Resource Partitioning in a Java™ Operating Environment

Mick Jordan

Resource Partitioning in a Java™ Operating Environment

Mick Jordan

SMLI TR-2006-161 December 2006

Abstract:
Managing the partitioning of resources between uncooperating applications is a fundamental requirement of an operating environment. Traditional operating environments only manage low-level resources which presents an impedance mismatch for internet-facing applications with service levels defined in terms of application-level transactions.

The Multi-tasking Virtual Machine (MVM) and associated Resource Management API (RM) provide basic mechanisms for managing multiple applications within a Java™ operating environment. RM separates mechanism and policy and takes the unusual position of delegating rate-based management of resources to the policy level. This report describes the design and implementation of policies that provide flexible resource partitioning among applications and shows their effectiveness using microbenchmarks and an application level benchmark. The latter demonstrates the partitioning of an application-specific resource among a set of application instances using exactly the same policies as used for machine-level resources.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:
mick.jordan@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

Resource Partitioning in a Java Operating Environment

Mick Jordan

Sun Microsystems Laboratories
16 Network Circle
Menlo Park,Ca 94025

Mick.Jordan@sun.com

1 INTRODUCTION

With the addition of a small number of extra features, the Java™ platform can provide a powerful *operating environment* for Java applications. Two critical features are support for multiple, isolated, applications, and the ability to control the allocation of resources to applications.

The Multi-tasking Virtual Machine (MVM) [CD01] implements a portable API for the creation and management of multiple applications that is defined by JSR 121 [JCP01]. In JSR 121 an application is referred to as an *isolate*, which conveys the important property of complete state isolation. This is in contrast to schemes for multiple applications that are built on the existing platform capability for multiple classloaders, which are complex and provide incomplete isolation.

The Resource Management (RM) API [CHS+05] builds on JSR 121 and provides a flexible and extensible framework for controlling resources at various levels in the platform. This is in contrast to traditional operating system facilities that are confined to managing low-level resources such as CPU and memory. RM provides a clear separation between the mechanism for resource acquisition and the policies that control usage. A simplified version of the RM API is in the process of being standardized by JSR 284 [JCP06].

RM provides a set of a basic mechanisms for defining the scope of resource control and establishing callbacks to policy code. It provides only very limited global controls, leaving more sophisticxated mechanisms to a higher-level software layer. This paper describes such a software layer that allows resources to be partitioned among a group of clients in a variety of ways that are convenient for administrators.

We begin with a description of the resource management (RM) system. This is followed by a description of the design and implementation of the resource partitioning mechanisms that are built on RM. We then present experimental results that demonstrate the effectiveness of the mechanisms. We conclude with an overview of related work, and ideas for improvement.

2 The Resource Management API

This section provides a brief overview of the main features of the RM API. An overarching design goal for RM was that existing applications can run without modification, even if the classes they depend on exploit the RM system. However, pro-active programs can use RM to learn about resource availability and consumption to improve the characteristics most important in the given case (response time, throughput, footprint, etc.). Application managers that need to control how resources are partitioned (e.g., application servers) can use RM for that purpose. The software described in this paper is primarily aimed at such application managers.

2.1 Basic Abstractions

The key abstractions of the RM API are discussed below.

Resource Management Context: The resource management *context* defines the scope in which resources are accounted. In this work the context is always an isolate since this provides for unambiguous ownership of resources. However, implementations are possible in which the context could be a thread, classloader or an entire Java™ virtual machine (JVM™).

Resource Domain: A *resource domain* encapsulates a usage policy for a resource. All contexts *bound* to a given resource domain are uniformly subject to that domain's policy for the underlying resource. A context cannot be bound to more than one domain for the same resource, but can be bound to different domains for different resources. Thus, two contexts can share a single resource

domain for, say, CPU time, but be bound to distinct domains for, say, JDBC connections.

Constraints and Notifications: The RM API itself does not impose any policy on a domain; policies are explicitly provided by policy software that is bound dynamically to a domain, using the RM API, typically by an application manager. A resource management policy for a resource controls when a computation may gain access to, or *consume*, a unit of that resource. The policy may specify *reservations* and arbitrary *consume actions* that should execute when a request to consume a given quantity of resource is made by a context bound to a resource domain. Consume actions may be defined to execute prior to the consume request and serve as programmable *constraints* that can influence whether or not the request is granted. Consume actions may also be defined to execute after the consume event and serve as *notifications* of resource consumption. Consume actions can cross context boundaries and typically do. That is, actions are usually set by a managing context on a resource domain that is bound to a managed context. Less common is an action set by a context to monitor its own consumption. Consume actions can be specified to be invoked one-time or every-time (persistent) a client makes a consume request, and whether they execute synchronously or asynchronously with respect to the client consume request. Constraint actions are typically persistent and always executed synchronously and notifications are typically executed asynchronously.

Multiple consume actions can be associated with a given resource domain and, for constraints, all actions have to approve the consumption for it to be granted. The set of consume actions constitute the policy associated with the domain.

