tree-based allocation is the height of the tree, which is $\lceil \lg n_c \rceil$, where n_c is the number of clients. Since the error for a pairwise A : B ratio is minimized when $A = B$, absolute error can be further reduced by carefully choosing client leaf positions to better balance the tree based on the number of tickets at each node.

4.2 Dynamic Modifications

Extending the basic hierarchical stride algorithm to support dynamic modifications requires a careful consideration of the effects of changes at each level in the tree. Figure 7 lists ANSI C code for performing a ticket modification that works for both clients and internal nodes. Changes to client ticket allocations essentially follow the same scaling and update rules used for normal stride scheduling, listed in Figure 5. The hierarchical scheduler requires additional updates to each of the client's ancestors, following the leaf-to-root path formed by successive parent links. Note that the root *pass* value used in Figure 7 effectively takes the place of the *global_pass* variable used in Figure 5; both represent the aggregate global scheduler pass.

```
/* dynamically modify node allocation by delta tickets */
void node_modify(node_t n, node_t root, int delta)
{
  int old_stride, remain;

  /* compute new tickets, stride */
  old_stride = n->stride;
  n->tickets += delta;
  n->stride = stride1 / n->tickets;

  /* done when reach root */
  if (n == root)
    return;

  /* scale remaining passes to reflect change in stride */
  remain = n->pass - root->pass;
  remain = (remain * n->stride) / old_stride;
  n->pass = root->pass + remain;

  /* propagate change to ancestors */
  node_modify(n->parent, root, delta);
}
```

Figure 7: **Dynamic Ticket Modification.** ANSI C code for dynamic modifications to client ticket allocations under hierarchical stride scheduling. A modification requires $O(\lg n_c)$ time to propagate changes.

Although not presented here, we have also developed operations to support dynamic client participation under hierarchical stride scheduling [Wal95]. As for *allocate()*, the time complexity for *client_join()* and *client_leave()* operations is $O(\lg n_c)$, where n_c is the number of clients.

5 Simulation Results

This section presents the results of several quantitative experiments designed to evaluate the effectiveness of stride scheduling. We examine the behavior of stride scheduling in both static and dynamic environments, and also test hierarchical stride scheduling. When stride scheduling is compared to lottery scheduling, we find that the stride-based approach provides more accurate control over relative throughput rates, with much lower variance in response times.

For example, Figure 8 presents the results of scheduling three clients with a 3 : 2 : 1 ticket ratio for 100 allocations. The dashed lines represent the ideal allocations for each client. It is clear from Figure 8(a) that lottery scheduling exhibits significant variability at this time scale, due to the algorithm's inherent use of randomization. In contrast, Figure 8(b) indicates that the deterministic stride scheduler produces precise periodic behavior.

5.1 Throughput Accuracy

Under randomized lottery scheduling, the expected value for the absolute error between the specified and actual number of allocations for any set of clients is $O(\sqrt{n_a})$, where n_a is the number of allocations. This is because the number of lotteries won by a client has a binomial distribution. The probability p that a client holding t tickets will win a given lottery with a total of T tickets is simply $p = t/T$. After n_a identical lotteries, the expected number of wins w is $E[w] = n_a p$, with variance $\sigma_w^2 = n_a p(1-p)$.

Under deterministic stride scheduling, the relative error between the specified and actual number of allocations for any pair of clients never exceeds *one*, independent of n_a. This is because the only source of relative error is due to quantization.

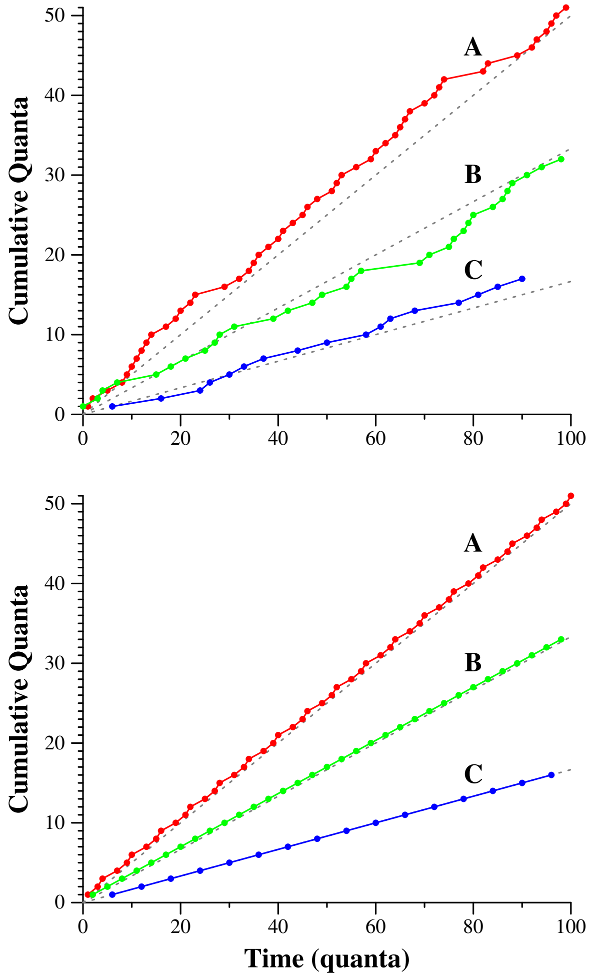

Figure 8: **Lottery vs. Stride Scheduling.** Simulation results for 100 allocations involving three clients, A, B, and C, with a 3 : 2 : 1 allocation. The dashed lines represent ideal proportional-share behavior. (a) Allocation by randomized lottery scheduler shows significant variability. (b) Allocation by deterministic stride scheduler exhibits precise periodic behavior: A, B, A, A, B, C.

