Proceedings of 2000 USENIX Annual Technical Conference

San Diego, California, USA, June 18–23, 2000

ISOLATION WITH FLEXIBILITY: A RESOURCE MANAGEMENT FRAMEWORK FOR CENTRAL SERVERS

David G. Sullivan and Margo I. Seltzer

Isolation with Flexibility: A Resource Management Framework for Central Servers

David G. Sullivan, Margo I. Seltzer
Division of Engineering and Applied Sciences
Harvard University, Cambridge, MA 02138
{sullivan,margo}@eecs.harvard.edu

Abstract

Proportional-share resource management is becoming increasingly important in today's computing environments. In particular, the growing use of the computational resources of central service providers argues for a proportional-share approach that allows resource principals to obtain allocations that reflect their relative importance. In such environments, resource principals must be isolated from one another to prevent the activities of one principal from impinging on the resource rights of others. However, such isolation limits the flexibility with which resource allocations can be modified to reflect the actual needs of applications. We present extensions to the lottery-scheduling resource management framework that increase its flexibility while preserving its ability to provide secure isolation. To demonstrate how this extended framework safely overcomes the limits imposed by existing proportional-share schemes, we have implemented a prototype system that uses the framework to manage CPU time, physical memory, and disk bandwidth. We present the results of experiments that evaluate the prototype, and we show that our framework has the potential to enable server applications to achieve significant gains in performance.

1 Introduction

In managing computational resources, an operating system must balance a variety of goals, including maximizing resource utilization, minimizing latency, and providing fairness. The relative importance of these goals for a particular system depends on the nature of the system and the ways in which it is used. For supercomputers running compute-intensive applications, the primary goal may be to maximize throughput, while for personal computers used to enhance a single user's productivity, the chief goal may be to maximize responsiveness.

In today's computing environments, users increasingly compete for the resources of server systems, whether to access central databases or to view content on virtually-hosted Web sites. On such systems, fairness becomes a critical resource-management goal. Proportional-share mechanisms allow this goal to be met by providing resource principals (users, applications, threads, etc.) with guaranteed resource rights. For example, customers who pay Internet service providers to virtually host their Web sites can be given rights to shares of the hosting machine that are commensurate with the prices they pay. Service providers who can make such guarantees can offer larger resource shares to principals willing to pay a premium for better quality of service.

Although its full promise is yet to be realized, thin-client computing is another domain in which proportional-share resource management is desirable. Administrators of such systems are often forced to host one application per server to provide predictable levels of service [Sun98]. Proportional-share techniques enable the consolidation of multiple applications onto a single server by giving each application a dedicated share of the machine.

A system that supports proportional-share resource management must *isolate* resource principals from each other, so that a given principal's resource rights are protected from the activities of other principals. To provide such isolation, a system must necessarily impose limits on the flexibility with which resource allocations can be modified. Such limits work well when the resource needs of applications are well-known and unchanging, because a system administrator can assign the appropriate resource shares and leave the system to run. Unfortunately, these conditions frequently do not hold. Even if the applications' current resource needs are adequately understood, they will typically change over time. For example, as a Web site's working set of frequently accessed documents expands, the site may require an increasing share of the server's disk bandwidth in order to offer reasonable responsiveness. Moreover, it would be preferable if system administrators could be freed from the need to make detailed characterizations of applications' resource needs. Ideally, the applications themselves should be able to modify their own resource rights in response to their needs and the current state of the system.

In this paper, we present extensions to the lottery-scheduling resource management framework [Wal94, Wal95, Wal96] that allow resource principals to safely overcome the limits on flexible allocation that proportional-share frameworks impose for the sake of secure isolation. Our extended framework supports both absolute resource reservations (*hard shares*) and proportional allocations that change in size as resource principals enter and leave the competition for a resource (*soft shares*). It also introduces a system of access controls to protect the isolation properties that lottery scheduling provides. And our framework offers the means for applications to modify their own resource rights without compromising the rights of other resource principals. One of these mechanisms, called *ticket exchanges,* allows applications to coordinate their use of the system's resources by bartering over resource rights with each other. Our extended framework thereby provides *isolation with increased flexibility*: the flexibility to safely overcome the limits on resource allocation that standard proportional-share frameworks enforce.

We have developed a prototype implementation of our framework in the VINO operating system [Sel96] and have used it, in conjunction with several proportional-share mechanisms, to manage CPU time, physical memory, and disk bandwidth. Our experiments demonstrate that the extended lottery-scheduling framework enables server applications to achieve improved performance under realistic usage scenarios.

This work makes several contributions. First, we extend the lottery-scheduling framework to securely manage multiple resources, providing both soft and hard resource shares. To our knowledge, our prototype is the first implementation of a proportional-share framework to support both types of shares for multiple resources. Second, we point out an important tension between the conflicting goals of secure isolation and flexible resource allocation, and we present mechanisms that allow for more flexible allocation while preserving secure isolation. Third, we illustrate the value of a system that can support dynamic adjustments to the resource allocations that applications receive.

In the next section, we review the original lottery-scheduling framework and describe how we extend it to securely support proportional sharing of multiple resources. In Section 3, we illustrate how lottery scheduling (like all proportional-share schemes) imposes both upper and lower limits on the resource allocations that clients can obtain, and we describe the mechanisms that we use to overcome both sets of limits while maintaining secure isolation. In Section 4, we describe our prototype implementation of the extended framework, including the scheduling mechanisms that we have chosen to employ. Section 5 presents experiments designed to evaluate the prototype and to test one of our mechanisms for flexibly adjusting resource rights. Finally, we discuss related work and summarize our conclusions.

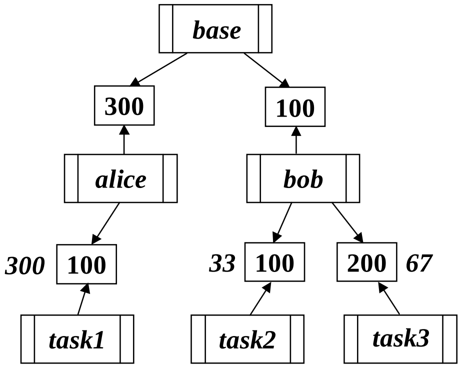

Figure 1. A sample resource hierarchy in which currencies provide isolation between the tasks of different users. The base values of the tasks' backing tickets are shown in italics.

2 Securely Managing Multiple Resources

2.1 The Original Framework

The resource management framework developed for lottery scheduling [Wal94, Wal95, Wal96] is based on two key abstractions, *tickets* and *currencies*. Tickets are used to encapsulate resource rights. Resource principals receive resource rights that are proportional to the number of tickets that they hold for a resource; changing the number of tickets held by a resource principal automatically leads to a change in its resource rights.

Tickets are issued by currencies, which allow resource principals to be grouped together and isolated from each other. Principals funded by a currency share the resource rights allotted to that currency; currencies thus enable hierarchical resource management.

Each currency effectively maintains its own exchange rate with a central *base currency,* and tickets from different currencies can be compared by determining their value with respect to the base currency (their *base value*). The more tickets a currency issues, the less each ticket is worth with respect to the base currency, and their total base value can never exceed the value of the tickets used to back the currency itself.

The sample currency hierarchy shown in Figure 1 illustrates these concepts. The *bob* currency is funded by 100 of the 400 base-currency tickets, and it thus receives rights to one-quarter of the resource. These rights are divided up by the tasks funded by *bob;* for example, *task3* holds 200 of the 300 *bob* tickets, and it thus receives rights to two-thirds of *bob*'s quarter share, or one-sixth of the total resource rights. In other words, *task3*'s 200 *bob* tickets have a base value of approximately 67 (two thirds of 100). If *task2* or *task3* forks off more tasks, causing the *bob* currency to issue more tickets, the value of its tickets will decrease, because its resource rights will be shared by a larger number of tasks. However, the resource rights of processes funded by other currencies will not be affected.

