# TopologyManager Feedback (Round 2)

**PUBLIC DOCUMENT**

15-July-2019
Kevin Klues (kklues@nvidia.com)

Link to **`TopologyManager`** Feedback (Round 1):
https://docs.google.com/document/d/1UbD1igeBA1sRuqwkQgJ5sJ9sf1rao_LLORWLhwSdGUo/edit#

Concrete Proposals:

- Proposal for returning a **`map`** from **`GetTopologyHints()`** instead of a slice:
  https://github.com/kubernetes/enhancements/pull/1131
  https://github.com/kubernetes/kubernetes/pull/80569

- Proposal to add **`GetPreferredAllocations()`** as optional device plugin API:
  https://github.com/kubernetes/enhancements/pull/1121

- Proposal to rename **`preferred`** policy to **`best-effort`**
  ~~https://github.com/kubernetes/kubernetes/pull/80301~~
  https://github.com/kubernetes/kubernetes/pull/80683

- Proposal to delay adding **`strict`** policy until after the initial **`alpha`** release
  (no formal proposal yet -- let's talk about this)

Bullet points 1 and 2 are internal API changes and can wait until after the initial **alpha** release to be decided on. Bullet points 3 and 4, I would consider blockers for the initial **alpha** release.

## Proposal for **`map`** from **`GetTopologyHints()`**:

At present, there is no way for a hint provider to return distinct hints for different resource types via a call to **`GetTopologyHints()`**. This means that hint providers that govern multiple resource types (e.g. the **`devicemanager`**) must do some sort of "pre-merge" on the hints it generates for each resource type before passing them back to the **`TopologyManager`**.

The logic behind this original design was to give components like the **`devicemanager`** the opportunity to perform these "pre-merges", in order to satisfy internal constraints amongst devices before bubbling their **`TopologyHints`** up to the **`TopologyManager`**.

The idea was that this "pre-merge" step could be used to resolve things like internal NVLINK dependencies amongst Nvidia GPUs, NIC to GPU alignment on the PCIe bus, etc., without exposing this information to the **`TopologyManager`** or other **`HintProviders`** via the shared **`TopologyHint`** abstraction**.** There were no concrete plans on how to actually perform resolutions like this (at least not in a generic way), but the plan was to keep the design open to such possibilities in the future.

With the introduction of the proposed **`GetPreferredAllocations()`** call outlined in the next section, performing "pre-merges" such as these should no longer be necessary (please see that section for details). As such, I think it is worth reconsidering what it means to be a **`HintProvider`**, and what the return value of the **`GetTopologyHints()`** should look like.

In an ideal world, I would expect every unique resource type to be its own **`HintProvider`**. This is already true for the **`CPUManager`** (and soon to be true for **`Memory`**), but because of the pluggable nature of the **`devicemanager`**, it is not possible for custom resources like **`nvidia.com/gpu`** or **`intel.com/sriov`** to be their own **`HintProvider`**.

This means that there is *no direct* merging of the affinities associated with resources such as **`nvidia.com/gpu`** or **`intel.com/sriov`** by the **`TopologyManager`.** Instead, the **`devicemanager`** performs its own custom “pre-merge” logic for these devices, passing a single set of hints to the **`TopologyManager`** to represent the combined affinities of both resource types.

This seems counter-intuitive, since there is no practical reason that a "pre-merge" should be necessary in this case -- it just happens to be necessary because of the way the current interface is designed. Furthermore, it adds unnecessary complexity to the system, and makes it harder to reason about how hints from different resource types are merged, because they are being merged at multiple levels.

To compensate for this, I propose the following change to the **`GetTopologyHints()`** interface to allow each resource type to pass back its own set of **`TopologyHints`**:

```
-  GetTopologyHints(pod v1.Pod, containerName string) []TopologyHint
+  GetTopologyHints(pod v1.Pod, containerName string) map[string][]TopologyHint
```

Where the returned **`map`** is indexed by the resource type instead of passing back a flat slice.

Extending the interface in this way, gives components like the **`devicemanager`** the ability to provide hints for each resource type as if they were their own **`HintProvider`**. That is to say, it allows the **`devicemanager`** to pass the responsibility of merging hints from all resource types into a single central location -- the **`TopologyManager`** -- instead of requiring a pre-merge step to be completed inside the **`devicemanager`** itself.

With this in place, **`HintProviders`** like the **`CPUManager`** and **`Memory`** can mostly go unchanged, except that they will now pass back a **`map`** with a single element indexed by "**`cpu`**" and "**`memory`**" respectively.

