Technical Disclosure Commons

Defensive Publications Series

October 2020

EFFICIENT UTILIZATION OF BARE METAL CORES WITH DYNAMIC MONITORING AND CALIBRATION

Chase Wolfinger

Bhavik Adhvaryu

Matt Seashore

Kashyap Kodanda Ram Kambhatla

Ian Wells

EFFICIENT UTILIZATION OF BARE METAL CORES WITH DYNAMIC MONITORING AND CALIBRATION

AUTHORS:
Chase Wolfinger
Bhavik Adhvaryu
Matt Seashore
Kashyap Kodanda Ram Kambhatla
Ian Wells

ABSTRACT

In existing cloud environments it is not possible to mix, on the same server at the same time, workloads that use part of a processor core, or that use cores on a best-effort basis, with workloads that must both be assigned to a single core and have that core dedicated to their use (i.e., nothing else runs on the core). To address these challenges and inefficiencies, techniques are presented herein that support a division of resources in a way that they can then be appropriately assigned to workloads. One logical pool of cores may be assigned for workloads requiring shared resources and another pool may be assigned for workloads requiring dedicated resources. The boundary between those pools may shift dynamically as, for example, additional resources are required.

DETAILED DESCRIPTION

Cloud performance is key in a number of areas, not the least of which is network functions virtualization (NFV), and within NFV perhaps more than in other circumstances there is a mix of control and data plane applications with differing performance needs. In order to satisfy those needs central processing units (CPUs) are used primarily in one of two different ways. In the first way, performance may be achieved by dedicating an entire processor core to a single task (avoiding unnecessary waste arising from context switching and competition between workloads). In the second way, a collection of cores may be shared between tasks using standard pre-emptive multitasking of the workloads, meaning that more tasks may be run but with fewer guarantees of performance consistency and some waste due to the sharing. This model is seen in, for example, OpenStack and Kubernetes [1].

However, in both cases the processor core allocation is enforced through a coarse-grained mechanism. Either a server is dedicated to single core workloads or it is shared [1]. In Kubernetes, some set of cores may be allocated at deployment time to running dedicated workloads, with the remaining cores shared among the balance of the workloads. Some cores are additionally assigned to platform management functions [1].

In existing cloud environments it is not possible to mix, on the same server at the same time, workloads that use part of a core, or that use cores on a best-effort basis, with workloads that must (a) be assigned to a single core and (b) have that core dedicated to their use (i.e., nothing else runs on the core).

To address these challenges and inefficiencies, techniques are presented herein that support a division of resources in two ways that may then be appropriately assigned to workloads. One logical pool of cores may be assigned to workloads requiring shared resources and another pool may be assigned to dedicated resources. The boundary between those pools may shift dynamically as, for example, additional resources are required. Host selection for a workload is performed based on available resources (as it is now done), but with a different awareness of the resources available.

Under the techniques that are presented herein, initially, the shared pool contains all of the available cores and the dedicated pool contains no cores. As well, when a core becomes free it is returned to the shared pool.

A workload may specify that it requires a certain number of cores. Using one of the following categories either all of the needed cores may be allocated in a single fashion or each core may be allocated independently:

Category 1. To be run as best-effort, in which case the workload is allocated into the shared resource pool.

Category 2. To be run on shared cores with a fixed minimum resource requirement that may be expressed as some percentage of core CPU time, in which case the workload is also allocated into the shared resource pool but with a confirmation that sufficient shared resources are available to satisfy that minimum requirement.

Category 3. To be run using dedicated cores, in which case cores are withdrawn from the shared pool, moved to the dedicated pool, and then allocated to the workload.

Each of the three approaches that were identified above are described below.

When a category 1 workload is placed on the host, it is assigned to the shared pool provided there are more resources in the shared pool in total than the current workload requires. If there are insufficient resources the host is deemed unsuitable and excluded from placement. The guarantee offered is that there is more than 0 CPUs available but with no guarantee of the specific number of available CPUs.

When a category 2 workload is placed on the host, it is also assigned to the shared pool and the host scheduler is configured to ensure that it gets an allotment of CPU resources at least as great as the request specifies. If the mandated resource requirement of all workloads including the new one exceeds the available CPU in the shared pool the host will again be declared unsuitable and ruled out at placement time.

When a category 3 workload is placed on the host, sufficient cores for it will be withdrawn from the shared pool, providing the shared pool is not reduced to the point that it cannot fulfill its category 1 and 2 workloads' needs (which again would make it unsuitable for placement). The withdrawn cores are moved to the dedicated pool and then allocated to the workload.

