Scheduler development update

George W. Dunlap
Citrix Systems R&D Ltd, UK
george.dunlap@eu.citrix.com

Abstract

The Credit Scheduler has served Xen well for many years now. However, recent changes in the computing field have shown some of its weaknesses. These include the advent of client hypervisors, the rise in the number of available cores, and the increasing importance of server power management. This talk will discuss challenges for the new scheduler, as well as goals and designs. We will then cover the current status of development, as well as future work and plans.

1. Introduction

The Credit Scheduler was added to the Xen tree in May of 2006. Since then, several changes have exposed weaknesses of the credit scheduler. These include the following: Client hypervisors, an increase in the number of cores, and the increased importance of power management for servers.

Client hypervisors, such as XenClient, will make workloads such as video and audio common in VMs running on Xen. Audio, and often video, do not require a great amount of CPU time; but they are very sensitive to latency. The Credit Scheduler's algorithm for handling latency-sensitive applications works tolerably well for some latency-sensitive workloads, like network. Unfortunately, for audio, its performance is unacceptable.

When the credit scheduler was written, systems with 16 cores were extreme; systems with 8 cores were still high-end. Now high-end server systems have 24 cores (4 sockets, 6 cores each), and may soon have 4 sockets, 8 cores each, and each core with 2 threads: 64 total schedulable entities. The load balancing algorithm for the credit scheduler doesn't scale well to this number of cores.

This document will discuss my work on a new general-purpose scheduler designed to replace the Credit Scheduler. Section 2 discusses design inputs: target workloads and systems, goals for the scheduler, the target interface, and some principles for what an ideal scheduler would do in common situations. Section 3 briefly describe some insights gained on a whole class of scheduling algorithms. Section 4 then describes one of the main algorithmic weaknesses of the Credit Scheduler with respect to latency-sensitive workloads. Section 5 then discusses the credit algorithm of the current working prototype. Finally, section 6 discusses the work still to be done for the credit algorithm and load balancing.

2. Design

2.1 Targets

There are three general use cases we wish to consider. The first is traditional server consolidation. We wish to tune performance for a system with 2 sockets, 6 cores per socket, 2 threads per core, giving us 24 logical CPUs. We will need to consider systems at least up to 4 sockets, 8 cores per socket, 2 threads per core, for a total of 64 logical CPUs. Servers will ideally be at around 80% utilization; but we should function well for up to 200% utilization. VMs will typically have a high number of virtual CPUs.

The second use case we want to consider is a virtual desktop server. Virtual desktop VMs will typically have only 1-2 virtual CPUs. We want to target our scheduler to work well for a server that has 8 VCPUs per physical core.

The last type of system we need to consider is a client system, such as XenClient. Such a system will typically be single socket, 2 cores, possibly with SMT. It will have 3-4 VMs including domain 0, and will be using PCI pass-through for video and audio. In addition to typical desktop workloads, audio and video will be important workloads.

2.2 Goals

There are several design goals we want to keep in mind. First is fairness: the ability of a VM to get its "fair share" of CPU. Fair share is defined by its scheduling parameters, such as weight, cap, and reservation (discussed below).

Fair share is more than simply allowing a VM to run for a given amount of time within some timeframe. For some workloads, like network data transfer, each bit of work (such as sending packets or ACKs) creates more work in the future. Not allowing a network workload to run in a timely manner thus prevents it from using its "fair share" of CPU time when it is made available.

Another goal is to work well for latency-sensitive workloads. Ideally, if a latency-sensitive workload uses less than its fair share of CPU, it should run as well when the system is loaded as it does when the system is idle. If it would use more than its fair share, then the performance should degrade as gracefully as possible.

Another aspect to consider is hyperthreads. A VCPU running on a core by itself will get a higher throughput than a VCPU sharing a core with another thread. Ideally, the scheduler should take this into account when determining the "fair share".

Finally, powering down cores or sockets into deeper sleep states can save power for relatively idle systems, while still allowing a system to perform at full capacity when needed. Our scheduler needs to either implement this power-vs-performance trade-off, or provide support for another system to do so.