While a lottery-scheduling resource hierarchy typically has a tree-shaped structure like the one shown in Figure 1, it can more generally take the form of any directed acyclic graph. The lottery-scheduling framework thus supports a greater variety of configurations than most other, recently proposed schemes for hierarchical resource management (see Section 6). For example, on a system like the one depicted in Figure 1, in which each user's applications are funded by a currency specific to that user, two or more users could pool their resources to support a single application that all of them are using (the system developed by Banga et al. [Ban99] also allows this).

2.2 Resource-Specific Tickets

Although prior implementations of lottery scheduling have focused exclusively on single resources (primarily the CPU), the original lottery-scheduling framework was designed to support multiple resources. Waldspurger [Wal95] considered two approaches to implementing a multi-resource system. In the first approach, tickets can be applied to any resource, allowing resource principals to shift tickets from one resource to another as needed, while in the second, tickets are resource-specific. Waldspurger favored the former approach because of its greater flexibility and simplicity. However, allowing principals to devote tickets to resources as they see fit violates the insulation properties of currencies, because it can lead to changes in the total number of tickets applied toward a given resource [Sul99a].

We therefore chose to use resource-specific tickets. To avoid the overhead of maintaining a separate currency configuration for each resource, we extend currencies to encompass all of the resources being managed. Concretely, this means that most pieces of currency state are maintained as arrays indexed by resource type. Similarly, many currency-related operations take a parameter that specifies the resource type.

2.3 Currency Brokers

For the lottery-scheduling framework to be secure in a multi-user setting, a system of access controls are needed. We encapsulate these controls in a *broker* associated with each currency. A broker stores the owner and group of the user who created the currency, along with a UNIX-style mode specifying who may perform various operations on the currency. Before these operations are carried out, the broker verifies that the current thread belongs to a user with the requisite permissions.

Like UNIX file modes, currency modes include three sets of permissions: one for the currency's owner, one for the currency's group, and one for all others. In a given set of permissions, the *f* bit indicates whether a user is allowed to fund the currency; the *c* bit indicates whether a user can "change" the currency by removing some of its funding or destroying it entirely; and the *i* bit indicates if a user is allowed to issue or revoke the currency's own tickets. This *fci* collection of bits is comparable to the *rwx* combination in UNIX file modes.

Table 1 provides more specifics about the permission checks that brokers perform. In most cases, superusers are allowed to override the ordinary permissions checks. If an attempt to fund a currency would lead to a cycle in the currency graph, the attempt is rejected.

Table 1. Permission checks performed by brokers

Operation	**Permission check**
create a currency	The caller must match both the user id and group id specified for the new currency[a].
destroy a currency	The appropriate *c* (*change*) bit must be set in the currency's mode.
fund currency A with tickets issued by currency B	The appropriate *f* (*fund*) bit in A's mode **and** the appropriate *i* (*issue*) bit in B's mode must be set.
take tickets issued by currency B away from currency A	Either the appropriate *c* (*change*) bit in A's mode **or** the appropriate *i* (*issue*) bit in B's mode must be set.

[a] Note that the user and group ids must be specified—and therefore checked—because superusers can create currencies that have ids other than their own.

2.4 Hard and Soft Resource Shares

The standard lottery-scheduling framework was primarily designed to support *soft* resource shares whose absolute value may change over time as principals enter and leave the competition for the resource. However, Waldspurger and Weihl pointed out that absolute, *hard* resource shares can be supported using the same framework by fixing the total number of tickets issued by the system [Wal96]. In particular, they proposed specifying hard shares by issuing tickets from a *hard currency* that maintains a fixed exchange rate with the base currency. When this hard currency issues additional tickets, some of the funding of other, "soft" currencies is transferred to the hard currency so that its exchange rate can be maintained.

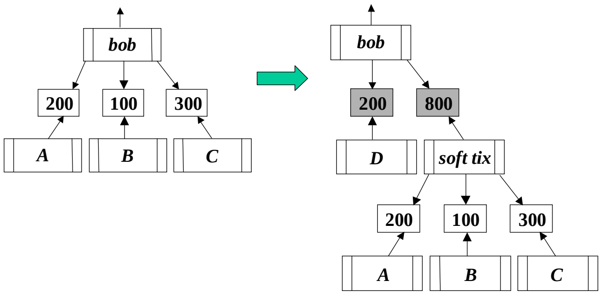

Figure 2: Offering Hard Shares of a Currency's Resource Rights. The *bob* currency issues a hard ticket to task D representing a fixed 20% (200/1000) of *bob*'s resource rights. As a result, a special currency (*soft_tix*) is created and used to fund *bob*'s soft tickets, isolating the hard tickets from changes in the number of soft tickets. The funding given to the *soft_tix* currency is adjusted as needed to ensure that the total number of hard tickets issued by *bob* remains fixed at 1000.

In our framework, we take a slightly different approach based on the notion of hard and soft *tickets*, and we allow resource principals to obtain hard shares from any currency. Under our approach, tickets issued by a currency are ordinarily *soft tickets* that specify soft shares of the currency's resource rights. However, when a currency issues a *hard ticket* to specify a fixed percentage of its resource rights, a separate currency is created and used to fund the currency's soft tickets (Fig. 2). The number of hard tickets used to fund this soft-ticket currency is adjusted as needed to ensure that the total number of the currency's hard tickets remains fixed.[1]

Our approach requires no extra overhead in the common case of a currency issuing only soft tickets, and yet it still allows hard tickets to be issued by any currency. Users could use hard tickets to give an application a fixed percentage of their resource rights, or to specify hierarchical reservations in which absolute shares from the base currency are divided into hard subshares. For a hard ticket to represent a fixed-share reservation of the actual resource, all paths from the root currency to the ticket must involve only hard tickets.

3 Isolation with Greater Flexibility

Currencies, like all mechanisms for providing isolation, necessarily impose limits on the flexibility with which resource allocations can be modified. In the following sections, we demonstrate that currencies enforce both upper *and* lower limits on resource allocations. We also describe the mechanisms that we have developed to safely overcome these limits so that applications can obtain allocations that better meet their differing and dynamically changing needs.

3.1 Problem: Currencies Impose Upper Limits

When a resource principal is funded by a currency other than the root currency, its resource rights can usually be increased by giving it extra tickets from that currency.[2] For example, in Figure 1, *task2*'s resource rights could be boosted by giving it 200 *bob* tickets rather than 100. However, doubling the tickets held by *task2* does not double its resource rights; rather, *task2* goes from having one-third of the *bob* currency's overall resource rights (a base value of 33) to having one-half of those rights (a base value of 50). This smaller increase reflects the fact that issuing additional *bob* tickets decreases their value. No matter how many currency tickets a resource principal receives, the resource rights imparted by those tickets cannot exceed the overall rights given to the currency itself. This upper limit is essential to providing isolation. Without it, the resource rights of principals funded by other currencies could be reduced.

Despite the need for the upper limits imposed by currencies, these limits may often be unnecessarily restrictive. This is especially true on central servers, because the large number of resource principals that a server must accommodate makes it difficult for a single allocation policy to adequately address their different and dynamically changing resource needs. Instead, some simple policy for ensuring fairness is likely to be used, such as giving users equal resource rights to divide among their applications, or allocating resource shares based on how much a user has paid.

3.2 Solution: Ticket Exchanges

Because certain resources may be more important than others to the performance of an application, applications may benefit from giving up a fraction of their resource rights for one resource in order to receive a larger share of another resource. We have therefore developed a mechanism called *ticket exchanges* that allows applications to take advantage of their differing resource needs by bartering with each other over resource-specific tickets. For example, a CPU-intensive application could exchange some of its disk tickets for some of the CPU tickets of an I/O-intensive application.

While ticket exchanges allow principals to obtain additional resource rights, they do so without compromising the isolation properties of the lottery-scheduling framework. As the scenario depicted in Figure 3 illustrates, only the resource rights of principals participating in an exchange are affected by it; the resource rights of non-participants remain the same.

