Advanced Topics in Configuration Management

Ryan Bak and Randy Kleinman
Hewlett Packard Enterprise
Bloomington, MN USA
ryan.bak@hpe.com, randy.kleinman@hpe.com

Abstract - **For the configuration of the latest generation of Cray supercomputers, the Configuration Framework Service (CFS) is a flexible framework used to prepare both images and booted nodes to meet their functional requirements. To help users get the most out of CFS, this paper will explore many advanced topics, such as the different modes of operation for CFS, configuration of both compute and non-compute nodes, how to configure CFS for best performance, and how to write the Ansible code for fast and efficient deployment of your configuration, as well as the differences between CFS and the previous generation Cray XC series system configuration management.**

I. Managing CFS in v1.3

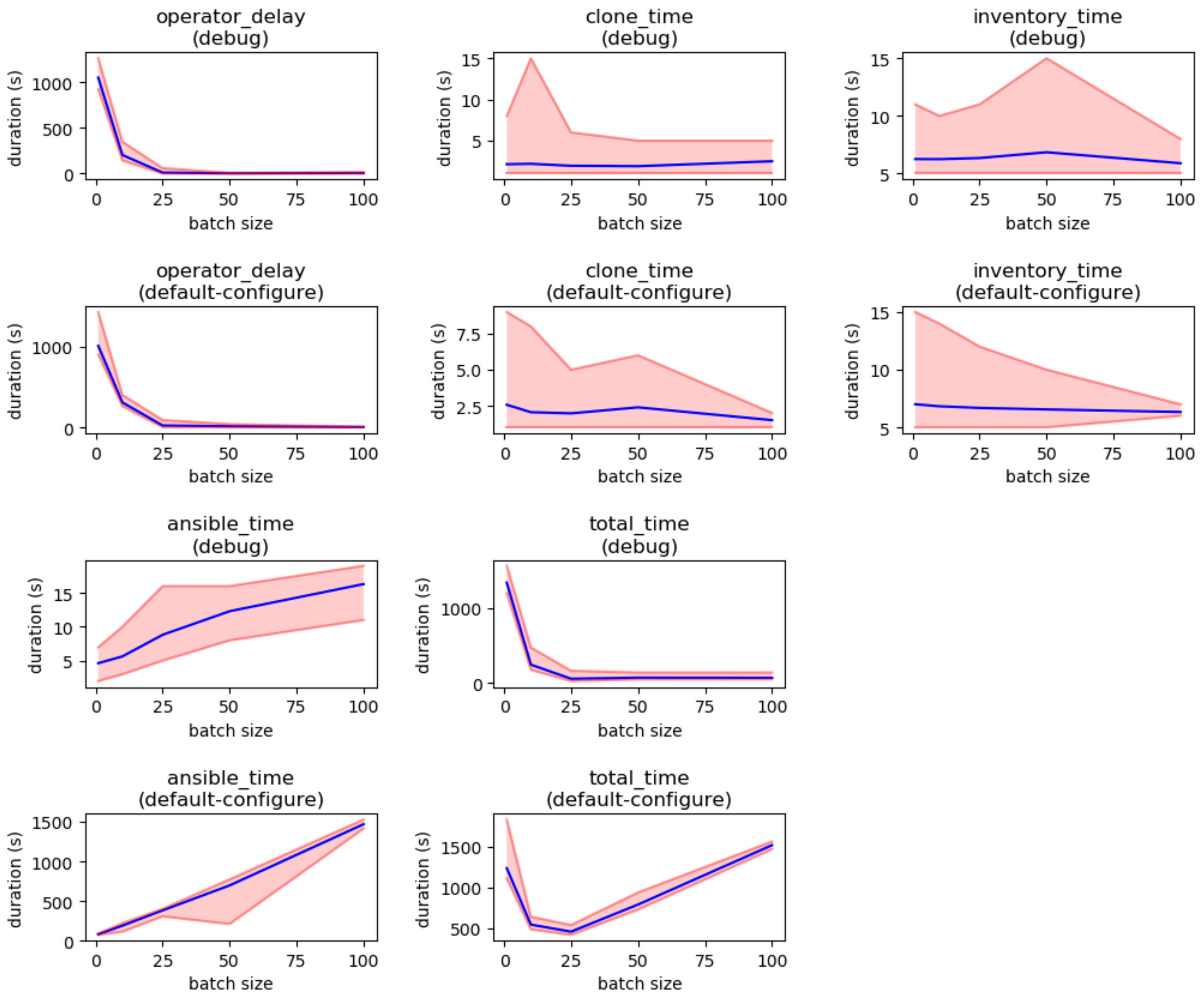
At the core of the Configuration Framework Service is Ansible, which CFS uses to apply configurations to its targets. Ansible provides users a familiar, open-source solution in which to write their configuration. CFS builds on top of Ansible's strengths, adding additional features to both add simplicity and power to the use-case of deploying configurations to large scale systems. This paper discusses the new CFS-Batcher feature which improves overall post-boot configuration time, why you would choose to have configuration applied as pre-boot image customization versus post-boot node personalization, how to manage different groupings of nodes, compares CFS to configuration solutions from previous Cray supercomputers, and describes some future areas of configuration improvement.

