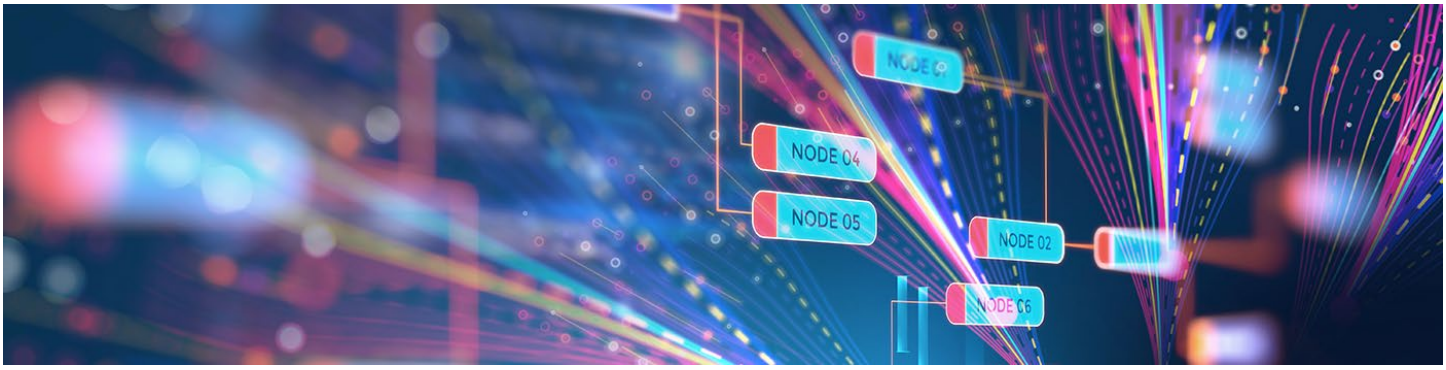
ALTAIR HERO – LEVERAGING ACCELERATOR SCHEDULING FOR EMULATION

Stuart Taylor – Altair / July 7, 2022



Altair® Hero™ 2.0 is a queuing solution for specialized compute resources, notably emulators, FPGAs, and GPUs. It provides an abstract definition of hardware resources and allows multiple jobs and multiple resources to share a common queue. Hero is based on the Altair® Accelerator™ batch queue system, and all of Accelerator's features are available in Hero. This document illustrates how users can apply those features to emulation job queuing.

Hero abstracts the emulator (and other hardware) into several resource levels. The lowest level is a LEAF. In Palladium, a leaf consists of some number of domains, and typically the granularity is a full board, half board, or quarter board. In ZeBu, a leaf typically represents a module. In a GPU, a leaf could be a slice or an entire GPU. A job requests some number of LEAF elements using the term SPAN. A SPAN5 job, for example, needs 5 LEAF elements. Normally, we need to specify a type of emulator as well since our job is likely to be compiled for a specific emulator. We do this by specifying an emulator group. An emulator group is expected to consist of emulators of the same type, and any emulator in the group can run the job assuming it has available resources.

The resulting resource looks like HERO:<GROUP>_<SPAN> and a job submitted to Hero will request a resource like:

```
%nc run -r HERO:Z4_SPAN5 – myEmulatorJob.sh
```

Here, the group is Z4 and this group will contain more ZeBu Z4 emulators. SPAN5 will request 5 ZeBu modules (leaves). Hero will look at a span of 5 modules on each emulator in the group and dispatch the job to the first available.

Similarly:

```
%nc run -r HERO:PD1_SPAN12 – myEmulatorJob.sh
```

will look for 12 leaf modules (perhaps representing 1/2 boards for a total of 6 boards) on a Palladium Z1 emulator.

A challenge for large emulators like Palladium and ZeBu is that a runtime host also needs to be selected. As part of the resource configuration step, runtime hosts are associated with a particular emulator host. Hero will manage both the emulator hardware and the associated runtime hosts and ensure that jobs are dispatched to an available runtime host that has access to available emulator resources.

The user only needs to specify the size of the job and the type of emulator required and both of these aspects are normally known. The runtime host is selected automatically, and the job script will be executed on that host. The job script will typically ask Hero for its emulator placement and then ensure that intrinsic emulator job uses that placement and *only* that placement.