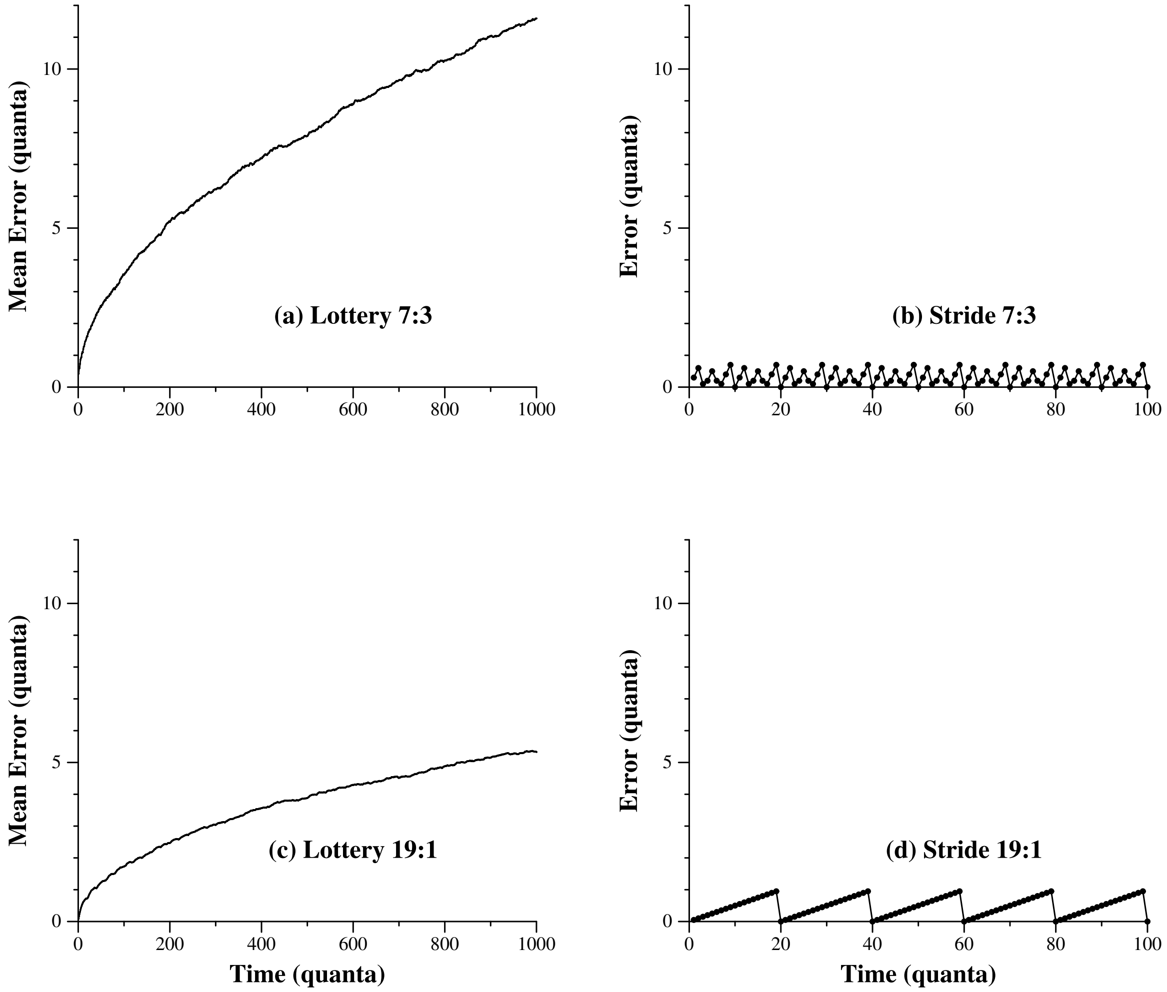

Figure 9: **Throughput Accuracy.** Simulation results for two clients with 7 : 3 (top) and 19 : 1 (bottom) ticket ratios over 1000 allocations. Only the first 100 quanta are shown for the stride scheduler, since its quantization error is deterministic and periodic. (a) Mean lottery scheduler error, averaged over 1000 separate 7 : 3 runs. (b) Stride scheduler error for a single 7 : 3 run. (c) Mean lottery scheduler error, averaged over 1000 separate 19 : 1 runs. (d) Stride scheduler error for a single 19 : 1 run.

Figure 9 plots the absolute error[7] that results from simulating two clients under both lottery scheduling and stride scheduling. The data depicted is representative of our simulation results over a large range of pairwise ratios. Figure 9(a) shows the mean error averaged over 1000 separate lottery scheduler runs with a 7 : 3 ticket ratio. As expected, the error increases slowly with n_a, indicating that accuracy steadily improves when error is measured as a percentage of n_a. Figure 9(b) shows the error for a single stride scheduler run with the same 7 : 3 ticket ratio. As expected, the error never exceeds a single quantum, and follows a deterministic pattern with period 10. The error drops to zero at the end of each complete period, corresponding to a precise 7 : 3 allocation. Figures 9(c) and 9(d) present data for similar experiments involving a larger 19 : 1 ticket ratio.

5.2 Dynamic Ticket Allocations

Figure 10 plots the absolute error that results from simulating two clients under both lottery scheduling and stride scheduling with rapidly-changing dynamic ticket allocations. This data is representative of simulation results over a large range of pairwise ratios and a variety of dynamic modification techniques. For easy comparison, the *average* dynamic ticket ratios are identical to the static ticket ratios used in Figure 9.

The notation $[A,B]$ indicates a random ticket allocation that is uniformly distributed from A to B. New, randomly-generated ticket allocations were dynamically assigned every other quantum. The *client_modify()* operation was executed for each change under stride scheduling; no special actions were necessary under lottery scheduling. To compute error values, specified allocations were determined incrementally. Each client's specified allocation was advanced by t/T on every quantum, where t is the client's current ticket allocation, and T is the current ticket total.

Figure 10(a) shows the mean error averaged over 1000 separate lottery scheduler runs with a [2,12] : 3 ticket ratio. Despite the dynamic changes, the mean error is nearly the same as that measured for the static 7 : 3 ratio depicted in Figure 9(a). Similarly, Figure 10(b) shows the error for a single stride scheduler run with the same dynamic [2,12] : 3 ratio. The error never exceeds a single quantum, although it is much more erratic than the periodic pattern exhibited for the static 7 : 3 ratio in Figure 9(b). Figures 10(c) and 10(d) present data for similar experiments involving a larger dynamic 190 : [5,15] ratio. The results for this allocation are comparable to those measured for the static 19 : 1 ticket ratio depicted in Figures 9(c) and 9(d).

Overall, the error measured under both lottery scheduling and stride scheduling is largely unaffected by dynamic ticket modifications. This suggests that both mechanisms are well-suited to dynamic environments. However, stride scheduling is clearly more accurate in both static and dynamic environments.

5.3 Response Time Variability

Another important performance metric is response time, which we measure as the elapsed time from a client's completion of one quantum up to and including its completion of another. Under randomized lottery scheduling, client response times have a geometric distribution. The expected number of lotteries n_a that a client must wait before its first win is $E[n_a] = 1/p$, with variance $\sigma^2_{n_a} = (1 - p)/p^2$. Deterministic stride scheduling exhibits dramatically less response-time variability.

Figures 11 and 12 present client response time distributions under both lottery scheduling and stride scheduling. Figure 11 shows the response times that result from simulating two clients with a 7 : 3 ticket ratio for one million allocations. The stride scheduler distributions are very tight, while the lottery scheduler distributions are geometric with long tails. For example, the client with the smaller allocation had a maximum response time of 4 quanta under stride scheduling, while the maximum response time under lottery scheduling was 39.

Figure 12 presents similar data for a larger 19 : 1 ticket ratio. Although there is little difference in the response time distributions for the client with the larger allocation, the difference is enormous for the client with the smaller allocation. Under stride scheduling, virtually all of the response times were exactly 20 quanta. The lottery scheduler produced geometrically-distributed response times ranging from 1 to 194 quanta. In this case, the standard deviation of the stride scheduler's distribution is three orders of magnitude smaller than the standard deviation of the lottery scheduler's distribution.

[7] In this case the relative and absolute errors are identical, since there are only two clients.

