On-Line Fair Allocations Based on Bottlenecks and Global Priorities

Yoel Zeldes
Department of Computer Science
The Hebrew University
91904 Jerusalem, Israel
yhotdog@gmail.com

Dror G. Feitelson
Department of Computer Science
The Hebrew University
91904 Jerusalem, Israel
feit@cs.huji.ac.il

ABSTRACT

System bottlenecks, namely those resources which are subjected to high contention, constrain system performance. Hence effective resource management should be done by focusing on the bottleneck resources and allocating them to the most deserving clients. It has been shown that for any combination of entitlements and requests a fair allocation of bottleneck resources can be found, using an off-line algorithm that is given full information in advance regarding the needs of each client. We extend this result to the on-line case with no prior information. To this end we introduce a simple greedy algorithm. In essence, when a scheduling decision needs to be made, this algorithm selects the client that has the largest minimal gap between its entitlement and its current allocation among all the bottleneck resources. Importantly, this algorithm takes a global view of the system, and assigns each client *a single priority* based on his usage of *all* the resources; this single priority is then used to make coordinated scheduling decisions on all the resources. Extensive simulations show that this algorithm achieves fair allocations according to the desired entitlements for a wide range of conditions, without using any prior information regarding resource requirements. It also follows shifting usage patterns, including situations where the bottlenecks change with time.

Categories and Subject Descriptors

C.2.3 [**COMPUTER-COMMUNICATION NETWORKS**]: Network Operations—*Network management*; D.4.1 [**OPERATING SYSTEMS**]: Process Management—*Scheduling*; K.6.2 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Installation Management—*Pricing and resource allocation*

General Terms

Design, Management, Performance

Keywords

Fairness; Bottlenecks; Resource allocation; Entitlements; Online algorithm; Priority inversion

ICPE'13, April 21–24, 2013, Prague, Czech Republic.
Copyright 2013 ACM 978-1-4503-1636-1/13/04 ...$15.00.

1. INTRODUCTION

Fairness is a basic issue in scheduling and resource management, and there are many different definitions of what “fairness” means. We use the definition that a fair system is one which allocates resources according to each client’s entitlement: a client that is entitled to 30% of the system should get 30% of the resources[1]. However, this definition is problematic, because clients may have uneven requirements. For example, if a client is entitled to 30% of the system, and he would like to use 70% of the network bandwidth but only 10% of the CPU cycles, what would be a fair allocation?

A system in which clients alternate between using multiple resources, e.g. the CPU, network, and disk, can be modeled as a queueing system. In queueing systems one finds that clients spend the majority of their time waiting in queue for a specific subset of the resources, namely the *bottleneck* resources. The degree to which clients make progress is constrained by the service they receive on these bottlenecks [14]. It has therefore been suggested that bottleneck resources are the most important resources in terms of scheduling and resource allocation [7, 5]. By controlling allocations of the bottlenecks, one controls allocations across the whole system.

Given the insight about focusing on bottlenecks, one can define a fair allocation as follows: *each client should be given his entitlement of the system bottleneck* [7, 5]. Thus if the network is the bottleneck, our example client will be cut beck and given only 30% of the bandwidth. But if the CPU turns out to be the bottleneck, and the network is not, then he can get all he wants, because his requirements on the bottleneck are lower than his entitlement.

This definition can be extend to the case where the system suffers from multiple bottlenecks by guaranteeing that each client receives his entitlement on *at least one* bottleneck. It has been shown that, for any combination of entitlements and requests, a fair allocation according to this definition exists [4]. However, the algorithm to find this allocation works off-line and assumes full information is available.

Our main contribution is to extend this result with an on-line algorithm, that *achieves a fair allocation without knowing each client’s requirements in advance*. In fact, it doesn’t even have to know which clients are going to be active and when. All it needs to know is their relative entitlements. It then serves the requests they make in an order that is based on their entitlements and the allocations they have received so far. Essentially, this is a simple greedy algorithm.

In the next section we provide more details about previous work and the required background. Section 3 then explains the details of the on-line algorithm. An evaluation using extensive simulations to

[1] Thus entitlements are similar to capacity or throughput guarantees in SLAs.

show that this algorithm indeed achieves the desired allocations is presented in Section 4. We conclude in Section 5.

2. BACKGROUND AND RELATED WORK

The two main considerations for scheduling algorithms are fairness and quality of service. In the context of interactive and real-time applications, quality of service (as reflected in response times and meeting deadlines) is paramount [8, 21]. We focus on fairness, which is more relevant to servers, especially in the context of clouds and virtualization where several servers are consolidated onto a common physical platform.

The notion of fairness is somewhat tricky to nail down. At its core lies a conflict between accounting, where we just count how much each client receives, and utilitarianism, where we consider the benefit derived from the allocated resources [20]. In the context of computing systems we typically do not know the benefits, so we are left with different forms of accounting. In particular, striving to satisfy given entitlements is known as "fair-share" scheduling. In this context, a major aspect of fairness is isolation between competing clients: guaranteeing a certain share of the resources to one client implies that this share will be received irrespective of the actions of other clients. For example, this may be important in real-time settings and to prevent certain denial-of-service attacks [15].

An example of a large-scale system based on such fairness considerations is the Intel NetBatch platform [23]. This is essentially a global enterprise grid system with hundreds of clusters and tens of thousands of machines in dozens of locations around the world. Intel business and engineering groups purchase computing resources according to their needs and make them available to other groups as part of the grid. However, each group retains ownership of their resources. This means that each group's jobs receive priority for running on their own machines: if no alternative resources are available, whatever is running on these machines will be suspended and the owner's jobs will run instead. Thus allocations reflect ownership, and by implication allocations reflect the entitlements as represented by the relative fraction of the resources that have been purchased by each group.

Many schemes have been devised to achieve fair-share scheduling. These include simple accounting where allocations are simply not made to clients who have achieved their entitlement [13], lottery scheduling where lottery tickets are distributed according to entitlements and the actual allocations are randomized [19], manipulation of a process's priority so as to nudge it towards achieving the desired allocation [6], and using the min-max principle to prioritize those clients who are lagging behind [17].

A whole class of fair allocation schemes is based on the notion of virtual time [22, 16]. This essentially means that time is counted at different rates for different clients, thereby leading to different but controlled allocations. A more sophisticated approach is to define fairness according to the divergence between what a client has received and what he was entitled to receive up to now [18]. This can also be used directly to prioritize scheduling decisions [8, 3].

All the above schemes are oblivious to the system state. There has been very little work considering scheduling decisions that are driven by resource contention. Unlike bottleneck-based scheduling (to be defined more fully below), this work is usually concerned more with preventing bottlenecks than with focusing on them. The idea is to prefer clients that make little use of contended resources, so as to reduce the danger of congestion [1, 11].

The schemes described above were typically applied to one specific resource, most commonly the CPU or the network. This leaves the question of how to control the joint allocation of multiple resources. The simplest approach is asset fairness. This means that we sum up the usage of the different resources, and ensure that this sum is proportional to the entitlement. However, this is inefficient in the sense that someone who is the sole client of an unpopular resource will be held back due to contention by others for a resource in which he is not interested.

Another approach is dominant-resource fairness [10, 9]. In this scheme each client is represented by the resource he wants the most of, which is called his dominant resource. The allocations of the dominant resources are then made proportional to the entitlements. A simple algorithm that achieves this has been shown, which is also "strategy proof". This means that a client cannot increase his allocation and his throughput by increasing his demands. However, note that as long as fairness (however one may decide to define it) is maintained, there is no way that the system can distinguish between a client that genuinely requires more resources and one that is trying to game the system. In particular, if a client can cause the system to give him a larger allocation, but this larger allocation is still considered fair, this is not necessarily a problem. It is therefore not clear that the property of strategy proofness as defined above is indeed appropriate.

Asset fairness and dominant-resource fairness are oblivious to the system state. The alternative is to monitor the system to identify the current bottlenecks, which may change from time to time [7, 5]. In bottleneck-based fairness an allocation is considered fair if all clients do not have justified grounds for complaining. This means that either they *receive all they want*, or else they *receive their entitlement on at least one bottleneck*. The claim that they cannot complain about this is based on the fact that bottlenecks are by definition contended resources, so giving more would have to come at the expense of other clients who also have their entitlements[2]. It has been shown that an allocation based on this principle is guaranteed to exist [4, 12]. Note that asset fairness and dominant resource fairness do not possess the "no justified complaints" property: in those schemes, a client may be prevented from receiving resources that could have been allocated without hurting anyone else.