Additional resources can also be specified along with the emulation resources. For example:

```
%nc run -r HERO:PD1_SPAN12 CORES/8 RAM/200G License:Xcelium — myEmulatorJob.sh
```

requests a runtime host with eight available cores and 200G of RAM along with an Xcelium software license in addition to the emulator resources. This job will only run when all are available.

While we haven't seen this is in practice, it's possible for Hero to accept jobs that specify alternative emulator resources. We assume that the job is compiled accordingly, and that the execution script can invoke the right design. An example might look like:

```
%nc run -r HERO:PD2_SPAN2 | HERO:PRX2 — myFlexiJob.sh
```

Either two leaves of a Palladium Z2 or a Protium X2 can run this job.

Job Control

Hero, like Accelerator, can control jobs by sending a sequence of signals to the process tree on the runtime host. A detailed understanding of the job characteristics is needed to get the desired behavior, though often we can get by with kill all processes in the job process tree. In emulation, this approach is unlikely to be successful and will often result in job remnants blocking the emulator resources. It's essential that the correct sequence and timing of signals in order to get a suspension, stop, or checkpoint is fully understood and then encoded into the job. Hero can support a complex sequence sent to a subset of processes over a period of several minutes and can be varied depending upon the job and emulator type. Hero, however, only understands the Unix process tree on the runtime host and any communication to the emulator itself must be done by the job script itself. A common approach is to have the job script trap signals and invoke the emulator-specific code/CLI; the job script should only exit when the emulator has cleared the job. Review the usage for 'nc stop' and the associated environment variable NC_STOP_SIGNALS.

Job Projects

It's often useful to associate a job with a named project. This can be used for accounting (give me a breakdown of emulator usage based on project) and as a factor in determining resource reservations and even preemption rules. Review the use of the variables LM_PROJECT and VOV_JOBPROJ and also the parameter to 'nc run' -jobproj.

Hero Wrapper and Resource Monitoring

Altair® Monitor™ is used to track the use emulation LEAF resources. The job wrapper hero_adapter logs the usage of the LEAF resources within Monitor. Monitor's rich reporting facility can then be used on emulator resources.

Fairshare Scheduling

The dominant mechanism for ranking jobs in order of attempted dispatch is Fairshare. Users are automatically associated with given a default Fairshare accounting node, but can override this with either an environment variable or a parameter to 'nc run'. Fairshare considers running jobs and recently run jobs as well as the weighting (share) of the accounting node. Emulation jobs can vary in size (i.e., SPAN1 vs. SPAN24) and it seems wrong to treat small jobs with the same weight as large ones. We recommend that the number of leaves (SPAN) used by the job is used to scale the jobs weight using the -fstokens parameter to 'nc run.' While the leaf size of a Palladium job may differ in terms of gates than a ZeBu4 leaf, for the purposes of scheduling, the resources are orthogonal and using the span value directly for fstokens is a useful simplification.

A typical job submission may look like:

```
nc run -g /proj/venus/dft -jobproj venus -r HERO:PZ1_SPAN4 -fstokens 4  -- hero_adapter myEmuJob.sh
```

Priorities

As has been mentioned already, Fairshare is the dominant scheduling strategy as this determines the job dispatch rank. Fairshare also automatically includes the user identity and if the user is taylor and the Fairshare group is /proj/venus/dft is expanded to /proj/venus/dft.taylor and because of this, each user will have their own rank. Job priority is a number between 1-15, with 15 being the highest priority, and can be set in 'nc run' using the -p option. However, because rank dominates and each user has their own rank, job priorities are normally only effective at ordering a given user's jobs. This can be useful if I have several jobs, say a priority four, in the queue and I want a new job to run before those (assuming resource availability). I can set that job to have a priority of, say, five and it will be considered before those. It should be understood though that your low priority job (say, one) can run before my high priority job (15) because Fairshare has ranked your job higher due to your limited use of resources or your weight in the Fairshare node.

Priorities do have use elsewhere in that they can be used to categorize jobs and can be used in the construction of preemption rules. For this reason, we recommend that the use of priorities is avoided until a clear usage model for them is adopted. It's easier to start using a new feature than it is to change existing usage.

Run Directories

Often emulators are kept separate from the main compute infrastructure. The file systems may not be shared. Several run options may be useful in this case.

