A Scheduling Scheme for Controlling Allocation of CPU Resources for Mobile Programs *

Manoj Lal Raju Pandey
Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{lal, pandey}@cs.ucdavis.edu

August 31, 1999

Technical Report TR-99-05

Abstract

There is considerable interest in developing runtime infrastructures for programs that can migrate from one host to another. Mobile programs are appealing because they support efficient utilization of network resources and extensibility of information servers. In this report, we present a scheduling scheme for allocating resources to a mix of real-time and non real-time mobile programs. Within this framework, both mobile programs and hosts can specify constraints on how CPU should be allocated. On the basis of the constraints, the scheme constructs a scheduling graph on which it applies several scheduling algorithms. In case of conflicts between mobile program and host specified constraints, the schemes implements a policy that resolves the conflicts in favor of the host. The resulting scheduling scheme is adaptive, flexible, and enforces both program and host specified constraints.

*This work is supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Project Agency (DARPA), Rome Laboratory, or the U.S. Government.

1 Introduction

There is increasing interest in computing models that support migration of programs. In these models, a program migrates to a remote host, executes there, and accesses the site's resources. For instance, Java [1] programs are increasingly being used to add dynamic content to a web page. When a user accesses the web page through a browser, the browser migrates Java programs associated with the page and executes them at the user's site. There are many other computing models that support mobility of programs. For instance, the remote evaluation [24] and the servlet [18] models support migration of programs by allowing one to upload a program to a remote site. The mobile programming model [6, 25] supports a more general purpose mobility model that also allows programs to migrate to other sites during their executions. The common element in all of these models is the ability of a runtime system to load externally defined user programs and execute them within the local name space of the runtime system.

The mobile programming model is appealing [6, 16] for several reasons: It allows optimization of resources. For example, if the mobile program encodes an application that must filter huge amounts of data, the application can migrate to the host with the data, execute there and then return with its results to the originating host. The mobile programming model also supports extensibility of information servers. In the traditional remote procedure call (RPC) model, an information server provides access to data through a fixed number of programs. By allowing the user to download arbitrary programs, the behavior of the server can be molded in infinite possible ways. Finally, the model is ideally suited for an extensible and dynamic distributed system structure like the internet. Within this structure, programs search for solutions on the basis of information that they contain. They are location independent, and hence scale well.

Although appealing from both system design and extensibility points of view, mobile programs have serious security implications[19, 12] [1] since they are created at remote sites and they access resources of the host by executing within the local name space of the host runtime system. Security problems occur when the mobile program tries to access and consume resources in a manner that violates the host defined policies (assuming such a policy exists). Mobile programs have the ability to maliciously disrupt the execution of programs at a site by reading and writing into their name spaces, by using unauthorized resources, by overusing resources, and by denying resources to other programs. For instance, a mobile program may try to read files that it is not allowed to read or consume more CPU resources than the host is willing to provide. Uncontrolled access to resources can be dangerous for the host if the mobile program is malignant or buggy. As a result, runtime systems are extremely vulnerable to misbehaving mobile programs. It is important for the runtime system at the host to protect the resources of the host from such mobile programs so that all mobile programs can execute in a safe and secure fashion.

One can think of a runtime system as a service provider that exports two kinds of resources to mobile programs: *system resources* and *conceptual resources*. System resources denote those resources that are implicitly allocated to mobile programs. Typical examples of system resources are memory and CPU. Conceptual resources are like objects and are explicitly defined and managed by a host. They have well-defined interfaces that external programs use to access the resources. For example, a host might provide an interface to a database repository and the incoming mobile program interacts with the database through this interface. In order to protect the host runtime system from mobile programs, the host must have the ability to control allocation of both system and conceptual resources. In this report, we focus on the problem of CPU allocation for mobile programs. CPU resource control protects against mobile programs that try to consume resources more than they are allowed. Mobile programs can stage a denial of service

[1] An example, "Ghost of Zealand" attack. For full details see `http://www.sevenlocks.com/SecurityDigest/SecurityDigestv2n01.htm` or `http://www.finjan.com/applet_alert.cfm`

attack by over-using resources, thereby denying resources to other programs and the runtime system. A mobile program can do this by spawning a number of programs and using more CPU resources through these additional programs.

CPU resource allocation has been studied extensively and several techniques have been proposed for solving the CPU resource allocation problem. The techniques have been primarily proposed in the form of a *scheduling algorithm* or a *scheduling scheme*. A scheduling algorithm is used to construct a schedule for a set of applications that have similar constraints on how CPU resources should be used. For instance, the Earliest Deadline First (EDF) [17] scheduling algorithm is used to schedule real-time applications with deadline based constraints.

A scheduling scheme is a way of composing different scheduling algorithms. A scheduling scheme constructs schedules for a set of applications with different kinds of CPU constraints. For instance, in CPU inheritance scheduling [9], real-time scheduling based on EDF is combined with multi-priority based round robin algorithm [27] for scheduling a mix of real-time and interactive applications.

While several scheduling algorithms and schemes have been proposed, most approaches have focussed on developing schedules that attempt to allocate resources to applications on the basis of constraints (such as lower bound and real-time deadline) that the applications specify. Further they are aimed at achieving general objective functions such as fairness, responsiveness and CPU utilization. However, in mobile systems, a host's objectives have additional components that can be described in terms of host-specific constraints.

The host-specific constraints are driven by two objectives: security and quality of service.

- Security: The security goals of a host are to protect the runtime system and mobile programs from a malicious or buggy mobile program. The host must stop the misbehaving program from getting unlimited or unauthorized access to CPU resources, otherwise such a program can cause denial of CPU to other programs by overusing CPU.
- QoS: The quality of service goals of a host are to provide differentiated services to different mobile programs. This is because the host may have preferences as to how it wants to allocate CPU. Further, it might want to control CPU usage by different mobile programs in order to optimize its own performance or in order to provide specific services. For instance, a host may get overwhelmed by mobile programs that access a popular (and free) service and thus, may not be able to provide resources to other mobile programs. This problem [21] is commonly observed in HTTP servers where requests for popular pages tend to overwhelm requests for less popular and possibly more important pages.

This report presents a CPU scheduling scheme that controls the allocation of CPU resources to a mix of real-time and non-real time mobile programs. The scheme includes the following:

- Specification of program and host-based constraints: Both mobile programs and hosts can specify how CPU resources should be allocated to programs. For instance, real-time mobile programs can request that the host allocate CPU based on specific real-time deadline constraints and hosts can set relative or precise limits on CPU resource usage. In addition, hosts can associate resource usage constraints on the basis of network domains, resources, or other host-specific groupings. Hosts can also specify allocation of resources that vary dynamically as the host state changes.
- Scheduling algorithms: The scheduling scheme consists of three scheduling algorithms: (1) an algorithm for enforcing shares and priority constraints on non-real time mobile programs, (2) an algorithm for enforcing deadline constraints on real-time mobile programs, and (3) an algorithm for enforcing security and limit constraints for both real-time and non real-time mobile programs.

- Algorithm composition policy: The algorithm composition policy defines how the different scheduling algorithms are integrated into a coherent scheme. Our composition policy is aimed at meeting a host's security needs and preferences first, followed by other requirements of mobile programs.

The scheme is general: it can be integrated within the HTTP servers, operating systems such as Solaris or Linux, and mobile programming runtime systems such as the Java Virtual Machine (JVM). For instance, we can include the scheme in HTTP servers to set priority or limit usage over HTTP requests. We can also include it as part of the Java runtime system or browser applications for controlling allocation of CPU resources to non-real time and real-time Java applets.

We developed a detailed simulation model for the scheduling scheme. We have also implemented the scheduling scheme within the Java Virtual Machine (JVM, JDK 1.1). The scheme provides an interface through which a host can specify resource access constraints on CPU usage by a set of mobile programs. At the same time, mobile programs can request different kinds of quality of service in the form of shares, deadlines and priorities. The results from the performance analysis show the following:

- The scheme protects the host by efficiently enforcing the security constraints.
- The host is able to provide different levels of services to different mobile programs.
- The scheme effectively integrates different algorithms for real-time and non-real time mobile programs in a coherent way.
- The scheme is modular and dynamic.

The report is organized as follows. In Section 2 we analyze the problem of CPU resource allocation for mobile programs in detail and define the problem in terms of a set of parameters. We discuss the problems with the existing schemes and examine what is desired from a CPU control scheme for mobile programs.

In Section 3, we present the scheduling scheme along with the different algorithms for satisfying constraints. We review the overall scheme in a piecewise as well as an integrated fashion. We describe the individual algorithms used to construct the scheme in detail.

Section 4 discusses the simulation model and the experiments conducted on the model. We analyze the performance behavior of the scheduling scheme. On the basis of the experiments we discuss the properties of the scheme. In Section 5 we discuss our results and experiences with integrating the scheme with JRTS. Section 6 reviews existing mechanisms for CPU scheduling and CPU resource control. Section 7 gives the concluding remarks and discusses future work.

2 Resource allocation problem

In this section, we look at the problem of CPU resource allocation for mobile programs in more detail. We discuss the different constraints that mobile programs and hosts can specify. We also examine the properties expected of a scheme to control CPU allocation for mobile programs.

2.1 Resource usage constraints

The primary goal of a scheduling scheme is to construct a schedule that satisfies a set of resource usage constraints. A resource usage constraint specifies how resources should be allocated. We classify resource usage constraints both according to how mobile programs want to consume resources and how a host wants to facilitate, as well as protect usage of these resources. This results in two different, possibly conflicting views of resource allocation – client (mobile program) view and server (host system) view. We consider both views of resource allocation and the problems that arise when composing them. We use the term $\sigma_t(P)$ to denote the resources associated with a program P in a *schedulable period* [2] t.

2.2 Client view

From the client's perspective, requests for resources are driven by how applications demand and use resources. Programs want to use as much CPU as possible so that they can perform their job quickly. These sentiments are expressed in terms of several quality of service (QoS) parameters on CPU.

- *Lower bound*: A lower bound constraint, l, associated with a program P specifies that, in case of contention, P must be allocated at least l% of CPU in each schedulable period. That is,
  $\langle \forall\ t : \sigma_t(P) \geq (l/100) \times t \rangle$.

- *Weight*: A weight constraint, w, associated with a program P specifies that P must get w% of CPU in each schedulable period. That is,
  $\langle \forall\ t : \sigma_t(P) = (w/100) \times t \rangle$.
  Note that if there are several programs, the total of weight constraints specified by the programs must be less than 100, since the total CPU usage by the whole system cannot go beyond 100%.

- *Share*: Shares are closely related to weights in that the relative amount of shares owned by a program define the program's weight constraint. Thus, if program P has s shares and the total number of shares in the system is S, P has a weight of $\frac{s}{S}$%. That is,
  $\langle \forall\ t : \sigma_t(P) = (\frac{s}{S}/100) \times t \rangle$.
  The difference between shares and weights is that shares are relative while weights are absolute. For example, if a program P has 20 shares while total number of shares in the system is 50, then P has 40% weight. Now if a new program P' joins the system and it is assigned 50 shares, the total shares in the system is 100. As a result, P has 20% weight now.