For example, the following class defines a callback that enforces a fixed limit on the consumption of a resource. This is an example of a callback that can be applied to any resource as it is independent of the resource characteristics.

```
class LimitCallback implements ConsumeCallback.Pre {
  private long limit;
  LimitCallback(long limit) { this.limit = limit; }
  public long preConsume(ResourceDomain d, long cu, long pu) {
    if (pu > limit) return cu; else return pu;
  }
}
```

The `preConsume` method is called with the domain associated with the request, the current usage, `cu`, and the proposed usage, `pu`. Returning `cu` (or less) is interpreted as denying the request; returning `au`, where `cu < au <= pu`, is interpreted as approving the request, only partially if `au < pu`.

The following code applies this callback to a resource domain `d`, with a limit of ten, specifying the action to be persistent and synchronous.

```
d.setConsumeAction(PERSISTENT, SYNCHRONOUS,
                   new LimitCallback(10));
```

2.2 Defining Resources

Resources can be exposed through the API in a uniform way, regardless of whether they are actually managed by the operating system (e.g., CPU time in JVMs that rely on kernel threading), the run-time system (e.g., heap memory), core classes (e.g., file and network resources), middleware (e.g., JDBC™ connections), or by the application itself. Retrofitting existing resource implementations to take advantage of the RM API is relatively easy and described below.

The ability to define high-level, middleware-specific or application-specific resources, is unique to RM. It is very convenient for administrators because it can be used to tie policies directly to key performance characteristics of a software component. For example, in [JS05], we define a resource that corresponds directly to one user-level transaction, i.e., as observed directly by an end-user. Policies can then be applied to this resource that affect the end-user experience in very predictable ways instead of having to estimate how controls on low-level resources translate into the end-user behavior.

A particular resource is represented by an instance of a Java class that must inherit from the `ResourceAttributes` class, which is part of the RM API, and provides standard methods that define the resource characteristics:

- **Disposable**: A resource is disposable if it can be returned and reused at a later time. For example, JDBC Connections are disposable but CPU cycles are not.
- **Unbounded**: there is no fixed limit on the amount of resource available. CPU cycles are an example of an unbounded resource and Memory is an example of a bounded resource.

- **Reservable**: Bounded resources may be reserved by a domain for future consumption, up to a pre-determined limit.
- **Scope:** RM captures the intended scope of a given resource through an attribute that can have one of two values: local or global.

2.3 Dispensers

Dispensers model the notion of a point of manufacture for a resource. For example, a storage management system "manufactures" memory. Dispensers record the total available quantity of a resource, manage resource reservations and coordinate the consume and unconsume actions associated with resource domains.

There is a clear relationship between the scope of a resource and the number of dispensers for the resource. For global resources there is one dispenser. Most resources are global. However, if a resource potentially has several points of manufacture, then it is deemed local and there will be one dispenser per point of manufacture. Local resources typically model software artefacts that could be instantiated multiple times within a single context, e.g., a J2EE™ servlet container.

2.4 Exposing Resources

To make a resource manageable through the RM API, one must modify the resource implementation and insert consume and unconsume calls where appropriate. For example, the following code, invoked upon an attempt to open a new physical JDBC connection, controls allocation of new connections:

```
ResourceDomain rd =
  ResourceDomain.currentDomain(JDBC_CONNECTIONS);
long val = rd.consume(1); //request unit of the resource
if (val >= 1) //go ahead, if the consume request succeeded
  return new connection ...
else //consume request failed – report the failure to the caller
  fail ...
```

For bounded resources it is vital to let the RM API's accounting know that the resource has been disposed of, by planting an appropriate unconsume call in the resource's implementation. In the case of JDBC connections the code in JDBC driver's close method is augmented with:

```
rd.unconsume(1); // "return" the resource
```

Existing applications run without modifications under the control of the RM API because they do not request resources explicitly. Instead, the implementations of the resources themselves are modified to interact with the RM API, so that resource consumption policies are taken into account when a resource is being used. This modification is transparent to clients of the resource.

2.5 Rate-based Management

The RM API provides no built-in support for managing the rate of consumption of resources. The rationale is that rate-based management can be achieved by scheduling mechanisms, for example, suspending the execution of a client of a resource as part of the consume action. This works acceptably for all resources except CPU, which is distinguished by not being allocated with explicit consume requests. Our current implementation of the CPU resource simulates consume requests using a polling thread that periodically wakes up, computes usage in the previous period and then invokes the consume action.[1] Note that in this case, it is the polling thread that invokes the consume action and not the actual application thread. Therefore a separate mechanism is required to suspend the application threads in order to control the rate of consumption.

3 RESOURCE PARTITIONING

The notion of partitioning implies the existence of a set of clients among which to divide up the resource, and a manager that controls the allocation among the clients.

Partitioning also implies that there is a well-defined limit on the total resource consumption. Some resources, for example, memory or CPU, have simple limits that change infrequently or not at all during an application's execution. Other resources, typically higher-level ones, for example, a JDBC transaction, have limits that vary and may be unknown a priori. Typically the limit will also depend on the availability of other resources. For example, the availability of CPU implicitly impacts the limits of many resources. Previous resource management systems were

[1] A more direct and efficient implementation would be possible if RM was more closely integrated with the thread scheduler.

limited to simple concrete resources and did not address the dynamic limits of higher level resources. In the RM environment it is necessary to provide a mechanism for informing the partitioning manager about dynamic limits.