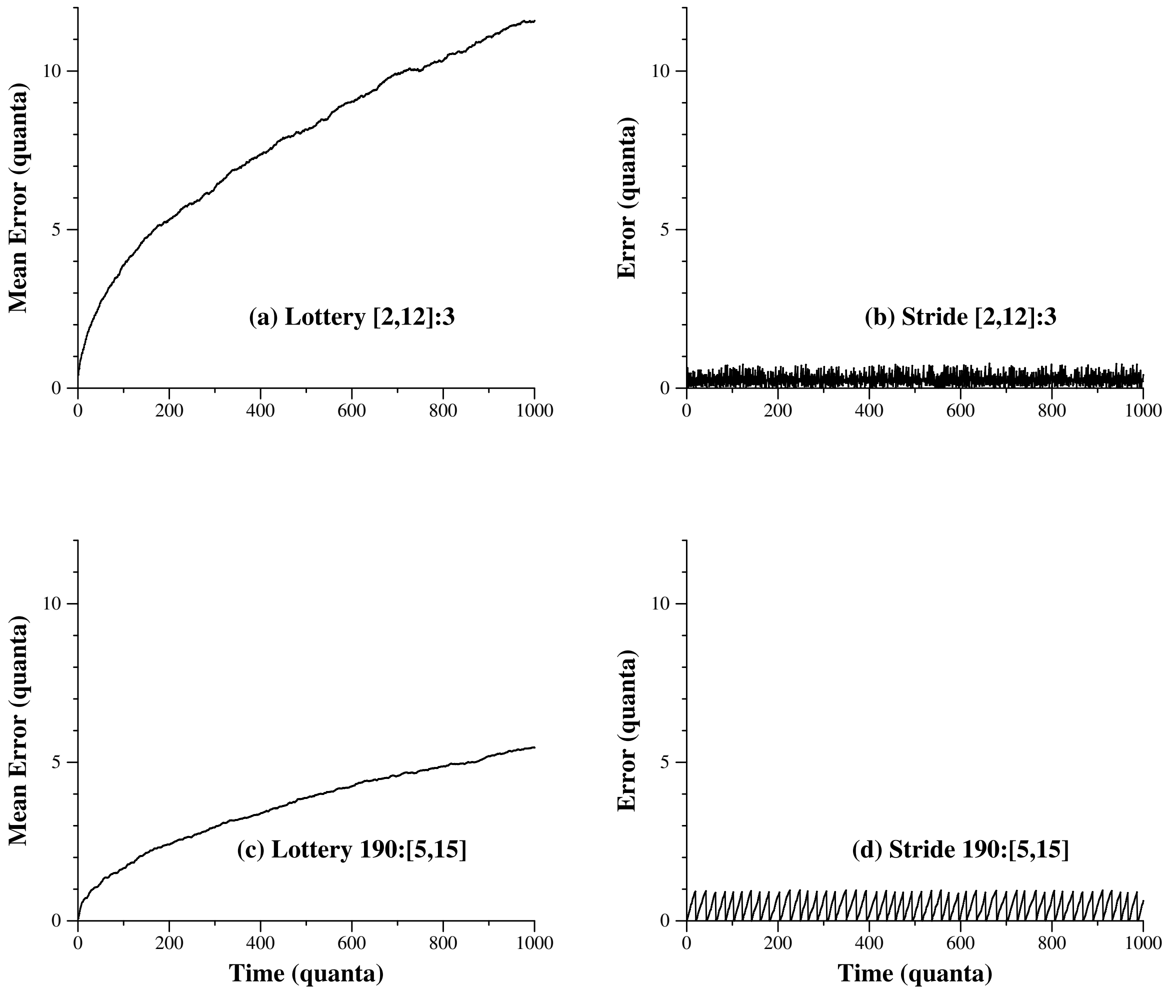

Figure 10: **Throughput Accuracy – Dynamic Allocations.** Simulation results for two clients with [2,12] : 3 (top) and 190 : [5,15] (bottom) ticket ratios over 1000 allocations. The notation $[A,B]$ indicates a random ticket allocation that is uniformly distributed from A to B. Random ticket allocations were dynamically updated every other quantum. (a) Mean lottery scheduler error, averaged over 1000 separate [2,12] : 3 runs. (b) Stride scheduler error for a single [2,12] : 3 run. (c) Mean lottery scheduler error, averaged over 1000 separate 190 : [5,15] runs. (d) Stride scheduler error for a single 190 : [5,15] run.