- -rundir allows you to specify the run directory for the job when it reaches the runtime host and finally starts. The job script or main entry point will have this set to its working directory prior to execution.
- -D tells the submission 'nc run' not to validate the directory since it may not be accessible from the submission host.
- -pre defines a script that will be run prior to the main job script and can be used to copy in necessary data to a filesystem local to the runtime host (and whose location will only be known at dispatch time).
- -post defines a script that will be run after the main job and can be used to stage data out of the runtime host.

Limits

While Altair does not recommend the use of job limits, most users of batch systems perceive them to have a value in protecting the shared resources being consumed by a single user or group. They are best used as part of a job class definition (described subsequently) as this enables an administrator to make global changes to the limits.

Containers

Hero 2.0 supports the use of container on runtime hosts, though practical experience is limited. It's recommended that the use of containers is deferred until the behavior and use case of Hero is more completely understood.

Environments

The main job script on the runtime host often needs a set of environment variables and two strategies are available. Either the job is associated with a named environment or the current submission environment is saved as restored when needed on the runtime host. A named environment might be called SA_PD which would set some parameters up suitable for simulation acceleration on a Palladium.

A job submission could look like:

```
% nc run -e SA_PD -r HERO:Z4_SPAN4 — hero_adapter myEmuJob.csh
```

It's possible to combine environments so:

```
% nc run -e SA_PD+XCELIUM -r HERO:Z4_SPAN4 — hero_adapter myEmuJob.csh
```

first sets up the SA_PD environment and then adds settings from the XCELLIUM definition. Finally, we can modify individual environment variables with the D (Define) and U (Undefine) mini environments:

```
% nc run -e SA_PD+XCELIUM+U,MAXRAM+D,XDBG=1  -r HERO:Z4_SPAN4 — hero_adapter myEmuJob.csh
```

In contrast, the SNAPSHOT environment saves the current submission environment which gets recreated on the remote runtime host. The following two commands are equivalent:

```
% nc run -e SNAPSHOT  -r HERO:Z4_SPAN4 — hero_adapter myEmuJob.csh
% nc run -r HERO:Z4_SPAN4 — hero_adapter myEmuJob.csh
```

As SNAPSHOT is the default environment. However, it's important to note that SNAPSHOT creates a file in the current directory or one defined by NC_SNAPSHOTDIR environment variable. This directory may not be accessible from the runtime host (see -rundir above). An alternative is to use the SNAPPROP environment, which encodes the environment as a property on the job instead of a file. Very large environments will have a significant impact on queue master process due to the high I/O load. The following are equivalent:

```
% nc run -e SNAPPROP  -r HERO:Z4_SPAN4 — hero_adapter myEmuJob.csh
% nc run -ep -r HERO:Z4_SPAN4 — hero_adapter myEmuJob.csh
```

Altair recommends named environments as it improves job reproducibility, but in some cases the submission scripts go to great lengths to create a custom environment for the job, and in those cases the SNAP approaches are more convenient.

Job Classes

Let's review what a typical job submission might look like if a user wanted to take advantage of the rich feature set in Hero:

```
nc run -D -rundir  /data/venus -pre copyin.sh -post copyout.sh -limit dft 8 -e
SA_PD+XCELIUM+U,MAXRAM+D,XDBG=1  -g /proj/venus/dft -jobproj venus -r HERO:PZ1_SPAN4 -fstokens 4
— hero_adapter myEmuJob.csh
```

That's a lot to get right. Of course, you can put that into a calling script, but that means as the usage model and administration of Hero evolves over time those scripts will need to be modified. Those scripts will have multiple owners and possibly the flows in which the scripts are resident are frozen. What we'd like to have is a layer where queue administrators can make any necessary configuration changes and make it easy for users to pick up those changes. That's where job classes come into play.

Effectively, a job class is a macro that gets evaluated during job submission. The job class is a Tcl file that, when evaluated, creates a job description array with the following contents:

```
VOV_JOB_DESC(check,directory)          = 0
VOV_JOB_DESC(env)                      = SA_PD+XCELIUM+U,MAXRAM+D,XDBG=1
VOV_JOB_DESC(fstokens)                 = 4
VOV_JOB_DESC(group)                    = /proj/venus/dft
VOV_JOB_DESC(jobproj)                  = venus
VOV_JOB_DESC(jpp)                      = fastest
VOV_JOB_DESC(limitres)                 = dft_8
VOV_JOB_DESC(nolog)                    = 1
VOV_JOB_DESC(postcmd)                  = copyout.sh
VOV_JOB_DESC(precmd)                   = copyin.sh
VOV_JOB_DESC(proplist)                 =
VOV_JOB_DESC(resources)                = HERO:PZ1_SPAN4 Limit:@USER@_dft_8
VOV_JOB_DESC(rundir)                   = /data/venus
VOV_JOB_DESC(wrapper)                  = vw hero_adapter
```

The group might be discovered from some environment variables in the submission environment or by looking up the user in some map. We use the fstokens value to drive the selection of the Hero span resource.