3.1 Fixed Partitioning and Reservations

If the total available quantity of a resource is constant over an interval, in particular, the lifetime of the application, then partitioning is straightforward, as one only needs to keep track of total consumption. This can be handled by simple callbacks that check against the total limit, as described earlier.

For reservable resources, the RM API provides built-in support for reservations, which are managed by the resource dispenser. However, not all resources are reservable. A reservation is really a special kind of partitioning mechanism that is intended to ensure that, at some future time, a minimum amount of resource can be allocated, while avoiding the overhead of allocating the resource in the interim. Reservations can be (temporarily) exceeded provided that there is sufficient quantity of a resource to satisfy all existing reservations and current usage.

In principle reservations could be implemented outside the dispenser using policies, provided they were coordinated by a manager with global knowledge equivalent to that of the dispenser.

3.2 Simple Rate-based Management

Simple rate-based management is provided in a utility class that is included with the basic RM API. It provides a callback that is parameterized by two arguments:

- A *time interval* over which the rate of consumption is to be controlled.
- A *limit* to the number of resource units that can be consumed in the time interval.

The constructor for this callback has the following signature.

```
LimitBasedCallback(long limit, long interval);
```

The callback suspends the thread that invoked the `consume` method as needed in order to ensure that no more than `limit` resource units are consumed in the given interval. The callback specializes the consumption of the CPU resource and suspends all the threads in the resource context as needed.

Note that the time interval parameter interacts with the maximum suspension value. For example, imagine a client that rapidly consumes a resource in the early part of the interval. Once it reaches the limit, it will be suspended for the remainder of the interval. If this behavior is repeated, the resource consumption will exhibit a bursty consumption pattern punctuated by long suspensions. Evidently this can be alleviated by choosing a smaller interval, but note that this is bounded by the measurement period of the resource. In particular, for the CPU resource, the interval cannot be made smaller than the period of the polling thread. In general, for sampled resources, the overhead of resource management increases as the measurement period decreases, leading to a trade-off between control precision and overhead.

At the other extreme, choosing a very long time interval makes rate-based management similar to the simple limit control described in section 2.1.[2] For example, we can limit a domain to consume at most 30MB of heap memory by specifying an interval that exceeds the lifetime of the application. The partitioning schemes described below can then be used to partition a fixed amount of memory between a varying number of applications.

This simple limit-based mechanism is unaware of other resource contexts and therefore cannot check any global limits, for example, whether the sum of the limits exceeds the total available. However, the mechanism can be extended to implement more sophisticated global controls as we will show below.

3.3 Share-based Management

The basic mechanism described above, while simple and straightforward to implement, is not an especially convenient interface for human administrators. It is more natural for humans to think in terms of percentages or similar partitions, such as tickets or shares [KL88], [WW94], [Sun06]. We will use the term *share-based* to refer to all such partitioning schemes henceforth.

[2] In the simple-limit case control returns with request denied, whereas in the rate-based case the thread is suspended.

Share-based mechanisms are convenient abstractions because they abstract away details of the underlying resource, such as absolute limits and intervals. For example, the need to divide up resources amongst a set of competing applications is easily expressed in such terms. In addition the abstraction is portable to a different computing environment which might have different absolute limits, for example, a faster CPU, therefore more cycles available in a given interval. On the other hand, a computer algorithm that is attempting to minimize a given application's resource usage might find the basic mechanism more appropriate precisely because it is specified in absolute terms.

An issue with all partitioning mechanisms is whether it is acceptable for a consumer to (temporarily) exceed its limit provided, of course, that there is excess resource available at the time. This issue is orthogonal to the basic mechanisms.

3.4 Quantifying Maximum Usage

Share-based mechanisms all require some way to define the maximum possible usage which, as we noted above, may be a dynamic value. Note also that a resource context may contain multiple threads. If the underlying machine hardware can execute some of these threads concurrently, then, depending on the resource, the maximum usage may be a function of the number of CPUs. Clearly, the number of available CPU cycles is a linear function of the number of CPUs. The partitioning manager attempts to determine the number of CPUs automatically but provides an interface to set the value explicitly, which can be used to artificially reduce the effective power of the machine. The interface is defined in terms of an *execution unit*, which is specified as a fraction of a physical CPU.[3] Note that an application must be multi-threaded to make effective use of multiple execution units. For example, consider a single threaded CPU-bound application executing on a machine with two processors. The maximum CPU usage in an interval `T` is therefore `2T`, when accounting for CPU in time units (as opposed to cycles). However, the single-threaded application can never use more than `T` units in the interval. Therefore, even if the application is limited to 50% of the (total) CPU, it will still run without suspension. If, however, the partitioning manager sets the effective number of CPUs to 1, this will limit the application to `T/2` units in the interval.

In some cases, the usage function may vary non-linearly with the number of execution units. For example, most web-based applications are not compute-bound, instead their throughput is closely related to the performance of the backend database. The database interface is connection-based and typically exhibits a non-linear relationship between the total throughput and the number of database connections. That is, as connections are added, the additional throughput per connection becomes progressively smaller and may, beyond a certain limit, turn negative.