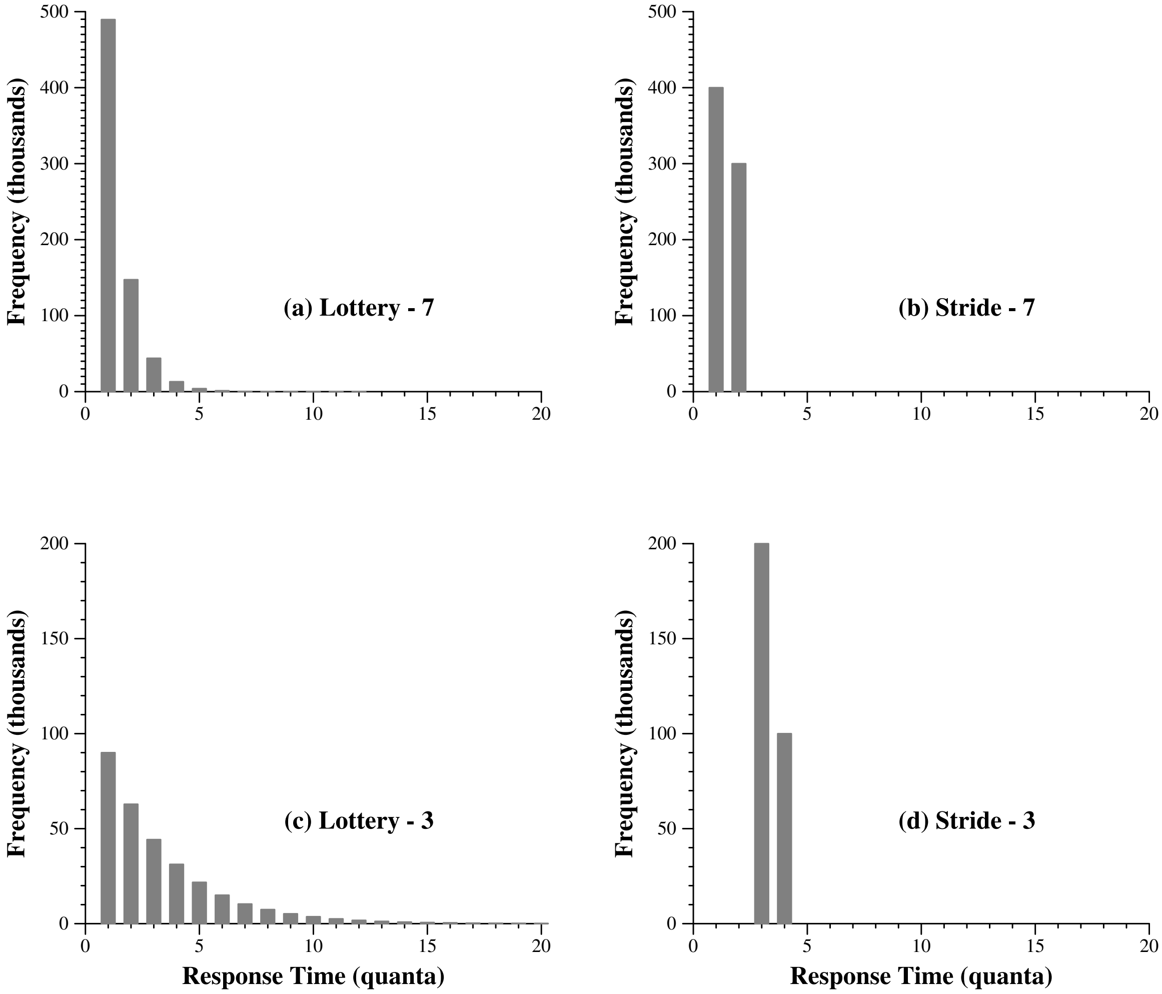

Figure 11: **Response Time Distribution.** Simulation results for two clients with a 7 : 3 ticket ratio over one million allocations. (a) Client with 7 tickets under lottery scheduling: $\mu = 1.43$, $\sigma = 0.78$. (b) Client with 7 tickets under stride scheduling: $\mu = 1.43$, $\sigma = 0.49$. (c) Client with 3 tickets under lottery scheduling: $\mu = 3.33$, $\sigma = 2.79$. (d) Client with 3 tickets under stride scheduling: $\mu = 3.33$, $\sigma = 0.47$.

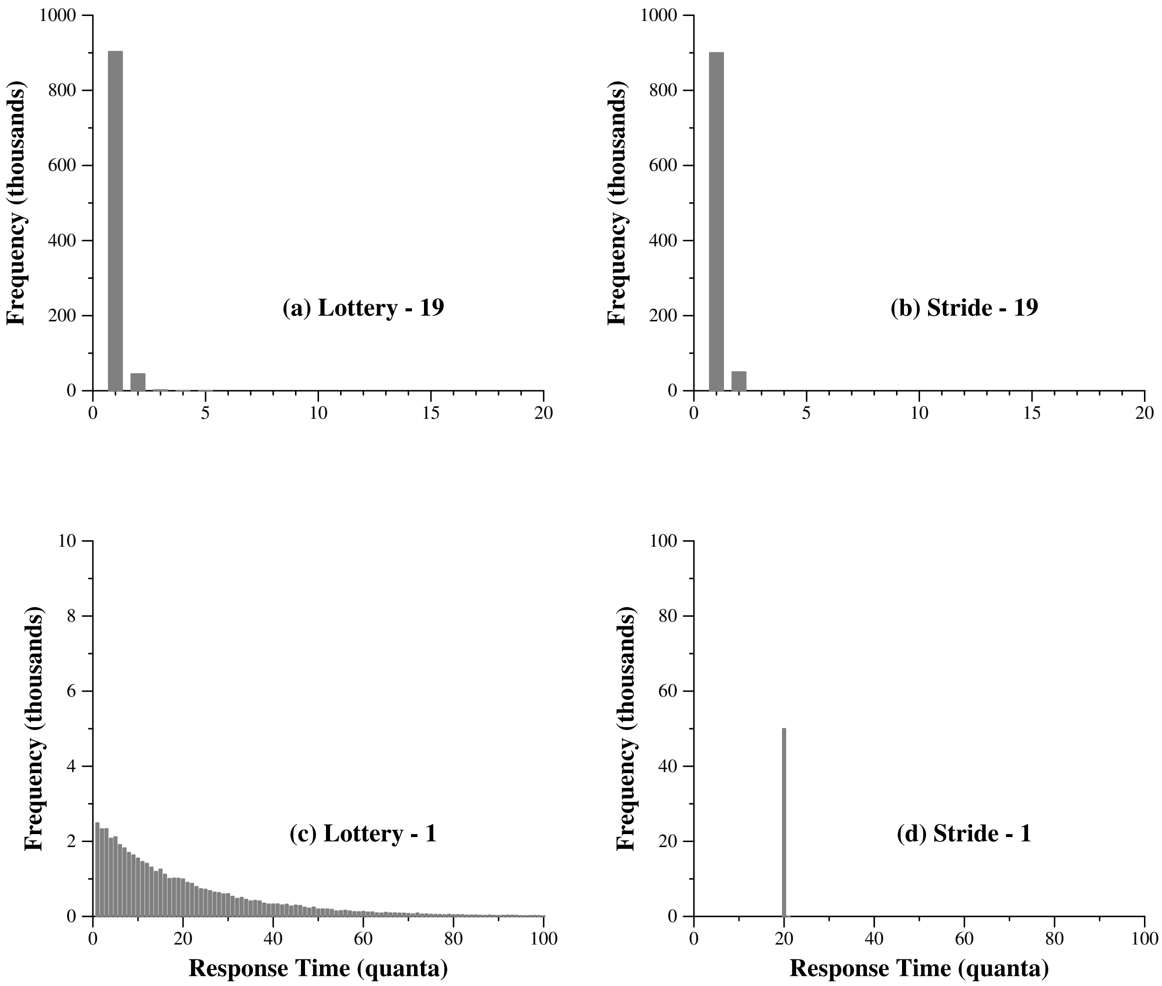

Figure 12: **Response Time Distribution.** Simulation results for two clients with a 19 : 1 ticket ratio over one million allocations. (a) Client with 19 tickets under lottery scheduling: $\mu = 1.05$, $\sigma = 0.24$. (b) Client with 19 tickets under stride scheduling: $\mu = 1.05$, $\sigma = 0.22$. (c) Client with 1 ticket under lottery scheduling: $\mu = 20.13$, $\sigma = 19.64$. (d) Client with 1 ticket under stride scheduling: $\mu = 20.00$, $\sigma = 0.01$.

5.4 Hierarchical Stride Scheduling

As discussed in Section 4, stride scheduling can produce an absolute error of $O(n_c)$ for skewed ticket distributions, where n_c is the number of clients. In contrast, hierarchical stride scheduling bounds the absolute error to $O(\lg n_c)$. As a result, response-time variability can be significantly reduced under hierarchical stride scheduling.

Figure 13 presents client response time distributions under both hierarchical stride scheduling and ordinary stride scheduling. Eight clients with a 7 : 1 : . . . : 1 ticket ratio were simulated for one million allocations. Excluding the very first allocation, the response time for each of the low-throughput clients was always 14, under both schedulers. Thus we only present response time distributions for the high-throughput client.

The ordinary stride scheduler runs the high-throughput client for 7 consecutive quanta, and then runs each of the low-throughput clients for one quantum. The hierarchical stride scheduler interleaves the clients, resulting in a tighter distribution. In this case, the standard deviation of the ordinary stride scheduler's distribution is more than twice as large as that for the hierarchical stride scheduler. We observed a maximum absolute error of 4 quanta for the high-throughput client under ordinary stride scheduling, and only 1.5 quanta under hierarchical stride scheduling.