[1] Note that even the base currency's soft tickets have a base value that can change over time as the number of its hard tickets changes.

[2] This is not always the case. If a resource principal is the sole recipient of a currency's tickets, giving it more tickets from the currency does not affect its resource rights.

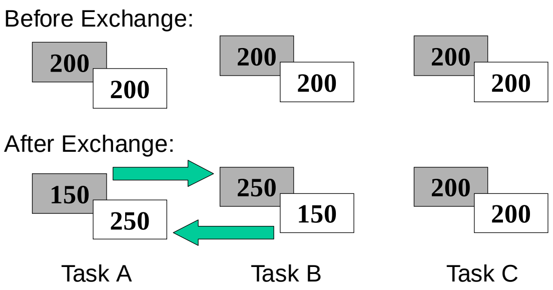

Figure 3: Ticket Exchanges Insulate Non-Participants. Tasks A and B exchange tickets. Task C is unaffected, because it still has one-third of the total of each ticket type.

Ticket exchanges are not, however, guaranteed to preserve the actual resource *shares* that non-participants received before the exchange. Because the principals involved in an exchange typically make greater use of the resource for which they obtain extra tickets than the principal who traded the tickets away, resource contention will likely increase. As a result, non-participants who previously received larger resource shares than their tickets guaranteed may see those shares reduced. For example, if a CPU-intensive process trades some of its disk tickets to a process that regularly accesses the disk, those previously inactive disk tickets will suddenly become active, and the disk tickets of other processes accessing the disk may decline in value.[3] However, principals should always receive at least the minimal shares to which their tickets entitle them.

Ticket exchanges and currencies complement each other. Exchanges allow for greater flexibility in the face of the upper limits imposed by currencies, while currencies insulate processes from the malicious use of exchanges. For example, a process could fork off children that use exchanges to give the parent process all of their tickets. With currencies, however, this tactic would only affect the resource rights of tasks funded by the same currency as the malicious process.

3.2.1 Determining and Coordinating Exchanges. Ticket exchanges enable applications to coordinate with each other in ways that are mutually beneficial and that may increase the overall efficiency of the system. Various levels of sophistication could be employed by applications to determine what types of exchanges they are willing to make and at what rates of exchange.

Certain types of resource principals may primarily need extra tickets for one particular resource. For example, consider two Web sites that are virtually hosted on the same server. Site A has a small number of frequently accessed files that it could keep in memory if it had additional memory tickets for its currency. Site B has a uniformly accessed working set that is too large to fit in memory; it would benefit from giving up some of its currency's memory tickets for some of A's disk tickets.

Applications could also apply economic and decision-theoretic models to determine, based on information about their performance (such as how often they are scheduled and how many page faults they incur) and the current state of the system, when to initiate an exchange and at what rate. This determination could be made by the application process itself, or by a separate *resource negotiator* process that monitors the relevant variables and initiates exchanges on the application's behalf. Resource negotiators are similar to the *application managers* proposed by Waldspurger [Wal95].

Applications are free to cancel exchanges in which they are involved. This allows them to take a trial-and-error approach, experimenting with exchange rates until they achieve an acceptable level of performance and adapting their resource usage over time.

Applications or their negotiators initiate exchanges by sending the appropriate information to a central *dealer*. The dealer maintains queues of outstanding exchange proposals, attempts to match up complementary requests, and carries out the resulting exchanges. If an exchange request cannot be immediately satisfied, the dealer returns a message that includes any proposals with conflicting exchange rates (e.g., process A requests 20 CPU tickets for 10 memory tickets, while process B requests 10 memory tickets for 10 CPU tickets). In this way, an application can decide whether to modify its proposed exchange rate and try again for a compromise deal. In environments where isolation is less important, the dealer could be modified to carry out exchanges that processes propose on the processes themselves (e.g., to take away 20 CPU tickets from a process and give it 20 memory tickets in return), giving an approach equivalent to the one suggested by Waldspurger (see Section 2.2).

Future research is needed to develop negotiators suitable for a wide variety of applications and environments. Among the questions that still need to be addressed are: How can a negotiator determine what exchanges are beneficial to its associated process? When should a negotiator accept a trade less desirable than the one it proposed? Will a system involving dynamic ticket exchanges be stable (i.e., how can oscillatory behavior be avoided)? Can general-purpose negotiators be written that avoid the need to craft one for each application? In addition, the central dealer must be designed to deal fairly with requests that have complementary but differing exchange rates.

[3] When a resource principal temporarily leaves the competition for a resource (e.g., when a thread is not performing I/O), its tickets are *deactivated*. As a result, the resource rights of other principals funded by the same currency or currencies are temporarily increased until the principal reenters the competition and its tickets are reactivated.

3.2.2 Carrying Out an Exchange. Once a complementary set of exchange requests is found, the funding of the resource principals involved must be modified to reflect the exchange. In a non-hierarchical system with only a base currency, this could be accomplished by directly modifying the number of tickets that the principals hold for the resources involved. However, the presence of non-base currencies complicates matters, because the base value of their tickets can change over time, whereas tickets used in exchanges should have a constant value.

To address this problem, we issue four tickets directly from the base currency: two for the amounts being exchanged, and two *negatively valued* tickets for the opposite amounts. For example, if task A trades disk tickets with a base value of 50 for some of task B's memory tickets with a base value of 20, then A is given 20 base-currency memory tickets and –50 base-currency disk tickets, and B is given 50 base-currency disk tickets and –20 base-currency memory tickets. Because scheduling and allocation decisions are based on the total base value of a principal's backing tickets, the negative tickets reduce the principals' resource rights by the amount they have traded away. And because principals cannot manipulate their own backing tickets, exchanged tickets cannot be misused (e.g., to reduce another currency's allocations). If a principal's tickets for a resource are temporarily deactivated (see footnote 3), the funding it has obtained through exchanges for that resource is temporarily transferred to the principal's other funders to preserve isolation.

An exchange is undone if one of the exchanging principals exits, cancels the exchange, or loses the resource rights that it traded away. This last scenario can occur if the tickets funding the principal decrease in value to the point that their base value is less than the value of the tickets that the principal gave away. In such cases, the principal's total base value for that resource becomes negative, and the exchange must be revoked.

3.3 Problem: Currencies Impose Lower Limits

Currencies can also impose lower limits on resource rights. These limits materialize when only one of the resource principals funded by a currency is actively competing for a resource. In such circumstances, that principal receives *all* of the currency's resource rights, no matter how few tickets have been used to fund it.

As a result, currencies make it difficult for lottery scheduling to support the semantics of the `nice` utility found on conventional UNIX systems. For example, a user running a CPU-intensive job may reduce its CPU funding as a favor to other users. But if the other tasks funded by the same currency are all idle, the CPU-intensive job will still get the currency's full CPU share (Fig. 4, *top*). The user would presumably be allowed to decrease the number of tickets backing the user currency itself, but then other jobs funded by that currency would be adversely affected when they became runnable.

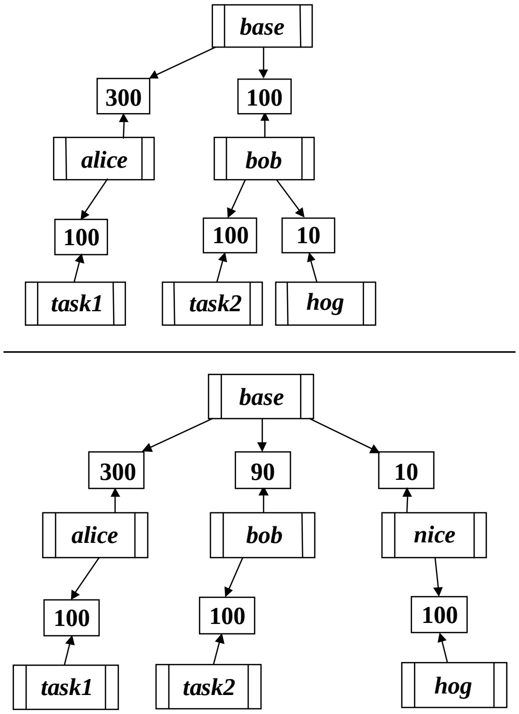

Figure 4: Currencies Impose Lower Limits. The user *bob* tries to lower the priority of *hog*, a CPU-intensive process, by giving it only 10 tickets (*top*). However, if *task2* becomes idle, *hog* will still receive all of *bob*'s resource rights. One solution is to fund *hog* directly from the base currency (*bottom*).