Assuming we've called the job class SA, we can now run the job with a much smaller set of parameters:

```
nc run -fstokens 4 -C SA -- myEmuJob.csh
```

The queue can also be configured to select a default job class in case one isn't selected.

Another approach is to have one job class per emulator-span combination (e.g., Z4SA1, Z4SA2, Z4SA16 for Z4 emulator jobs with 1, 2, and 16 modules respectively). While this can bloat the number of job classes, effective use of include files and symlinks can keep things manageable. It's possible, for example, for a job class to introspect what name it was called by which allows parameterization via symlinks.

The job class is an important attribute of the job (even if it isn't configured to do much). Job classes can be used in job reporting and preemption and along with jobproj, Fairshare group and user provide is one of the four primary categorization axes. The ability to act as a shim between job and queue allows teams to evolve usage models without excessive rework. Please use job classes, even when starting out.

Advanced Scheduling and Policies

Altair's general view is that the scheduler should just be allowed to run and attention paid to Fairshare and its weights to influence rank and dispatch order. However, this falls short in two areas: first the scheduler struggles to dispatch a large SPAN job when there are an inexhaustible supply of single SPAN jobs regardless of rank (priority inversion) and second, the design team may want run certain jobs at specific times perhaps for interactive debug sessions. Other common reasons for influencing the native scheduler's dispatch order include the perception of ownership of a certain emulator or some subset of boards on a scheduler.

Quantitative Reservations and Limits

The simplest methods to control the scheduler involve the use of limits and reservations on quantitative resources. Hero has two types of emulator resource accounting: named resources and quantitative resources. A named resource identifies a specific location on the emulator (a board, module, or group of domains). Named resources have quantity one and they are associated with one job or are available. Quantitative resources are a count of indistinct emulator resources. We might have 32 modules, and we may reserve eight of them for a specific team, but we have not specified which modules – any eight will do.

Hero uses a complex set of resource maps. One of them selects a specific emulator and required span. For example, a SPAN4 request will result in this map for cluster0:

```
HERO:JOBCOUNT_cluster0/1 HERO:LEAFCOUNT_cluster0/4 HERO:SUB_cluster0_SPAN4 HERO:RTSELECT_cluster0
```

And a similar SPAN2 request:

```
HERO:JOBCOUNT_cluster0/1 HERO:LEAFCOUNT_cluster0/2 HERO:SUB_cluster0_SPAN2 HERO:RTSELECT_cluster0
```

The counter for cluster0 leaves is HERO:LEAFCOUNT_cluster0 and the SPAN4 resource needs four of them and the SPAN2 request needs two. This counter provides us a mechanism to reserve some number of modules on cluster0 for some entity. These consumable resources come from the Accelerator code base and we can use regular Accelerator commands to reserve these resources. For example:

```
%nc cmd vovresourcemgr reserve HERO:LEAFCOUNT_cluster0 USER alice,bob 4 5m "special project"
vovresourcemgr 12/28/2021 18:18:51: message: Reserving 4 tokens of HERO:LEAFCOUNT_cluster0  for
USER alice,bob for the next 5m
```

Any job that Alice or Bob are able to start on that cluster, while the reservation is active, will consume all or part of the reservation. Other users will be able to start jobs on that cluster but will not be able to consume the last four leaves. The reservation does not specify which four leaves; this allows the scheduler more flexibility. The downside, though, is that there is no guarantee that the four reserved leaves are suitable for a SPAN4 job.

Time-based Reservations

A common use case with emulators is the need for some jobs to work with a known set of resources at a specific time. This might be for a time-critical simulation or for interactive work. The design may be compiled to use these specific named resources too and might require a relatively large span of resources (say SPAN24). We might want to treat at least some of the emulation resources a bit like conference rooms where we book named resources into the future for different teams and purposes. This is often termed Outlook scheduling (from the ubiquitous calendar application). The ability to mix different kinds of scheduling on the same set of emulators can often bring the best of higher utilization provided by batch scheduling along with the determinism from Outlook-like technique.