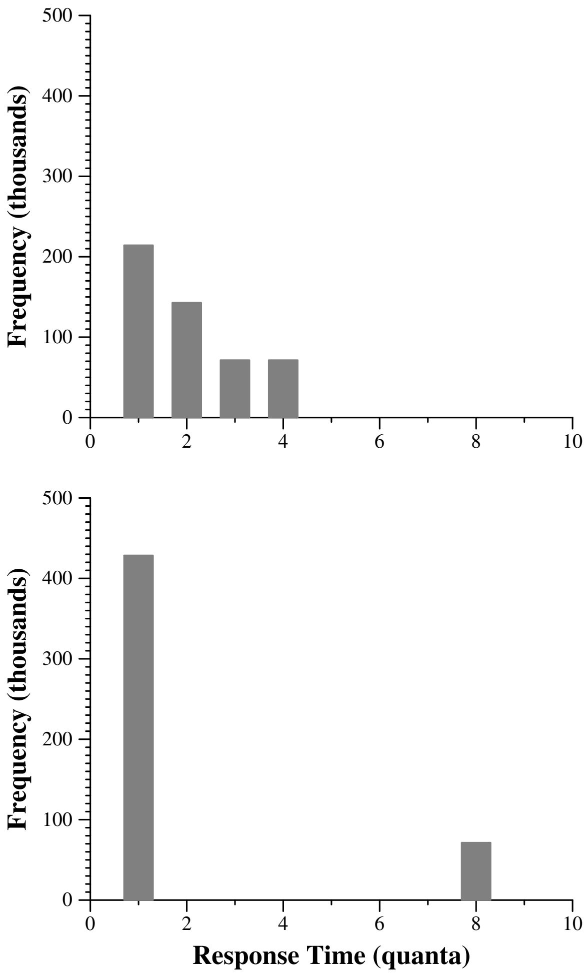

Figure 13: **Hierarchical Stride Scheduling.** Response time distributions for a simulation of eight clients with a 7 : 1 : . . . : 1 ticket ratio over one million allocations. Response times are shown only for the client with 7 tickets. (a) Hierarchical Stride Scheduler: $\mu = 2.00$, $\sigma = 1.07$. (b) Ordinary Stride Scheduler: $\mu = 2.00$, $\sigma = 2.45$.

6 Prototype Implementations

We implemented two prototype stride schedulers by modifying the Linux 1.1.50 kernel on a 25MHz i486-based IBM Thinkpad 350C. The first prototype enables proportional-share control over processor time, and the second enables proportional-share control over network transmission bandwidth.

6.1 Process Scheduler

The goal of our first prototype was to permit proportional-share allocation of processor time to control relative computation rates. We primarily changed the kernel code that handles process scheduling, switching from a conventional priority scheduler to a stride-based algorithm with a scheduling quantum of 100 milliseconds. Ticket allocations can be specified via a new

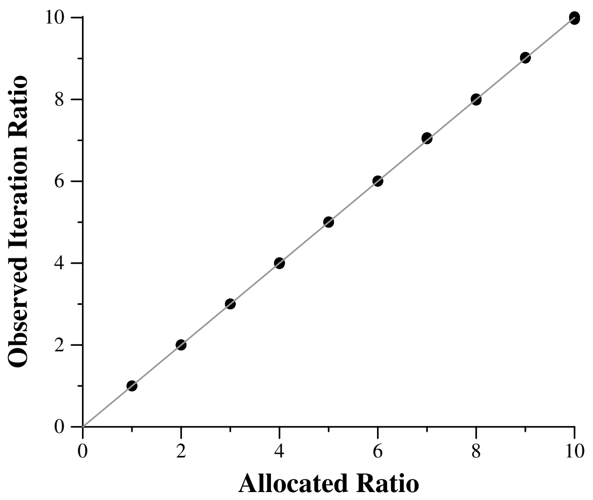

Figure 14: **CPU Rate Accuracy.** For each allocation ratio, the observed iteration ratio is plotted for each of three 30 second runs. The gray line indicates the ideal where the two ratios are identical. The observed ratios are within 1% of the ideal for all data points.

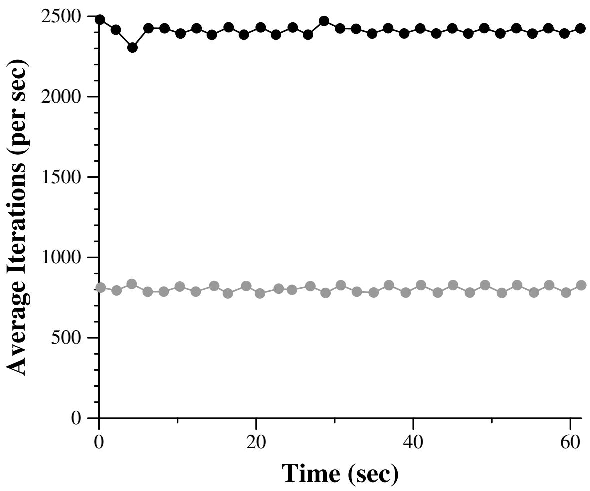

Figure 15: **CPU Fairness Over Time.** Two processes executing the compute-bound `arith` benchmark with a 3 : 1 ticket allocation. Averaged over the entire run, the two processes executed 2409.18 and 802.89 iterations/sec., for an actual ratio of 3.001:1.

`stride_cpu_set_tickets()` system call. We did not implement support for higher-level abstractions such as ticket transfers and currencies. Fewer than 300 lines of source code were added or modified to implement our changes.

Our first experiment tested the accuracy with which our prototype could control the relative execution rate of computations. Each point plotted in Figure 14 indicates the relative execution rate that was observed for two processes running the compute-bound `arith` integer arithmetic benchmark [Byt91]. Three thirty-second runs were executed for each integral ratio between one and ten. In all cases, the observed ratios are within 1% of the ideal. We also ran experiments involving higher ratios, and found that the observed ratio for a 20 : 1 allocation ranged from 19.94 to 20.04, and the observed ratio for a 50 : 1 allocation ranged from 49.93 to 50.44.

Our next experiment examined the scheduler's behavior over shorter time intervals. Figure 15 plots average iteration counts over a series of 2-second time windows during a single 60 second execution with a 3 : 1 allocation. The two processes remain close to their allocated ratios throughout the experiment. Note that if we used a 10 millisecond time quantum instead of the scheduler's 100 millisecond quantum, the same degree of fairness would be observed over a series of 200 millisecond time windows.

To assess the overhead imposed by our prototype stride scheduler, we ran performance tests consisting of concurrent `arith` benchmark processes. Overall, we found that the performance of our prototype was comparable to that of the standard Linux process scheduler. Compared to unmodified Linux, groups of 1, 2, 4, and 8 `arith` processes each completed fewer iterations under stride scheduling, but the difference was always less than 0.2%.