3.4 Solution: Limited Permission to Issue Base-Currency Tickets

While upper limits are necessary for providing isolation, lower limits are an undesirable side-effect of isolation. These limits could be overcome by funding CPU-intensive applications directly from the base currency (Fig. 4, *bottom*). However, for reasons of security, currency access controls (Section 2.3) would ordinarily prevent unprivileged users from issuing base-currency tickets.

To circumvent this restriction, our framework allows a user to issue base-currency tickets as long as the total value of currencies owned by that user never exceeds some upper bound. In this way, users can give up a small amount of their currencies' funding and then issue that same amount from the base currency to fund resource-intensive jobs. This approach leaves the user's total resource rights unchanged (preserving the isolation of other users), and it allows jobs to run at a reduced priority without crippling the rest of the user's applications. Section 4.6 describes how we implement this policy.

4 Prototype Implementation

We have implemented the extended framework in VINO-0.50 (`www.eecs.harvard.edu/~vino/vino`), and we use it to manage CPU time, memory, and disk bandwidth. In the following sections, we describe the key details of our implementation, including the scheduling and allocation mechanisms that we employ.

4.1 Threads and Currencies

A thread can be thought of as a special type of currency. Like all currencies, threads hold backing tickets, and they also issue temporary *transfer tickets* to threads on which they are waiting (see Section 4.3). By having VINO's thread class inherit from its currency class[4], threads can issue and receive tickets using the same methods as non-thread currencies, and other methods (e.g., the one that recursively computes a ticket's base value) can also treat threads and currencies interchangeably. Threads also reuse currency data members to keep track of the tickets that they hold and have issued.

Currencies that are also threads are identified by the process id of the thread. All other currencies are identified by a unique currency identifier (cid).

4.2 Currency Configuration and Permissions

By default, the system creates one currency for each active user of the system, and it funds these *user currencies* equally from the base currency. Each user's currency in turn funds the tasks of that user. The user currencies are created by means of a function, `cid_for_client()`, that is invoked when a process changes its real user id (uid); this function uses a uid to cid mapping to determine which currency should be used to fund the process. If no mapping exists for a given user, a new currency is created and funded. In either case, the process' existing funding is revoked and replaced with funding from the appropriate user currency. Once a user's login shell is correctly funded, processes forked by that shell are funded by the same currency. More generally, child processes are funded by the issuer of the first ticket in the parent's list of backing tickets for that resource.[5]

Each user currency is owned by the corresponding user and has a currency mode (see Section 2.3) that allows the user to manipulate it using a set of system calls added for this purpose. A user's tasks receive equal ticket allocations by default; the user can modify these allocations and thereby alter the relative resource rights of the tasks. Users can also create currencies and fund them with tickets from their user currency. However, they cannot increase the funding of their user currency itself, because they do not have permission to issue other currencies' tickets. Thus, each user's tasks are securely isolated from the tasks of other users, and each user has the same total resource rights.

Other currency-configuration policies could also be specified. On extensible operating systems like VINO [Sma98], superusers could safely download specialized versions of the `cid_for_client` function (which is also called when a process' real group id (gid) changes) to specify arbitrary configuration schemes based on uid and gid, as well as alternative access-control policies for currencies. Approaches that do not involve extensibility could also be employed to accommodate a more limited range of possible configurations and access controls.

4.3 Managing CPU Time

Our prototype uses the original lottery-scheduling algorithm [Wal94] to schedule the CPU, randomly choosing an active ticket and traversing the runnable queue to find the thread that holds the ticket. In searching for the winning thread, the system computes the base value of each thread's CPU backing tickets. We cache these base values to avoid unnecessary computation, although the cached values must be invalidated whenever a change occurs in a currency's count of active tickets (e.g., when a thread starts up, blocks, or exits).

Our prototype also uses two other features of the original lottery-scheduling framework, *compensation tickets* and *ticket transfers*. Compensation tickets are issued to threads who do not use their full quantum, temporarily inflating their resource rights to give them a higher probability of being chosen when they next become runnable. Ticket transfers occur when a thread blocks attempting to acquire a kernel mutex or to allocate memory. Because the thread is itself a currency (see Section 4.1), it issues a ticket and uses it to fund the thread on which it is waiting (the holder of the mutex or the pageout daemon), transferring its resource rights to that thread. This can reduce the time that the waiting thread spends blocked, and it prevents priority inversion from occurring. When the thread is made runnable again, its transfer tickets are revoked.

A number of deterministic algorithms, including stride scheduling [Wal95] and EEVDF [Sto96], can also be used within the lottery-scheduling framework to provide increased accuracy and lower response-time variability for interactive processes. Because our work primarily addresses ways of overcoming the limits that proportional-share frameworks impose on flexible resource allocation, the algorithms used are not crucial.

[4] VINO is written in C++.

[5] Actually, if a process holds tickets from more than one currency, its children should be funded by all of these currencies. We plan to extend our implementation to deal with this case.

4.4 Managing Memory

Effective proportional-share memory management is complicated by the difficulty of determining which processes are actively competing for memory and by the undesirability of a strict partitioning of memory among processes. When scheduling the CPU, threads that are blocked are simply ignored and the values of their tickets are not counted. Our data [Sul99b] indicates that a similar approach to memory management is not effective. When the system experiences heavy memory pressure, any process that blocks, even momentarily, can lose a large number of pages to the activity of the pageout daemon, resulting in erratic paging behavior and poor throughput. The obvious alternative, namely leaving the memory tickets of all processes active at all times, is also not viable, because pages belonging to idle processes tend to remain in memory indefinitely, reducing the number of pages available to active processes and effectively partitioning the total memory of the system. We have therefore chosen to give memory guarantees only to privileged processes that explicitly request them. Other processes compete for the unreserved portion of memory, which we ensure comprises at least five percent of the memory not wired by the kernel. While this approach is limited (e.g., it can easily lead to thrashing), it allows us to experiment with the resource trade-offs that applications can make.

Processes running as root can obtain hard memory shares from the base currency. Once a currency is funded with memory tickets, resource principals with appropriate permissions can obtain soft or hard subshares of its allocation. As with any resource, hard shares are obtained using the `reserve()` system call and soft shares using the `fund()` system call. In the common case of obtaining a hard share from the base currency, users simply specify the size in kilobytes of the memory share they are requesting. In other cases, users either specify a number of memory tickets (for soft shares) or a percentage of the issuing currency's share (for hard shares).

To maintain guaranteed shares, we altered the behavior of VINO's pageout daemon so that pages owned by processes that have not exceeded their memory guarantee are not reclaimed. This approach does not limit processes to their memory shares, but merely ensures that they can receive at least that amount. In the absence of memory pressure, processes receive as much memory as they need. If a process holds soft memory tickets, the number of pages to which it is entitled can change dynamically as the value of its tickets changes. The pageout daemon thus needs to compute the current base value of the processes' memory tickets; cached values are used whenever possible.

4.5 Managing Disk Bandwidth

Our prototype supports proportional sharing of disk bandwidth using the hierarchical YFQ algorithm [Bru99b]. YFQ approximates ideal proportional sharing by maintaining a virtual time function and per-disk queues of outstanding disk requests for each resource principal. Each of the queues has an associated *finish tag* that reflects the principal's past disk activity, its current share of the disk, and the length of the request at the head of the queue. Requests from queues with the smallest finish tags are forwarded to the device driver in small batches that can be reordered by the driver or device to achieve better aggregate throughput.