Aspects of the techniques that are presented herein may be summarized as supporting a straightforward implementation implying that you start with all cores running best-effort workloads, but if at any point you wish to pin a workload to a core you remove a core from the best-effort pool – providing you have enough cores remaining to satisfy any constraints. Thus, one could say for the unpinned workloads 'provided I am delivering at least X cores to the workload and the sum of the requirements for all unpinned workloads is X cores, I could withdraw cores from the best-effort pool and use them for pinned workloads. If I could not satisfy that constraint I can't run a pinned workload here, and if at any point I deliver 0 cores to the best-effort workloads, such that none will run, that is also a failure.'

Aspects of the techniques that are presented herein may be explicated with reference to the high-level process flow diagram that is presented in Figure 1, below.

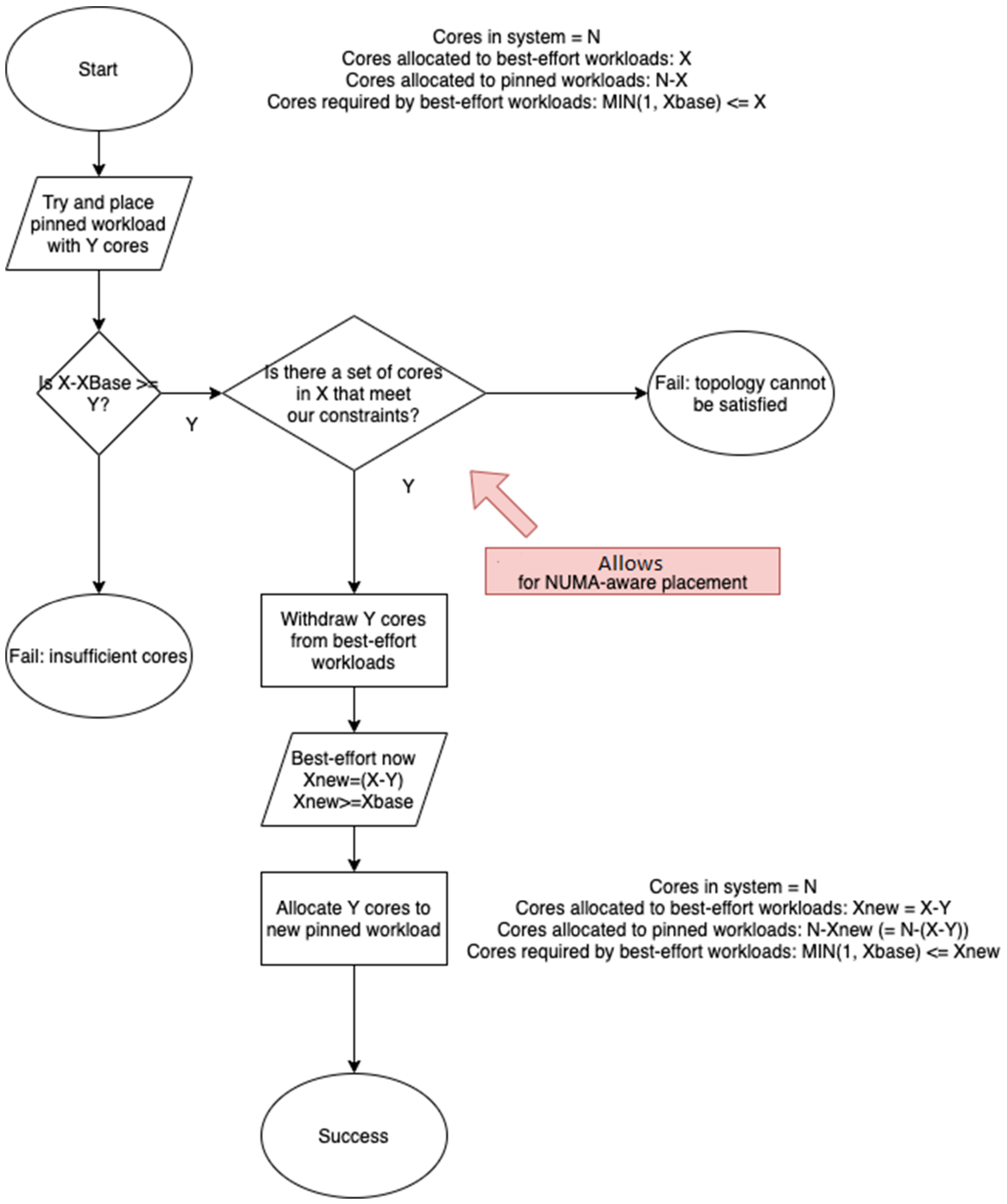
Start
Cores in system = N
Cores allocated to best-effort workloads: X
Cores allocated to pinned workloads: N-X
Cores required by best-effort workloads: MIN(1, Xbase) <= X
Try and place pinned workload with Y cores
Is X-XBase >= Y?
Y
Is there a set of cores in X that meet our constraints?
Fail: topology cannot be satisfied
Y
Allows for NUMA-aware placement
Fail: insufficient cores
Withdraw Y cores from best-effort workloads
Best-effort now Xnew=(X-Y) Xnew>=Xbase
Allocate Y cores to new pinned workload
Cores in system = N
Cores allocated to best-effort workloads: Xnew = X-Y
Cores allocated to pinned workloads: N-Xnew (= N-(X-Y))
Cores required by best-effort workloads: MIN(1, Xbase) <= Xnew
Success