However, neither the standard Linux scheduler nor our prototype stride scheduler are particularly efficient. For example, the Linux scheduler performs a linear scan of all processes to find the one with the highest priority. Our prototype also performs a linear scan to find the process with the minimum pass; an $O(\lg n_c)$ time implementation would have required substantial changes to existing kernel code.

6.2 Network Device Scheduler

The goal of our second prototype was to permit proportional-share control over transmission bandwidth for network devices such as Ethernet and SLIP interfaces. Such control would be particularly useful for applications such as concurrent `ftp` file transfers, and concurrent `http` Web server replies. For example, many Web servers have relatively slow connections to the Internet, resulting in substantial delays for transfers of large objects such as graphical images. Given control over relative transmission rates, a Web server could provide different levels of service to concurrent clients. For example, tickets[8] could be issued by servers based upon the requesting user, machine, or domain. Commercial servers could even sell tickets to clients demanding faster service.

We primarily changed the kernel code that handles generic network device queueing. This involved switching from conventional FIFO queueing to stride-based queueing that respects per-socket ticket allocations. Ticket allocations can be specified via a new `SO_TICKETS` option to the `setsockopt()` system call. Although not implemented in our prototype, a more complete system should also consider additional forms of admission control to manage other system resources, such as network buffers. Fewer than 300 lines of source code were added or modified to implement our changes.

Our first experiment tested the prototype's ability to control relative network transmission rates on a local area network. We used the `ttcp` network test program[9] [TTC91] to transfer fabricated buffers from an IBM Thinkpad 350C running our modified Linux kernel, to a

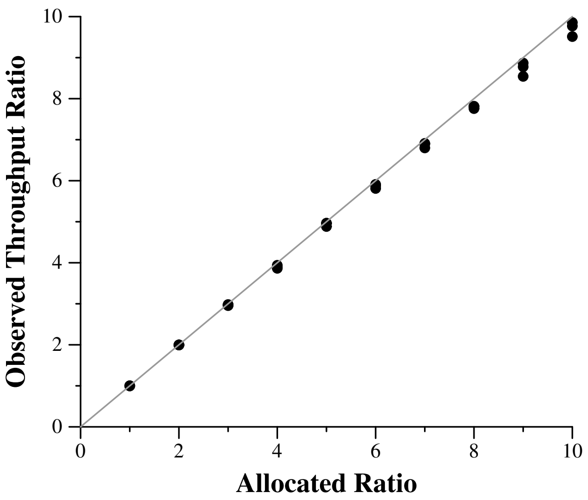

Figure 16: **Ethernet UDP Rate Accuracy.** For each allocation ratio, the observed data transmission ratio is plotted for each of three runs. The gray line indicates the ideal where the two ratios are identical. The observed ratios are within 5% of the ideal for all data points.

DECStation 5000/133 running Ultrix. Both machines were on the same physical subnet, connected via a 10Mbps Ethernet that also carried network traffic for other users.

Each point plotted in Figure 16 indicates the relative UDP data transmission rate that was observed for two processes running the `ttcp` benchmark. Each experiment started with both processes on the sending machine attempting to transmit 4K buffers, each containing 8Kbytes of data, for a total 32Mbyte transfer. As soon as one process finished sending its data, it terminated the other process via a Unix signal. Metrics were recorded on the receiving machine to capture end-to-end application throughput. The observed ratios are very accurate; all data points are within 5% of the ideal. For larger ticket ratios, the observed throughput ratio is slightly lower than the specified allocation. For example, a 20 : 1 allocation resulted in actual throughput ratios ranging from 18.51 : 1 to 18.77 : 1.

To assess the overhead imposed by our prototype, we ran performance tests consisting of concurrent `ttcp` benchmark processes. Overall, we found that the performance of our prototype was comparable to that of standard Linux. Although the prototype increases the length

[8] To be included with `http` requests, tickets would require an external data representation. If security is a concern, cryptographic techniques could be employed to prevent forgery and theft.

[9] We made a few minor modifications to the standard `ttcp` benchmark. Other than extensions to specify ticket allocations and facilitate coordinated timing, we also decreased the value of a hard-coded delay constant. This constant is used to temporarily put a transmitting process to sleep when it is unable to write to a socket due to a lack of buffer space (`ENOBUFS`). Without this modification, the observed throughput ratios were consistently lower than specified allocations, with significant differences for large ratios. With the larger delay constant, we believe that the low-throughput client is able to continue sending packets while the high-throughput client is sleeping, distorting the intended throughput ratio. Of course, changing the kernel interface to signal a process when more buffer space becomes available would probably be preferable to polling.

of the critical path for sending a network packet, we were unable to observe any significant difference between unmodified Linux and stride scheduling. We believe that the small additional overhead of stride scheduling was masked by the variability of external network traffic from other users; individual differences were in the range of $\pm 5\%$.

7 Related Work

We independently developed stride scheduling as a deterministic alternative to the randomized selection aspect of lottery scheduling [Wal94]. We then discovered that the core allocation algorithm used in stride scheduling is nearly identical to elements of rate-based flow-control algorithms designed for packet-switched networks [Dem90, Zha91, ZhK91, Par93]. Despite the relevance of this networking research, to the best of our knowledge it has not been discussed in the processor scheduling literature. In this section we discuss a variety of related scheduling work, including rate-based network flow control, deterministic proportional-share schedulers, priority schedulers, real-time schedulers, and microeconomic schedulers.

7.1 Rate-Based Network Flow Control

Our basic stride scheduling algorithm is very similar to Zhang's *VirtualClock* algorithm for packet-switched networks [Zha91]. In this scheme, a network switch orders packets to be forwarded through outgoing links. Every packet belongs to a client data stream, and each stream has an associated bandwidth reservation. A *virtual clock* is assigned to each stream, and each of its packets is stamped with its current virtual time upon arrival. With each arrival, the virtual clock advances by a *virtual tick* that is inversely proportional to the stream's reserved data rate. Using our stride-oriented terminology, a virtual tick is analogous to a *stride*, and a virtual clock is analogous to a *pass* value.

The VirtualClock algorithm is closely related to the *weighted fair queueing (WFQ)* algorithm developed by Demers, Keshav, and Shenker [Dem90], and Parekh and Gallager's equivalent *packet-by-packet generalized processor sharing (PGPS)* algorithm [Par93]. One difference that distinguishes WFQ and PGPS from VirtualClock is that they effectively maintain a *global* virtual clock. Arriving packets are stamped with their stream's virtual tick plus the *maximum* of their stream's virtual clock and the global virtual clock. Without this modification, an inactive stream can later monopolize a link as its virtual clock caught up to those of active streams; such behavior is possible under the VirtualClock algorithm [Par93].

Our stride scheduler's use of a *global_pass* variable is based on the global virtual clock employed by WFQ/PGPS, which follows an update rule that produces a smoothly varying global virtual time. Before we became aware of the WFQ/PGPS work, we used a simpler *global_pass* update rule: *global_pass* was set to the pass value of the client that currently owns the resource. To see the difference between these approaches, consider the set of minimum pass values over time in Figure 2. Although the *average* pass value increase per quantum is 1, the actual increases occur in non-uniform steps. We adopted the smoother WFQ/PGPS virtual time rule to improve the accuracy of pass updates associated with dynamic modifications.