Hero provides a mechanism to reserve specific leaf modules.

```
% hero_reserve HERO:Z4_cluster0_U(0..7).M(0..7) -duration 6m -jobproj myProj
% hero_reserve HERO:Z4_cluster0_U1.M1 1484612764  forever 1 -jobproj myProj -group g1,g2
% hero_reserve HERO:Z4_SPAN5 -duration 2h -user John,Bob -after NOW -before tomorrow
```

The hero_reserve command will return a reservation ID. We can use the reservation ID to find out which boards and emulator it found if we allowed it flexibility (the third example).

```
nc cmd vovselect why,reservestart,reserveend from reservations where id==002036658
HERO:Z4_SPAN5 HERO:EMUL_cluster1_SPAN5 HERO:PART_cluster1_SPAN5_6  1640720885   1640728085
```

This tells us what was reserved and when. We've not seen the HERO:PART resource component yet but it precisely identifies the leaves used in the reservation and should be consumed by the job. We can find out the leaves referred to by part by using the following command:

```
nc cmd vovselect map from resourcemaps where name==PART_cluster1_SPAN5_6
HERO:LEAF_cluster1_U0.M0 HERO:LEAF_cluster1_U0.M1 HERO:LEAF_cluster1_U0.M2 HERO:LEAF_cluster1_U0.M3
HERO:LEAF_cluster1_U1.M0
```

That, combined with the reserveend and reservestart, now gives us the reservation's "what" and "when."

There are, however, two remaining problems: first, forcing a job to use the reservation at the appointed time; and second, removing other jobs that are using the resources. For a long-standing reservation on a busy emulator, neither are likely to be an issue since there would be no other free slots for the job, and other jobs would likely have already completed. But for shorter reservations, reservations that are periodic (e.g., 1–5 p.m. every day), reservations that change owner frequently, or an emulator that frequently has idle slots, we need some forcing strategy to get the desired behavior.

Job-to-reservation Binding

The idea of binding a job to a reservation isn't yet baked into Hero, but the underlying facilities are present. Here's one simple, effective scheme.

We start by reserving for a Fairshare group. This has several advantages over other reservation types in that it allows multiple users and job classes to make use of the reservation and fits with the primary dispatch ranking employed by the scheduler.

```
hero_reserve HERO:Z4_SPAN12 -duration 60m -group /proj/mars -after 4pm -before tomorrow
```

is one example. Our job could be in the /proj/mars directly or could be assigned to a group below that /proj/mars/dft.

We can reserve for a project team (this example) while still have the accounting for sub teams unaffected. Or we could be more precise with the reservation:

```
hero_reserve HERO:Z4_SPAN12 -duration 60m -group /proj/mars/soc -after 12pm -before tomorrow
```

which would prevent the mars dft team from using this reservation.

To bind a job to the reservation we make use of the Fairshare subgroup option '-sg.' Other fields like TOOL (nc run -tool) and JOBNAME (nc run -N), can also be used.

```
nc run -sg RESERVE_000012651 -r HERO:Z4_SPAN12 -- hero_adapter myjob.sh
```

Assuming 000012651 was the ID returned by the hero_reserve command, the Fairshare subgroup marks the job as wanting to be bound to the reservation. We do the binding in the VncPolicyValidateResources callback, a standard Accelerator feature which is called just before the job is submitted. Here's some sample code to do the binding – a few lines, but nothing complicated.

```
proc VncPolicyValidateResources { resList } {
  global VOV_JOB_DESC
  global env
  set resprepend {}
  if {[info exists VOV_JOB_DESC(subgroup)]} {
    if [regexp {RESERVE_([0-9]+)} $VOV_JOB_DESC(subgroup) -> resid] {
      vtk_select_loop -select id,reservestart,reserveend,reservegroup,why -from reservations -where
"id==$resid" {rid, rstart, rend, rgroup, why } {
        set VOV_JOB_DESC(schedule,date) $rstart
        set VOV_JOB_DESC(deadline) $rend
        if {$rgroup != ""} {
          set VOV_JOB_DESC(group,final) ""
          set VOV_JOB_DESC(group) "$rgroup"
          VncSetGroup [vtk_logname]
          set partix [lsearch -glob $why HERO:PART*]
          set emulix [lsearch -glob $why HERO:EMUL*]
          if { $partix > -1 && $emulix > -1 } {
            set emulres [lindex $why $emulix]
            set partres [lindex $why $partix]
            vtk_resourcemap_get $emulres ra
            set emulmap $ra(map)
            #HERO:SUB_cluster0_SPAN4
            regsub {HERO:SUB_[\w]+} $emulmap $partres partialres
            # put partial res as first element so as to generate a LEAF based wait reason
            set firstix [lsearch -glob $partialres HERO:*]
            set firstel [lindex $partialres $firstix]
            set swapix [lsearch $partialres $partres]
            lset partialres $firstix $partres
            lset partialres $swapix $firstel
            regsub {HERO:[\w]+} $resList $partialres newres
            puts "Constrained reservation: $partialres is solution"
            set resList $newres
          }
        }
      }
    }
  }
  return "$resList"
}
```