To support custom maximum usage functions, the partitioning manager provides a method, `setMaxUsageFunction`, that registers an object which must implement the following interface:

```
public interface MaxUsageFunction {
  public long setMaxUsage(ResourceAttributes resource,
                          int executionUnits,  long interval);
}
```

The returned value should be the maximum usage that is possible for the given resource in the given interval on a machine with the given number of execution units.

Since the maximum possible usage for a resource is a global value across all domains, it is typically set by a management module that has a global view of the system.

Simple usage functions that cover common cases are available in a pre-defined library. The common case is a function that is independent of the resource and scales linearly with the interval and number of execution units. This is the default function for a resource unless overridden by the `setMaxUsageFunction` method.

To determine the nature of a non-linear maximum usage function it is necessary to measure the maximum operation rate with different numbers of execution units. Ideally, these measurements would be performed online and provided dynamically to the partitioning manager through the `setMaxUsageFunction` method.

3.5 Partitioning Manager Callbacks

The partitioning manager provides a set of progressively more sophisticated callbacks to

[3] This mechanism could be used as a higher-level form of partitioning between multiple partitioning managers.

implement shared-based rate management. These are specified as Java interfaces, allowing a variety of implementation classes that provide appropriate factory methods for the callbacks.

The top-level interface provides the following methods that are common to all callbacks.

```
interface IntervalBasedCallback
    extends ConsumeCallback.Pre {
  long getInterval();
  void setInterval(long interval);
  boolean getExceedLimitWhenUnderUtilized();
  void setExceedLimitWhenUnderUtilized(boolean b);
  void terminating();
}
```

The first two methods relate to the interval that this callback is operating under. The second two methods relate to whether the usage can exceed the limit (which is defined explicitly or implicitly in the sub-interfaces) if the total resource usage is below the permitted maximum. The terminating method supports fast update of global state when the resource domain associated with the callback is itself being terminated. While a partition manager should periodically detect terminated domains, an application manager can call this method to force the partitioning manager to adapt immediately to the termination of a domain and possibly allocate the freed resources to other domains.

3.6 Limit-based Partitioning Callback

The equivalent of the simple limit-based callback described in section 3.2 is provided by an interface that inherits from IntervalBasedCallback. The interval is inherited from the super-interface.

```
interface LimitBasedCallback
      extends IntervalBasedCallback {
  void setLimit(long interval);
  long getLimit();
}
```

The essential difference between the partioning manager's implementation of this interface and the callback described in section 3.2 is that the manager is able to check global limits across the set of participating domains.

3.7 Percentage-based Partitioning Callback

Partitioning using percentages is easier than working with absolute limits, since it avoids having to know the maximum usage limit. It is therefore more portable and resilient to changes in the underlying system. The percentage-based callback interface is as follows:

```
interface PercentageBasedCallback
    extends IntervalBasedCallback {
  void setPercentage(int percentage);
  int getPercentage();
}
```

Again, the interval is inherited from the super-interface.

3.8 Share-based Partitioning Callback

In a share-based scheme, each context is given a certain number of shares and resource consumption is divided among the contexts according to the number of shares each possesses. Share-based partitioning is even easier to work with than percentage-based partitioning because it automatically adapts to a variable number of clients. When clients join or leave the managed set, the number of shares of the resource allocated to the remaining clients is automatically recomputed. The share-based interface is as follows:

```
interface ShareBasedCallback
    extends IntervalBasedCallback {
  void setShares(int shares);
  int getShares();
  int getTotalShares();
}
```

The getTotalShares method is a convenience method that returns the total number of shares across all share-based callbacks registered for the resource associated with this callback. Again, the interval is inherited from the super-interface.

4 PARTITIONING MANAGER IMPLEMENTATION

4.1 Overall Structure

The partitioning manager implementation is 1500 lines of code, including comments, and is contained in one class, with nested classes and factory methods for each of the three partitioning schemes and associated callbacks described in the previous section.

The partitioning manager uses a global data structure to track the set of domains and callbacks that are active. The data structure is a map from resource type to another map from resource

domain to the callback implementation class. This data structure supports operations that require access to all registered callbacks for a resource, for example, when a new callback is registered, to check that the total usage across all domains does not exceed the maximum possible usage.

A separate global map is maintained from resource type to the MaxUsageFunction and a method is provided for registering an object that implements the interface. The default value for this object is an instance of the LinearMaxUsageFunction described earlier.

Additional methods are provided to enquire of and set the number of execution units that the manager assumes are available.

4.2 Callback Factory Methods

While the callback interfaces are defined as siblings, the factory methods utilize implementation classes that exploit the relationship between shares, percentages and limits to reuse code using class inheritance. Each callback interface has a corresponding implementation class, e.g.:

```
class KindBasedCallbackImpl
   implements KindBasedCallback {
}
```

where Kind is either Limit, Percent or Share.

The factory methods have the following general form:

```
KindBasedCallback newKindBasedCallback (
  ResourceDomain domain,
  boolean exceedLimitWhenUnderUtilized,
  kindSpecificArgs) {
}
```

The ResourceDomain argument is required so that the manager can maintain the set of domains that are being controlled. This set forms the basis for determining share-based partitioning. The domain object implicitly identifies the resource type, which is available from the getResourceAttributes method of the ResourceDomain object. The resource type is used to determine information such as maximum usage information. The callback specific arguments correspond to the state values that are specific to the callback, i.e., either the limit, percentage or share count.