To the best of our knowledge, our work on stride scheduling is the first cross-application of rate-based network flow control algorithms to scheduling other resources such as processor time. New techniques were required to support dynamic changes and higher-level abstractions such as ticket transfers and currencies. Our hierarchical stride scheduling algorithm is a novel recursive application of the basic technique that exhibits improved throughput accuracy and reduced response time variability compared to prior schemes.

7.2 Proportional-Share Schedulers

Several other deterministic approaches have recently been proposed for proportional-share processor scheduling [Fon95, Mah95, Sto95]. However, all require expensive operations to transform client state in response to dynamic changes. This makes them less attractive than stride scheduling for supporting dynamic or distributed environments. Moreover, although each algorithm is explicitly compared to lottery scheduling, none provides efficient support for the flexible resource management abstractions introduced with lottery scheduling.

Stoica and Abdel-Wahab have devised an interesting scheduler using a deterministic generator that employs a bit-reversed counter in place of the random number generator used by lottery scheduling [Sto95]. Their algorithm results in an absolute error for throughput that is $O(\lg n_a)$, where n_a is the number of allocations. Allocations can be performed efficiently in $O(\lg n_c)$ time using a tree-based data structure, where n_c is the number of clients. However, dynamic modifications to the set of active clients or their allocations require executing a relatively complex "restart" operation with $O(n_c)$ time complexity. Also, no support is provided for fractional or nonuniform quanta.

Maheshwari has developed a deterministic *charge-based* proportional-share scheduler [Mah95]. Loosely based on an analogy to digitized line drawing, this scheme has a maximum relative throughput error of one quantum, and also supports fractional quanta. Although efficient in many cases, allocation has a worst-case $O(n_c)$ time complexity, where n_c is the number of clients. Dynamic modifications require executing a "refund" operation with $O(n_c)$ time complexity.

Fong and Squillante have introduced a general scheduling approach called *time-function scheduling (TFS)* [Fon95]. TFS is intended to provide differential treatment of job classes, where specific throughput ratios are specified across classes, while jobs within each class are scheduled in a FCFS manner. Time functions are used to compute dynamic job priorities as a function of the time each job has spent waiting since it was placed on the run queue. Linear functions result in proportional sharing: a job's value is equal to its waiting time multipled by its job-class slope, plus a job-class constant. An allocation is performed by selecting the job with the maximum time-function value. A naive implementation would be very expensive, but since jobs are grouped into classes, allocation can be performed in $O(n)$ time, where n is the number of distinct classes. If time-function values are updated infrequently compared to the scheduling quantum, then a priority queue can be used to reduce the allocation cost to $O(\lg n)$, with an $O(n \lg n)$ cost to rebuild the queue after each update.

When Fong and Squillante compared TFS to lottery scheduling, they found that although throughput accuracy was comparable, the waiting time variance of low-throughput tasks was often several orders of magnitude larger under lottery scheduling. This observation is consistent with our simulation results involving response time, presented in Section 5. TFS also offers the potential to specify performance goals that are more general than proportional sharing. However, when proportional sharing is the goal, stride scheduling has advantages in terms of efficiency and accuracy.

7.3 Priority Schedulers

Conventional operating systems typically employ *priority* schemes for scheduling processes [Dei90, Tan92]. Priority schedulers are not designed to provide proportional-share control over relative computation rates, and are often ad-hoc. Even popular priority-based approaches such as *decay-usage scheduling* are poorly understood, despite the fact that they are employed by numerous operating systems, including Unix [Hel93].

Fair share schedulers allocate resources so that users get fair machine shares over long periods of time [Hen84, Kay88, Hel93]. These schedulers are layered on top of conventional priority schedulers, and dynamically adjust priorities to push actual usage closer to entitled shares. The algorithms used by these systems are generally complex, requiring periodic usage monitoring, complicated dynamic priority adjustments, and administrative parameter setting to ensure fairness on a time scale of minutes.

7.4 Real-Time Schedulers

Real-time schedulers are designed for time-critical systems [Bur91]. In these systems, which include many aerospace and military applications, timing requirements impose absolute deadlines that must be met to ensure correctness and safety; a missed deadline may have dire consequences. One of the most widely used techniques in real-time systems is *rate-monotonic scheduling*, in which priorities are statically assigned as a monotonic function of the rate of periodic tasks [Liu73, Sha91]. The importance of a task is not reflected in its priority; tasks with shorter periods are simply assigned higher priorities. Bounds on total processor utilization (ranging from 69% to nearly 100%, depending on various assumptions) ensure that rate monotonic scheduling will meet all task deadlines. Another popular technique is *earliest deadline scheduling*, which always schedules the task with the closest deadline first. The earliest deadline approach permits high processor utilization, but has increased runtime overhead due to the use of dynamic priorities; the task with the nearest deadline varies over time.

In general, real-time schedulers depend upon very restrictive assumptions, including precise static knowledge of task execution times and prohibitions on task interactions. In addition, limitations are placed on processor utilization, and even transient overloads are disallowed. In contrast, the proportional-share model used by stride scheduling and lottery scheduling is designed for more general-purpose environments. Task allocations degrade gracefully in overload situations, and active tasks proportionally benefit from extra resources when some allocations are not fully utilized. These properties facilitate adaptive applications that can respond to changes in resource availability.

Mercer, Savage, and Tokuda recently introduced a higher-level *processor capacity reserve* abstraction [Mer94] for measuring and controlling processor usage in a microkernel system with an underlying real-time scheduler. Reserves can be passed across protection boundaries during interprocess communication, with an effect similar to our use of ticket transfers. While this approach works well for many multimedia applications, its reliance on resource reservations and admission control is still more restrictive than the general-purpose model that we advocate.

7.5 Microeconomic Schedulers

Microeconomic schedulers are based on metaphors to resource allocation in real economic systems. *Money* encapsulates resource rights, and a *price* mechanism is used to allocate resources. Several microeconomic schedulers [Dre88, Mil88, Fer88, Fer89, Wal89, Wal92, Wel93] use auctions to determine prices and allocate resources among clients that bid monetary funds. Both the *escalator algorithm* proposed for uniprocessor scheduling [Dre88] and the distributed *Spawn* system [Wal92] rely upon auctions in which bidders increase their bids linearly over time. Since auction dynamics can be unexpectedly volatile, auction-based approaches sometimes fail to achieve resource allocations that are proportional to client funding. The overhead of bidding also limits the applicability of auctions to relatively coarse-grained tasks. Other market-based approaches that do not rely upon auctions have also been applied to managing processor and memory resources [Ell75, Har92, Che93].

Stride scheduling and lottery scheduling are compatible with a market-based resource management philosophy. Our mechanisms for proportional sharing provide a convenient substrate for pricing individual time-shared resources in a computational economy. For example, tickets are analogous to monetary income streams, and the number of tickets competing for a resource can be viewed as its price. Our currency abstraction for flexible resource management is also loosely borrowed from economics.