2.3 Interface

The target interface we wish to implement is similar to the Credit scheduler's interface. Like the Credit Scheduler, each VM will have a *weight* and a *cap*. In addition, each VM will have a *reservation*.

In the absence of cap or reservation, each VM will get CPU time relative to other CPUs according to its weight. If some VMs are using less than their "fair share" of CPU time, the remaining time will be divided among other VMs according to their weight.

A cap will put an upper limit on how much CPU time a VM can get, even if there is more processing speed available. This is expressed as a percentage of one CPU.

A reservation is a minimum amount of CPU time a VM can get, expressed as a percentage of one CPU. A VM will get either its "fair share" according to weight, or its reservation, whichever is greater. The tools must guarantee that the sum of all reservations in the system does not exceed the amount of CPU available.

2.4 Principles

This section contains some conventional wisdom and personal observation in the form of principles in which to evaluate the design of a scheduler.

The ideal treatment for computationally intensive tasks which are not latency-sensitive is to have long time-slices. This allows them to most effectively use the processor cache. Switching more frequently can reduce cache effectiveness; but experiments have shown a "floor" such that if you switch often enough, one VCPU doesn't have enough time to flush the cache before the other one runs. In that case, the two workloads are effectively sharing the cache.

Note that this principle of giving computationally intensive tasks long time-slices is still true if multiple cores are sharing an L2 cache. If you have two cores sharing a cache, and each one has a workload running for a long times-slice, the two workloads will share the cache. But if you have two cores, each switching quickly between two workloads, then all four workloads will share the cache.

Ideal treatment for latency-sensitive tasks which are using less than their fair share is to run them as soon as possible. Low-running latency-sensitive tasks rarely use much cache.

When a latency-sensitive task with a regular wake/sleep cycle would use more than its fair share, the ideal would be to lengthen the sleep cycle long enough to make that its fair share. For example, if a latency-sensitive task runs at a 50% duty cycle (run for 1ms, sleep for 1ms), and its fair share is only 33%, the ideal thing to do would be to let it run for 1ms, sleep for 1ms, and then wait for 1ms. This should allow most latency-sensitive workloads to degrade gracefully as the amount of CPU time available to them decreases.

3. Some theory

In the process of exploring alternate algorithms, I discovered that there are a large number of algorithms that are essentially *transformations* of each other. The same basic mechanism and problems must be solved by them all.

This class of algorithms can be defined thus:

- Each schedulable entity (VCPU in this case) has a value associated with it. (In the Credit Scheduler, this is credit; in the BVT scheduler for example, this is "virtual time".)
- This value is modified as a function of time spent running. It may also be modified with time spent blocked or waiting on a runqueue, and with real time passing.

The key problem to solve with any algorithm of this type is how to deal with VMs that don't use their "fair share". If you add credit (or time or whatever) at a constant rate, and some VMs don't use all of their CPU time, the values will tend to diverge. VMs which use all of their time will get more and more "behind", and VMs which don't will get more eand more "ahead".

I explored several solutions to this problem. The first was to attempt to guess in advance how much credit a VM might use, and only give it as much as I thought it would use. Unfortunately, the only metric I had to do the estimate was how much time the VM *had used*. But there are many reason why a VM may not have run as much as it could have, including not being given enough credits to begin with.

Another set of options I explored was to change the "earn rate" and "burn rate" of credit according to how many VCPUs were currently waiting for the runqueue. Conceptually, this is very satisfying: the credit "burned" always equals the credit "earned".

Unfortunately, from a computational perspective this method is impractical. It's not uncommon for VCPUs to wake, run for only a few thousand cycles, and sleep again, only to wake a few thousand cycles later. Each time a VCPU wakes or sleeps, the rate of "earning" or "burning" credit changes, meaning that all active VCPUs need to have their credits updated according to their weight in proportion to the weight of all other VCPUs currently running. It's just not practical.