A principal's disk tickets are active whenever it has an outstanding request. To adjust to dynamic changes in the number of active tickets, the base value of a principal's disk tickets is recomputed (using cached values if possible) whenever a request reaches the head of its queue, and this value is used to compute the queue's new finish tag.

4.6 Emulating Nice

To support the semantics of `nice`, we created a utility that runs with root privileges and executes programs with funding from the base currency. This utility reduces the funding of the caller's user currency by the amount requested for the new job, thus preserving isolation. The utility actually creates a new currency for the task, funds that currency with the requested number of tickets, and uses the new currency's tickets to fund the task (see Fig. 4, *bottom*). This level of indirection is needed in case the task spawns any children; if so, they will share the funding of the new currency. While this utility would typically be used to give a small percentage of the CPU to long-running, CPU-intensive jobs, it can be used with other resources as well. It successfully overcomes the lower limits imposed by currencies without employing VINO's extensibility mechanism, as we had originally planned [Sul99a]. The other broker methods could also be overridden using similar setuid-root utilities.

4.7 Carrying Out Exchanges

As discussed in Section 3.2.1, a number of challenging questions must be answered before a system that fully supports dynamic ticket exchanges can be built. At this point, we have implemented a framework that allows us to easily test the effects of ticket exchanges and thereby gain insight into the issues involved.

We provide two mechanisms for experimenting with exchanges. First, users with appropriate permissions can employ our `reserve()`, `fund()`, and `unfund()` system calls to implement *static* exchanges, preset modifications to the default ticket allocations.

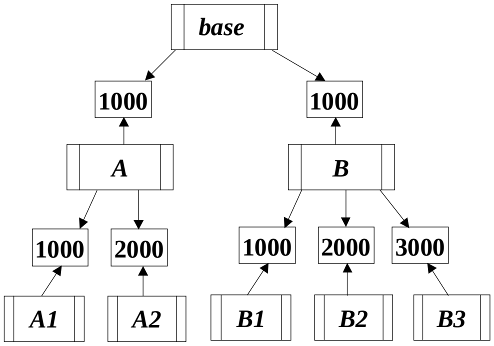

Figure 5. The CPU funding used for the experiment described in Section 5.2. Currencies A and B receive equal funding from the base currency, which they divide among the tasks they fund. A2 receives twice the funding of A1, B2 receives twice the funding of B1, and B3 has three times the funding of B1.

Second, we have implemented a simple central dealer in the kernel, and we allow applications to propose exchanges dynamically using a system call (`exch_offer`) added for this purpose. When an exchange is carried out, the four tickets involved (see Section 3.2.2) are linked to each other in a circular queue so that the exchange can be invalidated when one of the principals exits, loses too much funding, or retracts the exchange. If one of the tickets is deleted, all four of them are, thereby cancelling the exchange.

5 Experiments

We conducted a number of experiments to test the effectiveness of our extended framework and to assess its ability to provide increased flexibility in resource allocation while preserving secure isolation. In the following sections, we first discuss tests of the proportional-share mechanisms that we implemented and demonstrate that they provide accurate proportional-share guarantees and effective isolation. We then present experiments that test the impact of ticket exchanges on two sets of applications.

5.1 Experimental Setup

All of these experiments were conducted using our modified kernel. We ran it on a 200-MHz Pentium Pro processor with 128 MB of RAM and a 256-KB L2 cache. The machine had an Adaptec AHA-2940 SCSI controller with a single 2.14-GB Conner CFP2105S hard disk.

5.2 Providing Shares of CPU Time

To test our implementation of the basic lottery-scheduling framework, we replicated an experiment from the original lottery-scheduling paper (Wal94, Section 5.5). We ran five concurrent instances of a CPU-intensive program (the dhrystone benchmark [Wei84]) for 200 seconds using the CPU funding shown in Figure 5. Principal B3 sleeps for the first half of the test, during which time its tickets are not active.

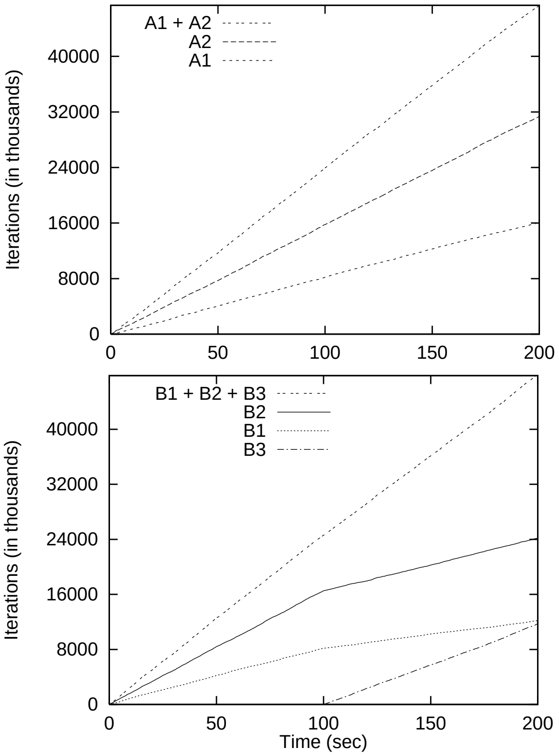

Figure 6: Hierarchical Proportional Sharing of CPU Time. Five CPU-intensive tasks, with funding shown in Figure 5, compete for the CPU. Shown above are the number of iterations completed by each task as a function of time. Task B3 sleeps for the first half of the test.

Figure 6 shows the number of iterations accomplished as a function of time for the jobs funded by each currency. In all cases, the relative levels of progress of the processes match their relative funding levels. When B3 awakes, its tickets are reactivated; as a result, the other tasks funded by currency B receive reduced CPU shares, while the tasks funded by currency A are unaffected because of the isolation that currencies provide.

5.3 Providing Memory Shares

The next experiment tests our prototype's ability to guarantee fixed shares of physical memory. To create enough memory pressure to force frequent page reclamation, we limited the accessible memory to 8 MB. After subtracting out the pages wired by the kernel as well as the desired number of free pages in the system, there were approximately 4.2 MB of memory that principals could reserve. We ran four concurrent instances of a memory-intensive benchmark; each instance repeatedly reads random 4-KB portions of the same 16-MB file into random locations in a 4-MB buffer. This load

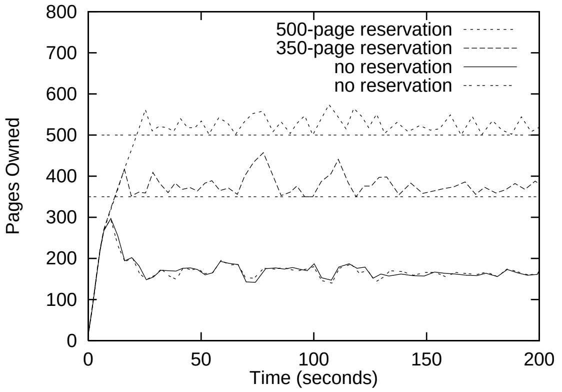

Figure 7: Providing Hard Memory Shares. Four memory-intensive tasks run concurrently on a system with approximately 4.2 MB of available memory. Two have guaranteed memory shares; two do not. Shown are the number of 4-KB pages owned by each process as a function of time.

keeps the pageout daemon running more or less continuously. We gave one of the processes a 2-MB memory guarantee (500 pages) and another a 1.4-MB guarantee (350 pages); the other two processes ran without memory reservations. Figure 7 shows the actual memory shares of the tasks as a function of time. The tasks with hard shares lose pages only when they own more than their guaranteed shares. The tasks without memory tickets end up owning much less memory than the ones with guaranteed shares.