In order to sharpen the differences between the above approaches, we suggest the fruit salad metaphor. Assume a fruit salad buffet with a bowl of diced apples, a bowl of diced oranges, a bowl of cherries, etc. Each diner is invited to create his or her own mix. Asset fairness then means that each diner gets a small bowl of the same size and can fill it in whatever way he chooses. This is intuitively fair but inefficient: if ten people like cherries, but only one each like apples, oranges, and bananas, why not let these three take everything that is there? Dominant resource fairness means each diner gets the same amount of his favorite fruit. This suffers similar inefficiency as asset fairness. The bottleneck-based approach means that each diner gets the same amount from *one of the fruits that ran out*, thus emphasizing the importance of those fruit which are in high demand, while explicitly allowing you more of fruit that nobody else wants. This analogy also shows the problem in finding a fair solution according to this definition: when you are making the allocations, how do you know which fruit are going to run out?

Given a scheduling scheme, the remaining question is how to apply it. The simplest approach is to schedule each resource individually. However, this may lead to undesirable results. For example, a client may use less than his entitlement of one resource, and more than his entitlement of another. Assume the first resource is a bottleneck, while the second is not. This client should therefore be prioritized in order to provide it with its due entitlement of the

[2] Note that a resource may become a bottleneck even if it is used by only one client. But in that case this client is fully utilizing the resource, so there is no more to allocate.

bottleneck resource. But when it is waiting for the second, non-bottleneck resource, it will receive a low priority. This will cause delays that affect the fairness as reflected by using the bottleneck resource. The solution to such mixed priorities is to use a global priority across the whole system. This global priority is *computed* based on the bottleneck resources, but it is *applied* to *all* the resources.

A building block that can be used to implement the idea of bottleneck-based fairness is the RSVT scheduler, which is a modular scheduler that can be "glued" to different resources to provide virtual-time-based prioritization [2]. This needs to be combined with a centralized monitoring facility that identifies bottleneck resources, and sets the global priorities. In this paper we fill in the algorithmic details and simulate how such an implementation may be expected to operate when faced with various scenarios of contending clients with different requirements. The actual implementation, and an evaluation of the overheads and performance characteristics, are left for future work.

A basic assumption of all the above works dealing with the allocation of multiple resources is that clients have constant resource usage profiles. For example, a client may always use twice as much disk as CPU, so if we allocate 30% of the CPU he will also use 60% of the disk bandwidth, and if we limit his disk usage to 20% we are effectively also limiting his CPU usage to 10%. This assumption is reasonable in cases where clients are performing essentially the same actions all the time, for example serving incoming requests that all have the same nature. However, our on-line algorithm does not rely on such persistence. Rather, it just tracks the cumulative usage of different resources by each client, regardless of the precise patterns in which requests are made. In particular, clients are free to change their behavior, and the algorithm will follow them and even identify changes in the set of bottlenecks.

3. THE ON-LINE ALGORITHM

Consider a set of clients with different entitlements, each with his unique resource usage profile. We know that a bottleneck-based fair allocation exists for any combination of entitlements and requirements. But the proofs given in [4, 12] are non-trivial, use complete information about the requirements in order to find the solution, and require the resource usage profile to be constant. The difficulty stems from the fact that we do not know in advance which resources will turn out to be bottlenecks. Moreover, it turns out that there may be multiple solutions, and even different solutions in which different resources become the bottlenecks.

An interesting question is therefore whether the principles of bottleneck-based fairness can be distilled into an on-line algorithm that will maintain a fair allocation despite not knowing what the requests are in advance. Obviously such an algorithm also does not know the bottlenecks in advance, but we assume it can discover them when they materialize, by monitoring the utilization of each resource. Thus it can also adjust if the bottlenecks change with time.

Our algorithm is based on the following two principles:

1. The "no justified complaints" condition requires that clients receive their entitlement on at least one bottleneck resource. This leads to the idea that a client's priority should be proportional to the divergence between his entitlement and his actual consumption, on the bottleneck resource where this divergence is *minimal*. The reason is that as we allocate more resources and enable him to run, this is where the gap will be closed first. A process that suffers from a larger minimal gap should be given priority in order to enable it to catch up and achieve its entitlement.
2. Given that there may be several bottlenecks and any one of them may be used to satisfy a client, a global viewpoint is required. Thus each client will be assigned a global, system-wide priority value, and scheduling decisions on all the different resources will be made based on these system-wide priorities.

While the algorithm based on these principles is simple (as shown below), it is not self evident. In particular, basing priorities on the minimal *gap* contradicts the commonly used max-min approach, which bases priorities on the minimal *consumption*, which is equivalent to using the *maximal* gap. But in the multi-resource scenario with fixed usage profiles this is problematic, because boosting the usage of the least-used resource also increases the usage of the heavily used resources, which may be impossible (or at least undesirable) if they too are bottlenecks.

Importantly, using global prioritization implies that scheduling on the different resources will be coordinated. This is in stark contrast with the prevailing methodology used today, where each resource has its own scheduler, and the scheduling is done based on a myopic view of each individual resource. As a result, we prevent situations in which a scheduler responsible for some resource that is not contended decides to run client A and let client B wait in the queue, due to some local efficiency consideration, when A is actually receiving all he wants while B is not receiving his due allocation on the system bottlenecks.

To formalize these ideas we need some notation. Let there be N clients and M resources. Client i has entitlement e_i, where $\sum_i e_i = 1$. Ideally, the fraction of the system that client i was entitled to receive up to time t is simply $a_i(t) = e_i t$. For each resource j, his consumption of resource j up to time t will be denoted $c_i(j,t)$. Both allocation and consumption are expressed in the same units, namely time using the resource, and these units apply equally to all resources (as opposed to units like bandwidth which are specific to a certain resource). Let $J(t)$ denote the set of resources that are bottlenecks at time t. Based on the above considerations, we calculate the global priority of client i as the minimal difference between the ideal allocation and the consumption, across bottleneck resources:

$$pri_i(t) = \min_{j \in J(t)} \{a_i(t) - c_i(j,t)\}$$

The queue of each resource will be sorted according to these global priorities, and the highest priority client will be selected. If several clients have the same global priority, the local scheduler may use its local considerations to break the tie. In either case, the selection is always for a limited time: a quantum of CPU time, the sending of a packet, the reading of a disk block, etc. This causes $c_i(j,t)$ to grow for the selected client, while at the same time $a_i(t)$ grows for all clients (but by less). As a result another client will most probably be selected the next time around, and if all the clients have the same entitlements our algorithm will produce simple round-robin scheduling.

But what happens if there are no bottlenecks? In this case we prioritize the different processes according to the difference between their allocation and their total consumption on all resources, namely we revert to asset fairness:

$$pri_i(t) = a_i(t) - \sum_j c_i(j,t)$$

Note, however, that this only affects the scheduling order and does not affect the allocations. When there are no bottlenecks there is no real contention and therefore all requests will be granted.

The above formalization cannot be used as described because it assumes that all clients arrived at the system at time 0 and have been active continuously ever since. In general, of course, this is not the case. Clients may come and leave at different times, and may suspend waiting on some external event. Therefore the following adjustments must be made.

First, at each time t only active clients are considered: clients who have terminated or have not arrived yet (or are suspended) will be excluded. Thus the entitlements need to be renormalized. Denoting the set of active clients by A, the entitlements used to calculate the priorities will be $e'_i = e_i / \sum_{i \in A} e_i$.

Second, if a new client enters the system at time t_0, his entitlement will be computed as $a_i(t) = e'_i(t - t_0)$. As e'_i may change with time, this is actually computed piecemeal by summing over intervals where e'_i is constant.

Third, handling clients who become suspended or do not use their full allocation is done similarly to RSVT [2]. The problem is that clients may divide their time across the resources in very different ways. Thus we may encounter situations where a client predominantly uses a resource that nobody else is interested in. As a result, his consumption may outstrip his entitlement (and there is no reason to prevent this, as otherwise the resource would stay idle). But if later some other client starts using this resource, and the resource becomes a bottleneck, then the first client will have a hugely negative priority and will be locked out for a long time. This should be prevented.

