Virtual Computers—A New Paradigm for Distributed Operating Systems *

Banu Özden† Aaron J. Goldberg Avi Silberschatz ‡

600 Mountain Ave,
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

The virtual computers (VC) paradigm enables the incorporation of predictability and choice into the design of an operating system. Predictability refers to the ability of the system to provide each user with a computing environment whose performance is independent of the behavior of other users. Choice refers to the ability of a user to select a computer system that meets that user’s specifications, needs or budget. In this paper, we introduce this new paradigm and show how the VC paradigm can be incorporated into the processor scheduling, and how the on-line schedulers can be effectively implemented.

1 Introduction

Personal computers provide two attractive features that are neglected in today’s distributed operating systems—predictability and choice. Predictability is the ability of the system to provide each user with a computing environment whose performance is independent of the behavior of other users. Choice refers to the ability of a user to select a computer system that meets that user’s specifications, needs or budget. We refer to the pair of features as ownership, since these two features together represent the rights of the computer owner.

In order to incorporate the concept of ownership into operating systems, we propose the virtual computers (VC) paradigm. In a system based on the VC paradigm, each user is promised a given quality of service, and the system seeks to provide each user with at least the level of service promised. One can view the service promised to a user as a virtual computer owned by that user. Ultimately, a user should receive the promised service independent of the number of users accessing the system, the location where the actual execution takes place, and where the user accesses the system. The users may be promised a different level of service of their choice corresponding to a different “type” of virtual computer, and each user may own more than one virtual computer.

The VC paradigm enables coexistence of virtual computers with different scheduling algorithms, which is important for the simultaneous handling of applications that require different scheduling policies (e.g., multimedia applications that require real-time scheduling can coexist with conventional applications). The VC paradigm can be realized on distributed systems based on various system models including the processor-pool model, the workstation model, and the time-shared computers model. By providing the users of a distributed system with the desirable properties of personal computers—predictability and choice—the VC paradigm encourages the users to use a distributed system as well as to share their computers with others. Furthermore, applying the VC paradigm to a single-user system provides the user with more than one virtual computer with different specifications, and applying it to time-sharing systems enables sharing resources predictably among users.

A system based on the VC paradigm will be referred to as a VC system. In order to implement a VC system, some of the issues in the design of a distributed operating system must be reconsidered including re-

---

* The research of Banu Özden and Avi Silberschatz was supported in part by the Texas Advanced Research Program under Grant No. ATP-209, the National Science Foundation under Grant Nos. IRI-9221301 and IRI-9312003, and grants from the IBM and Hewlett-Packard corporations.

† A PhD candidate in the Department of Computer and Electrical Engineering at the University of Texas at Austin where the work was done.

‡ On leave from the Department of Computer Sciences, at the University of Texas at Austin, where the work was done.
source management, naming, protection, and service provisions. This paper concentrates on the basic processor scheduling aspects of resource management.

2 The VC Paradigm

In a VC system, each user owns one or more imaginary computers—virtual computers. The virtual computer is only a description of a computer and may not correspond to any real computer in the system. The description includes the CPU type (CPU speed, etc.) and a local scheduling algorithm (e.g., first come first served, round robin, earliest deadline first). The real computers used in a VC system may be workstations, multiprocessors or mainframes depending on the system model. For our purpose, the system simply consists of a set $P = \{P_1, P_2, \ldots, P_m\}$ of processors.

Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of virtual computers. The virtual computers may differ in terms of their specification. Assignment of different virtual computers to users will usually depend on factors such as user’s needs, seniority, and budget. Suppose that task $T_i$ submitted to a virtual computer is executed on a computer which is equivalent to the virtual computer. Such an execution is referred to as the execution on the virtual computer. The abstract execution of a task $T_i$ on a virtual computer can be characterized by two attributes: arrival time $a_i$ and completion time $\hat{c}_i$ on the virtual computer. The arrival time is the time when $T_i$ is submitted to the system. The completion time is the time when $T_i$ would have been completed if it were executed on a real computer equivalent to the virtual computer. Clearly, $\hat{c}_i$ depends on the local scheduling algorithm of the virtual computer, the length of the task $T_i$, and the load of the virtual computer. We denote the actual completion time of $T_i$ in the system by $c_i$. We model periodic tasks as a sequence of independent tasks with separate arrival times and completion times.