4.3 Callback Implementation Classes

The LimitBasedCallbackImpl class forms the base of the implementation class hierarchy and contains the majority of the state needed to manage the scheduling that is necessary to limit resource consumption. In particular, both the percenta-based and the share-based implementation classes transform the percentage or share into a limit, which allows the scheduling code to operate using only the limit data. For example, for the percent-based scheme, the permitted limit in the interval is given by the following expression:

```
(maxUsageFunction.getMaxUsage(
   resource,
   executionUnits,
   interval) * percentage) / 100;
```

The PercentBasedCallBackImpl class inherits from LimitBasedCallbackImpl and adds only one state variable to record the percentage allocated to the callback. If the percentage is changed, the limit is recomputed and the setLimit method of the superclass is called. This method checks that the change will not exceed the maximum possible usage and otherwise throws an exception.

The ShareBasedCallbackImpl class inherits from the PercentBasedCallBackImpl class and, similarly, adds only one state variable, namely the number of shares allocated to the callback. It is straightforward to transform the ratio of the share count to the total share count into a percentage and pass that value to the constructor for the percent-based superclass which, in turn, transforms it into a limit for its superclass. However, unlike the limit-based and percent-based schemes, the share-based scheme must dynamically adjust the percentages for all registered share-based callbacks whenever a callback is added or removed. The global callback map is used to recalculate the total share count and then perform the adjustment.

Note that it is straightforward to permit a mixture of limit-based and percent-based callbacks as both define fixed usage amounts. In principle, it would also be possible to include share-based callbacks in the mix, by allocating the usage unused by the limit-based and percent-based callbacks to the share-based callbacks, and dividing that value based on the share counts. However, our current implementation disallows such mixing for simplicity, as it is not clear that it is useful in practice.

4.4 Handling Consume Requests

The mechanism for handling consume requests is common to all resource types and handled entirely in the LimitBasedCallbackImpl class.

The general approach to managing the rate of consumption is to ensure that the requesting threads in a domain do not consume more resource than the domain has been allocated in the defined interval. For example, if the maximum usage for a resource is 100 in a 1 second interval and a domain has been allocated 20% of that usage, the domain's consumption must be limited to 1 unit every 50ms. If the domain consumes 1 unit at time 0 and then attempt to consume a second unit at time 1, the thread associated with the request will be suspended for 49ms before being allowed to continue. The implementation maintains the consumption history over the interval period defined for the callback, to ensure that the allowed consumption in the interval is not exceeded.

As noted in section 2.4, the polling-thread mechanism for handling the CPU resource complicates the algorithm slightly for two reasons. First, the consumption has already occurred at the point of the consume request and, second, all the theads in the domain must be suspended to ensure the correct usage. However, this has only a minor impact on the overall algorithm.

4.5 Overconsumption

As noted earlier, it is an option to allow a domain to exceed its allocation if the resource is being underused. This is specified at the time the callback is registered. This option adds overhead to the processing of the consume request once a domain's usage exceeds its nominal limit. In that case, the global map must be consulted to compute the total usage across all domains in the history interval of the requesting domain. If this is less than the maximum usage, the request is granted. The overconsumption is recorded in the history so that, if the utilization in other domains increases during the history interval, the domain may experience a subsequent lowered rate of consumption to bring its average consumption down to its limit.

5 PERFORMANCE MEASUREMENTS

To test the efficacy of the mechanisms described above we performed experiments with two micro-benchmarks and one application benchmark.

5.1 Hardware

The hardware consisted of a Sun 280R server, with two 1015Mhz CPUs and 4GB of main memory, and a Netra-T12 server with twelve 900 Mhz CPUs and 96GB of memory. All machines were connected by a 1Gb ethernet.

5.2 Software

The RM implementation that we used is an extended, experimental, version of the Multi-tasking Virtual Machine (MVM). MVM implements the isolate abstraction of JSR 121 in a single JVM process, efficiently sharing the meta-data between multiple isolates. The majority of RM is implemented in the Java programming language, with some MVM support for the implementation of low-level resources, e.g., CPU and memory.

In MVM the context for resource management is an isolate. This choice makes accountability unambiguous, as each resource in use has exactly one owner and resources are recycled correctly when isolates terminate.[4]

The experimental setup uses *service containers* [JS05] to host and manage the test software. A service container can contain multiple isolates, all of which are subject to the same set of resource management policies. Different service containers may have different policies. A resource policy is an object that encapsulates an RM callback, with provision for the callback arguments to be supplied as strings generated, for example, from an XML configuration file or a command line.

5.3 CPU Microbenchmark

To test the behavior of the system in the face of a set of changing clients, we used a simple microbenchmark that runs for a given time, executing a loop performing some simple arithmetic computations and maintaining a count of the number of loop iterations.

[4] In [CHS+05] we discuss problems associated with designating class loaders or threads/thread groups as principals in resource management schemes.