Conversely, consider a resource that nobody uses to a great degree. This resource is mostly idle, and for all clients their consumption will lag way below their allocation. Then, if the client with the highest entitlement starts to use this resource extensively and turns it into a bottleneck, and assuming this is the only bottleneck, he will lock out all others, because his allocation had outgrown his consumption more than for all others. This too should be prevented.

The above scenarios can be summarized as follows. Ideally, we want to make allocations based on entitlements, and have the consumption follow these allocations. But if the consumption does not follow the allocation, and becomes either too big or too small relative to the allocation, we need to bound this difference [22]. The way to do this is simple: instead of accumulating allocations and consumptions since the client arrived, we only consider a window of the last T seconds. In such a limited window allocations and consumptions reflect the most recent entitlements and actual use, but they cannot diverge by more than T.

An important concern for online algorithms is their overhead. Our algorithm is centralized in the sense that it utilizes global information for its prioritization. Hence it faces the danger of not scaling for large systems. However, we note that only the accounting needs to be performed on each and every dispatch decision, and this can be incorporated into the mechanism performing the dispatch — for example, the CPU scheduler or the mechanisms controlling packet sending and disk I/O. The prioritization which uses this information can be limited to a desired granularity, e.g. once every so many seconds, as we indeed do in the simulations.

4. SIMULATION RESULTS

In this section we describe some of our simulation results, focusing on those that best illustrate various features of the scheduling algorithm.

4.1 The Simulator

To assess the behavior of the algorithm presented above we wrote an event-based simulator. This simulates the evolution of a system supporting several processes (or a server with several virtual machines). The processes represent the clients. Each process runs a "program", which specifies its resource usage. The programs are composed of multiple iterative phases, where each iteration consists of using different resources in the system one after the other. Use of a resource is described by a tuple of the form $\langle res,\ mean,\ width,\ dist\rangle$. res is the resource being used. The other three parameters describe the distribution of service times when the resource is used ($dist$ can be UNIForm, EXPonential, or NORMal; $mean$ is obviously the mean, and $width$ is half the range for UNIF or the standard deviation for NORM). As each use of each resource is simulated, the service time for this instance is selected at random from the specified distribution. We typically use the normal distribution, because the service times are expected to be influenced by many independent factors.

```
# Priority:
34
# Requests:
loop 1
CPU_CS 2 0 NORM
loop 40
DISK_CS 0.1 0.05 NORM
CPU_CS 0.1 0.05 NORM
```

Figure 1: *Example process program in the simulation.*

An example program is shown in Fig. 1. This process has a relative priority (entitlement) of 34. Its program consists of using the CPU for precisely 2 seconds, and then performing 40 iterations of using the disk for 0.1 seconds and then the CPU for an additional 0.1 seconds, both with a standard deviation of 0.05 seconds.

Each process that wishes to use a certain resource is passed on to that resource's scheduler. Each scheduler keeps track of every process's usage of its resource in the last 3 seconds (This is the T mentioned above). A global prioritization agent collects the prioritization data from the schedulers of bottleneck resources so as to create the global priorities. Bottleneck resources are those that have been active for more than 90% of the time in the last 3 seconds (these are all configurable parameters). Thus the identification of a bottleneck will lag 3 seconds behind the time it actually became a bottleneck.

Allocations can either be continuous, or, if the resource name ends with _CS (allows context switching), they are done in quanta of 0.1 seconds. This may represent a scheduling quantum, the sending of a packet, or the reading of a disk block. Note, however, that the resource names and the service times should not be taken too literally. Obviously in many real systems CPU quanta are much shorter than disk access times. We use "CPU", "disk", and "network" to easily refer to the resources, but they could just as well be abstracted as "R1", "R2", etc. Our goal in the simulations is to probe the behavior of the algorithm under diverse conditions, not to approximate specific benchmarks.

The simulator as described has the limitation that each process uses one resource at a time. This is not completely realistic, as a process may spawn multiple parallel threads on a multiprocessor system, and asynchronous I/O can be done in parallel to computation. Note, however, that this is a limitation of only the simulator and not of the algorithm itself.

4.2 Simple Experiments

The first experiments are meant to demonstrate that the simulator works as expected on simple scenarios with a single resource, and to explain the way we present the simulation results.

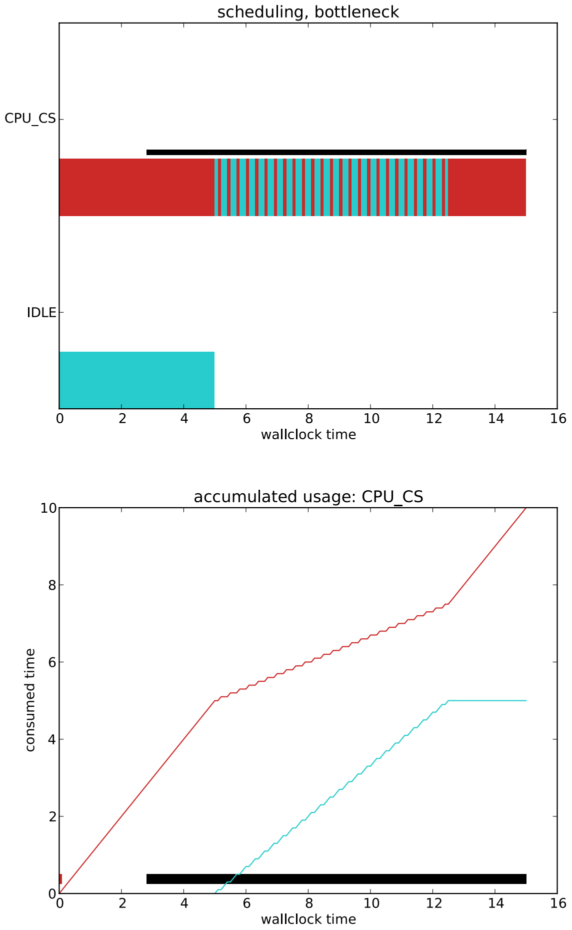

Figure 2: *An example of two processes with different priorities contending for a resource. Service times are deterministic.*

Each experiment involves one or more processes, using one or more resources. In the graphs, the horizontal axis represents wallclock time. The first graph for each experiment is actually a Gnatt chart, with a lane for each resource. This lane is color-coded according to which process is using the resource at each instant; no coloring indicates that the resource is idle. A black line segment above the lane identifies those intervals in which the resource is considered to be a bottleneck. The second graph shows the cumulative resource usage by each process. Each process is represented by a line, with the same color as in the Gnatt chart. If a process is continuously active, on whatever resource, this will be a straight line with slope 1. When a process waits for a resource, this is represented by a horizontal line segment. When two or more processes share a resource the effect is to produce a slope smaller than 1, but this is actually a sequence of small steps. We occasionally also look at the cumulative usage of a select resource, rather than all resources together.

A simple example is shown in Fig. 2, involving two processes. The red process has a relative priority of 33% and wants to use the CPU for 10 seconds, while the blue one has 67% and wants 5 seconds. Initially only the red process runs, and gets full use of the CPU (note that we use the "idle" resource to delay the blue process). 5 seconds later, when the blue process starts, their relative shares become governed by their priorities. This is achieved by giving the blue process two quanta each time, while the red process only receives one. As the blue process has a higher priority it terminates first, again leaving the CPU exclusively to the red process. A generalization to four processes that start together with priorities of 10%, 20%, 30%, and 40% is shown in Fig. 3. As they all want to run for 10 seconds, the one with the highest allocation terminates first, and then the others in decreasing order. As each one terminates, the CPU is divided among the remaining processes according to their designated shares.