- *Deadline*: Real time constraints in the form of $<S,E,T>$ for a program P specify that between times S and E, P must get T amount of CPU. That is,
  $\langle \sum_{t \geq S}^{t \leq E} \sigma_t(P) = T \rangle$.
  Such constraints are specified by programs requesting real time processing such as multimedia applications, or programs that are going to control a unique piece of external equipment, for example a machine tool. A common requirement of real-time programs is that the latency and delay is bounded and if the program is not able to meet its deadline then it is not worth scheduling the program at all.

2.3 Server view

From the server's perspective, two concerns govern the allocation of resources – allocating CPU to mobile programs according to their demands and tightly controlling allocation of resources. Two factors unique to mobile environments accentuate the latter concern. First, a level of distrust exists between hosts and mobile programs since the mobile programs and the hosts typically belong to different administrative domains.

[2] We divide the time line into schedulable periods and specify resource usage constraints in terms of these schedulable units.

Mobile programs can maliciously disrupt a host by using unauthorized resources, by over-using resources, and by denying resources to other programs. Second, in distributed systems such as the web, hosts may provide varying degree of services to clients. A host may differentiate requests from different clients and allocate resources to these requests in accordance with the kinds of services the host wants to provide. For instance, a host may reserve 85% of its resources to mobile programs that originate from paying customer sites and allocate the rest for other programs.

Within the server view, resource control thus involves differentiating and categorizing mobile program requests, and allocating specific amounts of resources to these categories of mobile programs on the basis of the host preferences.

In the mobile programming models [6, 25], the resource allocation problem has an additional dimension: A mobile program can circumvent resource control by migrating to another host and then returning to its previous host for more resources. Resource usage constraints, thus, apply not only to specific executions but to all executions of a program. We refer to such constraints as *lifetime constraints*. An issue closely related to lifetime constraints is that of uniquely identifying and authenticating a mobile program when it re-migrates to the host system. We do not address this issue in this report though we recognize that without sound authentication, a mobile program can easily change its identity and carry out a denial of service attack.

Another important issue in CPU control is that of enforcing constraints that vary *dynamically* with the state of the host. This is because Web-based systems are inherently dynamic; the level of trust placed by the host on a particular external site can change depending on the state of the runtime system. For example, if a remote site sends several mobile programs to the host, it is possible that the remote site is trying to stage a denial of service attack. In such a case, the trust level of the remote site must be reduced and CPU allocation done accordingly. Further, the ability to change resource allocations dynamically allows a host to utilize its resources more effectively. For instance, a host site may change its allocation policy depending on the demand for different kinds of services by mobile programs. Such a scheme will also be useful in HTTP servers where the system can dynamically decide to allocate more resources to service requests for popular pages.

To summarize, the resource allocation problem for mobile programs includes additional resource usage constraints arising out of the server view:

- *Enforcing client constraints:* Such as lower bounds, deadlines, shares and weights.
- *Upper bounds:* To prevent denial of service attacks, hosts specify and enforce upper bounds on resource consumption. An upper bounds constraint, u, associated with a program P specifies that, within each schedulable period, P will be allocated at most u% of CPU. That is,
  $\langle \forall\ t : \sigma_t(P) \le (u/100) \times t \rangle$.
  A host can also specify upper bounds in absolute form, meaning that P is allocated at most u seconds of CPU time.
- *Lifetime constraints:* Hosts enforce lifetime constraints in order to control resource consumption over the whole lifetime of mobile programs. A lifetime constraint, l associated with a program P specifies that P can get at most l seconds of CPU time over all executions of P.
- *Priority:* Given a set of programs to schedule, select the program with the highest priority. Priorities are used to classify programs and to set importance to different classes of programs. For example, in UNIX SVR4 [10] real-time programs occupy higher priority levels, system management programs are at the next level of priority and finally time sharing programs occupy the lowest priority levels.

- *Shares and weights:* Hosts may also want to define share based constraints for mobile programs.

Upperbound and lifetime constraints form the basis for enforcing protection of CPU resources. Priorities and Shares provide differential levels of QoS to different categories of clients.

2.4 The scheduling problem

The CPU resource control problem for mobile programs is, therefore, one of developing a scheduling scheme that, given a set of client and server resource usage constraints, schedules the programs so that the constraints are enforced. In order to enforce the client and server constraints, a scheduling scheme should have the following properties:

- `Flexible:` The scheme should allow for policies to be varied from host to host. Different hosts might want to enforce different constraints. For instance, a particular host may want to allocate 60% of CPU resources for mobile programs accessing a particular knowledge base database. Another host may want to assign a lower priority to such mobile programs as compared to another set of mobile programs which gather information on the current stock situation.
- `Modular:` The scheme should allow for setting constraints for a set of mobile programs relative to the rest of the system. For instance, if a host allocates 40% of CPU to mobile programs originating from site *A* and remaining 60% to mobile programs from site *B*, any changes in CPU allocation for programs in *B* should not affect allocation for programs in *A*.
- `Dynamic:` The scheme should be able to dynamically adapt to the state of the host system in order to enforce dynamic constraints.
- `Security:` The scheduling scheme must make sure that the security policy of the host is never breached. The scheme must, therefore, strictly enforce the upper bound and lifetime constraints even if it means that the quality of service constraints are not satisfied.
- `Conflict resolution:` It is quite possible that a client requests resources more than the host is willing to provide. A scheme must, therefore, include a set of policies, called *algorithm composition policies*, that specify how conflicts among different resource usage constraints are resolved.
- `Efficiency:` The scheduling scheme should have a low overhead in making scheduling decisions.

3 Resource allocation scheme

In this section we describe our scheduling scheme in detail. We first describe the overall approach for scheduling resources to mobile programs.

- Construction of scheduling graph: The scheme partitions mobile programs into real-time (deadline) and non real-time programs. It captures group-subgroup relationships among mobile programs along with various constraints to construct a scheduling graph.
- Application of algorithms: The scheme applies three algorithms to the scheduling graph: (i) an algorithm that enforces upper bound and lifetime constraints; (ii) an algorithm that enforces share and priority constraints; and (iii) an algorithm to enforce real-time deadline based constraints.

- `Monitoring of system state`: Since the host can specify constraints as a function of state variables, the scheduling scheme monitors the state of the system and adapts to the changes in the resource constraints by modifying the scheduling graph.

In the following sections, we describe the individual algorithms and how they are composed to build the scheduling scheme.

3.1 Framework for specifying constraints

We first describe how hosts and clients can specify resource usage constraints. For the purpose of this study we have considered the following kinds of resource usage constraints: share, priority, real-time deadline, lifetime and upperbound constraints. Also, we consider upper bounds that are specified in absolute form only and not relative upper bounds. We believe that other constraints, such as weights and relative upper bounds can be easily added to our model. We have developed a runtime interface and a specification language that programs and hosts can use to specify resource usage constraints.

3.2 Group definition

We use the notion of *groups* [19] as the basis for associating resource constraints with a single or a set of mobile programs. A group is an individual mobile program or a collection of groups. For instance, a group `ucdavis.edu` denotes all mobile programs that originate from this domain. This group may contain subdomains such as `cs.ucdavis.edu` and `ece.ucdavis.edu`. The notion of groups and subgroups results in a hierarchical partitioning of mobile programs. The definition of a group need not necessarily be based on network domains. A group can be defined on the basis of the kinds of services a host provides. For instance, mobile programs accessing a particular database can be made part of a database group irrespective of the network domains the programs originate from.

Clients can define their own groups and determine constraints for individual jobs within the group. When a new mobile program becomes part of a group, the group owner decides the constraints for the mobile program.

3.3 Constraint specification

The specification language provides for specifying the following kinds of constraints: real-time constraints based on deadlines, non real-time constraints, and upper bounds. We describe them below.

3.3.1 Non real-time constraints

Resource allocation for non real-time mobile programs is done on the basis of share or priority constraints. Each rule specifies the type of constraint it is enforcing (share or priority) and also a set of `<condition,value>` tuples. A condition is specified as a `<type,type range>` tuple, where type specifies a runtime system variable, and the range specifies the range of values of the type for which the condition will be true. For example, conditions can be of the following form:

$Condition_1$: `<number of mobile programs, 10-20>`
$Condition_2$: `<number of mobile programs, default>`

This specifies $Condition_1$ to correspond with the number of mobile programs being within the range 10 and 20. $Condition_2$ corresponds to number of mobile programs not being within the range 10 and 20. A parent group can now specify a priority or share based rule for a child group as follows:

> $Rule_1$: <group.shares, <$Condition_1$, 30>, <$Condition_2$, 70»

$Rule_1$ specifies that if $Condition_1$ holds, then the group has 30 shares, while if $Condition_2$ holds then it has 70 shares. Hosts can thus specify constraints as functions of state variables, and the scheduling scheme monitors the state of the system and adapts to the changes.

3.3.2 Real-time constraints

Resource allocation for real-time mobile programs is done on the basis of deadline based constraints. The host defines a real-time guarantee group (RT_G) which contains all mobile programs whose deadline based constraints can be specified. Each program has a non real-time scheduling constraint (in the form of priority or share) that is applied within the program's scheduling group. If the program makes a real-time request, and if that request can be satisfied by the scheduler, then the program is scheduled as part of the RT_G group. Resource allocation for real-time programs is done on the basis of the following rules:

> RT_G.upperbound = val1 (specified by the host)
> group.RT_Gbandwidth = val2 (requested by the group)
> mobileprogram.deadline = $<S,E,T>$ (requested by the mobile program)

The first rule specifies that the upper bound on the time reserved for RT_G is val1. The second rule specifies that group has reserved val2 bandwidth within RT_G for allocating to group's mobile programs. The final rule specifies the deadline based constraint as requested by a mobile program. The rules are described in more detail later.

3.3.3 Upperbound constraints

A host can specify upper bound and lifetime constraints through the following rules:

> UpperboundRule.members = < group1, group2, ... >
> $Rule_2$: <UpperboundRule.value, <$Condition_1$, val1>, <$Condition_2$, val2»
> LifetimeRule.members = < group1, group2, ... >
> $Rule_3$: <LifetimeRule.value, <$Condition_1$, val1>, <$Condition_2$, val2»

The first rule specifies the groups on which the upperbound constraint is applied. The second rule specifies that the upperbound has val1 value if <condition> is true else it has val2 value. The same holds for lifetime constraints.

3.4 Construction of scheduling graph

The scheduling scheme first builds a *scheduling graph* from resource usage constraints. We show an example of a scheduling graph in Figure 1. The scheduling graph contains three subgraphs:

- Real-time,
- Non real-time, and

- Upper-bound.