The most effective way I've found of handling divergence is with a *reset* event. A reset event, in order to be effective, must do two things. First, it must discard the extra value accumulated by those "ahead"; secondly, it must tend to converge back to zero those who are "behind". Both the BVT scheduler and the Credit Scheduler have reset events; I will describe the Credit Scheduler's reset event in the next section, and my prototype scheduler's reset event in the section following.

4. Credit Scheduler

There are many aspects of the current scheduler which make it less effective at scheduling latency-sensitive VMs. These include long time slice (30ms), sorting by priority rather than credit, and the probabilistic debiting of credit 10ms at a time. Each of these could be addressed individually. But the key issue is how it deals with the reset condition.

In order to deal with latency-sensitive workloads the Credit Scheduler classifies all VCPUs into two categories: active or non-active.

Active VMs earn credits every 30ms according to their weights, and burn credits as they run. Active VMs can be in either priority `UNDER`, meaning they have positive credit, or `OVER`, meaning negative. VCPUs in `UNDER` will always run ahead of VCPUs in priority `OVER`. Scheduling within a priority is round-robin.

Non-active VMs do not earn or consume credits as they run. As soon as a non-active VM is woken, it is set to priority `BOOST`, which allows it to run ahead of any VCPUs in `UNDER` or `OVER`.

Movement between the two classifications happens as follows. If an active VM is not using all of its fair share, it will slowly accumulate credit. Once it reaches a certain threshold (30ms worth of credit), it is marked inactive. Its credits are discarded (set to zero), and the next time it wakes it will be marked `BOOST`.

A VM is marked active again if it is interrupted by a tick. Tick interrupts in the Credit Scheduler happen every 10ms. The tick serves two purposes: to probabilistically debit credit[1], and to detect VCPUs which need to be marked active. The comment from the tick code of the Credit Scheduler makes the purpose clear: "If the VCPU is found here, then it's consuming a non-negligible amount of CPU resources and should no longer be boosted."

The problem with this is that probabilistically speaking, every VCPU, no matter how little CPU it's using, will eventually be interrupted by a tick. For example, suppose an audio VM consumes 5% of the CPU while playing MP3s. This means that each time a tick fires, there's a 5% chance that the audio VM is running. So we would expect, on average, the tick to interrupt the audio VM every 20 ticks or so. Since ticks happen every 10ms, we can expect a VM using only 5% of the CPU to stay in the `BOOST` priority for 200ms before switching to being actively accounted.

So any VM which is not using its fair share will flip back and forth between "active" and "non-active": When active, it will accumulate credit until it reaches the credit limit, and be marked inactive; when inactive, it will eventually be interrupted by a tick

[1] This has been changed in the unstable branch after 3.4; credit is now debitied on every schedule according to how long the VCPU has run.

and marked active. At the point where it is made active again, it has zero credit, so it will almost always begin by going into OVER, scheduled behind all other active VMs; even when it goes back into UNDER, it's still competing in a round-robin fashion with all VMs that haven't used up their credit yet.

5. New scheduler prototype

5.1 Runqueue per L2 cache

The first design change made with the new scheduler is to have one runqueue per L2 cache[2], rather than one runqueue per logical processor. Having separate runqueues per processor, and migrating between them, is mainly to prevent unnecessary migration, which can cause ineffective use of processor caches. Having one per processor made sense when each processor had its own large caches. However, most cores within a socket now share an L2 cache, so migrating between logical processors on the same core should have little effect on cache efficiency. (Conventional wisdom is that L1 caches are so small that there will be nothing still in the cache after even a short context switch away. They are therefore not worth considering.)

Sharing a runqueue within a socket will give near-perfect load balancing within that socket. It has also allowed me to test the new credit algorithm on single-socket boxes without having to worry about load-balancing at this point.

5.2 New credit algorithm

The new credit algorithm has the following basic properties. All VCPUs start with the same default amount of credit. Credit is burned at different rates, based on the weight; the VM with the highest weight burns credit at the default rate, and VMs with lower weight burn credit faster. When VMs wake up, they are inserted into the runqueue sorted by weight.