As a bonus, this change also allows the **`TopologyManager`** to recognize which resource type a set of hints originated from, should this information become useful for policies in the future.

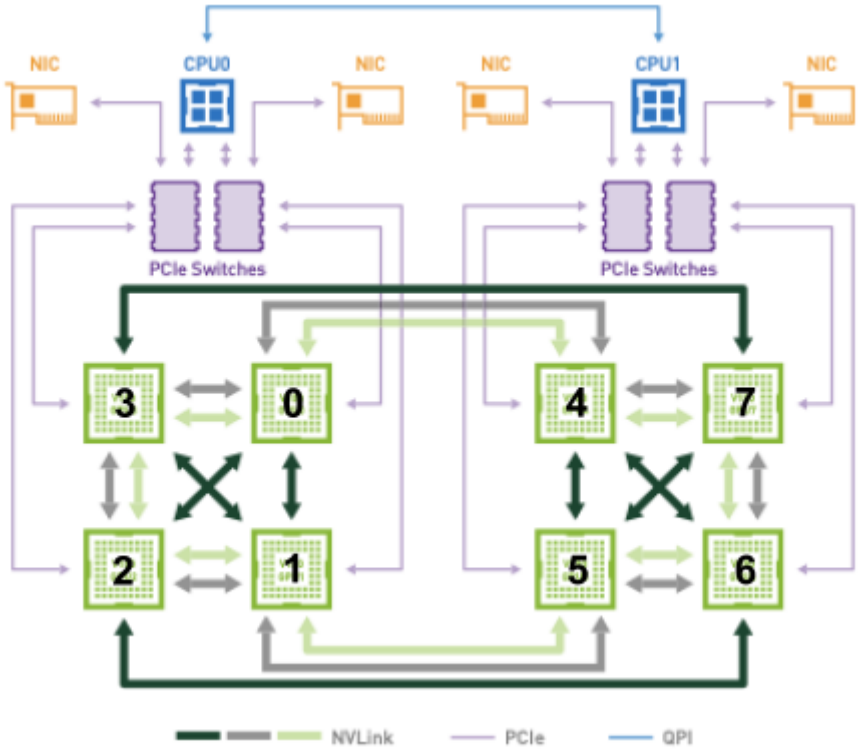
## Proposal for **`GetPreferredAllocations()`**:

This proposal allows a device plugin to forward lists of preferred allocations to the **`devicemanager`** so it can incorporate this information into its **`TopologyHint`** generation, as well as help influence its final allocation decision once all **`TopologyHint`**s have been merged.

One way to look at this proposal, is to think of it as a way of generating *intra-device* allocation preferences from each plugin without having to expose any device specific topology information (e.g. NVLINK topologies) to the kubelet.

In this way, the **`TopologyManager`** can be restricted to only deal with *inter-device* topology constraints (e.g. NUMA node, PCIe bus, etc.), while still having a way of incorporating device-specific topology constraints into its allocation decisions.

As a concrete example, consider the following DGX-1 node below:




The DGX-1 is a Deep Learning AI server from Nvidia that has 8 GPUs, with 4 GPUs per NUMA node and a set of high-bandwidth connections between pairs of GPUs called NVLINKs. The more NVLINKs that exist between a pair of GPUs, the larger the communication bandwidth, and thus the better the performance. As such, the number of NVLINKs that exist between a set of GPUs imposes a GPU specific topology constraint that must be considered when allocating GPUs to a container for high performance.

As a concrete example, a request for 2 GPUs on the DGX-1 node above would better be served by the pair (0,3) than the pair (0,1) because two NVLINKs exist between (0,3) and only one NVLINK exists between (0,1). Without a mechanism to consider these NVLINK constraints in some way, the **`TopologyManager`** (and ultimately the **`devicemanager`**) will end up weighing these two pairs equally since they can both be allocated from the same NUMA node.

Using **`GetPreferredAllocations()`** gives device plugins like the Nvidia GPU plugin, a mechanism to influence allocation decisions made by the **`devicemanager`** to compensate for situations like these, without exposing any internal details about why a plugin prefers one pairing of devices over another.

Continuing with the example above, if all GPUs on the DGX-1 node were currently available, a call to **`GetPreferredAllocations()`** with a request size of 2 would pass back the following (i.e. a list of all available GPU pairs that have 2 NVLINKs between them):

```
{(0, 3), (0, 4), (1, 2), (1, 5), (2, 3), (2, 6), (3, 7), (4, 7), (5, 6), (6, 7)}
```

The **`devicemanager`** would then *prefer* allocations from this list both when generating hints to pass to the **`TopologyManager`**, as well as when doing its final allocation after all **`TopologyManager`** constraints have also been considered.