The real-time subgraph is a single node (a real-time guarantee group, RT_G) containing all mobile programs that specify deadline based constraints. The RT_G graph consists of all real-time mobile programs as children of the root node of the real-time guarantee graph ($RT_G Root$). The construction of RT_G is different from the non real-time subgraph (NRT_G in Figure 1). The difference arises due to the nature of constraints for real-time and non real-time programs. Constraints specified for non real-time programs are relative to each other (priorities, shares). This is not the case with real-time programs where constraints are in terms of absolute time. Scheduling real-time programs requires that every program be allocated so that their deadlines are satisfied. Therefore, all real-time programs are at the same level of the hierarchy.

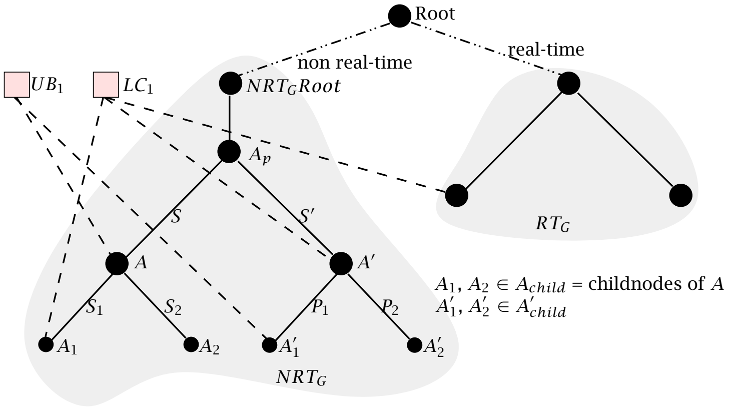


Figure 1: The scheduling graph

NRT_G is a hierarchical graph where each node denotes a group and each edge denotes a group-subgroup relationship. Mobile programs are at the leaves of NRT_G. The edges of NRT_G are annotated to denote shares or priority constraints associated with groups. For instance, in Figure 1, the label on edge (A_p, A) specifies that group A has S shares within group A_p. Similarly, the label on (A', A'_1) specifies that mobile program A'_1 has priority P_1 within group A'. The share or priority constraints for each node/group is relative to its parent group. For instance, group A's share S of CPU resources are with respect to the CPU resources allocated to its parent group, A_p. This results in a modular allocation of CPU: Any allocation of CPU to a child group depends only on the allocation to the parent group. In this way, changes in CPU allocation to a child group affect only the siblings. Programs within the child group do not affect resource allocations to other non-overlapping groups.

The upper bound subgraph represents the security constraints. Nodes in this subgraph denote specific upper bound and lifetime constraints. Edges link these constraints to the relevant groups and mobile programs. Upper bound and lifetime constraints are general in a sense that they can encapsulate more than one node in the scheduling graph. Moreover, the nodes encapsulated by a particular constraint need not be at the same level. For instance, as shown in Figure 1 the upper bound constraint UB_1 applies to group A and mobile program A'_1. There is a need for such constraints so that the host can control mobile programs belonging to different levels in the hierarchy. For example, suppose a site wants to impose an upper bound constraint that mobile programs accessing a particular database be allocated at most 10% of CPU. Such mobile programs may exist in different groups and may span multiple subtree domains.

As the runtime system starts, it initializes the scheduling graph – it creates a graph consisting of empty RT_G and possibly non-empty NRT_G. When a new client program arrives, the host creates a new group node (C) for the client, specifies constraints for C, and adds C at an appropriate place in the scheduling graph. The client program can create subgroups under C and define resource usage constraints for the subgraph under C. The host can specify upperbound and lifetime constraints for the subgraph. It can and also override constraints specified by C and specify its own constraints for the subgraph under C. In this manner the scheduling scheme maps client and server constraints to the graph.

3.5 Application of algorithms

An important aspect of the scheduling scheme is the algorithm composition policy that resolves conflicts among host and mobile program resource usage constraints. Our scheduling scheme implements an algorithm composition policy that always resolves conflicts in favor of server constraints. The policy is summarized as follows:

1. It first ensures that the security related constraints are always enforced. It always applies the upper bounds algorithm first in order to enforce the upper bound and lifetime constraints even if it means that the mobile programs do not get their requested CPU allocation or that some deadlines are missed.
2. Next, it enforces host-specified priority and share constraints in order to implement host's preferences.
3. non real-time jobs are then allocated according to their relative shares, whereas real-time jobs are scheduled so that their deadlines constraints are met.

We now describe the scheme in more detail. The scheme partitions the continuous time line into small *quantum time chunks* (see Figure 2). Within each quantum time chunk, mobile programs from the real-time group are scheduled according to their reservations. The reservations fix the times when CPU is allocated to real-time programs. This is shown as shaded parts in a quantum time chunk. The remaining time is allocated to non real-time programs. The scheduling of non real-time mobile programs starts from the root node of NRT_G graph (NRT_GRoot). The scheme traverses from NRT_GRoot to one of the leaves of the graph.

In Figure 3, we describe the overall working of the schedule function for one single quantum time chunk. In the next sections we describe the individual algorithms.

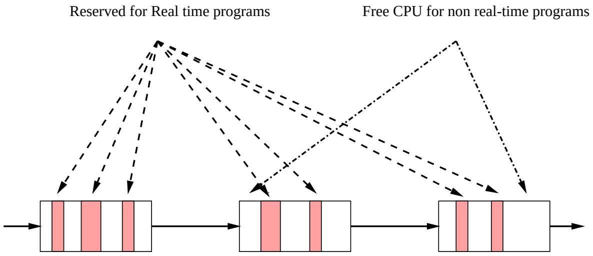

Reserved for Real time programs
Free CPU for non real-time programs

Figure 2: List of quantum time chunks with reservations for real-time programs

3.6 Scheduling of non real-time programs

The crux of the algorithm for non real-time programs is the decision making at each non-leaf node. At each non-leaf node, the algorithm considers the constraints associated with the children nodes of the node. If the children nodes have priority based constraints, the algorithm selects the child node with the highest

```
for (;;) {
    t = next_time_quantum();
    while (in current quantum time chunk) {
        if (time to schedule a real-time mobile program) {
            Check for upper bound and lifetime constraints;
            schedule the mobile program;
            update lifetime and upper bound constraints;
        }
        else {
            //schedule from the remaining (non real-time) hierarchy
            currentnode=Root of the non real-time programs;
            while (currentnode is not a leaf node) {
                Check for upper bound and lifetime constraints;
                if (constraints of children nodes based on priority) {
                    currentnode = select childnode with earliest priority;
                }
                else if (constraints of children nodes based on shares) {
                    currentnode = select childnode on the basis of shares;
                }
            }
            schedule the leaf node;
            update lifetime and upper bound constraints;
        }
    }
}
```

Figure 3: The scheduling scheme

priority. If the children nodes have share based constraints, the algorithm selects a child node on the basis of the share allocations or the children nodes.

The algorithm to allocate CPU on the basis of share based constraints extends the ideas in the SMART scheduling algorithm [20] to a hierarchy. SMART defines two numbers for each application – a *virtual time (VT)* and a *virtual finish time (VFT)*. The notion of VT and VFT was developed in fair queuing algorithms for congestion control in network protocols [15] and has been used in CPU scheduling in SMART and Stride scheduling[29].

We extend the notion of virtual time to define three entities – an *upper virtual time (UVT)*, a *virtual finish time (VFT)*, and a *lower virtual time (LVT)*. First, we present the intuition behind virtual time and virtual finish time and give their formal definitions as they are used in the SMART system. We then define UVT, VFT and LVT.

In the SMART system applications are partitioned into different priority queues. Applications within each priority queue have shares. The system associates a VT with with each application and priority queue.

- *VT of priority queue P:* Initially:

$$VT_P(t) = 0 \tag{1}$$

  At a later time, if an application within P was initiated for execution at time (τ) and is currently (t) executing:

$$VT_P(t) = VT_P(\tau) + \frac{t - \tau}{\sum\limits_{A \in P_{member}} S_A} \tag{2}$$

  where A is an application in P and S_A represent A's shares.

- VT of application: When an application A joins the priority queue P for the first time at time t:

$$VT_A(t) = VT_P(t) \tag{3}$$

At a later time, if A was initiated for execution at time (τ) and is currently (t) executing:

$$VT_A(t) = VT_A(\tau) + \frac{t - \tau}{S_A} \tag{4}$$

where S_A represent A's shares.

The virtual time of an application measures the degree to which the application has received its proportional share of CPU on the basis of its share allocation. The difference between $VT_A(t)$ and $VT_P(t)$ gives a measure of whether A has received its share-based allocation. If $VT_A(t)$ is less then $VT_P(t)$, A has received less than its share and vice-versa. The virtual time for an application advances at a rate inversely proportional to the number of shares it holds. If an application has a large number of shares, its virtual time will increase at a smaller rate and, therefore, the application will be scheduled more often to make its virtual time same as that of the queue.

The virtual finish time of an application refers to its virtual time if the application had been selected for the currently executing time quantum.

- VFT of application: When application A joins queue P at time τ:

$$VFT_A(\tau) = VT_P(\tau) + \frac{Q}{S_A} \tag{5}$$

where Q is the time quantum. Later, when A has been scheduled for some time and now is going to be stopped (t):

$$VFT_A(t) = VFT_A(\tau) + \frac{Q}{S_A} \tag{6}$$

where τ is the time when VFT_A was last changed.

A property of virtual finish time is that it does not change while the application is executing. It changes only when the task is rescheduled. *The scheduling algorithm selects the application with the smallest virtual finish time from the highest priority queue for scheduling.*

To extend the idea of virtual time to a hierarchy, we define three quantities: upper virtual time (UVT), virtual finish time (VFT) and lower virtual time (LVT) for each node in the hierarchy. The reason we require UVT and LVT is that in NRT_G, each internal node is both a child node and a parent node. UVT of the internal node is compared with the LVT of the parent node to select the child node that should be scheduled.

Assume that the algorithm has reached an internal node A, and the children nodes of A have share based constraints associated with them. Let A_p be the parent of A, and let A own S_A shares under A_p. Let A_{child} be the set of children nodes of A. Also let each A_i in A_{child} own S_i shares.

- LVT: The lower virtual time at A is used for selecting from one of A's children.
Initially, when A joins the hierarchy:

$$LVT_A(t) = 0 \tag{7}$$

Later, if a mobile program from the subtree within A was initiated for execution at time τ and is currently (t) executing:

$$LVT_A(t) = LVT_A(\tau) + \frac{t - \tau}{\sum\limits_{a \in A_{child}} S_a} \tag{8}$$

- UVT: When A joins the hierarchy for the first time at time t:

$$UVT_A(t) = LVT_{A_p}(t) \tag{9}$$

  Later, if a mobile program from the subtree within A was initiated for execution at time τ and is currently (t) executing:

$$UVT_A(t) = UVT_A(\tau) + \frac{t - \tau}{S_A} \tag{10}$$

- VFT: The VFT of a node A is its UVT had A been selected for the current quantum. When A joins the hierarchy for the first time at time t:

$$VFT_A(t) = UVT_A(t) + \frac{Q}{S_A} \tag{11}$$

  where Q is the quantum size. Later, when a mobile program from within A was initiated for executions at time τ and now (t) some other program is going to be scheduled:

$$VFT_A(t) = VFT_A(\tau) + \frac{Q}{S_A} \tag{12}$$

The algorithm selects the child node with the earliest virtual finish time (VFT). To summarize the scheduling algorithm for non real-time programs: The scheduling of non real-time programs starts at NRT_GRoot(Figure 1). The algorithm traverses down the tree till it reaches a leaf. At an internal node A, the algorithm looks at the constraints associated with the children nodes of A. If the children nodes have priorities, the algorithm selects the child node with the highest priority. If children nodes have shares, the child node with the earliest VFT is selected. If the node selected is a mobile program, it is scheduled for execution, otherwise the process is repeated.