II. CFS-Batcher

As of the 1.3 release, the most recent addition to CFS's features is the CFS-Batcher. This is a new service running in Kubernetes that serves two purposes: It breaks up configuration sessions into smaller Ansible runs, and also supports a partially declarative model, adding the ability to automatically configure components when they start, and retrying failures on a component level so that individual components don't prevent the rest from configuring. While users can still create individual CFS sessions, which are still useful when targeting small numbers of components in a one-off case such as configuring images, the preferred method of using CFS is now to use the new /components endpoint. Here users will find information on all nodes in their systems, along with information about what configuration has been applied to them and what the desired configuration state is. By setting the desired configuration state for a component, users will not only apply that configuration to the component now, but also anytime in the future that the component reboots.

The normal workflow starts by creating a session through the Boot Orchestration Service (BOS). If CFS is enabled in the BOS session template, BOS will set the desired configuration state for all targeted components. In the case of a reboot or boot operation, the state is set prior to issuing the power command, and the configuration is set along with a flag in CFS that temporary disables automatic configuration so that CFS doesn't attempt to configure the nodes before they are powered up. When the node comes up, the CFS-State-Reporter, a package installed in Cray-provided boot images (e.g. Compute and User Access Nodes), checks in with CFS and alerts it that the component is both available for configuration, and has no configuration currently applied. CFS-Batcher monitors the configuration state of all nodes, and when it detects nodes in a state where the desired and current configurations do not match, it starts the process of reconfiguring it. The node is put into a batch, which then becomes a CFS session once the batch is full, or a maximum wait time has expired. Because the configuration on each component has already been set prior to any power commands, there is no need for all components to be up prior to starting configuration. As components come up and report in, CFS-Batcher will immediately start the configuration process for these nodes, reducing overall wait time. The "thundering herd" problem is also avoided because all CFS sessions do not start at exactly the same time. At the end of the CFS run, an Ansible callback module then reports back the success or failure of the configuration for each component and retries are started if necessary.

The CFS-Batcher is configurable. Everything from the batch size to the number of retries can be set by the user via the /options endpoint. By default, these parameters have been tuned to best balance Ansible performance and overall performance for our default playbook. The graphs below show how Ansible performs better when you are targeting fewer nodes at a time. Across every playbook tested, the time it takes to complete an Ansible run scales linearly with the number of nodes being configured. We set the default at 25 nodes per batch, because although it is possible to get better performance in the Ansible phase of the CFS session by using smaller batch size, the overall time is increased. Testing showed that a batch size of 25 nodes was the fastest overall once the competing calls to HSM for inventory, competing calls to our Version Control Service (VCS) to clone the Ansible content, and the extra



amount of time needed for the operator when scheduling more CFS jobs at the same time are accounted for.

Two tests are shown here. The first is running a simple playbook with only a debug task, the second is the site.yml playbook provided with v1.3. Both tests were run with varying batch sizes (x-axis) on a 1024 node system running Shasta v1.3 software and show the time for each phase in seconds (y-axis). Blue is average, and the shaded red area shows minimum and maximum range for the results. The phases shown are:

1. operator_delay – The time from the creation of the CFS session until the CFS session job starts.
2. clone_time – The time taken for the git-clone container to clone the Ansible content.
3. inventory_time – The time taken for the inventory container to build the Ansible inventory
4. ansible_time – The time for the Ansible container to run once the Ansible content and inventory are in place.
5. total_time – The total time from BOS session creation to CFS session completion.

The combination of two phases define the overall time CFS will take: the operator_delay and ansible_time. The operator_delay, which is the time is takes for the CFS-Operator to schedule jobs for all batches, can be high with large numbers of batches, such as when using small batch sizes. On the other hand, the ansible_time, the time it takes Ansible to run, increases with batch size. The result is a total_time graph with a minimum at some batch size. The ideal batch size occurs at this location, where both the overhead of scheduling more sessions and the time it takes Ansible to run are minimized.