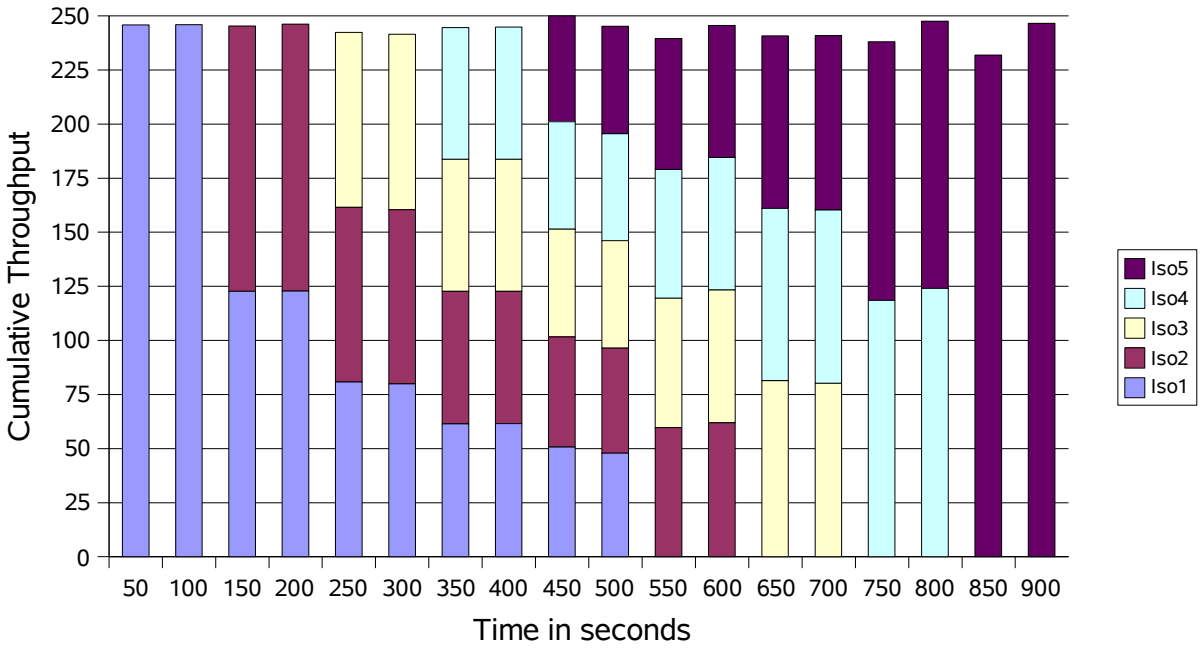

Figure 1: CPU microbenchmark showing individual and cumulative throughput as the number of concurrent isolates varies over time.

The test consists of a variable number of service containers each hosting an instance of the microbenchmark that is set to run for 500 seconds. The initial number of service containers is 1, and a new container is added and started every 100 seconds, up to a limit of 5, after which the number of containers is reduced every 100 seconds back down to zero. Each container has an associated resource policy that specifies a share-based policy for the CPU resource, with each container having the same number of shares. Therefore, when there are N containers active, each should get 1/N of the CPU. Figure 1 shows the results of this test, which match the expected behavior.

5.4 Memory Microbenchmark

This benchmark shows how control over the maximum usage can be used to share a fixed rate of memory allocation over a given time period. For example, imagine there are three domains and we want to limit the combined rate of allocation to 100MB per minute. Further, assume that we want to share this rate among the domains in the ratios 1:3:6. We can achieve this by registering a MaxUsageFunction object that returns a value of 100MB in an interval of 60 seconds, independent of the number of execution units, and then applying the share-based policy with the above share counts.

The benchmark runs for sixty seconds, with a variable number of domains, each of which continuously allocates a sequence of byte arrays, the size of each array being determined by a random number generator. All domains use the same random number seed so they follow the same allocation pattern. Table 1 shows the results from the average of five runs with three domains. The total memory allocated is slightly below the expected 100MB maximum usage, but the share for each domain is very close to the expected value.

	D1	***D2***	***D3***	***Total***
Allocated (MB)	9.85	29.55	59.09	98.49
Share	1.02	3.03	5.95	10