Figure 1: High-level Process Flow Diagram

To help illustrate aspects of the techniques presented herein, consider the following hypothetical example along with the high-level process flow diagram that was presented in Figure 1.

In the hypothetical example eight cores are needed for a pinned workload, there are more than eight cores allocated to best-effort workloads, and there are no non-uniform memory access (NUMA) topology requirements.

The best-effort workloads are set up to consume the cores in the best-effort pool. This implies that all threads in those workloads can run on any of the cores in the pool, and that they will float between them as they need CPU time (the operating system (OS) scheduler taking care of that).

Having established there are sufficient cores to run the pinned workload and not starve the best-effort workloads, one locates eight cores in the best-effort set and withdraws them from that set. In withdrawing them, one re-pins the best-effort workloads and they now utilize the smaller remaining set of cores in the best-effort pool. Note that it was already established that the smaller remaining set of cores is sufficient for their needs.

The idle withdrawn cores are now tied to the pinned workload. For a pinned virtual machine (VM) workload that would mean that each virtual CPU (vCPU) is pinned to a specific physical core out of the set. For a container workload, one would grant the cores to the new container workload and let it decide as to how it would want to pin its threads to cores. In either case, the new workload has the full attention of all of the cores and receives a fixed and known quantity of CPU time without any interruptions from other workloads pre-empting the CPU at unpredictable intervals and for unknown time periods. This is generally required in scenarios like high performance NFV workloads to avoid input buffer overflow. (Note that if one is sharing a core then a thread might stop and the OS might run one of the other workloads, so a thread could be paused for long enough for an input queue to overflow.)

One particular way in which aspects of the techniques that are presented herein might be implemented would involve the creation of a 'shared' control group (cgroup) on an underlying Linux kernel whose CPU membership is updated with the list of cores currently in the shared pool each time the pool size changes, ensuring that such workloads will never find their way onto dedicated cores. Allocations within the shared pool might have suballocations (e.g., a percentage of CPU) within such a cgroup, implemented as child cgroups. Dedicated cores may be placed onto dedicated cores using either cgroups or CPU pinning, but importantly the dedicated cores are isolated from shared-core workloads.

Since one knows that a dedicated core is in the dedicated group for the lifetime of its workload, therefore its core can never be taken away during that time, and hence there is never a question of having to move or rebalance the dedicated cores due to a removal of the core from the dedicated pool.

A management workload may be considered as either a shared or a dedicated resource. One possible simplification might be to have the management workload allocated to a core at the outset which never finds its way into the shared pool. Thus, one knows that the management workload is there for the life of the machine, as this allocation is done on initialization when no workloads are present.

Employing aspects of the techniques that are presented herein a host may run both shared and dedicated workloads, the workloads may be allocated to a known slice of the machine, and shared workloads will not interfere with dedicated workloads.

Aspects of the techniques that have been presented herein provide a range of advantages, including, for example:

1. A division of the set of cores in the CPU into shared and dedicated pools.
2. Use of the shared pool for workloads that can share cores.
3. Use of the dedicated pool for dedicated cores.
4. Adapting the pool size dependent on demand.
5. Considering whether a compute host in the cloud (whether VM, or container, or other) is suitable for placement as a workload based on whether its allocation could be adjusted to fit the workload.

In summary, in existing cloud environments and on the same server at the same time, it is not possible to mix, workloads that use part of a processor core, or that use cores on a best-effort basis, with workloads that must both be assigned to a single core and have that core dedicated to their use (i.e., nothing else runs on the core). To address these challenges and inefficiencies, techniques have been presented that support a division of resources in a way that they can then be appropriately assigned to workloads. One logical pool of cores may be assigned for workloads requiring shared resources and another pool may be assigned for workloads requiring dedicated resources. The boundary between those pools may shift dynamically as, for example, additional resources are required.

References:

[1] S. Balaji, and C. Doyle, “Feature Highlight: CPU Manager”, July , 2018, “https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/”