3.7 Scheduling of real-time programs

The scheduling of real-time mobile programs is based on the scheduling algorithm in Rialto [14]. Rialto uses a *precomputed scheduling graph* to implement continuously guaranteed CPU reservations with application defined periods, and to guarantee time constraints. Applications make *CPU reservations* in the form of "reserve X units of time out of every Y units". Real-time applications request CPU resources by specifying *time constraints* of the form <S,E,T>. On the basis of the CPU reservations, Rialto constructs a `Rialto scheduling graph`. The nodes in the `Rialto scheduling graph` indicate either reserved time periods for applications or free time not reserved for any application. The time constraints for threads are then satisfied from the reserved time periods and from any free time that might be available.

The real-time scheduling in our scheme differs from the problem solved in Rialto in several ways: Our problem is a simpler instance of Rialto where we don't consider continuous *CPU reservations* of the form "reserve X units ....". Instead we define CPU reservations over discrete base period, ie, quantum time chunks. With the above modification in the problem statement, there is no need of computing the `Rialto scheduling graph`. However, the RT_G scheme is more general, since CPU reservations for time constraints can be carried out from any place in the base period rather than from some fixed locations in the Rialto scheduling graph. Also, there are additional constraints in the form of upper bounds.

Within each quantum time chunk, the real-time programs are scheduled according to their reservations. The reservations fix the times at which CPU is allocated to real-time programs (Figure 2). In this section we look at the algorithm used for making reservations for real-time mobile programs.

Resource allocation for real-time programs is done on the basis of the rules described earlier. We first describe the rules in more detail.

- RT_G.upperbound = val1: An upper bound on the time reserved for RT_G within each quantum time chunk. This prevents starvation of non real-time programs.
- group.RT_G-bandwidth = val2: Groups can reserve bandwidth within RT_G so that deadline based constraints for member mobile programs can be satisfied from the reserved bandwidth.
- mobileprogram.deadline = $<S,E,T>$: A mobile program within a group can request that its time constraints be satisfied by utilizing the bandwidth reserved for its parent group. If there is no bandwidth reserved for the parent group, the program will get only unreserved RT_G bandwidth to satisfy its constraints.

The scheduling algorithm allocates time within the quantum time chunks to satisfy reservation requests. The use of quantum time chunks is similar to the notion of *slot lists*[22]. While the slot list method considers only real-time applications, our scheduling scheme integrates the idea of slot lists with scheduling for non real-time programs as well. Moreover, in RT_G scheduling, CPU time available in each quantum time chunk is constrained by upper bound on RT_G.

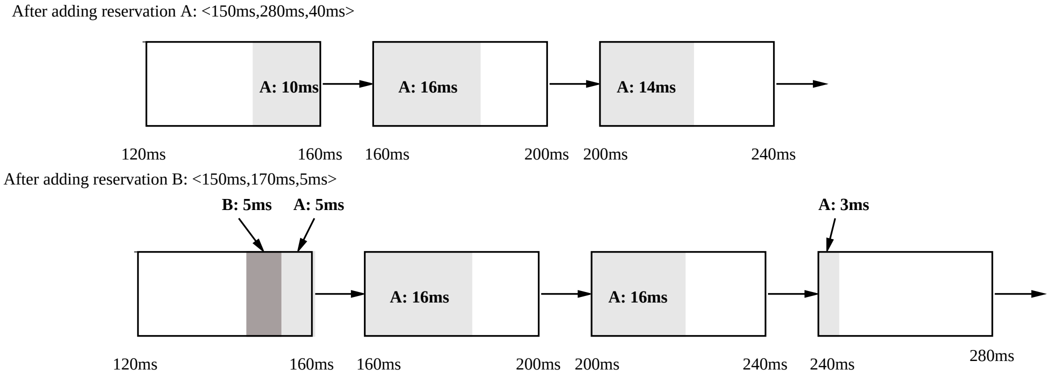


Figure 4: List of quantum time chunks for two reservations

The real-time algorithm first reserves the bandwidth for each group in each quantum time chunk. For each <S,E,T> constraint, the scheduling algorithm makes reservations in the quantum time chunks (Figure 2) that fall within times S and E. The algorithm reserves the computation time T from within the parent group's reserved bandwidth, if any, and any free unreserved RT_G bandwidth that might be available within the quantum chunk. It does so by creating reservation nodes in each quantum time chunk. The reservation nodes specify the start time, the time reserved, and the mobile program for which the time has been reserved.

We demonstrate the features of the RT_G scheduling algorithm by examples. Assume that the size of a quantum time chunk is $40ms$. Also assume that a host specifies an upper bound of 40% ($16ms$) on RT_G in each quantum time chunk. This means that a real-time program can get at most $16ms$ CPU in a quantum time chunk. A group RT_A can reserve 25% ($4ms$) of RT_G time in each quantum time chunk for RT_A's member real-time programs. In this case a program A_1 that belongs to RT_A group gets $4ms$ of reserved time and $12ms$ of unreserved time in each quantum time chunk. On the other hand, a mobile program B that is not a member of RT_A gets at most $12ms$ of RT_G time in each quantum time chunk.

When a new real-time program arrives, the algorithm performs a feasibility check to determine if the deadline request can be met. It goes through the list of quantum time chunks, reserving any available RT_G time for the request. If the program's deadline cannot be met, any reservation made for the program is freed. In the process of carrying out the feasibility checks, the algorithm performs a rearrangement of any reservations already made for earlier programs so that the deadline based constraints are added in the Earliest Deadline First (EDF) [17] order. Using EDF for adding new reservations improves the algorithm so that the number of reservation requests satisfied is increased. In Figure 4, we show how the algorithm makes reservations for two requests: reservation A (<150,280,40>) and reservation B (<150,170,5>) in that order. The size of a quantum time chunk is $40ms$. The host has specified an upper bound of 40% ($16ms$) for RT_G. We assume that all the CPU time ($16ms$) for RT_G is available to the mobile programs, that is a group has not reserved any bandwidth from the RT_G group. When request A is made, the algorithm greedily reserves any RT_G time available to A. When request B arrives, the algorithm rearranges the reservation for A so that the constraints of B can also be satisfied. This is done because B has an earlier deadline. If rearrangement is not done, then B cannot be guaranteed its constraints because all RT_G time has been used by A.

Figure 5 describes the algorithm for making a real-time reservation.

3.8 Resource usage control

The upper bound subgraph captures the upper bound and lifetime constraints on groups and mobile programs. Each security node in the graph maintains the usage information for the groups and programs that the node monitors. As the scheduling scheme traverses the scheduling graph, it checks the security node associated with a node before it applies any scheduling algorithm to the node. If selecting a program from within that node will cause an upper bound or a lifetime constraint to be violated, the particular internal node is not selected. For example, assume that the scheme decides to schedule a program in the subtree under A_p in Figure 1. Before it decides between nodes A and A', the scheme checks with the security nodes that control A and A' (UB_1 for A and LC_1 for A') to ensure that the two nodes do not violate any constraints. The scheme then employs the selection algorithm as described earlier to select one of the two.

In order to control CPU usage by programs in the real-time (RT_G) group, the host can specify upper bound on the CPU time available to the RT_G group in each quantum time chunk. In addition, there can be upper bounds and lifetime constraints on individual real-time mobile programs. As for non real-time jobs, the scheme checks for any violation of these constraints before it applies the real-time scheduling algorithm.

Note that since the resource requirements of real-time mobile programs are known before hand during reservation, it appears that checks for security constraints can be performed during the reservation phase itself. We do not use this technique because the upperbound constraints that control a real-time program might also control CPU allocation for other non real-time groups (Figure 1). It means that the checks for upper bounds cannot be applied because the CPU usage for the non real-time programs in the future cannot be predetermined.

4 Implementation and Performance Analysis

To assess the behavior of the scheduling scheme, we first implemented the scheme as part of a simulation engine and conducted several experiments using the simulation engine to analyze the performance behavior of the scheme. Once we were satisfied with the scheme, we then implemented the scheduling scheme within the Java virtual machine (JVM). We analyzed its behavior within the JVM as well. In this section, we describe the simulation engine. The next section describes the implementation within the JVM.

```
int reserve(S,E,T)
{
    //get the set of real time requests interfering with
    //the current reservation, and that have later deadlines than
    //the current request.
    I = set of interfering requests;
    Remove_reservations(I); // remove reservations for I
    //see if current request can be satisfied
    int result = Try_adding(S,E,T);
    if (result = true) {
        //if previous requests can still be satisfied
        int result = Try_adding(I);
        if(result = true)
            return result;
        else {
            //not able to satisfy previous requests
            //with the new one, revert to earlier situation
            Remove_reservations(S,E,T);
            Try_adding(I);
            return false;
        }
    }
    else {
        // not able to satisfy new request, revert to earlier situation
        Remove_reservations(S,E,T);
        Try_adding(I);
        return false;
    }
}

int Try_adding(S,E,T) //add the new request <S,E,T>
{
    t = get_time_quantum(S);
    t'= get_time_quantum(E);
    while (all T not reserved) {
        Check for upper bound on RT_G in current quantum time chunk;
        if (upper bound on RT_G not reached) {
            reserve any RT_G time available;
        }
        if(all T not reserved) {
            t = next_time_quantum();
            if(t > t') {
                //unable to satisfy the current request
                return false;
            }
        }
    }
    return true;
}
```

Figure 5: Algorithm for making real-time reservations

The primary goals of the experiment were to address the following issues:

- How effective is the scheme in satisfying both real-time and non real-time constraints?
- How does the scheduling scheme behave when upper bounds and lifetime constraints are enforced?
- What is the scheduling behavior of a system when the resource allocation constraints are changed dynamically?

We first describe the simulation engine. Next we describe the different experiments in detail.

4.1 The simulation engine

The simulation engine provides an API for creating groups, specifying group memberships, constraints and mapping the constraints to the groups. After reading the various specifications, it builds a scheduling graph, creates virtual threads for mobile programs, and simulates the scheduling of the virtual threads. We simulate time by keeping a virtual timer. Whenever a virtual thread is selected for scheduling, we advance the virtual time by the scheduling quantum and charge the quantum to the virtual thread.

The simulator is organized into two sets of APIs: the interface and the hierarchy.