5.4 Providing Shares of Disk Bandwidth

We tested our implementation of the YFQ algorithm for proportional-share disk scheduling by running five concurrent instances of an I/O-intensive benchmark (iohog) that maps a 16-MB file into its address space and touches the first byte of each page, causing all of the pages to be brought in from disk. Each copy of the benchmark used a different file. Throughout the test, each process almost always has one outstanding disk request. We limited YFQ's batch size to 2 for this and all of our tests to approximate strict proportional sharing. We gave one process a 50% hard share of the disk (i.e., one-half of the base currency's hard disk tickets), while the other four tasks received the default number of disk tickets from their user's currency. Figure 8 shows the number of iterations that each process accomplishes over the first 100 seconds of the test. Because one task has reserved half of the disk, the other four tasks divide up the remaining bandwidth and effectively get a one-eighth share each. Thus, the process with the hard share makes four times as much progress as the others; when it has finished touching all 4096 of its file's pages, the other four have touched approximately 1000 pages (a 4.1:1 ratio).

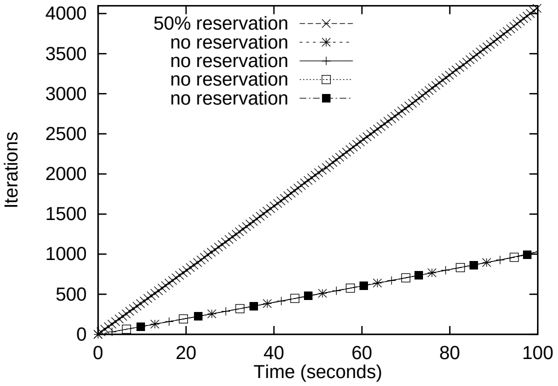

Figure 8: Providing Proportional Shares of Disk Bandwidth. Five I/O-intensive tasks compete for the disk. One of them receives a 50% hard share, while the others receive equal funding from their user's currency and thus divide up the other 50% of the bandwidth. Each iteration corresponds to paging in one 4-KB page of a memory-mapped file.

5.5 Ticket Exchanges: CPU and Disk Tickets

To study the impact of ticket exchanges, we first conducted experiments involving the CPU-intensive dhrystone benchmark [Wei84] and the I/O-intensive iohog benchmark (see Section 5.4). In the first set of runs, we gave the benchmarks allocations of 1000 CPU and 1000 disk tickets from the base currency. We then experimented with a series of one-for-one exchanges in which dhrystone gives iohog *n* disk tickets in exchange for *n* CPU tickets, where *n* = 100, 200, …, 800. To create added competition for the resources—as would typically be the case on a central server—we ran additional tasks (one dhrystone and four iohogs) in the background during each experiment. Each of the extra tasks received the standard funding of 1000 CPU and 1000 disk tickets.

Figure 9 shows the performance improvements of the exchanging applications under each exchange, in comparison to their performance under the original, equal allocations. Dhrystone benefits from all of the exchanges, and the degree of its improvement increases as it receives additional CPU tickets. Iohog also benefits from all of the exchanges, but the degree of its improvement decreases for exchanges involving more than 500 tickets. While dhrystone does almost no I/O and can thus afford to give up a large number of disk tickets, iohog needs to be scheduled in order to make progress, and thus the benefit of extra disk tickets is gradually offset by the loss of CPU tickets. However, both applications can clearly benefit from this type of exchange, which takes advantage of their differing resource needs.

We also examined the effect of the ticket exchanges on the non-exchanging tasks. As discussed in Section 3.2, the resource *rights* of these tasks should be preserved, but their actual resource *shares* may be affected.

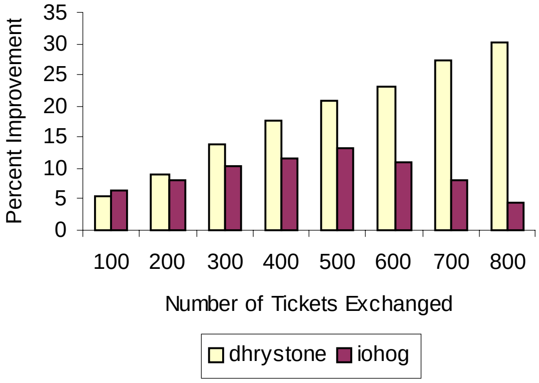

Figure 9: Performance Improvements from Ticket Exchanges. A CPU-intensive task (*dhrystone*) exchanges disk tickets for some of the CPU tickets of an I/O-intensive task (*iohog*). The improvements are with respect to runs in which both tasks receive the default ticket allocations. All results are averages of five runs.

Such an effect is especially likely in these experiments, because the two benchmarks rely so heavily on different resources. For example, dhrystone uses almost no disk bandwidth. As a result, the iohogs obtain more bandwidth than they would if the dhrystones were competing for the disk. However, when the exchanging iohog receives some of the exchanging dhrystone's disk tickets, it obtains rights to a portion of this "extra" bandwidth, and the other iohogs thus end up with smaller bandwidth shares. Exchanges affect the CPU share of the non-exchanging dhrystone in the same way.

However, the non-exchanging processes should still obtain at least the resource shares that they would receive if all of the tasks were continuously competing for both resources. To verify this, we used `getrusage (2)` to determine each task's CPU and disk usage during the first 100 seconds of each run. The results (Fig. 10) show that the minimal resource rights of the non-exchanging processes are preserved by all of the exchanges. The top graph shows the CPU shares of both the exchanging and non-exchanging dhrystones, and the bottom graph shows the disk-bandwidth shares of the exchanging and non-exchanging iohogs.[6] Because there are seven tasks running during each test, the non-exchanging tasks are each guaranteed a one-seventh share (approximately 14.3%). The non-exchanging iohogs are affected less than the non-exchanging dhrystone because each of them loses only a portion of the bandwidth gained by the exchanging iohog. In general, as the number of tasks competing for a resource increases, the effect of exchanges on non-exchanging tasks should decrease.

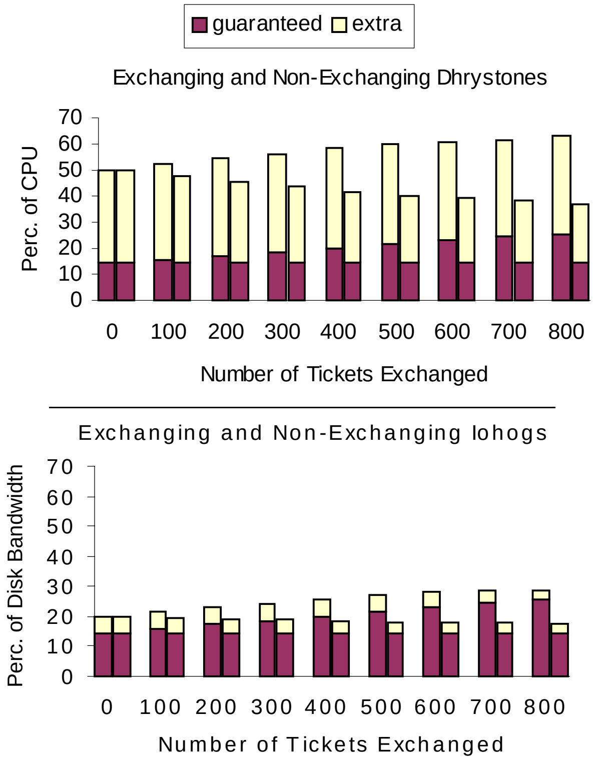

Figure 10: Resource Shares under Exchanges. Shown are the CPU shares of the exchanging and non-exchanging dhrystones (*top*) and the disk-bandwidth shares of the exchanging and non-exchanging iohogs (*bottom*). The dark portion of each bar represents the share guaranteed by the task's tickets, while the full bar indicates its actual share. In each pair, the left bar is the exchanging copy, and the right bar is the non-exchanging copy. All results are averages of five runs.