The following subsections go into more detail about how this would be accomplished.

## Integration into hint generation:

1) Explicitly define the default strategy for setting the preferred field of a **`TopologyHint`**. By default, a hint provider will set **`preferred = true`** on hints with the smallest **`SocketAffinity`** possible, and **`preferred = false`** on all others.

2) Enumerate all of the available devices managed by a **`HintProvider`** and figure out what NUMA affinity the current request could be allocated with. Generate a unique hint per unique NUMA affinity, following the default strategy for setting the preferred field of a **`TopologyHint`**, as outlined above.

3) If the **`devicemanager`** has a plugin that implements **`GetPreferredAllocations()`**, reset the value of the **`preferred`** field for each hint as follows. First, walk through all hints and set **`preferred = false`**. Next walk though each preferred allocation returned from **`GetPreferredAllocations()`** and calculate the NUMA affinity for that allocation. Walk back through each hint, setting **`preferred = true`** on hints with an affinity mask matching that of a preferred allocation.

## Integration into allocation strategy:

If a plugin does not implement **`GetPreferredAllocations()`**, then we should simply follow the device allocation strategy that exists today (i.e. allocate devices directly from the available devices list). If however, **`GetPreferredAllocations()`** is implemented, then one of the preferred allocations should be chosen over simply pulling devices at random from the available devices list.

There are 4 cases to consider:

- **`TopologyManager`** disabled, **`GetPreferredAllocations()`** not implemented
- **`TopologyManager`** enabled, **`GetPreferredAllocations()`** not implemented
- **`TopologyManager`** disabled, **`GetPreferredAllocations()`** implemented

- **`TopologyManager`** enabled, **`GetPreferredAllocations()`** implemented

With the **`TopologyManager`** disabled and **`GetPreferredAllocations()`** unimplemented, the existing strategy is to simply pull devices from the front of the available devices list -- this should go unchanged.

With the **`TopologyManager`** enabled and **`GetPreferredAllocations()`** unimplemented, the strategy is to pull devices from the available devices list, such that they have the desired NUMA affinity -- this should also go unchanged.

With the **`TopologyManager`** disabled and **`GetPreferredAllocations()`** implemented, the new strategy should be to prefer allocations from the list returned by **`GetPreferredAllocations()`** if possible, and fall back to pulling devices from the front of the available devices list if not.

With the **`TopologyManager`** enabled and **`GetPreferredAllocations()`** implemented, the new strategy should be to prefer allocations from the list returned by **`GetPreferredAllocations()`** such that they have the desired NUMA affinity, then fall back to pulling devices at random from the available devices list, such that they have the desired NUMA affinity.

# Proposal to rename **`preferred`** to **`best-effort`**

At present, there are two proposed **`TopologyManager`** policies: **`preferred`** and **`strict`**.

The **`TopologyManager`** KEP states that:

> *The **`TopologyManager`** supports two modes: **`strict`** and **`preferred`** (default). In **`strict`** mode, the pod is rejected if alignment cannot be satisfied.*

However, the exact semantics of how each of these policies should behave are a bit fuzzy.

After reading the rest of the KEP (as well as looking through the code where this policy is implemented) it is clear that the "**`preferred`**" policy actually implements more of a "**`best-effort`**" policy, where it attempts to align device allocations from multiple **`HintProviders`** on *at least one* NUMA node (but does not *guarantee* alignment across multiple NUMA nodes if that is what some of the **`HintProviders`** require).

The algorithm for this "**`best-effort`**" policy goes as follows:

1) Take the cross-product of hints generated by each **`HintProvider`**

2) For each set of hints in the cross-product, take the bitwise-and of the **`SocketAffinity`** for each hint to produce a merged **`SocketAffinity`**.

3) For each set of hints in the cross-product, inspect the **`preferred`** field of each hint.

    a) If all hints have **`preferred == true`**, then set **`preferred = true`** in the merged hint.
    b) If even one hint has **`preferred == false`**, then set **`preferred = false`** in the merged hint.

4) Select the "**`best-hint`**" from the set of all merged hints with the following criteria:

    a) Start with a default hint of **`SocketAffinity = nil`**, **`preferred = false`**
    b) Prefer hints with **`preferred == true`**
    **c)** Prefer hints that have at least 1 bit set in the **`SocketAffinity`**
    d) Prefer hints with *less* bits set in the **`SocketAffinity`**
    e) Prefer hints with lower-ordered bits set in the **`SocketAffinity`**

5) Always admit the pod, regardless of what the merged hint result is

This strategy makes sense under many scenarios, and is obviously a useful policy to implement.