Interface

The interface API provides for specifying

- Group creation/deletion: `CreateGroup()`, `DeleteGroup()`.
- Thread creation/deletion: `CreateThread()`, `DeleteThread()`.
- Constraint specification: Priority, shares, upper bound, lifetime constraint, deadline.
- Group membership mapping: `AddtoGroup()`, `RemovefromGroup()`.
- Constraint to Group mapping: `SetGroupConstraint()`, `SetThreadConstraint()`.

The interface API builds the structures containing all the information for the hierarchy to construct the scheduling graph.

Hierarchy

The hierarchy API provides the following:

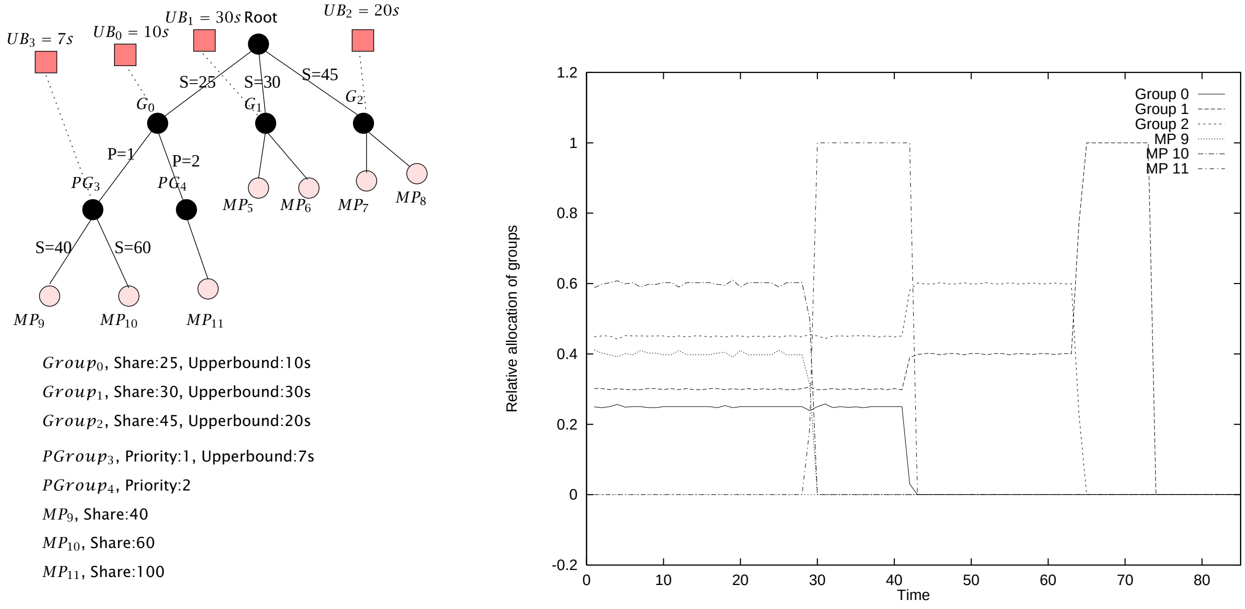
- Reading in the group structure built by the interface API and constructing the scheduling graph: `AddGroup()`, `RemoveGroup()`.
- Looking at the various constraints from the interface structure and setting them in the scheduling graph. For example, a priority rule in the interface structure says that, between times *A* and *B* priority is P_1 and otherwise priority is P_2. The scheduling graph will just specify the current priority status.
- The scheduling primitives:
  - `SCHEDULE()` to select an eligible thread.
  - `AddThread()` to add a thread.

- `SuspendThread()` to suspend a thread.
- `MakeThreadRunnable()` to make a suspended thread runnable.
- `RemoveThread()` to remove a thread.
- `ReAddThread()` to simulate a mobile program re-migrating to the system. In such a case, lifetime constraints are consulted to verify if the thread can be added to the system or not.

A *Tree* structure maintains the non-real time graph (NRT_G). Each node in the scheduling graph contains specifications for the priority or share constraints, a reference to the upperbound and lifetime constraints for the node and the number of threads within the subtree under the node. The real time graph (RT_G) is maintained as a queue of reservation requests.

When a new thread is added or removed, the interface API is consulted to get the values of the constraints, in case they depend on the number of threads. The scheduler also maintains a list of time based events to implement time based dynamic constraints. The scheduler periodically consults with the event list to update any constraints. SCHEDULE function is very similar to the Scheduler function described in Figure 3. For experimental purposes, the time quantum is set to $5ms$.

4.2 General scheduling behavior



(a) The scheduling graph

(b) Relative allocation of CPU for groups and mobile programs

Figure 6: General scheduling behavior of the scheme

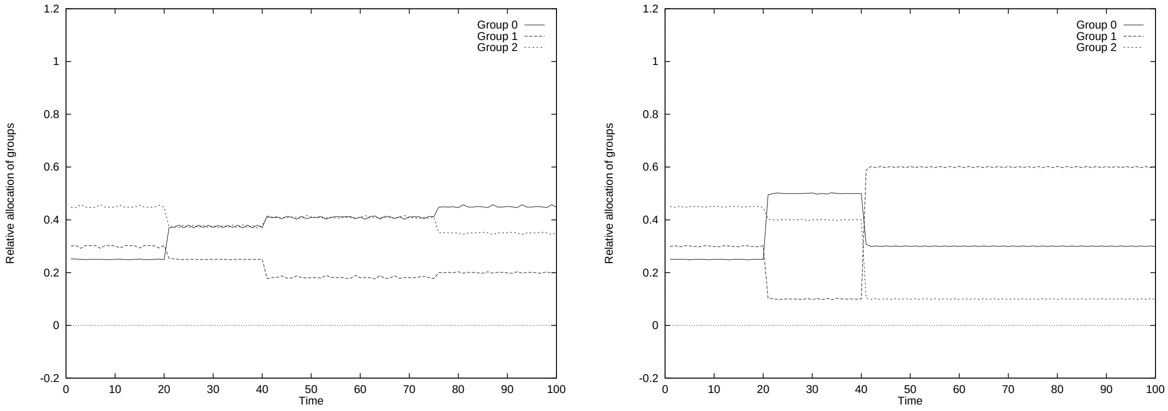
The first experiment demonstrates how the scheme schedules groups of mobile programs that are constrained by shares, priorities and upper bounds. Further, it shows how upper bound constraints interact with shares and priority constraints. In Figure 6(a), we show the hierarchy constructed from the client and host resource usage constraints. In Figure 6(b), we show the relative CPU allocations of groups G_0, G_1 and G_2. We also show the relative CPU allocations of mobile programs MP_9, MP_{10}, and MP_{11}.

Between times $(0, 40s)$, G_0, G_1 and G_2 get 25%, 30% and 45% of the CPU respectively which matches their share allocations. At time $40s$, G_0 reaches its upper bound. This results in relative allocation for G_1 and G_2 to increase to 40% and 60% respectively that corresponds to the share ratio of $30 : 45$. When the upper bound of G_2 is reached, G_1 is the only group and it gets all the CPU resources till its upper bound is achieved as well.

Within G_0, the relative allocations of mobile programs MP_9 and MP_{10} are 40% and 60% respectively, according to their share allocations. MP_{11} is not scheduled in the beginning because it belongs to a lower priority group. At time $28s$, upper bound for PG_3 is achieved and then mobile programs from PG_4 are scheduled till the upper bound for G_0 is reached.

The scheme, thus, effectively implements relative allocations of resources within hierarchies of groups. Further, it enforces upper bounds constraints as well. Note that changes in CPU allocation to MP_9, MP_{10} and MP_{11} (programs in G_0) do not affect the allocation to G_1 or G_2. This highlights the modularity of the scheme.

4.3 Dynamic nature of the scheme



(a) Constraints as a function of number of mobile programs

(b) Constraints as a function of time

Figure 7: Dynamic nature of the scheme

In the second set of experiments, we show that the scheme dynamically adapts to changes in resource constraints. We use the scheduling graph of Figure 6(a) for the experiments. The share and the priority constraints are as specified in the graph. We remove the upper bound constraints for these experiments. The first experiment, depicted in Figure 7(a), demonstrates the allocation of CPU to the programs when resource usage constraints depend on the number of mobile programs:

G_0: If $n_{mp} < 4$, the group has 25 shares, otherwise it has 45.
G_1: If $n_{mp} < 3$, the group has 30 shares, otherwise it has 20.
G_2: If $n_{mp} < 3$, the group has 45 shares, otherwise it has 35.

Here n_{mp} denotes the number of mobile programs within a group. At time $20s$, a new mobile program is added to PG_4, resulting in a change in relative allocation. Another program arrives at time $40s$ and is added to G_1. Another program is added to G_2 at time $75s$. During time periods $(0, 20)$ and $(20, 40)$, the relative allocations for groups $G_0 : G_1 : G_2$ are 25 : 30 : 45 and 45 : 30 : 45 respectively. These allocation match the specified constraints. Similarly, the relative allocations during the periods $(40, 75)$ and $(75, 100)$ also satisfy the share constraints.

The second experiment (Figure 7(b)) demonstrates the allocation of CPU when constraints are time dependent. The share specifications for the groups are as follows:

Time 0 to 20s: The share allocation for $< G_0, G_1, G_2 >$ is <25,30,45>.
Time 20 to 40s: The share allocation is <50,10,40>.
Time 40 to 100s: The share allocation is <30,60,10>.

Between time $(20, 40)$ the relative CPU allocation is 0.5 : 0.1 : 0.4; corresponding to the shares values. The CPU allocation changes as the share allocation changes.

4.4 Lifetime constraints

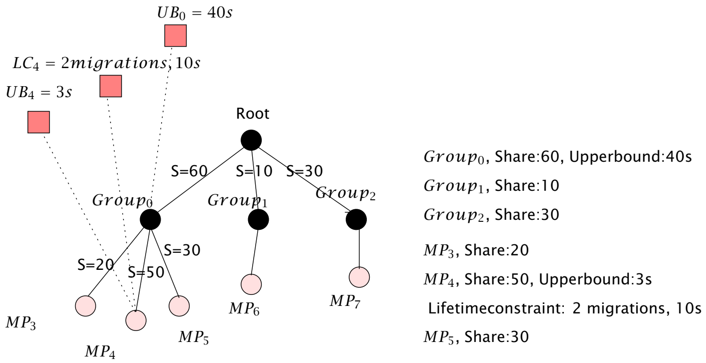


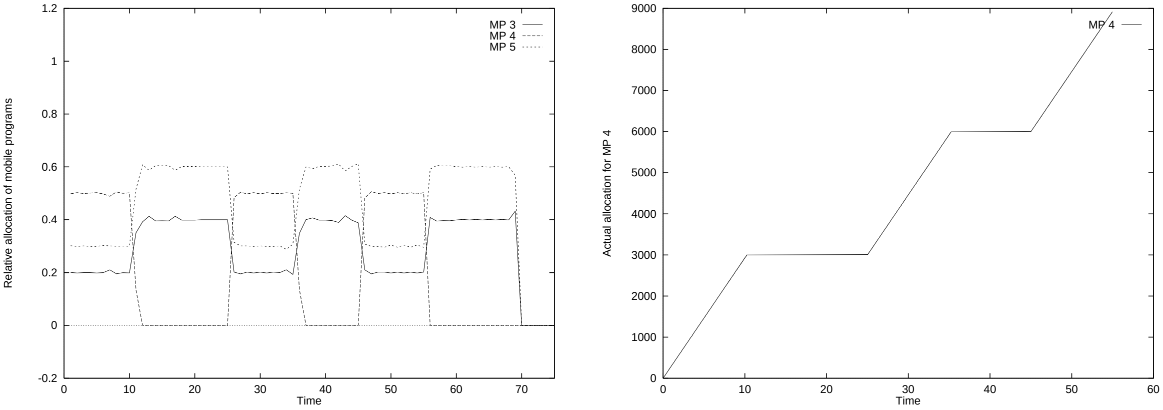
Figure 8: The scheduling graph for lifetime constraints