The main actions are to replace the resource SPAN request with the solution found by the resource reservation and to adjust the schedule time of the job to coincide with the start of the reservation. This will ensure that the job will consume the resources intended by the reservation.

Next, we need something to stop other jobs that are consuming the reserved leaves when the reservation becomes active. In this example, though, we only do the stop if a job is present that is bound to the reservation. We do this by creating a preemption rule dynamically as part of a Hero task. Tasks are run about every minute or so.

The task is mostly defined here. Note we look for active reservations and create a matching preemption rule. The rule itself does the preemption and leverages the intelligence built into the preemption system to stop only jobs that have resources required by the preempting job.

```
set FIELDS id,isactive,duration,quantity,type,reservegroup,reservestart,reserveend,why
set FILTER {type==resourcemap reservegroup!="" isactive==1}
set POOL FSGROUP_RESERVE_HELPER
array set ISACTIVE {}
vtk_select_loop -select $FIELDS -from reservations -where $FILTER [split $FIELDS ,] {
    puts "Active fsgroup reservation $id $reservegroup $why"
    set PR_NAME "RESERVE_$id"
    set PR_ID [vtk_preemptrule_find $POOL $PR_NAME]
    set ISACTIVE($PR_NAME) $PR_ID
    if {$PR_ID != 0} {
        puts "already have rule $PR_NAME"
        continue
    }
   puts "Create rule $PR_NAME for $reservegroup $why"
    VovPreemptRule \
  -pool   $POOL \
  -rulename $PR_NAME \
  -ruletype "FAST_FAIRSHARE" \
  -order  50 \
  -debug 1 -enabled 1 -fireonce 0 \
  -waitingfor "HERO:*" \
  -numjobs "1" \
  -sortjobsby "PRIORITY ASC, AGE ASC" \
  -preempting "FSSUBGROUP~RESERVE_" \
  -preemptable "GRABBEDRESOURCES~HERO" \
  -killage 52w \
  -resumeres "" \
  -resumedelay "5" \
}
```

For completeness, here's a clean-up script that can be added to the task to remove preemption rules that are no longer required:

```
# cleanup rules for expired reservations
vtk_generic_get preemptrules PR
foreach id $PR(list) {
    set rname $PR($id,name)
    if {[regexp {^RESERVE_} $rname] } {
        if [info exists ISACTIVE($PR($id,name))] {
            puts "$rname is an active rule"
        } else {
            puts "$rname is an inactive rule and should be deleted"
            vtk_preemptrule_delete $id
        }
    } else {
        #puts "$rname is not in scope"   }
```

Our reservation system is mostly complete. It actively kills jobs that are using the reserved resources, but only if the intended reservee is present. We can make reservations in the future and have jobs start at the intended time. This is particularly useful for efficiently starting interactive jobs, as load time can be estimated and the user can access the prompt at the reservation time and load time.

Like booked but unused conference rooms, unused reservations are wasteful. It's useful to cancel such reservations, perhaps 15 minutes into the reservation. We can use the why field from the reservation and pull out the PART component, then using this name check to see whether the reservation is in use. For example, we can look at active reservations.

```
vovselect id,why,isactive from reservations where "isactive==1 why~PART"
002037149    HERO:Z4_SPAN    5 HERO:EMUL_cluster1_SPAN5 HERO:PART_cluster1_SPAN5_3    1

vovselect name,total,inuse from resourcemaps where "type==HERO name==PART_cluster1_SPAN5_3"
PART_cluster1_SPAN5_3      1      1
```

This shows the PART resource to be used. If it wasn't the inuse field would be zero and if that were the case and the reservestart field from the reservations. This also gives us the facility to generate reservation utilization reports should they be of interest.