8 Conclusions

We have presented stride scheduling, a deterministic technique that provides accurate control over relative computation rates. Stride scheduling also efficiently supports the same flexible, modular resource management abstractions introduced by lottery scheduling. Compared to lottery scheduling, stride scheduling achieves significantly improved accuracy over relative throughput rates, with significantly less response time variability. However, lottery scheduling is conceptually simpler than stride scheduling. For example, stride scheduling requires careful state updates for dynamic changes, while lottery scheduling is effectively stateless.

The core allocation mechanism used by stride scheduling is based on rate-based flow-control algorithms for networks. One contribution of this paper is a cross-application of these algorithms to the domain of processor scheduling. New techniques were developed to support dynamic modifications to client allocations and resource right transfers between clients. We also introduced a new hierarchical stride scheduling algorithm that exhibits improved throughput accuracy and lower response time variability compared to prior schemes.

Acknowledgements

We would like to thank Kavita Bala, Dawson Engler, Paige Parsons, and Lyle Ramshaw for their many helpful comments. Thanks to Tom Rodeheffer for suggesting the connection between our work and rate-based flow-control algorithms in the networking literature. Special thanks to Paige for her help with the visual presentation of stride scheduling.

References

[Bur91] A. Burns. “Scheduling Hard Real-Time Systems: A Review,” *Software Engineering Journal*, May 1991.

[Byt91] Byte Unix Benchmarks, Version 3, 1991. Available via Usenet and anonymous `ftp` from many locations, including `gatekeeper.dec.com`.

[Che93] D. R. Cheriton and K. Harty. “A Market Approach to Operating System Memory Allocation,” Working Paper, Computer Science Department, Stanford University, June 1993.

[Cor90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[Dei90] H. M. Deitel. *Operating Systems*, Addison-Wesley, 1990.

[Dem90] A. Demers, S. Kehav, and S. Shenker. “Analysis and Simulation of a Fair Queueing Algorithm,” *Internetworking: Research and Experience*, September 1990.

[Dre88] K. E. Drexler and M. S. Miller. “Incentive Engineering for Computational Resource Management” in *The Ecology of Computation*, B. Huberman (ed.), North-Holland, 1988.

[Ell75] C. M. Ellison. “The Utah TENEX Scheduler,” *Proceedings of the IEEE*, June 1975.

[Fer88] D. Ferguson, Y. Yemini, and C. Nikolaou. “Microeconomic Algorithms for Load-Balancing in Distributed Computer Systems,” *International Conference on Distributed Computer Systems*, 1988.

[Fer89] D. F. Ferguson. “The Application of Microeconomics to the Design of Resource Allocation and Control Algorithms,” Ph.D. thesis, Columbia University, 1989.

[Fon95] L. L. Fong and M. S. Squillante. “Time-Functions: A General Approach to Controllable Resource Management,” Working Draft, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, March 1995.

[Har92] K. Harty and D. R. Cheriton. “Application-Controlled Physical Memory using External Page-Cache Management,” *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[Hel93] J. L. Hellerstein. “Achieving Service Rate Objectives with Decay Usage Scheduling,” *IEEE Transactions on Software Engineering*, August 1993.

[Hen84] G. J. Henry. “The Fair Share Scheduler,” *AT&T Bell Laboratories Technical Journal*, October 1984.

[Kay88] J. Kay and P. Lauder. “A Fair Share Scheduler,” *Communications of the ACM*, January 1988.

[Liu73] C. L. Liu and J. W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *Journal of the ACM*, January 1973.

[Mah95] U. Maheshwari. “Charge-Based Proportional Scheduling,” Working Draft, MIT Laboratory for Computer Science, Cambridge, MA, February 1995.

[Mer94] C. W. Mercer, S. Savage, and H. Tokuda. “Processor Capacity Reserves: Operating System Support for Multimedia Applications,” *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[Mil88] M. S. Miller and K. E. Drexler. “Markets and Computation: Agoric Open Systems,” in *The Ecology of Computation*, B. Huberman (ed.), North-Holland, 1988.

[Par93] A. K. Parekh and R. G. Gallager. “A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case,” *IEEE/ACM Transactions on Networking*, June 1993.

[Pug90] W. Pugh. “Skip Lists: A Probabilistic Alternative to Balanced Trees,” *Communications of the ACM*, June 1990.

[Sha91] L. Sha, M. H. Klein, and J. B. Goodenough. “Rate Monotonic Analysis for Real-Time Systems,” in *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. M. van Tilborg and G. M. Koob (eds.), Kluwer Academic Publishers, 1991.

[Sto95] 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, Norfolk, VA, April 1995.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems*, Prentice Hall, 1992.

[TTC91] TTCP benchmarking tool. SGI version, 1991. Originally developed at the US Army Ballistics Research Lab (BRL). Available via anonymous `ftp` from many locations, including `ftp.sgi.com`.

[Wal89] C. A. Waldspurger. "A Distributed Computational Economy for Utilizing Idle Resources," Master's thesis, MIT, May 1989.

[Wal92] 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*, February 1992.

[Wal94] C. A. Waldspurger and W. E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[Wal95] C. A. Waldspurger. "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," Ph.D. thesis, MIT, 1995 (to appear).

[Wel93] M. P. Wellman. "A Market-Oriented Programming Environment and its Application to Distributed Multicommodity Flow Problems," *Journal of Artificial Intelligence Research*, August 1993.

[Zha91] L. Zhang. "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," *ACM Transactions on Computer Systems*, May 1991.

[ZhK91] H. Zhang and S. Kehav. "Comparison of Rate-Based Service Disciplines," *Proceedings of SIGCOMM '91*, September 1991.

A Fixed-Point Stride Representation

The precision of relative rates that can be achieved depends on both the value of $stride_1$ and the relative ratios of client ticket allocations. For example, with $stride_1 = 2^{20}$, and a maximum ticket allocation of 2^{10} tickets, ratios are represented with 10 bits of precision. Thus, ratios close to unity resulting from allocations that differ by only one part per thousand, such as $1001 : 1000$, can be supported.

Since $stride_1$ is a large integer, *stride* values will also be large for clients with small allocations. Since *pass* values are monotonically increasing, they will eventually overflow the machine word size after a large number of allocations. For a machine with 64-bit integers, this is not a practical problem. For example, with $stride_1 = 2^{20}$ and a worst-case client $tickets = 1$, approximately 2^{44} allocations can be performed before overflow occurs. At one allocation per millisecond, centuries of real time would elapse before an overflow.

For a machine with 32-bit integers, the *pass* values associated with all clients can be adjusted by subtracting the minimum pass value from all clients whenever an overflow is detected. Alternatively, such adjustments can periodically be made after a fixed number of allocations. For example, with $stride_1 = 2^{20}$, a conservative adjustment period would be a few thousand allocations. Perhaps the most straightforward approach is to simply use a 64-bit integer type if one is available. Our prototype implementation makes use of the 64-bit "`long long`" integer type provided by the GNU C compiler.