This experiment demonstrates how the scheme enforces control defined by lifetime constraints. Figure 8 shows the scheduling graph. MP_4 has lifetime constraint of 2 migrations, and a total lifetime usage of $10s$. MP_4 also has upper bound of 3s. At time $t = 20s$, MP_4 leaves the host site. It migrates back at $t = 25s$. It again leaves at $t = 40s$ and migrates back at $t = 45s$. It finally leaves at $t = 55s$ and is not allowed to execute when it migrates back for the third time. Figure 9(a) shows the relative allocations of mobile programs within the group. The relative allocations for MP_3 and MP_5 go up when MP_4 leaves. Allocation to any of the programs within the group stops when the group's upper bound is reached. Figure 9(b) shows the actual allocation for MP_4. The execution of MP_4 stops when its upper bound is reached. Then when it migrates back, it again gets allocated till its upper bound is reached. After the third time, the mobile program is not allocated anymore since its lifetime constraint of 2 migrations has been achieved.

4.5 Real-time programs

In the third set of experiments, we test the scheme's effectiveness in enforcing deadline based constraints for real-time programs.

For the first experiment (Figure 10(a)) we simulate the execution of an application that displays real time video streams from the local storage. The video input stream contains frames in JPEG compressed format at 15 frames/sec. We assume that the estimated execution time per frame to be about 30ms [20]. The application makes reservation requests for each frame within a 100msec period. If the reservation



(a) Relative allocation of mobile programs

(b) Actual allocation for mobile program 4

Figure 9: Results for lifetime constraints

is granted then the application displays the frame; otherwise it skips the frame. The graph (Figure 10(a)) shows how the upper bounds and reserved bandwidth affect real time applications. The individual plots in the figure show the number of JPEG frames rendered per second as a function of the reserved bandwidth for the application. The different plots correspond to the upper bound set on the RT_G group in each quantum time chunk. As the amount of reserved bandwidth decreases, the number of frames rendered/second also decreases.

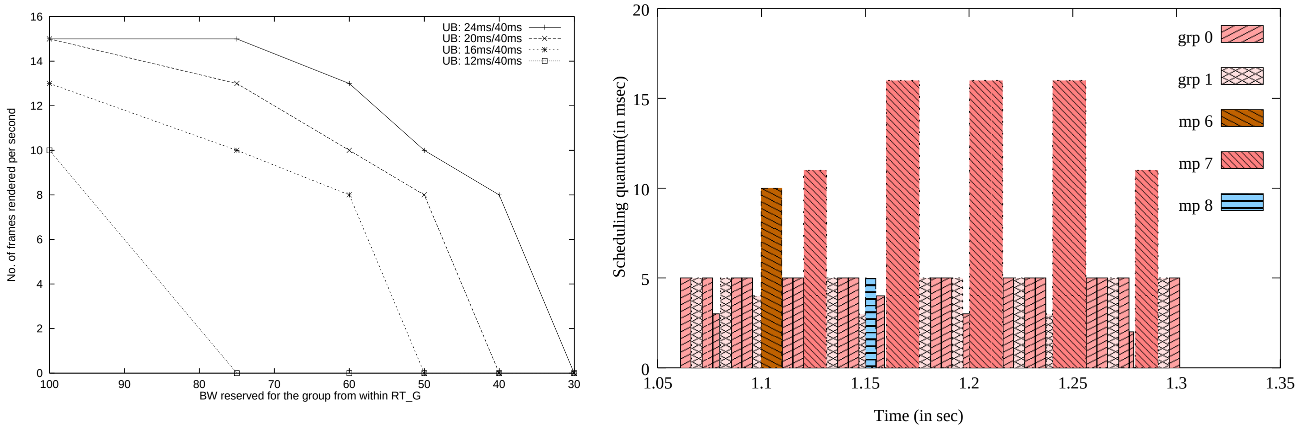
The second experiment (Figure 10(b)) demonstrates how the scheduling of real-time programs takes place in the presence of non-real time programs. There are two non-real time groups: $Group0$ and $Group1$ have shares 40 and 20, respectively. There are three programs with real-time reservations:

MP6: <1.100s,1.150s,10ms>
MP7: <1.120s,1.500s,70ms>
MP8: <1.150s,1.180s,5ms>

The host specifies an upper bound of 40% ($16ms$) on the RT_G group for each quantum time chunk of $40ms$. The plot shows that the real-time programs are allocated according to their reservations. At the same time non-real time programs are allocated according to their shares. Also, since there is an upper bound on RT_G group, real-time programs cannot starve the non-real time programs ($Group0$ or $Group1$), even though real-time programs are scheduled for more than $5ms$ (the default time quantum) at a given time.

5 Scheduling Scheme in Java Runtime System

We modified the Java virtual machine (Solaris JDK version 1.1) to incorporate our scheduling scheme. The modified JVM contains an API for specifying groups, subgroups, and various resource usage constraints. In addition, it includes a thread API for managing and scheduling threads. We have integrated the notion of groups in our scheduling scheme with that of *ThreadGroups* in JVM. The current implementation does not include support for scheduling real-time programs since the JVM currently does not have support for real time programs.



(a) No. of frames rendered/sec as a function of bandwidth reservation

(b) Scheduling for a combination of real time and non-real time programs

Figure 10: Experiments for real time constraints

In this section we describe the details of the integration of the scheduling scheme within the JVM. We first describe the original JVM scheduling scheme, and then the modifications made in the JVM scheduling mechanism to incorporate our scheduling scheme.

5.1 The original JVM scheduling scheme

The default scheduling scheme in the JVM is based on simple round robin based priority queues. The native C code for thread scheduling is provided in $solaris/java/green\_threads$ subdirectory of the JVM source tree. The main data structure is a $runnable\_queue$ that consists of runnable threads sorted in terms of their priority. The scheduler picks up the first thread from the $runnable\_queue$ for execution. Apart from being in a $runnable$ state, a thread can be in $suspended$, $monitor\_wait$, $condvar\_wait$, and $monitor\_suspended$ states. $Monitor\_wait$ state implies that the thread is waiting to enter a monitor, and is inserted in $monitor\_waitq$, a queue of threads waiting on the monitor. $Monitor\_suspended$ state means that the thread was suspended while in the monitor, and such a thread is placed in $suspend\_waitq$ queue. A thread in $condvar\_wait$ state is a thread that is executing $wait()$ inside a monitor and is placed in $condvar\_waitq$ queue.

There are two kinds of threads: system threads and user threads. System threads are always created when the virtual machine executes and they assist in performing various system tasks such as timing, garbage collection and clock management. Examples of system threads are the clock manager, the time slicer, the garbage collector and the idle threads. User threads are created to execute the user programs. There is at least one user thread created for executing a Java program.

The JVM assigns priorities to various threads and uses these priorities for scheduling. Priority values for user threads range from a $minimum\_priority$ (normally 0) to a $maximum\_priority$ (normally 10). The clock manager thread has the highest priority of $maximum\_priority + 2$. The clock manager is essentially a thread running at maximum priority that manages a database of timeouts. There are two primary functions for the clock manager: to suspend a thread for a period of time (thread_sleep), and to notify a condition variable after a timeout has expired.

The time slicer thread is only run if the $-ts$ flag is used when the runtime is started. It runs at priority $maximum\_priority + 1$, which is greater than any user thread but less than the clock manager thread. The time slicer helps in round robin execution of user threads that have the same priority.

The idle thread runs at a priority of $minimum\_priority - 1$. It runs when no other thread is being scheduled. The garbage collector (GC) runs as follows: the idle thread is running at a priority lower than anything else, including the GC thread. If the GC thread wakes up, it remembers the the number of times the idle thread has run and then goes to sleep for a second. When the GC thread wakes up again, if the idle count is higher than it had been, it means that the idle thread ran while the GC thread slept. This is a reasonably good indication that nothing was going on during the last one second, and that nothing will be going on in the near future. The garbage collector then starts to run. If the idle thread has not been run while the GC thread slept, the GC thread goes back to sleep again.

5.2 Integration of Scheduling Scheme

The JVM is provided with an API to interface with our scheduling scheme. The API consists of following kinds of methods:

- `AddThread():` Add a new thread into the hierarchy.
- `RemoveThread():` Remove a thread from the hierarchy.
- `SuspendThread():` Suspend a thread.
- `MakeThreadRunnable():` Make the thread state runnable.
- `AddThreadGroup():` Add a new thread group to the hierarchy.
- `RemoveThreadGroup():` Remove a thread group to the hierarchy.
- `AddBWRule():` Add a share based rule.
- `AddPriorityRule():` Add a priority based rule.
- `SetUB():` Set upperbound constraint.
- `SetLC():` Set lifetime constraint.
- `AddtoUB():` Add thread group to an upperbound class.
- `AddtoLC():` Add thread group to a lifetime constraint class.
- `ClearUB():` Clear an upperbound constraint.
- `ClearLC():` Clear a lifetime constraint.

We have modified the original JVM code in two places to implement our scheduling scheme:

The code modifications in $solaris/java/green\_threads$ subdirectory change the JVM scheduler for user threads: If a thread is not a system thread, it is not added to the $runnable\_queue$. It is added to our scheduling hierarchy using the API. For instance, in `sysThreadCreate()` (thread creation) and `sysThreadExit()` (thread exit), `AddThread()` and `RemoveThread()` methods are invoked. Whenever user thread state changes from *runnable* to *suspended* and vice-versa, the API functions (`SuspendThread()` or `MakeThreadRunnable()`) are called.

Modifications of Java classes in $share/java/java/lang$ integrate the notion of groups in our scheduling scheme with that of $ThreadGroups$ in JVM. We have added some additional methods within the ThreadGroup object ($ThreadGroup.java$) to specify resource usage constraints in the form of priority and shares for subgroups and member threads. These methods ultimately invoke functions in the family `AddBWRule()` or `AddPriorityRule()` in our scheduling scheme. Some already existing methods have also been modified: `add(ThreadGroup)` and `add(Thread)` methods in JVM are modified to invoke `AddThread()` and `AddGroup()` methods in our scheduling scheme. Similarly, `remove(ThreadGroup)` and `remove(Thread)` methods in JVM have been changed to invoke `RemoveThread()` and `RemoveThreadGroup()`.