5.6 Ticket Exchanges Between Database Applications: Memory and Disk Tickets

We further experimented with ticket exchanges using two simple database applications that we developed using the Berkeley DB package [Ols99]. Both applications emulate a phone-number lookup server that takes a query and returns a number; when run in automatic mode, they repeatedly generate random queries and service them. One of the applications (*small*) has a 4-MB database with 70,000 entries, while the other (*big*) has a much larger, 64-MB database with 2^{20} entries. Both applications use a memory-mapped file as a cache.

We ran these applications concurrently for a series of 300-second runs. We disabled the update thread for the sake of consistency, because its periodic flushing of dirty blocks from the applications' cache files can cause large performance variations. To emulate the environment on a busy server, we created added memory pressure—limiting the available memory to 16 MB—and we ran four iohogs in the background. After subtracting out the pages wired by the kernel and the system's free-page

[6] The four non-exchanging iohogs have approximately equal shares. In each case, the graphed value is the smallest share of the four.

target, there was approximately 11.1 MB of memory that principals could reserve. When *small* runs alone, it uses up to 8 MB as a result of double buffering between the filesystem's buffer cache and its own 4-MB cache. With only 70,000 entries, it makes a large number of repeated queries, and it should thus benefit from additional memory tickets that allow it to cache more of its database. On the other hand, *big* uses a smaller, 500-KB cache because it seldom repeats a query; it should benefit from more disk tickets.

We started by giving the applications equal allocations: 1000 CPU tickets, 1375 hard memory tickets[7], and 1000 disk tickets, all from the base currency. We then experimented with exchanges in which *small* gives up some of its disk tickets for some of *big*'s memory tickets, trying all possible pairs of values from the following set of exchange amounts: {100, 200, …, 800}[8]. The iohogs had 1000 CPU and 1000 disk tickets each.

While the exchanges in Section 5.5 were preset, the exchanges in these experiments were proposed and carried out dynamically using the `exch_offer()` system call (see Section 4.7). *Big* proposes the exchange as soon as it starts running, but *small* waits until it has made 10,000 queries (approximately one-third of the way through the run), at which point the exchange is carried out. By waiting, *small* is able to use its original disk-ticket allocation to bring a portion of its database into memory quickly, at which point it can afford to exchange some disk tickets for memory tickets.

Small benefits from most of the exchanges, including any in which it obtains 400 or more memory tickets. It fails to benefit when it gains only 100 memory tickets (not shown), or when it gives away a large number of disk tickets for 300 or fewer memory tickets (Fig. 11, *top*). Because *small* can only fit about three-quarters of its database in memory with this allocation, it cannot afford to give away a large number of disk tickets. When *small* obtains 700 or 800 memory tickets, it can hold all of its database in memory, and it thus sees performance gains of over 1000 percent (Fig. 11, *bottom*). *Big* likewise benefits from most of the exchanges, including any in which it obtains disk tickets worth 600 or more.

It is interesting to note that these applications cannot simply specify an exchange *ratio*, such as two disk tickets for every one memory ticket, because what constitutes an acceptable ratio depends on the number of tickets being exchanged. For example, *small* should not accept a ratio of 2 disk for 1 memory if only 100 or 200 memory tickets are offered, but it should accept exchanges with this ratio to obtain 300 or more memory tickets. More generally, what constitutes an acceptable exchange depends heavily on the environment in which the tasks are running. For example, because tasks need to wait until a synchronous I/O completes before

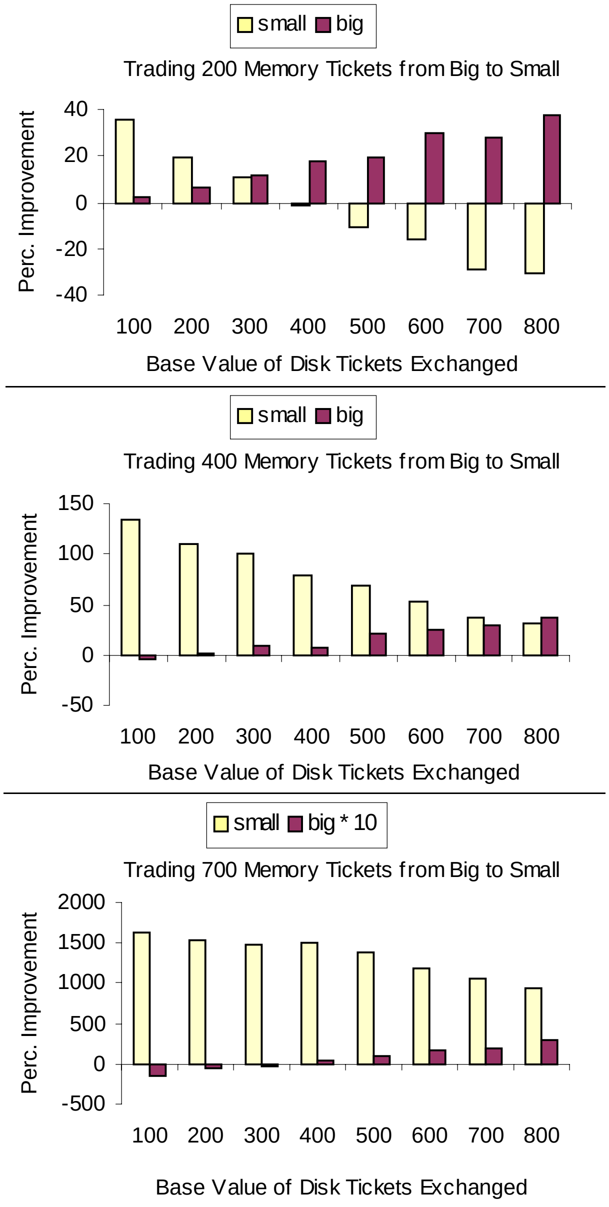

Figure 11. Results of exchanges in which an application with a large working set (*big*) exchanges memory tickets for some of the disk tickets of a similar application with a small working set (*small*). The graphed changes compare the number of requests serviced in a 100-s interval after the exchange has occurred with the requests serviced during the same interval with no exchange. Results are averages of at least five runs. There is a different vertical scale for each graph, and the values for *big* in the third graph are scaled by 10 to make them more visible. See related work [Sul99b] for graphs of the other exchanges.

[7] Each hard memory ticket from the base currency represents one page of physical memory, so 1375 tickets confer a 5.5-MB reservation.

[8] Because these exchanges were carried out by the kernel, the values given represent the *base value* of the tickets exchanged, whereas the values given in Section 5.5 are the *number* of soft tickets exchanged.

enqueueing a new one, they receive at most 50% of the bandwidth in the absence of prefetching. Therefore, without extra tasks competing for the disk, *big* cannot benefit from extra disk tickets, because it already obtains 50% of the disk by default. Applications like *big* will need to use negotiators that can assess the current system conditions before proposing an exchange.

6 Related Work

In addition to lottery scheduling, other frameworks can be used to provide proportional-share management of multiple resources. In particular, Rialto's *activities* [Jon97], Eclipse's *reservation domains* [Bru98, Bru99a], Verghese et al.'s *Software Performance Units (SPUs)* [Ver98], and Banga et al.'s *resource containers* [Ban99] function similarly to currencies in their ability to isolate resource principals from each other.

Reservation domains and resource containers also share lottery scheduling's ability to support hierarchical resource management. However, the hierarchies supported by reservation domains are limited to a tree-shaped structure in which the resource shares of non-leaf domains are divided among their children. As discussed in Section 2.1, lottery scheduling allows resource principals to be funded by more than one currency and to thus share the resource rights of multiple currencies. Resource containers similarly allow threads to be multiplexed over several containers and to receive their combined allocations.

Moreover, most of these alternative frameworks only support hard shares; resource principals that lack a reservation either share the remaining CPU capacity equally (as in Rialto and Eclipse) or are scheduled according to a traditional time-sharing scheduling discipline. Lottery scheduling, on the other hand, can support both hard and soft shares. In their prototype implementation, resource containers were used with both fixed-share CPU guarantees and time-sharing, but they could potentially be used to support soft proportional-share guarantees as well.