Table 1: Memory microbenchmark results

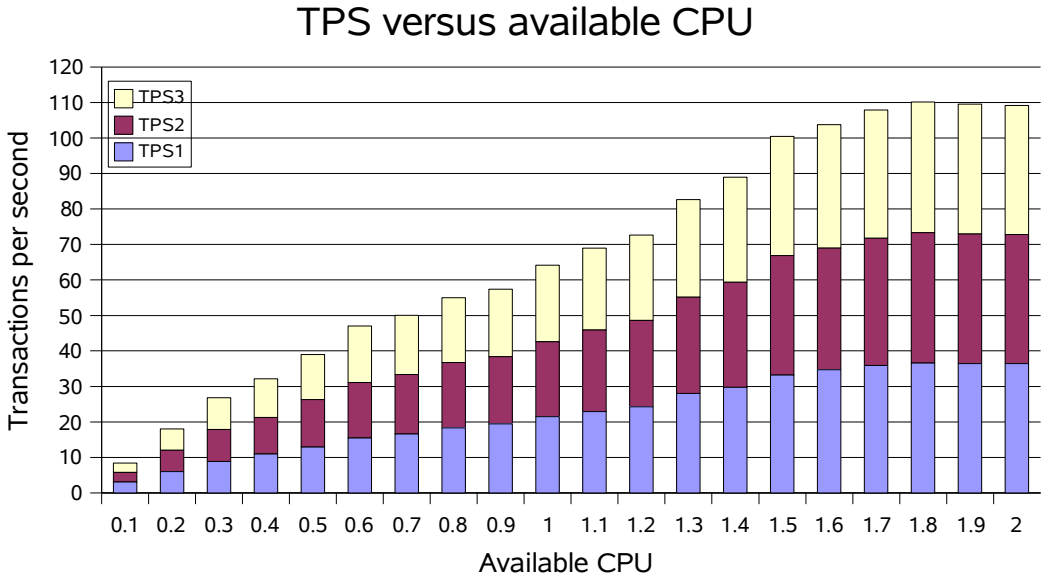

Figure 2: RUBiS transactions per second versus available CPU units for three domains.

5.5 RUBis Benchmark

The RUBiS benchmark [RUBiS] simulates an auction trading website. The catalog of available items and the details on registered users is stored in an SQL database. The benchmark was originally written to compare different middleware frameworks, so a variety of implementations are available. We use the variant in which the operations that can be performed by users are implemented as separate servlets. The servlet implementation does little caching so the performance is essentially dependent on the database interaction.

RUBiS can be driven interactively through a web browser but comes with a tool that simulates a set of interactive clients and produces a file of URLs that can be fed into a load generator for stress testing.[5] We used the Siege load generator [Siege05] in our tests.

The performance specification for web-based applications is generally defined as a service-level agreement (SLA) using metrics that relate to the client's view of the application, e.g., the number of operations per second (OPS), constraints on operation latency, etc. A load generator like Siege provides OPS and latency data for every test run. There is a complex relationship between these metrics and the underlying machine resources, such as CPU and memory, and it is non-trivial to provision a service container using the underlying resources and be confident that the SLA will be met, without adopting a strategy of over-provisioning.

A unique feature of RM is the ability to define high-level resources. In previous work [J06a], we modified RUBiS to define an application-specific resource, namely a RUBiS operation as seen from the client's perspective. Given the existence of the RUBiS operation resource, it is then possible to provision service containers that deliver specific RUBiS operation rates using any of the available resource management policies. Note that working with high-level resources does not change the complex dependencies on low-level resources but, provided that these are not under-provisioned, it offers a simpler and more accurate way to provision a given service container. However, to avoid over-committing resources it is still necessary to know the maximum possible usage on a given system. Recall also that, in order to use a share-based policy with a resource, it is

[5] By default this includes many URLs that correspond to static images displayed in the web pages. We removed these in order to focus on the database operations.

necessary to provide the system with a function that returns the maximum usage for a given number of execution units in a given interval.[6]

To test the effectiveness of the mechanisms for share-based policies on high-level resources, we ran tests with three service containers, each running an instance of our modified RUBiS hosted on JBoss 4.0.2 [JBOSS].

We gathered the data for the maximum usage function with a series of runs that varied the number of execution units and measured the maximum throughput under load. Each run was ten minutes long, preceded by a one minute warmup phase. In this test the CPU resource is divided equally among the containers, which then contribute equally to the total transaction rate. The results are shown in Figure 2.[7] Note that, initially, the relationship is essentially linear but ultimately reaches a maximum, beyond which adding extra CPU resources does not improve the throughput. For edample, although not shown in the figure, the aggregate throughput is similar at four CPU units to that at two.

The second test applied the share-based policy to the RUBiS operation resource, in the ratio 1:3:6, at selected numbers of execution units. The maximum usage data from the previous test was fed in as an external parameter, allowing the partition manager to compute the correct limit based on the share value and the maximum usage. The CPU resource remained equally distributed among the three domains. The resulting transaction rate shares, normalized with respect to domain 1 are shown in table 2.

CPU	*D1*	*D2*	*D3*
0.4	1.0	1.69	2.23
0.8	1.0	3.0	4.6
1.2	1.0	3.0	5.29

Table 2: Measured share of transaction rate with equal CPU per domain.

[6] In effect this is one way in which the relationship between the high-level resource and a low-level resource, namely CPU, is established.

[7] Note that the increment in the X-axis is 0.1 of a CPU. For maximum flexibility the number of execution units can be stated as a fractional number of CPU units.

Note that the expected ratio was never achieved for domain D3 that should deliver a share of six. This occurs because that domain was under-provisioned with CPU due to its equal share policy.[8] A similar problem occurs with domain D2 at 0.4 CPU units. This problem can be solved either by allowing overconsumption of unused CPU or by allocating CPU more appropriately, e.g., in the same ratio as the RUBiS shares. Table 3 shows the effect of the latter approach. Note that the ratios are now as expected, although there is some variance at low CPU levels.

CPU	*D1*	*D2*	*D3*
0.4	1.0	3.13	6.29
0.8	1.0	3.02	6.01
1.2	1.0	3.01	6.0

Table 3: Measured share of transaction rate with CPU shared in ratio 1:3:6.

It is interesting to ask whether the given RUBiS share rate could be achieved merely by setting the equivalent share policy on the CPU resource alone. Table 4 shows the results from this test.