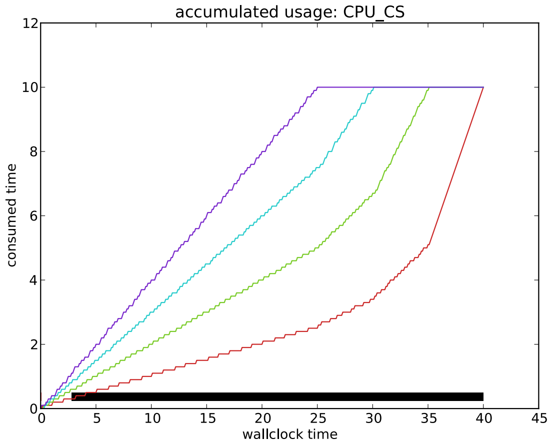

Figure 3: *An example of four processes with different priorities contending for a resource. Service times are deterministic.*

Note the smooth transition as the blue process starts running in Fig. 2. This reflects two features of the scheduling algorithm. First, when the red process runs alone, its effective relative priority is 100% because there is no other process in the system. As it indeed receives full use of the CPU, it does not accumulate any lag between its entitlement and its consumption. Second, when the blue process starts, it is initialized with zero entitlement and consumption. As a result both processes start from an equal footing, and immediately receive allocations according to their relative entitlements.

4.3 Properties of the Scheduling Algorithm

Fig. 4 shows how the algorithm adjusts when different processes use different resources. Here we have 3 processes, that repeatedly (5 times) use the CPU for 0.1 second and then some other resource for 10 seconds. The blue process has an entitlement of 70%, and uses the network. The green and red processes have entitlements of 20% and 10%, respectively, and use the disk. As we see, the blue process actually receives nearly 100% of the network, despite having an entitlement of only 70%, because no other process requests it. Green and red receive approximately 67% and 33% of the disk, which are also much higher than their entitlements, but maintain the correct ratio. The CPU is hardly used, and therefore does not affect these allocations.

Note that due to the randomization used in the simulations the total time in any specific run does not necessarily sum to exactly 50 seconds of resource use. However, as shown in the bottom plot, the average of multiple runs does come out right. The error bars show the distribution of values observed in the individual runs. Importantly, in the initial part of the simulation, when all processes are active, there are no such variations. In the sequel, we will typi-

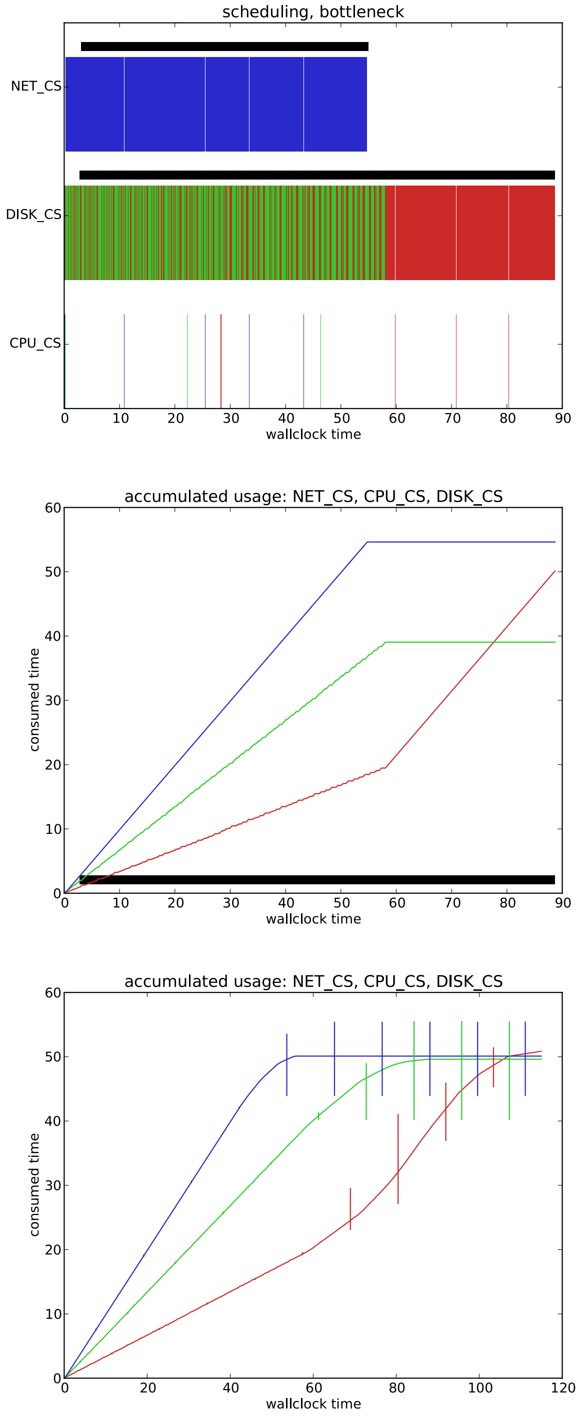

Figure 4: *An example of three processes using different resources, and thus enjoying a higher effective entitlement. Service times are randomized, with the top two graphs showing a specific run while the bottom one shows an average of many runs.*

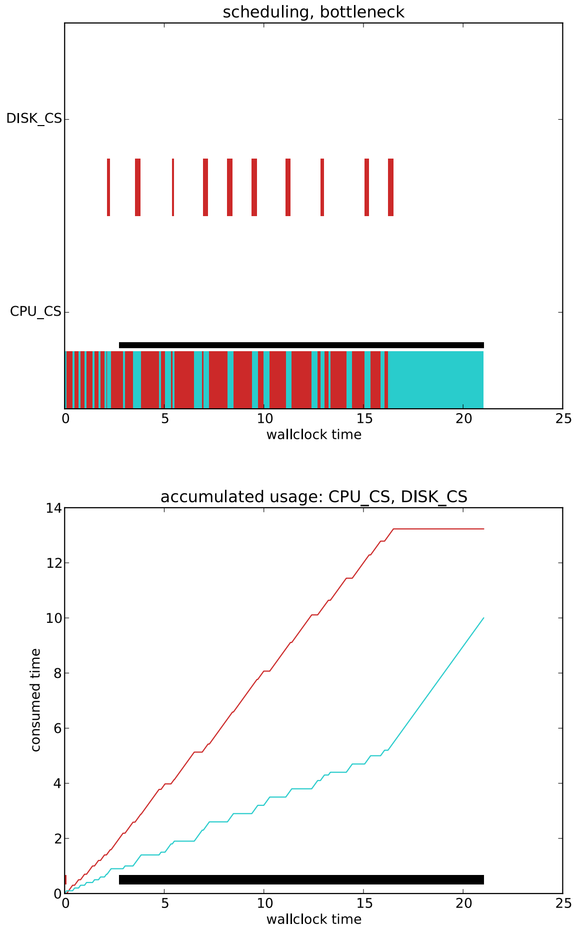

Figure 5: *Example where the grace period increases the ideal allocation also when a resource is not used. Service times are randomized.*

cally use averages and error bars to show the results of randomized experiments.

An important feature of the algorithm is the way it handles processes that skip from one resource to another, or become suspended. On one hand, we want continuity: if a process does not use a resource for a short time, its allocation should nevertheless continue to grow, so as to be available once it requests to use this resource again. On the other hand, we do not want the allocation to grow excessively relative to the consumption, so as to avoid situations that give the process an unbeatable priority that will allow it to lock out all other processes.

Following the RSVT scheduler [2] we define a grace period of 1 second during which the allocation continues to grow. The effect of this grace period is demonstrated in the following two experiments. In both experiments, the red process represents the user and has an entitlement of 70%, and the blue process represents some background activity with an entitlement of only 30%. In the first experiment (Fig. 5) the red process alternates between using the CPU for 2 seconds and then accessing the disk for 0.5 second. This is shorter than the grace period, so the allocation of the CPU continues to rise, and when he returns he gets exclusive access for a short while in order to make up for the deficit in consumption. In the other experiment (Fig. 6) the red process uses the CPU for 2