III. Adding Custom Content to CFS

The Ansible content that CFS runs is managed in the Version Control Service (VCS), which uses the open source self-hosted Git service Gitea (https://gitea.io). This makes it easy for users to add or modify content and track any changes made. However, although adding and modifying content is easy, it's much more difficult to write Ansible code that is efficient at scale. This is especially true if there is confusion about how and when the Ansible code is run.

Perhaps the biggest factor in writing fast Ansible plays is choosing whether to put a given task in the image customization or node personalization phase of configuration. Image customization is when changes are made to an image pre-boot, and is good for installing packages, adding files, or other tasks that would apply the same configuration to a large group of nodes. Running tasks at this stage prevents them from needing to be run on every node individually, which not only saves time in the short run by reducing the number of targets the task is run against, but also in the long run by ensuring the task never needs to be run again so long as the image that was customized is reused.

On the other hand, node personalization runs after a node boots and is generally less efficient. It should be reserved for tasks that have different results on every node, or for tasks that require running services. Some tasks can only be executed post-boot, but in order to keep configuration time low, it's important to move tasks to image customization whenever possible. Tasks can distinguish which mode is being run by using the cray_cfs_image variable as documented in the admin guide. This value evaluates to true when Ansible plays are run in a CFS session that targets one or more images hosted by the Image Management Service (IMS). See the "Managing Hardware" section for more information about this mode. It is also important that all tasks check this variable, either at the task level or role level, for the situation when a playbook will be run for both image customization and node personalization. Cray-authored plays and roles are grouped by functionality and therefore will contain checks for cray_cfs_image within a single playbook and within individual Ansible roles.

Beyond the choice of image customization and node personalization, users should also aim to write as efficient as possible Ansible code. There are many guides that include tips for writing effective, reusable, and efficient Ansible code. These are readily available online or in our documentation. As a result, this paper will not cover them, but the principles are important to incorporate, and users are encouraged to take the time to find this information.

It is important to take into consideration the CFS framework that Ansible is running in when writing configuration content for the Shasta system. CFS has several features that separate it from using stand-alone Ansible. One important consideration is that when run through BOS or when setting components desired state, components are automatically split into batches which are independent Ansible runs. When batching, there's no coordination between Ansible running in the separate batcher jobs. So, if components rely on a certain strict order of tasks across all nodes, you may have to build coordination into the plays. This could mean adding a task to wait on system state or coordination via a flag, although both approaches would sacrifice some of the performance benefits of batching.

Likewise, due to batching, any tasks that specify run_once will run once per batch. If it's important that the task run only once for an entire host group (and not just a subset within a single batch), a better solution would be to specify the host for the task, so that only the batch that contains the task ends up running the role.

In both these cases, because CFS-Batcher also enforces state, any solution will have to take into account that a single node could reboot and have the configuration reapplied at any time, and there is no guarantee the all nodes will configure at the same time.

Finally, tasks should ideally only make changes once, even if the playbook is run multiple times. It is common to apply a new version of a playbook on top of an already applied playbook, and it will save time in subsequent runs if not every task runs the second time. This is usually accomplished using the "when" keyword, but can be done other ways, such as building the logic into the module if custom modules are used.

IV. Managing Hardware

When it comes to managing the components that Ansible is running against, CFS provides several options. The special case of the four options is "image" which is used for image customization, and lets you specify the image and any of the hardware groups it will belongs to when booted. The remaining three are all used in post-boot scenarios on live nodes. First is "spec" or command line inventory. This is similar in usage to "image", and lets you specify the component(s) to run against on the command line, as well as the hardware groups that the components belong to. While "spec" is useful for one-off configuration runs, "repo" is a more long-term solution, and simply refers to a standard Ansible inventory file that is stored in the VCS repo along with the Ansible content.

The most common inventory method is "dynamic" inventory. This is generated from information stored in the system's Hardware State Manager (HSM) and as a result requires no additional management by the user. All components in HSM are automatically included in the inventory and assigned to Ansible groups based on their hardware roles. These are automatically determined so that dynamic inventory works out of the box, but Ansible groups are also generated for the inventory based on HSM partitions and groups/labels, giving users more control over the inventory, while not having to maintain a separate inventory in a git repository. One important thing to note is that while HSM distinguishes between hardware roles, partitions, and groups, Ansible does not support this distinction, and everything is compressed into a single set of groups. This means that using a group name that is the same as a hardware role name, such as Compute, will create a conflict and this should be avoided. In this instance the priority is given to the hardware role first, and then partitions, and lastly to groups, and the lowest priority will simply be ignored if there is a naming conflict.