Priority Inversion and Preemption

A classic problem in scheduling occurs when a high priority but difficult (it has large resource requirements) to dispatch job is deferred in preference to lower priority easy to dispatch jobs. The scheduler never sees a contiguous span of resources sufficient for the high-priority job but can always dispatch the smaller job. If there's a never-ending supply of smaller jobs, then the high priority job will never get dispatched.

There are a few strategies we can employ to address the problem, and choosing the best is often dependent on the kind of workload we have.

An easy approach is to use different job classes for the small jobs and big jobs, perhaps SA1 and SA16. We can then reserve half the emulator for the big job class:

```
hero_reserve HERO:Z4_SPAN16 -duration forever -jobclass SA16
```

A permanent reservation like this one is obviously effective, but it is obviously wasteful too. Perhaps what we want is to only apply the reservation when big jobs are waiting. We can use a resource reservation preemption rule to do this.

```
VovPreemptRule \
    -pool     "PriorityInversion" \
    -rulename "SA16cluster0" \
    -ruletype "RESERVE_RESOURCES" \
    -order    50 \
    -debug 1 -enabled 1 -fireonce 0 \
    -waitingfor "HERO:LEAF_cluster0_U4.M0 HERO:LEAF_cluster0_U4.M1 HERO:LEAF_cluster0_U4.M2
HERO:LEAF_cluster0_U4.M3 HERO:LEAF_cluster0_U5.M0 HERO:LEAF_cluster0_U5.M1 HERO:LEAF_cluster0_U5.M2
HERO:LEAF_cluster0_U5.M3 HERO:LEAF_cluster0_U6.M0 HERO:LEAF_cluster0_U6.M1 HERO:LEAF_cluster0_U6.M2
HERO:LEAF_cluster0_U6.M3 HERO:LEAF_cluster0_U7.M0 HERO:LEAF_cluster0_U7.M1 HERO:LEAF_cluster0_U7.M2
HERO:LEAF_cluster0_U7.M3" \
    -reservefor "JOBCLASS SA16" \
    -reservenum "1" \
    -reservetime "60" \
    -reservetype "JOBID" \
```

This rule will work well if the jobs that are using the leaf resources exit reasonably quickly. If they don't, we may need another rule to reserve another 16 leaves. This can also become wasteful. Another approach is to limit the emulator to either big jobs or short-running small jobs.

```
vovresourcemgr reserve  HERO:JOBCOUNT_cluster0  JOBCLASS SA16,SA1SHORT 16 forever "priority
inversion"
vovresourcemgr 12/29/2021 20:58:09: message: Reserving 16 tokens of HERO:JOBCOUNT_cluster0  for
JOBCLASS SA16,SA1SHORT for the next forever
```

is an example of improving the chances of the reservation being successful. Here we expect SA1SHORT to implement an autokill to keep SA1 jobs short. Increasing the count to 32 would be more effective but also more restrictive. Effective job dispatch latency reduction techniques always have an impact on utilization.

Finally, we can use a preemption rule to kill jobs that are in the way. We can use a similar rule to the one used for clearing a time-based reservation, but care is needed to make sure that all jobs "in the way" are preemptable; otherwise, the system may start preempting them piecemeal and then find that there's one that it can't preempt due to an age or non-preemptable flag. A combination of job class restrictions (reservations) and preemption may be more effective approach.

Composite Regressions and Job Chains

For many designs, a regression suite is likely to consist of many separate tests. The tests may be quite short perhaps a few seconds. In contrast it may take five to ten minutes to load the design on to the emulator; clearly we don't want to load the design each time we run a test. In earlier versions of Hero, the approach was to group all the tests together to form a single large job, with the design load being done just once at the beginning of the composite job. There are a few problems with this approach: visibility is lost on individual jobs – we can't see which test is running and which have passed or failed; we can't control the tests so the ability to autokill a test if its runtime becomes extraordinary long is removed; we can't stop or pause the regression on a test boundary; and incremental reruns are impossible. While it's possible to code these things into the master regression job, it would be more convenient to let the batch system take care of these things.