A VC system seeks to guarantee predictable performance to each user. We define the predictable performance in terms of the performance which the user could have obtained on the corresponding virtual computer. However, it is not sufficient to guarantee predictable performance in terms of typical performance measures, such as the average response time and the average throughput. This is because individual tasks submitted to a virtual computer can be delayed far beyond their completion times on the virtual computer, even if the average performance on the system is better than the average performance on the virtual computer.

Since cumulative performance measures do not adequately address predictability, the VC paradigm is based on stronger guarantees so that the performance of each individual task will be predictable. Such guarantees can be expressed in terms of deadlines. Hence, each task $T_i$ is assigned a deadline $d_i$, which is the time by which $T_i$ must complete. The assignment of deadlines depends on the the level of service guarantees. Various types of VC systems can be defined, each with different levels of guarantees. The types of VC systems and, hence, the assignment of the deadlines is discussed in Section 3.

A VC system is a set of tasks $T = \{T_1, T_2, \ldots, T_k\}$ that must be actually scheduled on $P$ under the constraints $a_i$ and $d_i$ for each task. In practice, there may be further constraints on tasks. For example, a resource constraint may be needed to specify on which processor(s) a task can be executed (e.g., if the task is a high bandwidth interactive application such as a 3D CAD tool), then it may be constrained to execute on the processor that is connected to the graphical display). In this paper, we consider only the time constraints.

3 VC System Types

A VC system seeks to provide each user with a level of service which is proportional to the one that could have been obtained on the user’s virtual computer. Each level of service defines a different type of a VC system. The discussions in this paper are confined to the following system types: The strict VC system, the average VC system, and the bounded VC system. The set of schedules for a bounded VC system contains the set of schedules for the average VC system, and the set of schedules for the average VC system contains the set schedules for the strict VC system.

3.1 Strict VC System

A strict VC system provides each user with a service which is at least as good as the one on the virtual computer. For each task, the constraint that the completion time of the task will be less than or equal to

\[ \text{average throughput at time } t \text{ as the average over } \left[ \frac{t}{T} \right] \text{ time intervals.} \]
the completion time of the task on the virtual computer is imposed. This constraint can be expressed by selecting the deadline for each task $T_i$ as:

$$d_i = \tilde{c}_i$$

A strict VC system schedules all the tasks in $T$ so that they meet their time constraints. We refer to schedules generated by a strict VC system as strictly feasible. The average response time for each user will be less than or equal to the average response time that could have been obtained on his virtual computer at any given time (see Figure 1). Similarly, the average throughput for the user will be greater than or equal to the average throughput on the virtual computer at any given time.

Obviously, the deadline of a task depends on the length of the task, CPU speed, the local scheduling algorithm, and the load of the virtual computer. Since the load changes dynamically, for some local scheduling algorithms, the deadline of a task is not fixed at the arrival time, even if the length of the task is known.

### 3.2 Average VC System

An average VC system provides each user with a level of service which is, on average, at least as good as that on the virtual computer while maintaining the delay of each individual task bounded. Specifically, an average VC system assumes a time interval $\tilde{t}$ and requires that:

1. The average throughput of tasks submitted to a virtual computer will be greater than or equal to the average throughput on the virtual computer at the end of every time interval of length $\tilde{t}$.

2. Each individual task can be delayed at most $\tilde{t}$.

This constraint can be expressed by

$$d_i = \lceil \frac{\tilde{c}_i}{\tilde{t}} \rceil \cdot \tilde{t}$$

for each task $T_i$.

This constraint disallows interchanging the completion times of any two tasks completed on the virtual computer in different time intervals. For example, suppose that task $T_i$ is completed on the virtual computer in time interval $[k \cdot \tilde{t}, (k + l) \cdot \tilde{t}]$ and task $T_j$ completed on the virtual computer in time interval $[(k + l) \cdot \tilde{t}, (k + l + 1) \cdot \tilde{t}]$. The average throughput will be equal to the average throughput on the virtual computer, if $T_i$ completes in time interval $[(k + l) \cdot \tilde{t}, (k + l + 1) \cdot \tilde{t}]$ and $T_j$ in $[k \cdot \tilde{t}, (k + 1) \cdot \tilde{t}]$. An average VC system disallows such cases. An average VC system schedules all the tasks in $T$ so that they meet their time constraints. We refer to schedules generated by an average VC system as average feasible. If a schedule is strictly feasible, it is also average feasible. However, the converse is not true.