The reset condition happens when the next VCPU in the runqueue has non-positive credits (i.e., ¡= 0). At that point, all VCPUs are re-set to the initial default amount of credit. (This is actually a simplification; more detail is explained below.)

Resetting the credits this way controls the rate at which credit is introduced into the system. When a single CPU-intensive task is running alone, it will be given credits much more often than when it is competing for CPU with other tasks. A VMs "fair share" is determined by how long it takes to burn through its credits. If a VM is using less than its fair share, then it will always have a higher credit than those using their full share. Whenever it wakes up, it will be scheduled immediately. All VMs using their full share will naturally divide, by weight, the CPU time that is not used by VMs not using their fair share.

If a latency-sensitive VM is using more than its fair share, then it will tend to cycle through two modes. It will run whenever it wants to until its credit is low enough that it won't run immediately upon waking. Then other VMs will run for a period of time, until they have burned enough credit so that the latency-sensitive VM has enough credit to run when it wants. As long as we make this "recharge" phase short enough, it should have little impact on the performance.

6. Future work

There is still a great deal of work to be done before the new scheduler is ready for production use.

[2] I say per L2 rather than per socket because some chip designs may have multiple L2s per socket, with some number of cores sharing each.

6.1 Credit algorithm

To begin with, the credit algorithm needs some more tweaking. As mentioned above, a latency-sensitive VCPU which is using more than its fair share will alternate between run-on-demand and a "recharge" period. More work is needed to determine the best way to implement this "recharge" period.

Another issue comes when a single VM has more than one workload, one of which is CPU-bound, and another of which is latency-sensitive. For instance, a XenClient user might be listening to music in his personal VM, while unbeknownst to him a flash advertisement running in Firefox is consuming 100% of the CPU. The challenge is to allow the latency-sensitive portion of the workload to run in a timely manner, while not allowing the VM to use more than its fair share.

The key observation is that latency-sensitive components of workloads, such as audio and network packet handling, is generally handled by the operating system in interrupt handlers. If a VM which has an outstanding interrupt is allowed to run, it will probably handle the latency-sensitive portion of its workload first. So an idea I plan on testing is to give VMs a 1ms boost when the VM has an outstanding interrupt.

Another issue is the following. As we said in section 2.4, CPU-bound VMs should be allowed to run for long time-slices, if possible, to maximize cache efficiency. If two CPU-bound processes are alternating, this is fairly easy: just give each a decently long time-slice.

However, if there is a third VM which runs for short periods of time but very frequently, it will effectively shorten the timeslice of the other two, so that they alternate much faster than they would otherwise. This is because the long-running VCPUs' credits are updated and they are sorted in the runqueue every time the third VM wakes up, rather than only after the full scheduling quantum has run.

A similar problem happens when a VM makes several short calls out to QEMU. Instead of running for its full quanta, it runs for a short time, and does a quick (10k cycle) yield to QEMU (either in domain 0 or in a stub domain). Since it has run, its credit is debited and it is inserted in the queue, possibly behind another VM. It would be ideal to allow the same VM to continue running, to allow it to more effectively use the cache.

Xia Yubin introduced a technique called *preempt-back*, in which a VM which was preempted to do a short task is allowed to run again immediately, instead of being put back in the runqueue in its place. Doing this effectively will require some more research and experimentation.

Finally, caps and reservations need to be implemented.

6.2 Load balancing

Having a single runqueue per L2 makes lower-level load balancing much simpler. Most single-socket systems will not need to load balance at all. What remains is to load balance across sockets. To do this I plan on borrowing ideas from Linux load-balancing.

The Linux scheduler divides is logical CPUs hierarchically into *domains*. Domains have different levels: e.g., there may be multiple threads in a core, multiple cores in a socket, and multiple sockets in a NUMA node. At each domain level, a task periodically balances work between those levels. Balancing at lower levels is done more frequently than at higher levels. To do load balancing, each runqueue keeps track of a measure of *CPU load* for different lengths of time in the past; this information is aggregated up to the level at which balancing is being done.

I plan on implementing something similar for Xen, but starting at the L2 runqueue level. As always, measurement and experimentation will be needed.