Most of the hardware roles currently available are managed by default. Configuration for both Compute and Application nodes (which includes nodes functioning as the User Access Nodes or UANs) are included in the site.yml playbook. This is used both for image customization, which is called via CFS directly, and later as part of a BOS session to personalize the node components.

Management nodes also have a default configuration provided, although this is handled by a different mechanism. Rather than being triggered by the system installation commands or through BOS, each of the Management non-compute nodes (NCNs) is managed by a Kubernetes daemonset. When the daemonset starts up on each node, it calls CFS to run the ncn-customization playbook against itself, and when the daemonset is deleted, CFS is called again with an ncn-customization-unload playbook to tear down changes that were made by the initial customization. The NCN daemonset is not a permanent solution and will be removed in future versions of the NCN software, but it does demonstrate another interesting way in which CFS can be appropriately used.

V. Comparison with Previous Configuration Solutions

CFS on Shasta systems introduces several major differences with the previous configuration management solution on Cray XC series systems. Many of these changes were the result of direct user feedback. The most notable change is the shift from building unconfigured images on the system and applying configuration solely during node boot as was done on XC series systems. In Shasta, CFS enables the user to customize a pre-built image prior to booting it. The intention of this change was to reduce the time required to fully configure and customize nodes, as well as provide users with a more realistic view of an image prior to boot. Through the usage of the cray_cfs_image variable, users can provide Ansible tasks that can differentiate between pre-boot image customization and post-boot node personalization to give the flexibility of running the proper configuration tasks when desired.

Another significant change is moving from directory-based config sets managed by the configurator tool on XC series systems to Git-based configuration content management in Shasta. Config sets contained mostly configuration data (playbooks could be added ad hoc), with the majority of Ansible plays being embedded in the boot images and the site playbook generated dynamically during boot time. By bringing all the Ansible plays and roles together with inventory into a single location in Git, the user can see a more complete picture of the configuration that will be applied.

CFS only runs Ansible plays that are available in Git repositories. This enables modern change management and familiar development workflows for configuration content as well as the simplicity of branching for applying test configurations to nodes. The inclusion of Gitea as the provided git service on the Shasta system also provides a Github-like user interface for managing changes through pull requests, branch permissions, etc.

Finally, the cray-ansible tooling on XC series systems ran Ansible locally against the node as it was booting. CFS instead runs in batches of nodes in the familiar and common Ansible “push” mode with a full-fledged inventory, including the batteries-included dynamic inventory provided by the HSM. This allows for fine-grained configuration host targeting without requiring a boot image to be rebuilt. Leveraging standard data targeting like Ansible’s inventory also allows for fine-grained configuration data differentiation at global, group, and host levels that was available before through Cray’s simple sync functionality only.

VI. Upcoming Improvements

There are many improvements coming for CFS as well, both new features, and performance improvements as we work to bring down the time it takes to configure a system. One of the big new features planned for the v1.4 release is support for multi-layer configurations. Rather than creating a CFS session for each playbook that needs to run to configure a system, it will be possible to request a single session that runs multiple playbooks from different git repositories. This will make it easier to track and manage changes to different parts of the configuration, by making it possible to keep each part in its own playbook and even in its own branch or repository. This feature is applicable when setting desired configuration states for components, so that users can not only apply multiple playbooks, but also ensure that the same set of playbooks will be reapplied if the component is restarted.

Performance improvements are also in development for the CFS-Operator in particular. As seen in the graphs above, this is currently a bottleneck in CFS. When creating large numbers of CFS sessions, such as when setting the batch size to 1, the CFS-Operator can take a long time to create all the necessary Kubernetes jobs. By reducing the time, CFS will be able to support not just larger scale systems without as much overhead, but also make using smaller batch sizes more practical and improve the time it takes for Ansible to run.

VII. Summary

CFS is a powerful tool for configuring large groups of components, but it is important to understand how it operates in order to use if effectively. This is especially true when it comes to achieving desired performance out of the recently added batching capabilities. The optimal CFS configuration and Ansible code will depend on the specifics of the system being configured, and the configuration being applied. However, by learning about both CFS and Ansible, and by following best practices, it is possible to efficiently apply configuration to a system of considerable scale.