The alternative approaches do provide advantages over our lottery-scheduling framework. In particular, activities and resource containers offer finer-grained resource management, addressing applications such as Web servers in which a single thread is associated with more than one independent activity. In addition, resource containers account for kernel-mode processing done on behalf of an activity. We plan to extend our lottery-scheduling framework to support these features.

Regardless of the framework used to provide proportional-share resource management, the need to isolate resource principals from each other necessarily involves imposing limits on allocations of the types described in Sections 3.1 and 3.3. Principals restricted to a particular activity, reservation domain, SPU, or resource container cannot obtain more than their group's overall resource rights. If only one principal in a group is actively competing for a reserved resource, it will receive the entire reservation, even if it would be preferable for it to receive less than that amount. Mechanisms like ticket exchanges would be needed to allow these frameworks to provide more flexible resource allocation while preserving secure isolation.

Verghese et al.'s work on SPUs explicitly addresses the need to provide both secure isolation and flexible allocation. However, their system starts by giving absolute resource shares to each SPU, and it gains added flexibility by dividing unused portions of these shares among SPUs that need additional resources. The original lottery-scheduling framework naturally supports this type of resource sharing by deactivating the tickets of idle tasks. Our extended framework provides added flexibility through ticket exchanges and a utility that emulates the semantics of nice. One advantage of SPUs is that they were designed for use with shared-memory multiprocessors. Extending the lottery-scheduling framework for use with SMPs remains future work.

Other systems have allowed applications to negotiate their resource usage with the operating system [Jon95, Nob97]. Our extended lottery-scheduling framework lets applications coordinate their resource usage with *each other*, as well as with the system as a whole.

Besides Waldspurger's own prototypes, others have implemented portions of the lottery-scheduling framework [Arp97, Nie97]. Petrou et al. [Pet99] retrofitted lottery scheduling into FreeBSD to schedule the CPU, extending the framework to better support interactive jobs. VINO currently has a small, 10-ms quantum, so such extensions have not been needed in our prototype. Petrou et al. also suggest an alternative approach to overcoming the lower limits that currencies impose.

As discussed in Section 4.4, our scheme for managing memory is a temporary one. The Nemesis operating system [Han99] provides a more complete solution that also allows applications to obtain guaranteed memory shares. Nemesis ensures complete isolation by requiring that applications handle their own page faults.

7 Conclusions

Our extended lottery-scheduling resource management framework gives applications increased flexibility in modifying their resource allocations while preserving the ability to isolate groups of processes. We believe that it could be particularly useful on systems in which many users compete for the resources of a central server, as in thin-client networks or Web servers used for virtual hosting. Ticket exchanges allow processes to adjust their allocations while insulating resource principals that do not take part in an exchange, and they enable applications to coordinate their resource usage with each other. Currency brokers provide secure access controls to currencies, while setuid utilities can be used to circumvent the default controls in ways that preserve isolation.

In order for our extended framework to be fully effective on large central servers, more work needs to be done to develop negotiators that can intelligently carry out ticket exchanges on behalf of users and applications. Developing such negotiators will be a challenging task, but one with potentially significant rewards.

Availability

Source code and binaries for the version of VINO used in this paper, as well as source code for the test programs, can be obtained from `ftp://ftp.eecs.harvard.edu/pub/vino/vino-usenix2000`.

Acknowledgments

This research was supported in part by a USENIX Association Scholarship. Robert Haas contributed to the development of our framework. Special thanks to Carl Waldspurger, the anonymous reviewers, and our shepherd, Yoonho Park, who all offered helpful comments on earlier drafts.

References

[Arp97] Arpaci-Dusseau, A.C., Culler, D.E., “Extending Proportional-Share Scheduling to a Network of Workstations,” *Proc. of the Int’l Conf. on Parallel and Distributed Processing Techniques and Applications*, June 1997.

[Ban99] Banga, G., Druschel, P., Mogul, J.C., “Resource Containers: A New Facility for Resource Management in Server Systems,” *Proc. of the Third Symposium on Operating Systems Design and Implementation,* February 1999.

[Bru98] Bruno, J., Gabber, E., Özden, B., Silberschatz, A., “The Eclipse Operating System: Providing Quality of Service via Reservation Domains,” *Proc. of the USENIX 1998 Annual Tech. Conference*, June 1998.

[Bru99a] Bruno, J., Brustoloni, J., Gabber, E., Özden, B., Silberschatz, A., “Retrofitting Quality of Service into a Time-Sharing Operating System,” *Proc. of the USENIX 1999 Annual Tech. Conference*, June 1999.

[Bru99b] Bruno, J., Brustoloni, J., Gabber, E., Özden, B., Silberschatz, A., “Disk Scheduling with Quality of Service Guarantees,” *Proc. of the Int’l Conf. on Multimedia Computing and Systems*, June 1999.

[Han99] Hand, S.M., “Self-Paging in the Nemesis Operating System,” *Proc. of the Third Symposium on Operating Systems Design and Implementation,* February 1999.

[Jon95] Jones, M.B., Leach, P.J., Draves, R.P., Barrera, J.S., “Modular Real-Time Resource Management in the Rialto Operating System,” *Proc. of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[Jon97] Jones, M.B., Rosu, D., Rosu, M-C., “CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities,” *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.

[Nie97] Nieh, J., Lam, M., “The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications,” *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.

[Nob97] Noble, B.D., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., Walker, K.R., “Agile Application-Aware Adaptation for Mobility,” *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.

[Ols99] Olson, M., Bostic, K., Seltzer, M., “Berkeley DB,” *Proc. of the USENIX 1999 Annual Tech. Conference*, June 1999.

[Pet99] Petrou, D., Milford, J., Gibson, G., “Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers,” *Proc. of the USENIX 1999 Annual Tech. Conference*, June 1999.

[Sel96] Seltzer, M., Endo, Y., Small, C., Smith, K., “Dealing with Disaster: Surviving Misbehaved Kernel Extensions,” *Proc. of the Second Symposium on Operating System Design and Implementation*, October 1996.

[Sma98] Small, C., *Building an Extensible Operating System*, Ph.D. thesis, Division of Engineering and Applied Sciences, Harvard University, October 1998.

[Sto96] Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S., Gehrke, J., Plaxton, C.G., “A Proportional-Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems,” *Proc. of the IEEE Real-Time Systems Symposium*, December 1996.

[Sul99a] Sullivan, D., Haas, R., Seltzer. M., “Tickets and Currencies Revisited: Extending Multi-Resource Lottery Scheduling,” *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, March 1999.

[Sul99b] Sullivan, D., Seltzer, M., “A Resource Management Framework for Central Servers,” Computer Science Technical Report TR-13-99, Harvard University, December 1999.

[Sun98] “Solaris Resource Manager 1.0: Controlling System Resources Effectively: A White Paper,” http://www.sun.com/software/white-papers/wp-srm/.

[Ver98] Verghese, B., Gupta, A., Rosenblum, M., “Performance Isolation: Sharing and Isolation in Shared Memory Multiprocessors,” *Proc. of the Eighth Int’l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[Wal94] Waldspurger, C.A., Weihl, W., “Lottery Scheduling: Flexible Proportional-Share Resource Management,” *Proc. of the First Symposium on Operating System Design and Implementation*, November 1994.

[Wal95] Waldspurger, C.A., *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, Ph.D. thesis, MIT/LCS/TR-667, MIT Laboratory for Computer Science, September 1995.

[Wal96] Waldspurger, C.A., Weihl, W., “An Object-Oriented Framework for Modular Resource Management,” *Proc. of the Fifth Int’l Workshop on Object Orientation in Operating Systems*, October 1996.

[Wei84] Weicker, R.P., “Dhrystone: A Synthetic Systems Programming Benchmark,” *Communications of the ACM*, October 1984.