New Java classes, UB and LC, implement upperbound and lifetime constraints. UB and LC implement methods to associate groups with upperbound and lifetime constraints (`AddtoUB()`), methods to set various constraints (`setUB()`), and methods to clear various constraints (`clearUB()`). Figure 11 presents the new classes and the additional methods in the ThreadGroup class.

5.3 Experimental results

We conducted several experiments on the JVM. The goals of these experiments were to examine the effectiveness of the scheduling scheme within the JVM in (i) satisfying non real-time constraints; (ii) enforcing upper bound and lifetime constraints; and (iii) satisfying constraints that change dynamically. We describe the experiments and the results below. In all the experiments, the time quantum is $10ms$.

Figure 12(a) shows the hierarchy on which the experiments have been conducted. Groups G_0, G_1 and G_2 have shares constraints. Groups PG_3 and PG_4 are subgroups of G_0 and are allocated on the basis of priority. Groups G_0, G_1 and G_2 may also be constrained by upper bounds.

5.3.1 General scheduler behavior

The first experiment demonstrates how the scheme schedules mobile programs constrained by shares and priorities. There are no upper bound constraints in this experiment. The following are the constraint values:

$S_0 = 30$ $S_1 = 60$ $S_2 = 10$
$P_3 = 1$ $P_4 = 2$
$S_5 = 30$ $S_6 = 70$ $S_7 = 40$ $S_8 = 60$
$S_9 = 10$ $S_{10} = 90$ $S_{11} = 100$

Figure 12(b) shows the relative CPU allocations of groups G_0, G_1 and G_2. Initially the relative allocations are according to the share values. At time ($t = 70s$), G_1 finishes execution and relative allocations to G_0 and G_2 change to S_0 and S_2. Later G_0 finishes execution and G_2 is the only group scheduled.

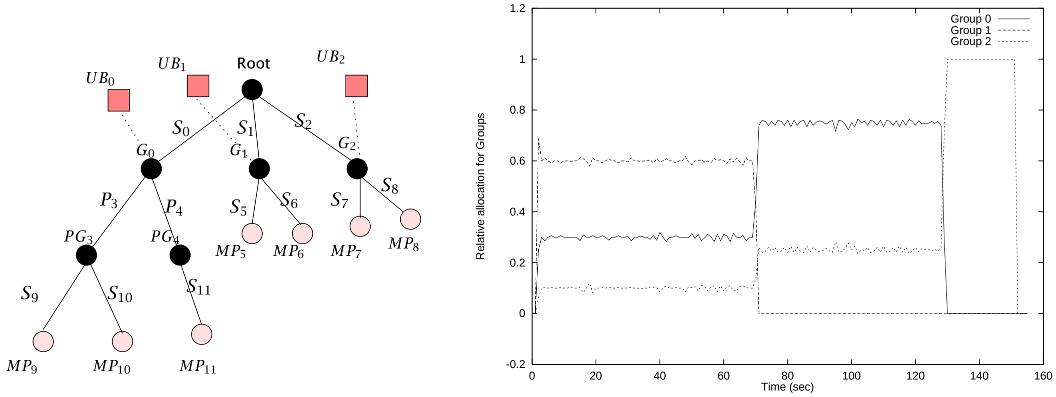
5.3.2 Relative allocation with upper bounds

This experiment demonstrates how allocations are made for groups in the presence of upper bound constraints. The following are the constraint values:

$UB_0 = 7s$, $UB_1 = 25s$, $UB_2 = 100s$
$S_0 = 30$ $S_1 = 60$ $S_2 = 10$
$P_3 = 1$ $P_4 = 2$
$S_5 = 30$ $S_6 = 70$ $S_7 = 40$ $S_8 = 60$
$S_9 = 10$ $S_{10} = 90$ $S_{11} = 100$

```
public class ThreadGroup {
    //create the root group for the hierarchy
    private final native void CreateRootGroup();

    private final native void CreateGroup(ThreadGroup t);
    private final native void RemoveGroup(ThreadGroup t);
    private final native void CreateThread(Thread t);
    private final native void RemoveThread(Thread t);
    public void Add_Priority_rule(Thread t, int value);
    public void Add_Priority_rule(ThreadGroup t, int value);
    public void Add_BW_rule(Thread t, int value);
    public void Add_BW_rule(ThreadGroup t, int value);

    // Init_rule methods invoked before changing the bw/priority rules
    public void Init_BW_rule(Thread t);
    public void Init_BW_rule(ThreadGroup t);
    public void Init_Priority_rule(Thread t);
    public void Init_Priority_rule(ThreadGroup t);
}

//Upper bound class
public final class UB {
    public void Add(ThreadGroup g);
    public void Add(Thread t);
    public void Rem_from_UB(ThreadGroup g);
    public void Rem_from_UB(Thread t);
    public void Add_rule(int upperbound);
    public int IsViolated();
    public void ClearViolated();
}

//Lifetime control class
public final class LC {
    public void Add(ThreadGroup g);
    public void Add(Thread t);
    public void Rem_from_LC(ThreadGroup g);
    public void Rem_from_LC(Thread t);
    public void Add_rule(int globalbound, int migration);
    public int IsViolated();
    public void ClearViolated();
}
```

Figure 11: The JVM Class Interface

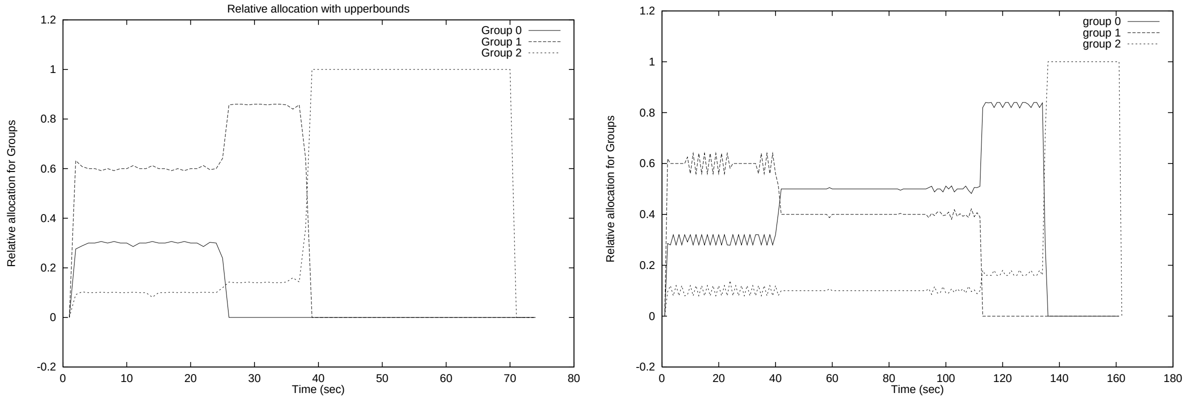


(a) The scheduling graph

(b) Relative allocation of CPU for groups

Figure 12: Scheduling scheme in JVM

Figure 13(a) shows the relative CPU allocations for the groups. Relative allocation for G_1 and G_2 increases once the upper bound for G_0 is reached ($t = 25s$). At time ($t = 40s$), the upper bound for G_1 is achieved and G_2 is the only group scheduled.



(a) Relative CPU allocation with upper bounds

(b) Relative CPU allocation: changing rules on the fly

Figure 13: General scheduling behavior of the scheme

5.4 Dynamic changes in allocation policies

This experiment demonstrates the relative group allocation as constraints are dynamically varied. After starting all the threads, the main thread goes to sleep. When the main thread wakes up, it changes the share specification for G_0 and G_1. There are no upper bound constraints. The following are the constraint values:

$S_0 = 30$ $S_1 = 60$ $S_2 = 10$ (initially)
$S_0 = 50$ $S_1 = 40$ $S_2 = 10$ (when main thread wakes up)
$P_3 = 1$ $P_4 = 2$
$S_5 = 30$ $S_6 = 70$ $S_7 = 40$ $S_8 = 60$

$S_9 = 10 \qquad S_{10} = 90 \qquad S_{11} = 100$

Figure 13(b) shows the relative CPU allocations of groups. At time $t = 40s$, the main thread changes the share values, and the relative allocation for groups changes. The result demonstrates that there can be a controller thread within the system that controls all other threads and change the allocation policies as required.

5.4.1 Dynamic Constraints

In this set of experiments, we show that the JVM dynamically adapts to changes in resource constraints. The first experiment, depicted in Figure 14(a), demonstrates the allocation of CPU to the programs when resource usage constraints depend on the number of mobile programs: There are no upper bound constraints. The following are the constraint values for the groups:

Constraint	Value	
S_0	30	
S_1	60	
S_2	10	if $n_{mp} < 3$
S_2	110	otherwise

The graph results can be explained as follows: at time ($t = 40s$), a new thread is added to G_2. As a result, the relative allocations for the different groups change. At $t = 120s$, one of the threads of G_2 finishes execution, and the relative allocation goes back to the initial values. Later, when all threads of G_1 have finished execution, the relative allocations for the other groups increase.

The second experiment (Figure 14(b)) demonstrates the CPU allocation when the constraints are time dependent. There are no upper bound constraints. The share specifications for the groups are as follows:

Constraint	Value	
S_0	30	
S_1	50	
S_2	20	if $time \in [0s, 47s]$
S_2	110	if $time \in [47s, \infty]$

Figure 14(b) shows the relative CPU allocations of the groups.

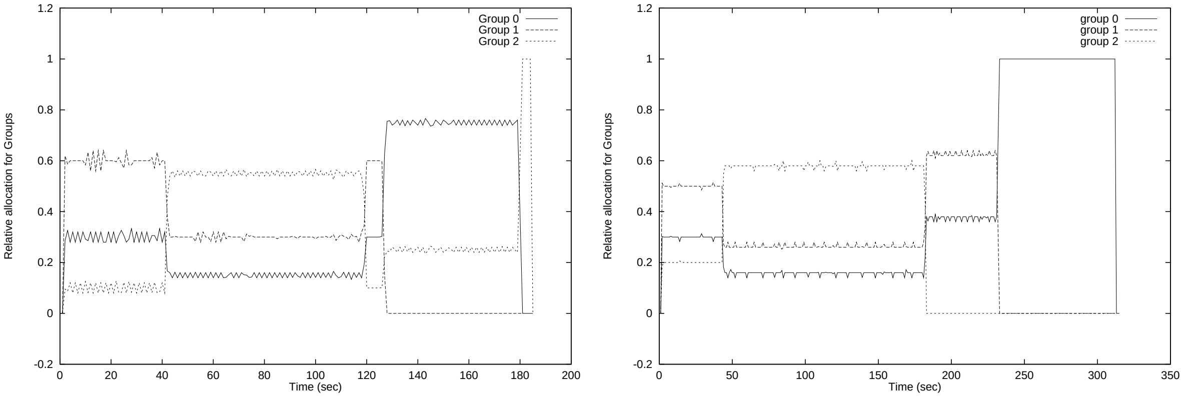
5.4.2 Clearing violated constraints

This experiment (Figure 15) demonstrates that any violated upper bound constraints can be cleared by a controller thread. The following are the constraint values:

$UB_0 = 4.5s,\ UB_1 = 12.5s,\ UB_2 = 500s$

$S_0 = 30 \qquad S_1 = 60 \qquad S_2 = 10$

After starting all the threads, the main thread goes to sleep. When the main thread wakes up, it selectively clears violated upper bounds for UB_0 and UB_1. Figure 15 shows the relative CPU allocations of groups. The results can be explained as follows:



(a) Relative CPU allocation with dynamic rules based on number of threads

(b) Relative CPU allocation with time based dynamic rules

Figure 14: Dynamic resource usage constraints

- At $t = 30$, UB_0 gets violated, and execution of G_0 stops.
- At $t = 40s$, UB_1 is violated and G_1 is suspended.
- At $t = 53s$, UB_0 is cleared and G_0 resumes execution.
- At $t = 67s$, UB_0 is again violated and G_0 is suspended.
- At $t = 71s$, UB_1 is cleared and G_1 resumes execution.
- At $t = 88s$, UB_1 is again violated and G_1 is suspended.
- At $t = 110s$, G_2 finishes execution.