The average response time for tasks submitted to a virtual computer will be either less than or equal to the sum of $\tilde{t}$ and the average response time on the virtual computer at any given time, whereas the average throughput for the tasks submitted to a virtual computer will be always greater than or equal to the one on the virtual computer at any multiples of $\tilde{t}$ seconds (see Figure 2).

### 3.3 Bounded VC System

In many circumstances, a user will tolerate delay if his tasks are delayed by a known bound. This delay can be imposed by the system to express the upper bound of the delay that a task can experience. In a bounded VC system, the deadlines are assigned as

$$d_i = \tilde{c}_i + b_i$$

where $b_i$ is either a constant or a function (of either time, or response time on the virtual computer or the
Figure 3: The containment relation between strictly, average and bounded feasible schedules.

The load on the system (i.e., \( b_i \)) is assumed to be bounded by a constant from above. A bounded VC system schedules all the tasks in \( T \) so that they meet their time constraints. We refer to schedules generated by a bounded VC system as bounded feasible. If a schedule is average feasible, it is also bounded feasible. However, the converse is not true. Figure 3 illustrates the containment relation between strictly, average and bounded feasible schedules of a given task set.

Suppose that the maximum \( b_i \) for the tasks submitted to a virtual computer is \( b_{\text{max}} \). At any given time, the average response time for these tasks will be either on or below the sum of \( b_{\text{max}} \) and the average response time that could have obtained on the virtual computer. Suppose that over \( k \) time intervals the average throughput on the virtual computer is \( \frac{a_k}{b_{\text{max}}} \), then the average throughput for the tasks submitted to virtual computer will be either less than or equal to \( \frac{a_k}{b_{\text{max}}} \).

This type of the VC system is called bounded, since the guaranteed performance to each user has a lower bound below the performance on the virtual computer to within a constant factor (see Figure 4).

4 Scheduling in VC Systems

The primary goal of scheduling in a VC system is to meet the VC constraints. In this section, we present on-line feasible scheduling algorithms for uniprocessor and multiprocessor VC systems. In this paper, we make a number of simplifying assumptions. We assume that the cost of starting, restarting and killing a task on any processor is negligible. We assume that tasks are mutually independent, preemptable, and the cost of preemption is negligible. We assume that tasks are computation intensive and only consider processing resources (CPUs). That is, we assume that the system always has sufficient amounts of other resources such as memory, secondary storage, and network bandwidth. We assume that the processors are identical and that the local scheduling algorithm of all virtual computers is FCFS. We are currently extending our results to allow the virtual computers to employ different local scheduling algorithms (e.g., fixed priority, deadline-based algorithms, and round-robin).

4.1 Feasible Scheduler for Uniprocessor VC Systems

We first examine on-line feasible schedulers for uniprocessor VC systems where the real processor is at least \( n \) times faster than the CPUs of the virtual computers. If the clock rate of the processor is \( c \), then the clock rate of the virtual computer will be \( \frac{c}{n} \). For example, consider a uniprocessor VC system where the clock rate of the processor is 100 MHz and the number of virtual computers is four. The clock rate of virtual computers will be 25 MHz. We know that there is always a feasible schedule that can be obtained on-line without knowing any characteristics of the tasks with the following algorithm.

The processor time is divided into time slices of length \( \frac{1}{c} \), which in turn is divided into \( n \) sub-slices of length \( \frac{1}{nc} \). One sub-slice of each time slice is reserved for a virtual computer. A sub-slice of a time slice is allocated to any virtual computer which is busy during this or a previous sub-slice of the same time slice, and which has not yet assigned to a sub-slice in this time slice. We refer to this algorithm as reservation with smallest time slice. This algorithm is strictly feasible.

\(^2\)This means that the clock of the real processor ticks at least \( n \) times in the time when the imaginary clock of virtual computer ticks once.
if we assume that at most one task will arrive in any given subslice. However, this algorithm is not practical due to the frequency of decision points and the cost of context switching. Therefore, we must search for a more practical algorithm that has less frequent decision points, and therefore yields less context switches. One such algorithm is the earliest deadline first (EDF) algorithm, which at each decision point executes the task with the earliest deadline, and breaks ties arbitrarily.