Hero 2.0 provides a scheme in which tests are treated as separate jobs and the design is only loaded when necessary. The tests themselves are submitted directly via 'nc run' and are associated with a job set and request a resource that references the design and a possibly a specific version or build of that design; the job set groups the tests into a regression and the resource request prevents the scheduler from dispatching the job to any available tasker. We can now put properties on to the sets to identify when to run the regression (now or sometime later) and also the SPAN of resources required to run the design. A script can now be run manually or periodically to look for job sets with those properties and for each set found and design loader job can be generated. The design loader job looks like a regular Hero job with a request for a certain emulator group and SPAN. When the resources become available the design loader is dispatched and it ensures that the appropriate image is loaded on to the right emulator and boards with assistance from the hero_adapter wrapper. While a regular Hero job would now start the emulation, the design loader instead starts a Hero tasker. The tasker is reserved for test jobs (using a job class reservation) and offers the resource strings (design name and version, perhaps) that are being requested by those test jobs.

The result is that the test jobs are now dispatched one-by-one to this special tasker and are used as test stimulus for the emulator image. The characteristics of this approach are as follows:

- The design is loaded only at the beginning of the regression
- Tests are visible as separate jobs and have separate return exit codes
- The status of the overall regression is immediately available
- Being individual jobs, the tests can be subject to specific runtime limits and autokill (prevents one test spoiling the entire run)
- The regression can be preempted on a test job boundary
- Incremental reruns of the regression are possible (subject to setup constraints that might be needed for certain groups of tests)
- The pre- and post- commands are now available for each test separately giving a mechanism to upload test performance and status to design metric databases
- Multiple instances of the design can be loaded on to different emulator resources and execute in parallel using the same regression test set (subject to design setup constraints)
- Tests can be added while the regression is running.

Contact support@altair.com to investigate this capability further. Demo code is available.

Longitudinal Reporting

Now that we've got our tests represented as separate jobs, we might want to run a **longitudinal report**. Such a report looks at a specific test and compares its behavior as the design changes. This can be useful to detect when a test started to become unstable or fail consistently. Such a report can be generated from a database system outside of Hero and we can use the post- command to help load that data efficiently. Additionally, Hero has its own jobs database, which with some preparation can also be made to run longitudinal reports. The challenge with the built-in database is that it's designed for aggregating data, but here, we want to keep the records separate and identify specific jobs. Supposing each of our test jobs has a unique name to identify the test, our job submission could look something like:

```
%nc run -C emulSwTest -set regression:venus:nightly -tool ddrsync -r venus_full_build_1234 –
test_ddrsync_fullmodel.sh
```

That defines the job as a test job (-C emulSwTest), puts it into the nightly regression set for the venus project, names the test ddrsync, and requires the design image venus_full_build_1234. The job script itself comes after the double, as normal. Of course, it's possible for the job class itself to populate most of these arguments, perhaps derived from environment variables.

To run the report, we will need some SQL:

```
nc cmd vovsql_query -e \

       "SELECT to_char(to_timestamp(endtime), 'YY-MM-DD HH24:mm:ss') AS
time,jobs.id,exitstatus,tools.name,resources.name,statuses.name,endtime-starttime

       FROM jobs,tools,resources,statuses

       WHERE starttime > 1645892997 AND tools.name='ddrsync'

       AND tools.id=jobs.toolid

       AND resources.id=jobs.resourcesid

       AND  statuses.id=jobs.statusid "


 {22-02-26 13:02:43} 964329900 0 ddrsync venus_full_build_1204 Done 10
 {22-02-26 13:02:50} 964329904 0 ddrsync venus_full_build_1214 Done 10
 {22-02-26 13:02:11} 964329923 1 ddrsync venus_full_build_1235 Failed 0
 {22-02-26 13:02:55} 964329907 0 ddrsync venus_full_build_1240 Done 10
 {22-02-26 13:02:04} 964329909 0 ddrsync venus_full_build_1241 Done 11
 {22-02-26 14:02:34} 964329926 1 ddrsync venus_full_build_1252 Failed 1
```

The advantage of making the resource string meaningful as well as unique to the design image can now be seen. We're able to see the design image version next to the job status – something to consider when using composite regressions and job chains. The database itself is Postgres and comes with a published schema.