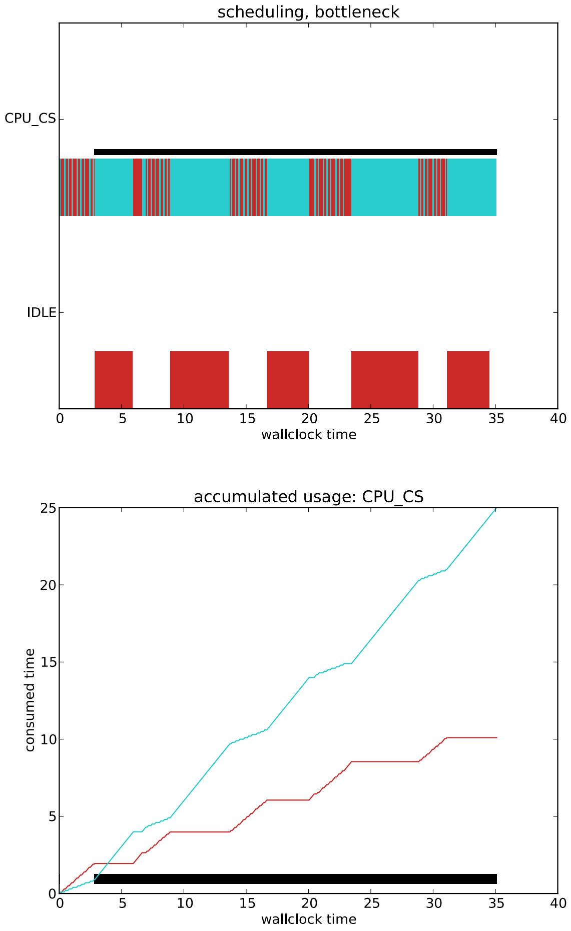

Figure 6: *Example where the grace period is not long enough to increase the ideal allocation also when a resource is not used. Service times are randomized.*

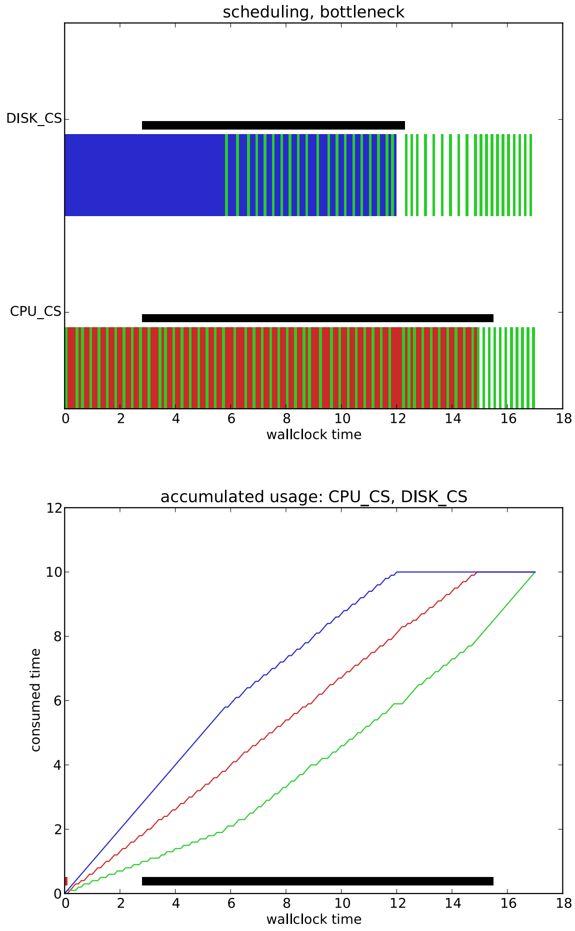

Figure 7: *An example of three processes with different priorities contending for two resources. Service times are deterministic.*

seconds and then suspends waiting for user input for 4 seconds on average. This is longer than the grace period, so when he returns he has to share the CPU with the blue process. To enable this to be seen clearly, we do not average multiple random runs but rather show a single run.

4.4 Further Examples of Controlled Allocations

A slightly more involved example is shown in Fig. 7. Here we have 3 processes with different priorities contending for the use of two resources:

- Red, with priority 40%, wants 10 seconds on the CPU
- Blue, with priority 40%, wants 10 seconds on the disk
- Green, with priority 20%, wants 2 seconds on the CPU and then 8 seconds on the CPU and disk alternately

The results of the run are as follows. Initially blue uses the disk with no contention, while red and green share the CPU according to their relative priorities. Then green starts using the disk. As a result we find that now all three processes make progress at the same overall rate. However, on each resource, green gets half what the other process gets. Thus red and blue receive their relative entitlements of 67% on their resources, and green receives his relative entitlement of 33% on each of the resources, for a total of 67%.

Fig. 8 shows what happens when the priorities are changed. In the first plot, all three processes have equal priorities of 33%. As a result the green process receives the same allocation as the others on each resource. But due to the fact that it uses both resources, its total rate of progress becomes double that of the others. This highlights the difference between asset fairness and bottleneck-based fairness: In bottleneck-based fairness, we are not bothered by the total allocations, but only by the allocations on each bottleneck.

The second plot shows what happens when the green process has a priority of 50, and the other two 25. We would then expect the green process to make progress at twice the rate on each resource, and at four times the rate in total because it uses both resources. However, this does not happen. The explanation is that when green alternates between the two resources, it leaves each of them idle half of the time. The other processes pick up the slack rather than leaving them idle. This illustrates the increased efficiency relative to dominant resource fairness.

Interestingly, these effects are observed only when the service times are all deterministic and equal, i.e. when the standard deviations in the distributions are 0 (and indeed this is the configuration shown in the graphs). When this is the case, the processes proceed

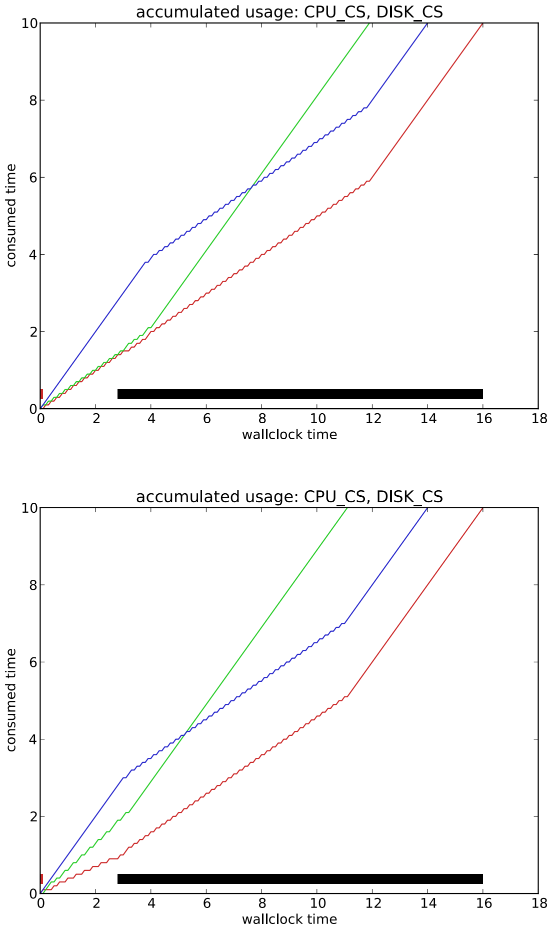

Figure 8: *The same scenario as in Fig. 7 but with different relative priorities. Service times are deterministic.*

in lockstep and the different allocations match perfectly. But if randomization is introduced, the synchronization is broken. The green is then not always available at the right instant to receive his share of each resource, but blue and red, which only use one resource each, are always there waiting. As a result green is cut back, and makes progress at about the same rate as the other two.