However, calling it **`preferred`** instead of **`best-effort`** seems like it will lead to much confusion since "**`preferred`**" doesn't actually convey any information about what the policy is doing under the hood, whereas **`best-effort`** does.

# Proposal to delay **`strict`** policy in **`alpha`** release

Likewise, the semantics around the behaviour of the **`strict`** policy are even less well defined.

At present, the **`strict`** policy is a simple extension of the **`preferred`** policy. I.e. its algorithm goes as follows:

1) Calculate the merged hint from the "**`best-effort`**" / "**`preferred`**" policy
2) If **`preferred == true`**, admit the pod, if **`preferred == false`**, don't

This is a very straightforward extension of the "**`best-effort`**" / "**`preferred`**" policy, but is it the right one? I'm not so sure.

In fact, I'd argue that the current semantics of "**`strict`**" are actually quite misleading. They encode a sort of "**`strict`**" equality across all **`preferred`** fields, but then allow a "**`best-effort`**" affinity across the **`SocketAffinity`** fields. Sort of a *half-strict* interpretation.

Would it be better to interpret "**`strict`**" as a policy that enforces a "**`strict`**" equality across both the **`preferred`** field *and* the **`SocketAffinity`** field (and, by extension, any other fields we introduce in the future)?

Such a policy would be useful, for example, to *guarantee* alignment of devices on multiple NUMA nodes (something that is not possible with the current **`best-effort`** policy). This strategy seems much more logical and easy to interpret than the current implementation.

Even this has its complications though. For example, consider the following scenario:

```
+------------+-----------------+-----------------+
| Providers  | Hints           | Proposed Strict |
+------------+-----------------+-----------------+
| Nvidia GPU | ((1, 1), True)  | Merged Hint:    |
|            |                 | ((1, 1), True)  |
| CPUManager | ((0, 1), True)  |                 |
|            | ((1, 0), True)  | Admit Pod: True |
|            | ((1, 1), True)  |                 |
+------------+-----------------+-----------------+
```

Such a scenario could arise if 3 GPUs are being requested on a 4 GPU system with 2 GPUs per NUMA node. The *only* possible way to allocate GPUs on this node is across 2 different NUMA nodes. There will never be a situation where there will be single NUMA node alignment.

As such, it's clear that this means that GPUs will be allocated from both NUMA nodes if he pod is admitted. However, does this mean that the CPUmanager ***must*** also allocate CPUs from both NUMA nodes? If so, in what distribution should it allocate them? If not, then how is this different than the "**`best-effort`**" policy? What if only a single CPU is requested?

Given all of this ambiguity, it feels premature to introduce a "**`strict`**" policy at the node level until we've talked through these details a bit more and figured out their exact semantics.

Moreover, I'm not convinced we would ever want to set such a "**`strict`**" policy at the node level. It seems too restrictive for common use-cases. Additionally, it adds unforeseen complications by exposing users, job controllers, and the scheduler to pod rejections that were previously not possible.

Instead, it seems like "**`strict`**" topology-aware allocation should be more of a pod-level decision that we use to *override* the lower-level "**`best-effort`**" policy when desired. This, however, opens up a whole other discussion that is out of scope of the current design.

As a side note, I've implemented a small python program that implements all of the different policies discussed here and compares their results for every combination of hints from 2 **`HintProviders`** on a 2 socket system.

Example output below:

```
+------------+-----------------+-----------------+------------------+------------------+
| Providers  | Hints           | Best Effort     | Current Strict   | Proposed Strict  |
+------------+-----------------+-----------------+------------------+------------------+
| Provider 0 | ((0, 1), True)  | Merged Hint:    | Merged Hint:     | Merged Hint:     |
|            |                 | ((0, 0), True)  | ((0, 0), True)   | ((0, 0), True)   |
| Provider 1 | ((1, 0), True)  |                 |                  |                  |
|            |                 | Admit Pod: True | Admit Pod: False | Admit Pod: False |
|            |                 |                 |                  |                  |
+------------+-----------------+-----------------+------------------+------------------+
| Provider 0 | ((1, 1), True)  | Merged Hint:    | Merged Hint:     | Merged Hint:     |
|            |                 | ((0, 1), True)  | ((0, 1), True)   | ((1, 1), False)  |
| Provider 1 | ((0, 1), True)  |                 |                  |                  |
|            | ((1, 1), False) | Admit Pod: True | Admit Pod: True  | Admit Pod: False |
|            |                 |                 |                  |                  |
+------------+-----------------+-----------------+------------------+------------------+

...
```