**Theorem 1:** Consider a uniprocessor VC system. If task lengths are known, then the EDF algorithm is strictly feasible.

**Proof:** Since the local scheduling algorithm is FCFS and length of tasks are known, the completion times of the tasks on the virtual computers, hence, the deadlines for a strict VC system are fixed and known at the task arrival time. It was shown that the EDF algorithm is optimal in the sense if there exists a feasible schedule, then the EDF algorithm generates a feasible schedule \(2\). Since on a uniprocessor VC system there exists always a schedule, the EDF algorithm is feasible. \(\square\)

**Theorem 2:** Consider a uniprocessor VC system. If the length of tasks are not known, then the only strictly feasible scheduling algorithms are those that have a decision point at every clock tick.

**Proof:** The proof relies on the fact that for any scheduling algorithm that sequences tasks in a given order without knowledge of task lengths, one can find a task set for which the algorithm fails. Consider a uniprocessor VC system with two virtual computers and assume the task set depicted in Figure 5, where \(l_i\) denotes the length of task \(T_i\) on the real processor. Note that the type of the local scheduling algorithm of the virtual computers does not affect the example, since only one task is submitted to each virtual computer. There are three possible ways of sequencing these tasks on the uniprocessor. The sequence depicted in Figure 6 is not a feasible schedule. Since the scheduler does not know the length of tasks, all tasks appear to be the same to the scheduler. Thus, if a scheduler algorithm executes tasks of virtual computers in a given order, one can always find another set of tasks for which the algorithm fails. \(\square\)

Figure 5: Tasks characteristics in Proof of Theorem 2.

Theorem 2 demonstrates that there can be no efficient strictly feasible scheduling algorithm if tasks lengths are not known. However, there exists an average feasible scheduling algorithm for a uniprocessor VC system. One such algorithm is the reservation with a fixed time slice (RFT) which we present below.

A FIFO queue is maintained for each virtual computer. The length of time slice is denoted by \(t^*\). During each period of length \(t^*\), each virtual computer is reserved at most \(\frac{t}{2}\) time units on the processor. A color, which is white initially, is associated with each virtual computer to indicate whether a virtual computer has utilized all the time reserved for itself during a time slice. If so, the color of the virtual computer becomes red, else it remains white. During each period of length \(t^*\), the scheduler executes the first available task from a white queue until the remaining reserved cycles for this virtual computer are finished or the task completes, which ever is first. In the first case, the color of the virtual computer is changed to red. If the task is not completed, then it is placed at the head of the queue. At the end of each period, all colors are set to white.

**Theorem 3:** For a uniprocessor VC system, the RFT with a time slice of length \(t^*\) is an average feasible scheduler over time intervals of length \(t^*\).

**Proof:** The algorithm delays each task at most \(t^*\) units of time. However, the number of tasks completed in each period is the same as the one on the virtual computer. \(\square\)

### 4.2 Feasible Scheduler for Multiprocessor VC Systems

In this paper, we only consider multiprocessor VC systems where the number of real processors is equal to or greater than the number of virtual computers, and the real processors and virtual computers have identical speed, whereas in \(2\) we cover heterogeneous virtual computers. In such a system, there exists always a strictly feasible schedule which can be obtained online without knowing the characteristics of the tasks by assigning each virtual computer to a processor, and

<table>
<thead>
<tr>
<th>Virtual Computer</th>
<th>Task</th>
<th>(a_i)</th>
<th>(l_i)</th>
<th>(d_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V_1)</td>
<td>(T_1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(V_2)</td>
<td>(T_2)</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 6: A nonfeasible schedule used in Theorem 2.
executing all tasks of the virtual computer on the same processor. However, this scheduler wastes idle cycles. Therefore, we search for on-line scheduling algorithms that utilize the idle cycles and remains still feasible.

One such algorithm is the least laxity first (LLF) algorithm which at each decision point executes the task with the minimum laxity. The laxity of a task at a given time is the difference between the elapsed time to the deadline of the task and the remaining length of the task. Ties are broken arbitrarily. The feasibility of this algorithm is based on the assumptions that the cost of the migration is negligible.