We now turn to a couple of more complex scenarios. A rather extreme example is shown in Fig. 9. Here four processes execute the same type of iterations 3 times each. The iterations are composed of 0.5 second network, 1 second CPU, and 2 seconds disk (as may happen when serving requests). The differences are in their priorities: purple has a priority of 97%, and the other three 1% each. Obviously purple's priority is much higher than the others, and indeed we find that it never has to wait for a resource (except to wait for the end of the current quantum) and makes the most progress. As a result it manages to finish its 3 iterations way ahead of the others. They are then left to contend with each other, and enjoy similar performance on average. However, each run may actually be somewhat different. Specifically, the in run depicted in the top two graphs of Fig. 9, blue was lucky and required slightly less CPU than green and red in the first iteration. As a result it managed to get to the disk first, and continued to run out-of-phase with the other two and to make better progress. Nevertheless, each process still

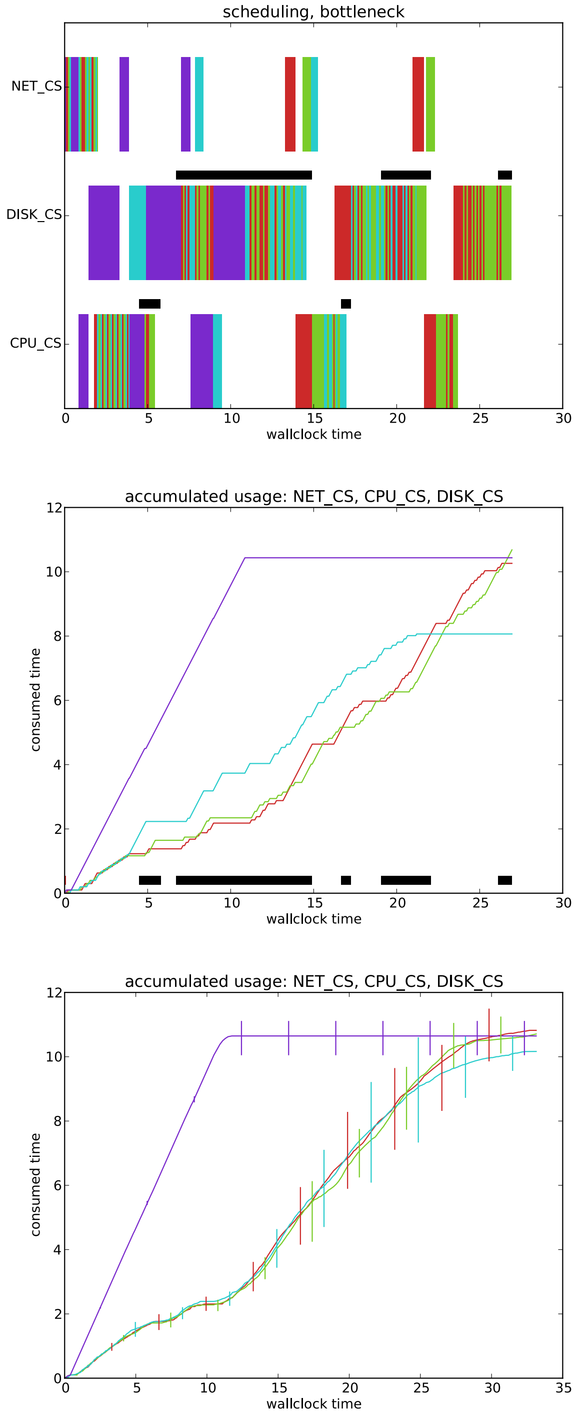

Figure 9: *An example of four processes with widely different priorities. Service times are randomized, with the top two graphs showing a specific run while the bottom one shows an average of many runs.*

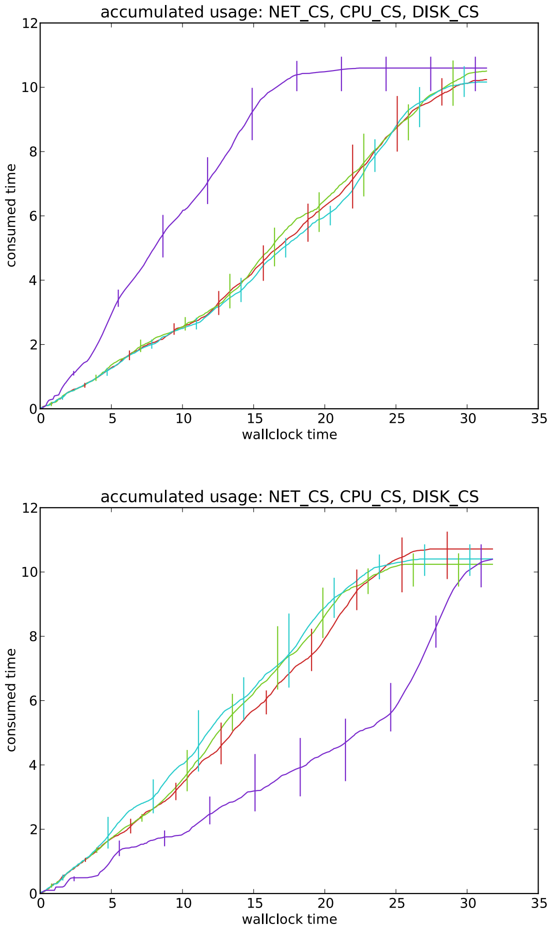

Figure 10: *Variations on the experiment shown in Fig. 9. Service times are randomized.*

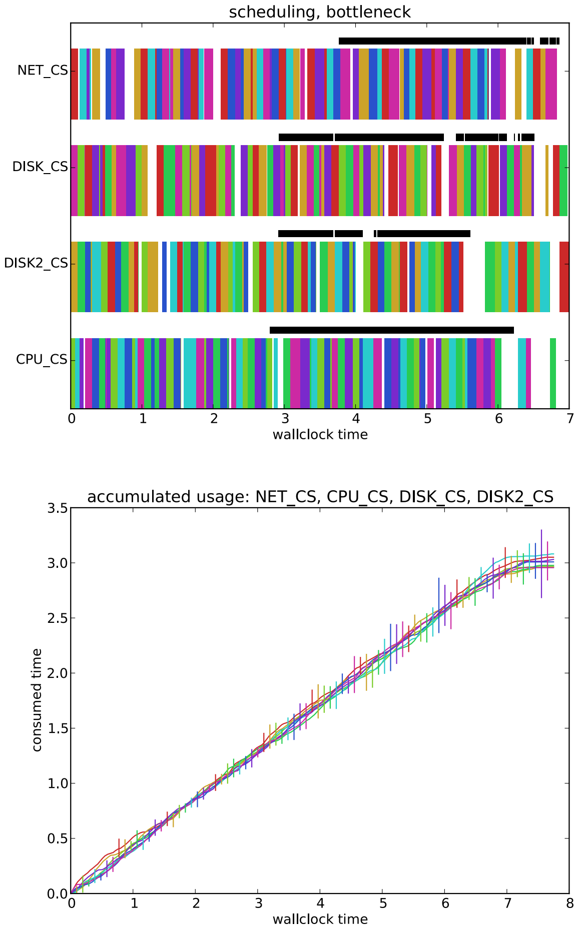

Figure 11: *An example where several different solutions are possible, and the algorithm gravitates towards the symmetrical solution. Service times are randomized.*

receives at least his entitlement on the bottleneck, in this case the disk.

Fig. 10 shows two variations on this experiment. In the first the priorities are not so extreme: purple is only 40%, and the other three are 20% each. In the second variation the priorities are inverted: purple is down to 10%, and the other three are 30% each. In these variations individual runs may be even more noisy than in the original experiment, but on average they are pretty clear. In the second variation purple appears to achieve more than $\frac{1}{3}$ of the others, because they sometimes need to wait for each other to release some resource, and thus leave another resource idle. This can be seen more clearly in longer runs, where the noise becomes smaller in relative terms, and therefore individual runs are very close to the average of many runs.

As mentioned in Section 3, there exist configurations in which several off-line solutions are in principle possible. Moreover, different solutions may involve different sets of bottleneck resources. One example of such a situation is the following. We have 8 processes and 4 resources. All the processes have equal entitlements, and they are arranged in pairs. Each pair uses 3 resources iteratively, for 0.1 seconds each time. The pattern is completely symmetric: if we arrange the resources in a circle, then each pair uses a different set of 3 consecutive resources on this circle. Possible solutions that abide by the no justified complaints criterion are as follows:

- All resources become bottlenecks, and all processes continuously use their 3 resources for $\frac{1}{6}$ of the time each.
- Two of the pairs run for 25% of the time on each of their resources, while the other two run for 37.5% of the time (this is why we need pairs: a single process cannot utilize 3 resources at 37.5% each). As a result two resources become bottlenecks, while the other two are only utilized 87.5% of the time. There are six such solutions with different sets of bottleneck resources.

Simulating this configuration with deterministic service times leads to the symmetric solution, with all resources utilized 100% of the time. The result of a randomized simulation is shown in Fig. 11. This is slightly less clear-cut than the deterministic version, and none of the resources maintain their bottleneck status without some gaps and idle periods. However, when observing the progress made by the 8 processes, they are found to achieve largely the same rate.