CPU	*D1*	*D2*	*D3*
0.4	1.0	1.54	1.9
0.8	1,0	1.48	1.62
1.2	1.0	1.49	1.66

Table 4: Measured share of transaction rate with only CPU shared in ratio 1:3:6.

While the ratios are broadly similar across the CPU range, they are far from the 1:3:6 ratio hoped for. This demonstrates that, for this application, CPU is not a good proxy for the RUBiS transaction resource.

6 RELATED WORK

The idea of scheduling based on shares has its roots in early time-sharing systems [KL88]. Waldspurger and Weihl introduced lottery scheduling and stride scheduling [WW94], [WW95] as mechanisms for efficient implementations of share-based resource management. Resource Containers, which are similar to resource domains, [BDM99] decoupled the traditional operating system binding between a process and re-

[8] The option to allow a domain to exceed its allocated CPU share was not enabled in these runs.

source management. Solaris™ 10 [Sun06] includes an optional fair-share scheduler that allocates CPU using a share-based scheme.

These systems were predominantly concerned with managing the CPU resource and had hard-wired management policies. We are not aware of any system that supports rate-based partitioning of high-level resources using flexible policies that operate outside the kernel.

7 FUTURE WORK

Evidently there are many variations of the simple share-based schemes described here that could be implemented using the same basic framework.

The main limitation of the current implementation is that it performs scheduling at user-level, using the thick-grain and rather imprecise Thread.sleep mechanism. For example, CPU usage is recorded at a grain of 10ms. In addition, tracing revealed that the wakeup from the sleep call frequently is later than expected and the overshoot can be as large as 10%. We attempt to compensate for this in the implementation by adjusting the recorded usage to reflect the actual rate.

Given an adequate and accessible thread scheduler interface, it would clearly improve both the fidelity and the performance overhead by interfacing more closely with the thread scheduler. This would be particularly beneficial for fine grained CPU management.

8 CONCLUSIONS

We have described the design and implementation of three related mechanisms that provide partitioned, rate-based management of resources for the Java platform. The mechanisms build on the basic RM framework developed in the Barcelona project and now being standardized in JSR 284.

The ease of implementing the partitioning manager confirms that the design choice in RM to not build rate-based management into the RM core but to delegate it to policies works adequately. However, finer control over thread scheduling would improve the fidelity of CPU management.

We demonstrated that the rate-based partitioning was effective across several resources with several benchmarks, including the RUBiS application-level benchmark. The latter benchmark demonstrated that high-level resources can be used to express effective policies while providing a more intelligible level of control to administrators tasked with ensuring that service levels are met.

9 ACKNOWLEDGEMENTS

We would like to thank Laurent Daynes, Glenn Skinner and Jeanie Treichel for their careful reviews and helpful comments.

10 REFERENCES

[BDM99] Banga, D., Druschel P., Mogul, J., R*esource Containers: A new facility for resource management in server systems*, Proc. 3rd OSDI, New Orleans, Feb. 1999.

[CD01] Czajkowski, G., and Daynes, L. *Multitasking without Compromise: A Virtual Machine Evolution.* 17th ACM OOPSLA'01, Tampa, Fl, October 2001.

[CHS+05] Czajkowski, G., Hahn, S., Skinner, G., Soper, P., and Bryce, C. *A Resource Management Interface for the Java™ Platform.* Software Practice and Experience, 2005; **35**: 1230157.

[JCP01] Java Community Process. JSR 121: Application Isolation API. http://jcp.org/jsr/detail/121.jsp.

[JCP06] Java Community Process. *JSR 284: Resource Management API*, http://jcp.org/jsr/detail/284.jsp.

[J06a] Jordan, M. *Policy-based management of a JDBC Connection Pool,* Sun Labs Technical Report TR-2006-151, February 2006.

[JBOSS] http://www.jboss.org

[JS05] Jordan, M,, Stewart C., *Adaptive Middleware for Dynamic Component-level Deployment*, Proc. 4th Workshop on Reflective and Adaptive Middleware Systems, Grenoble, France, 2005.

[KL88] Kay, J. and Lauder, P. *A Fair Share Scheduler*, Communications of the ACM, January 1988.

[RUBiS] Rice University Bidding System. http://rubis.objectweb.org

[Sun06] Sun Microsystems Inc., *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*, http://docs.sun.com/app/docs/819-2450.

[Siege05] http://www.joedog.org/siege/

[WW94] Waldspurger, C., Weihl, W., *Lottery Scheduling: Flexible Proportional-Share Resource Management*, Proc. 1st OSDI, Nov. 1994.

[WW95] Waldspurger, C., Weihl, W., *Stride Scheduling: Deterministic Proportional-Share Resource Management*, MIT/LCS/TM-528, 1995.

ABOUT THE AUTHOR

Mick Jordan is a Senior Staff Engineer at Sun Microsystems Laboratories. His interests include programming languages, programming environments, persistent object systems and systems software. He has a Ph.D in Computer Science from the University of Cambridge, UK. He was a member of the team that designed and implemented the Modula-3 programming language. The work described here was done in the context of the Barcelona project which investigated and prototyped technologies to support a Java Operating Environment.