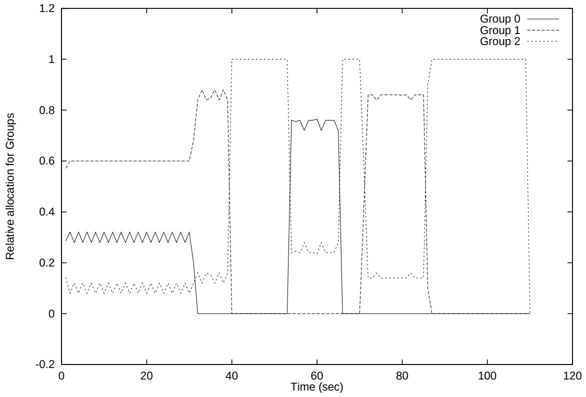


Figure 15: Relative CPU allocation with upper bounds being cleared and reset

A host can use such a facility to implement *overuse callbacks*, actions to be executed whenever any of the security constraints are violated. The callback methods can, for instance, reduce the priority of the offending thread, or relax the limits set on the thread so that the thread can complete its execution.

6 Related Work

The subject of resource scheduling in general and CPU scheduling in particular has been widely studied. [5, 2] present a taxonomy of the different CPU scheduling algorithms. The scheduling techniques range from simple algorithms such as first come first served and priority queues [27] to more general, flexible and modular schemes [13, 9, 11, 28]. We compare our scheduling scheme and the algorithms with only those approaches that we believe are closest to our approach.

6.1 Scheduling schemes

Several scheduling schemes [13, 9, 11] have looked at providing modular control by statically separating scheduling policies for different classes of applications. The policies are combined using priorities or proportional sharing. CPU inheritance scheduling[9] allows threads in a hierarchy to define their own scheduling policies within the subtree under the thread. This results in a very flexible and decentralized scheduling mechanism where different schemes are applied on parts of the subtree to contribute to the overall scheduling scheme. The scheduling scheme described in this thesis is a centralized one. This enables a host to monitor and control external mobile programs more effectively. Further, the above schemes rely on a single scheduler servicing both real-time and conventional applications. This results in a static scheduling hierarchy that is primarily based on different classes of applications. Our scheme, on the other hand, is adaptive in that it allows the host to define classes of applications based not only on constraints but also on other parameters such as network domains.

Many commercial systems [27] provide fixed priority scheduling for real-time applications in order to combine scheduling of real-time applications with conventional tasks. The problem with these schemes is that they end up starving the non real-time applications while not providing any guarantees regarding the real-time tasks. Other systems provide timely execution of real-time tasks on the basis of some hierarchical partitioning [9, 11]. However, these schemes are not based on deadlines and do not provide guarantees to the real-time programs. Some schemes [20] have implemented deadline based schemes. However, they do not provide any guarantees or resource reservations.

In our scheme, the non real-time scheduling algorithm is based on scheduling algorithm used in SMART [20]. We have extended the algorithm to enforce share and priority based constraints over a hierarchical scheduling graph. Real-time scheduling in our scheme is based on Rialto[14] which provides for deadline based resource reservations and guarantees. Our schemes extends the Rialto scheme with upper bounds on the CPU time available to real-time applications.

The notion of upper bounds constraint has been studied in several forms. For instance, many version of the Unix operating system provide system calls (e.g., `setrlimit`) for specifying limits on resource consumption. VINO [23], an extensible operating system, provides similar control over allocation of resources. The scheme in our approach supports an adaptive and fine-grained control more suited for the mobile programming environment.

6.2 Mobile programming systems

CPU resource control schemes have been proposed for mobile programs [26, 3, 4]. These systems propose solutions for effective utilizations of resources by mobile programs. In these systems, client and server resource usage constraints are not defined directly in terms of lower bound, upper bounds, shares etc. Instead allocation of resources is based on an economic model. In these models, hosts set prices on consumption of resources, whereas mobile programs use money (digital or otherwise) to buy the usage of resources. A host, thus, allocates resource to a mobile program on the program's ability to buy these resources.

In such schemes the problem being solved is slightly different. The goal of these schemes is to have mobile programs efficiently utilize the host resources to prevent wastage. In our case, the goal is to protect host resources from misbehaving mobile programs. While such schemes can be used to enforce lifetime constraints, a mobile program can cause denial of service attacks if it owns a lot of cash. This brings up the issue of cash protection and cash management. Also, since the cost set for resources is uniform for all mobile programs, it is difficult to define policies in which a host can control allocation of resources on the basis of its preferences or trust relationship. Our approach differs in both the mechanisms used for specifying and enforcing policies. We believe that the economic model can be easily modeled in terms of upper bounds, lower bounds, shares and priority constraints.

JRes [8] is a scheme for controlling allocation of different kinds of resources (CPU, memory etc) within the Java runtime system. JRes uses binary editing to enforce simple upper bound constraints on Java programs. Our scheme differs from JRes model in that our scheme not only enforces upper bound constraints, but also performs CPU scheduling based on other constraints. We have not used binary editing but implemented the scheme by changing the scheduler within the JVM.

7 Conclusion and future work

In this paper we have highlighted the need for a CPU scheduling scheme that addresses the security and quality of service requirements of a host. We present a CPU scheduling scheme that addresses these needs. The scheme presents an environment for specifying CPU resource usage constraints. Mobile programs specify shares, priority and deadline constraints. Hosts specify shares, priority, upper bound and lifetime constraints. The scheme constructs a scheduling hierarchy to apply a set of algorithms that enforce the various constraints. The non-real time algorithm enforces share and priority based constraints. The real time algorithm enforces deadline constraints. The upper bounds algorithm enforces the security constraints specified by the host. Any conflict between the client and server constraints is resolved by an algorithm composition policy that always favors the server constraints.

Experiments show that our scheduling scheme provides modular resource control. The scheme is flexible so that the host system can selectively trust some mobile programs. The scheme enforces protection in the form of lifetime constraints and upper bounds, and is dynamic in the sense that the trust level and allocation can change any time.

We plan to extend our current work in several directions. The first is to extend the scheme so that it can be applied to user-defined resources. We also intend to make the scheme extensible so that arbitrary resource usage constraints, their scheduling algorithms, and algorithm composition policies can be specified and composed.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[2] K. M. Baumgartner and B. W. Wah. Computer scheduling algorithms: Past, present, and future. *Information Sciences*, 57-58:319-345, 1991.

[3] Jonathan Bredin, David Kotz, and Daniela Rus. Market-based resource control for mobile agents. In *AGENTS '98, Proceedings of the second international conference on Autonomous agents*, pages 197–204, May 1998. Available at http://www.acm.org/pubs/articles/proceedings/ai/280765/p197-bredin.pdf.

[4] Jonathan Bredin, David Kotz, and Daniela Rus. Economic markets as a means of open mobile-agent systems. In *Proceedings of the Workshop "Mobile Agents in the Context of Competition and Cooperation (MAC3) at Autonomous Agents '99"*, May 1999. Available at http://agent.cs.dartmouth.edu/papers/index.html.

[5] T. Casevant and J. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions of Software Engineering*, 14:141-154, 1988.

[6] D. Chess, C. Harrison, and A. Kreshenbaum. Mobile agents: Are they a good idea? Technical report, IBM T. J. Watson Research Center, 1995.

[7] Davis Chess, Benjamin Grosof, Colin Harrison, Davis Levine, Colin Parris, and Gene Tsudik. Itinerant agents for mobile computing. *IEEE Personal Communications*, pages 34–49, October 1995.

[8] G Czajkowski and T von Eicken. JRes: A resource accounting interface for java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, 1998.

[9] Bryan Ford and Sai Susarala. CPU inheritance scheduling. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[10] Berny Goodheart and James Cox. *The Magic Garden Explained, The Internals of Unix System V Release 4: An Open Systems Design.* Prentice Hall, 1994.

[11] Pawan Goyal, Xinguong Guo, and Herrich M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[12] B. Hashii, M. Lal, S. Samorodin, and R. Pandey. Securing systems against external programs. *IEEE Internet Computing*, 2(6):35–45, November - December 1998.

[13] G. J. Henry. Fair share scheduler. *AT&T Bell Laboratories Tech. Journal*, October 1984.

[14] M. B. Jones, D. Rosu, and M-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. *16th ACM Symposium on Operating Systems Principles*, October 1997. St. Malo, France.

[15] S. Keshav. *Congestion Control in Computer Networks.* Phd thesis, U. C. Berkeley, 1991.

[16] Joseph Kiniry and Daniel Zimmerman. A hands-on look at java mobile agents. *IEEE Internet Computing*, pages 21–30, July 1997.

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

[18] Sun Microsystems. Servlet specification. http://java.sun.com/products/servlet/2.1/index.html.

[19] Natraj Nagaratnam and Steven B. Byrne. Resource access control for an internet user agent. In *Proceedings of the USENIX 3rd Conference on Object-Oriented Technologies*, 1997.

[20] Jason Nieh and Monica Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. *16th ACM Symposium on Operating Systems Principles*, October 1997.

[21] R. Pandey, J. Fritz Barnes, and R. Olsson. Supporting quality of service in http servers. In *Proceedings of the Seventeenth Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 247–256, Puerto Vallarta, Mexico, June 1998. ACM.

[22] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, August 1992.

[23] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, 1996.

[24] J.W. Stamos and D.K. Gifford. Remote evaluation. *ACM Transactions on Programming Langua es and Systems*, 12(4):537–565, October 1990.

[25] Rommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3), September 1997.

[26] Christian F. Tschudin. Open resource allocation for mobile code. Computer Science Department, University of Zurich, Switzerland.

[27] Uresh Vahalia. *UNIX Internals, The New Frontiers*. Prentice Hall.

[28] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional share resource management. In *Proceedings of the USENIX 1st Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, November 1994.

[29] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional share resource management. Technical report, MIT Laboratory for Computer Science, June 1995.