4.5 Allocations Based on Bottleneck Usage

In the previous examples the bottlenecks did not play a dominant role in the scheduling. However, there are situations where the

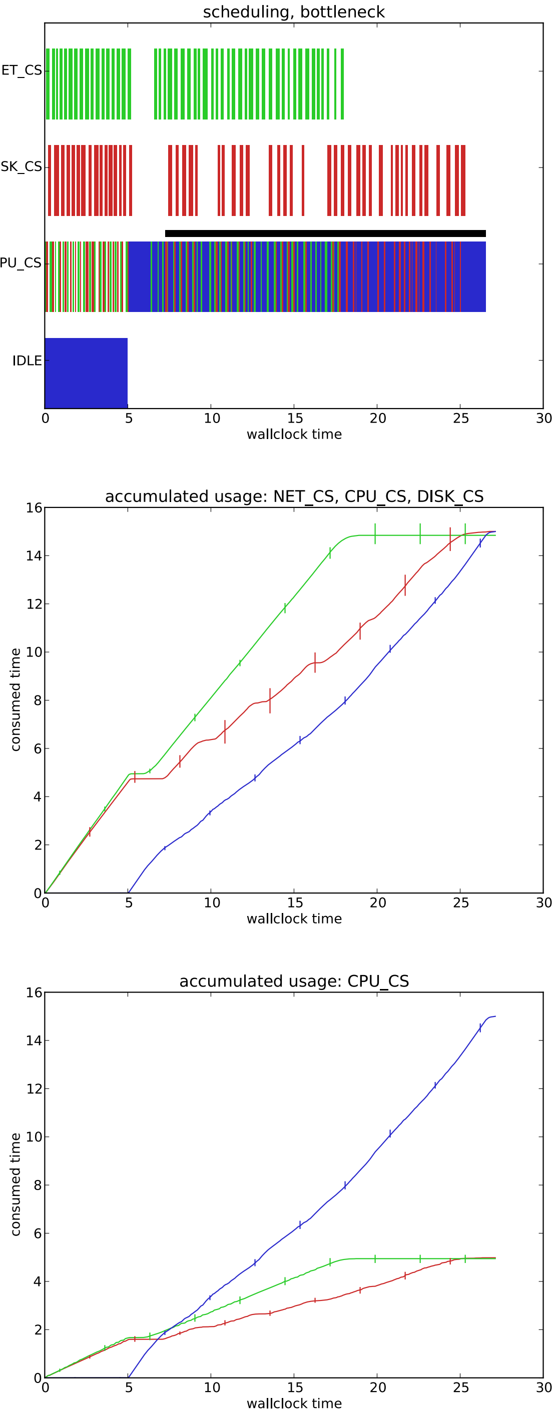

Figure 12: *An example where allocations change once a resource becomes a bottleneck. Service times are randomized.*

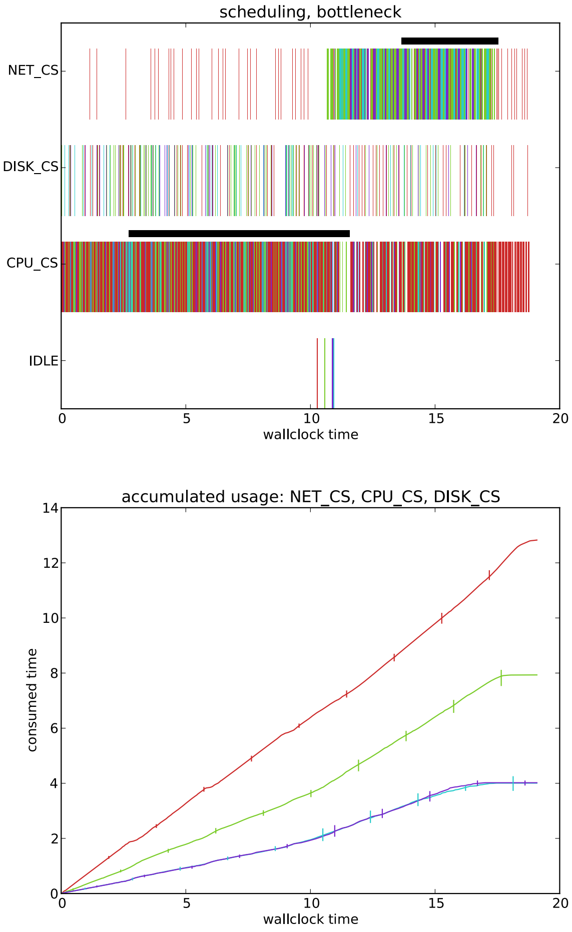

Figure 13: *An example where the focus shifts from one bottleneck to another. Service times are randomized.*

bottleneck resource actually has a decisive effect. An example is shown in Fig. 12. We start with 2 processes, with entitlements 33 and 67. The red process uses 1 second of CPU per 2 seconds of disk. The green process uses 1 second of CPU per 2 seconds of network. These requirements to not stress the system, so each runs continuously, and all 3 resources are 67% utilized. As there is no contention, the entitlement do not influence the allocations. After 5 seconds a third process (blue) becomes active. This process has entitlement 100, and only uses the CPU. The CPU then becomes a bottleneck. As a result we allocate 50% of the CPU (on average) to the blue process, 17% to the red process, and 33% to the green process. For the red process this is half what he got before, and as a result the usage of the disk also drops to half what it was. For the green process it happens to be exactly what he received before the blue process arrived, so nothing much changes.

Another example shows how the algorithm manages to track a shift from one bottleneck to another (Fig. 13). There are 4 processes, with entitlements of 50, 25, 12.5, and 12.5. The first uses the CPU 80% of its time, and the disk and network 10% each. The other three initially iterate between the CPU and disk, but then add the network too, for 70% of each iteration. As a result the CPU is the bottleneck in the first part of the simulation, but the network becomes the bottleneck in the later part, leading to a change in relative allocations.

Our last example is derived from the recent paper by Ghodsi et al. about using dominant resource fairness in network settings [9]. In that paper they suggest a test case where two processes each predominantly uses a different resource, and speculate that bottleneck-based scheduling would lead to strong oscillations as the system tries to satisfy them in turn. They further conjecture that a third pro-

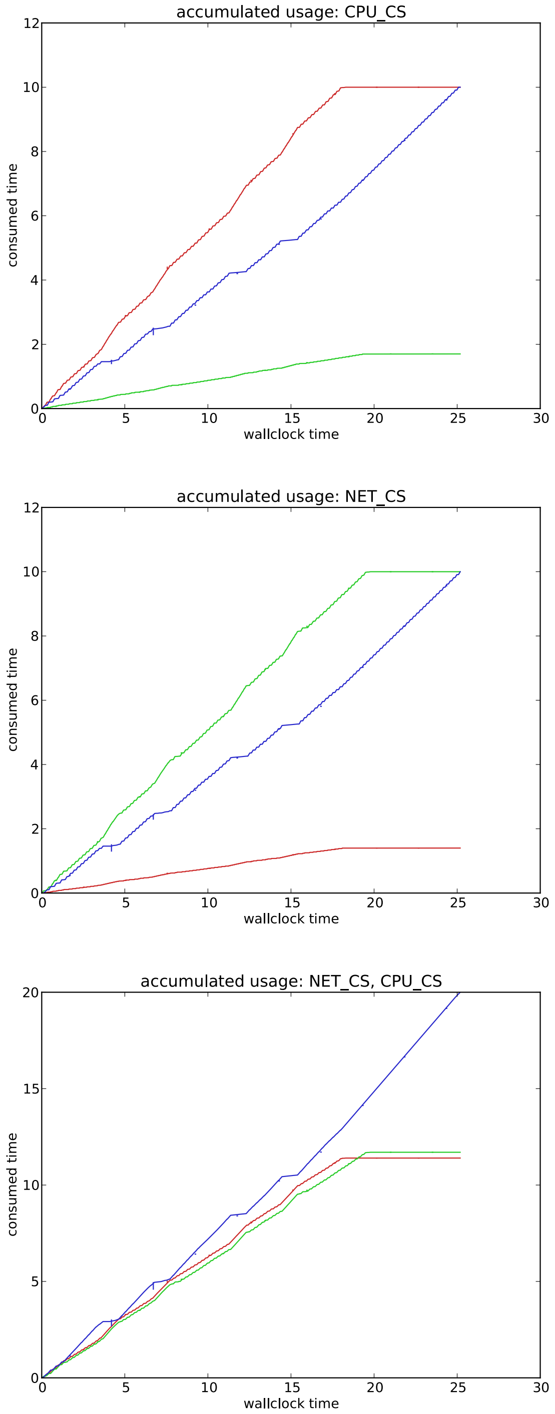

Figure 14: *Results for processes with complementary requirements. Service times are randomized.*

cess that wants both resources would not receive its due allocation at all. We implemented this scenario, with one process requesting 0.014 seconds on the network for each 0.1 second quantum on the CPU, a second requesting 0.1 seconds of the network for 0.017 of the CPU, the a third requesting equal use (in 0.1 second quanta) of both. All three processes have equal entitlements.