**Theorem 4:** Consider a multiprocessor VC system. If the local scheduling algorithms of the virtual computers are FCFS and task lengths are known, then the LLF algorithm is strictly feasible.

**Proof:** At any given time, there can be at most \( n \) tasks with zero laxity. Since there are at least \( n \) processors, all tasks with zero laxity will always be scheduled. Hence, each task will complete by its deadline. \( \square \)

In the case where the cost of migration is not negligible, then the LLF algorithm can be modified as follows. At any decision point, if selection of the least laxity task for execution preempts a task, that task is killed and requeued at the head of the queue of the corresponding virtual computer. We call this algorithm the least laxity first with restarts (LLFR).

**Theorem 5:** Consider a multiprocessor VC system where the cost of migration is not negligible. If the local scheduling algorithms of the virtual computers are FCFS and task lengths are known, then the LLRF algorithm is strictly feasible.

**Proof:** Since there can be at most \( n \) tasks with zero laxity, at any given time, the task which is killed and will be restarted at a future time cannot have zero laxity at the time it is killed. \( \square \)

If task lengths are not known, average or bounded feasible schedulers can be derived [?]. In a VC system, the primary goal of the scheduling is to meet the VC constraints. The secondary goal is to optimize a secondary performance metric subject to meeting deadlines. A scheduling algorithm is said to be *optimal* if it optimizes the secondary performance criterion subject to VC constraints. Hence, an optimal scheduler chooses the best among the feasible schedules. Derivation of optimal schedulers for a given secondary performance metric is beyond the scope of this paper.

## 5 Comparison to Distributed and Real Time Systems

In this section, we highlight the main differences between scheduling in VC systems and traditional scheduling in distributed and real time systems. In traditional general-purpose distributed operating systems, the common performance criteria are average throughput, average response time, and idle processor cycles [?]. However, the notion of providing predictable performance to each user is not emphasized. Below, we present examples to demonstrate that algorithms which balance load, minimize average response time, or minimize idle cycles do not simultaneously satisfy the time constraints that the VC paradigm imposes on tasks (despite the fact that there is a schedule that meets these constraints). In all the examples, a strict VC system with two virtual computers (\( V_1 \) and \( V_2 \)) and two processors (\( P_1 \) and \( P_2 \)) is used. The processing speeds of the real processors and virtual computers are identical. The deadlines of tasks are assigned by the a strict VC system.

**Example 1:** Consider the three tasks in Figure 7 where \( T_1 \) and \( T_2 \) are submitted to \( V_1 \) and \( T_3 \) is submitted to \( V_2 \). Figure 7 gives the task arrival times \( \langle a_i \rangle \), lengths \( \langle l_i \rangle \) and deadlines \( \langle d_i \rangle \). Suppose that system balances load on the processors \( P_1 \) and \( P_2 \). Figure 8 depicts the schedule on processors \( P_1 \) and \( P_2 \) under load balancing. As illustrated in Figure 8, task \( T_3 \) misses the deadline imposed by a strict VC system, completing at \( t = 600 \) rather than \( t = 400 \), although the LLF algorithm would have generated a strictly feasible schedule. \( \square \)

**Example 2:** Consider the four tasks with the characteristics described in Figure 7. Assume that

<table>
<thead>
<tr>
<th>Virtual Computer</th>
<th>Task</th>
<th>( a_i )</th>
<th>( l_i )</th>
<th>( d_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_1 )</td>
<td>( T_1 )</td>
<td>0</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>( V_1 )</td>
<td>( T_2 )</td>
<td>100</td>
<td>500</td>
<td>600</td>
</tr>
<tr>
<td>( V_2 )</td>
<td>( T_3 )</td>
<td>200</td>
<td>200</td>
<td>400</td>
</tr>
</tbody>
</table>

**Figure 7:** Tasks characteristics in Example 1.

\[ T_1 \]
\[ T_3 \]
\[ T_2 \]

<table>
<thead>
<tr>
<th>( P_1 )</th>
<th>( T_1 )</th>
<th>( T_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_2 )</td>
<td>( T_2 )</td>
<td></td>
</tr>
</tbody>
</table>

| 0 | 100 | 400 | 600 |

**Figure 8:** Schedule on \( P_1 \) and \( P_2 \) in Example 1.