The simulation results show that the concerns were unfounded (Fig. 14). As expected, both the CPU and network become bottlenecks, and retain this status until one of the processes terminates. While some oscillations do occur, they are small. Moreover, these oscillations involve the third process, which tends to get ahead of the other two rather then not receiving its due entitlement. The scheduler responds by occasionally stopping it momentarily to allow the other two to catch up. The erroneous prediction seems to follow from a mindset where at each instant only one of the first two processes is active. But in reality their usage profiles are actually complementary, and the system quickly falls into a pattern where this is exploited.

5. CONCLUSIONS

We define fair allocations of resources based on bottlenecks, and specifically, require that each process receive at least its entitlement on at least one bottleneck resource. Previous work showed that for any combination of entitlements and requirements it is possible to find a fair allocation according to this definition. However, the proof was based on an off-line procedure that assumed full knowledge and a static configuration. We now augment this with a dynamic on-line algorithm that achieves a fair allocation without prior knowledge, and adjusts to changing conditions.

The algorithm itself is essentially a greedy algorithm. In a nutshell it can be described as follows:

1. Define a global (system-wide) priority order, and schedule processes according to this order on all resources.
2. The priority of a process is the minimum of its priorities on the different bottleneck resources.
3. The priority on each bottleneck resource is calculated based on the lag between what the process is entitled to receive and what it had actually received so far.

The main innovations in this algorithm are that it uses a global view, and that it focuses on the system bottlenecks. The prioritization of processes is system-wide: each process has a *single* priority, which is calculated based on *all* the bottlenecks, and is valid for *all* the resources. This produces coordinated scheduling decisions, thus preventing resource-specific schedulers from counteracting each other and causing priority inversions.

The algorithm provides the conceptual framework for an online bottleneck-based scheduling and allocation mechanism, that accommodates shifting usage patterns and provides allocations according to pre-defined entitlements. In future work we intend to incorporate these ideas into a working system, by using an RSVT scheduler [2] to control each resource, in conjunction with a monitoring facility that will identify the system bottlenecks. The monitoring will involve a recording of the periods during which resources are busy, and identifying those that are busy a large fraction of the time (e.g. above 90%). To reduce overheads, the calculation of priorities will be done at a fixed granularity, e.g. once a second. These priorities will then be used by all dispatch decisions until new priorities are computed.

Apart from the implementation, there is also more to be done regarding the algorithm itself. Group accounting (e.g. multiple threads in a process that share their entitlement and allocations, or asynchronous I/O leading to concurrent use of multiple resources by the same process) is a challenging and interesting issue. Further on, we would also like to investigate additional resources that might become bottlenecks, such as cache space, bus bandwidth, accelerators such as GPGPUs, and memory.

6. ACKNOWLEDGMENTS

This research was supported by the Israel Science Foundation (grant no. 28/09) and by an IBM faculty award.

7. REFERENCES

[1] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren, "*An opportunity cost approach for job assignment in a scalable computing cluster*". *IEEE Trans. Parallel & Distributed Syst.* **11(7)**, pp. 760–768, Jul 2000.

[2] T. Ben-Nun, Y. Etsion, and D. G. Feitelson, "*Design and implementation of a generic resource sharing virtual time dispatcher*". In 3rd *Ann. Haifa Experimental Syst. Conf.*, May 2010.

[3] A. Chandra, M. Adler, P. Goyal, and P. Shenoy, "*Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors*". In 4th *Symp. Operating Systems Design & Implementation*, pp. 45–58, Oct 2000.

[4] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial, "*No justified complaints: On fair sharing of multiple resources*". In 3rd *Innov. Theor. Comput. Sci.*, pp. 68–75, Jan 2012.

[5] N. Egi, A. Greenhalgh, M. Handley, G. Iannaccone, M. Manesh, L. Mathy, and S. Ratnasamy, "*Improved forwarding architecture and resource management for multi-core software routers*". In 6th *IFIP Intl. Conf. Network & Parallel Comput.*, pp. 117–124, Oct 2009.

[6] D. H. J. Epema, "*Decay-usage scheduling in multiprocessors*". *ACM Trans. Comput. Syst.* **16(4)**, pp. 367–415, Nov 1998.

[7] Y. Etsion, T. Ben-Nun, and D. G. Feitelson, "*A global scheduling framework for virtualization environments*". In 5th *Intl. Workshop System Management Techniques, Processes, and Services*, May 2009.

[8] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Process prioritization using output production: scheduling for multimedia*". *ACM Trans. Multimedia Comput., Commun. & App.* **2(4)**, pp. 318–342, Nov 2006.

[9] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "*Multi-resource fair queueing for packet processing*". In *ACM SIGCOMM Conf.*, pp. 1–12, Aug 2012.

[10] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "*Dominant resource fairness: Fair allocation of multiple resource types*". In 8th *Networked Systems Design & Implementation*, pp. 323–336, Mar 2011.

[11] F. Guim, I. Rodero, and J. Corbalan, "*The resource usage aware backfilling*". In *Job Scheduling Strategies for Parallel Processing*, E. Frachtenberg and U. Schwiegelshohn (eds.), pp. 59–79, Springer Verlag, 2009. Lect. Notes Comput. Sci. vol. 5798.

[12] A. Gutman and N. Nisan, "*Fair allocation without trade*". In 11th *Autonomous Agents & Multiagent Syst.*, Jun 2012.

[13] G. J. Henry, "*The fair share scheduler*". *AT&T Bell Labs Tech. J.* **63(8, part 2)**, pp. 1845–1857, Oct 1984.

[14] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[15] A. Mancina, D. Faggioli, G. Lipari, J. N. Herder, B. Gras, and A. S. Tanenbaum, "*Enhancing a dependable multiserver operating system with temporal protection via resource reservations*". *Real-Time Syst.* **43(2)**, pp. 177–210, Oct 2009.

[16] J. Nieh, C. Vaill, and H. Zhong, "*Virtual-Time Round Robin: An O(1) proportional share scheduler*". In *USENIX Ann. Technical Conf.*, pp. 245–259, Jun 2001.

[17] B. Radunović and J.-Y. Le Boudec, "*A unified framework for max-min and min-max fairness with applications*". *IEEE/ACM Trans. Networking* **15(5)**, pp. 1073–1083, Oct 2007.

[18] D. Raz, H. Levy, and B. Avi-Itzhak, "*A resource-allocation queueing fairness measure*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 130–141, Jun 2004.

[19] C. A. Waldspurger and W. E. Weihl, "*Lottery scheduling: Flexible proportional-share resource management*". In 1st *Symp. Operating Systems Design & Implementation*, pp. 1–11, USENIX, Nov 1994.

[20] M. E. Yaari and M. Bar-Hillel, "*On dividing justly*". *Social Choice and Welfare* **1(1)**, pp. 1–24, May 1984.

[21] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "*Redline: First class support for interactivity in commodity operating systems*". In 8th *Symp. Operating Systems Design & Implementation*, pp. 73–86, Dec 2008.

[22] L. Zhang, "*Virtual clock: A new traffic control algorithm for packet switching networks*". In *ACM SIGCOMM Conf.*, pp. 19–29, Sep 1990.

[23] Z. Zhang, L. T. X. Phan, G. Tan, S. Jain, H. Duong, B. T. Loo, and I. Lee, "*On the feasibility of dynamic rescheduling on the Intel distributed computing platform*". In 11th *Intl. Middleware Conf. (Industrial Track)*, pp. 4–10, Nov 2010.