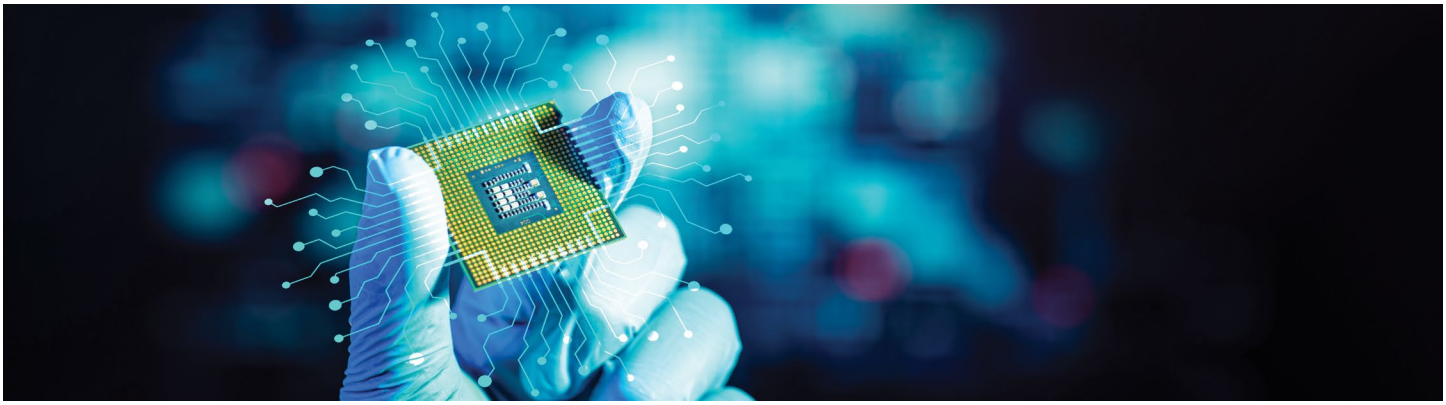
PUSHING PPA WITH ALTAIR® FLOWTRACER™ – ADVANCED USAGE MODELS FOR PHYSICAL CHIP DESIGN

Stuart Taylor and Kirk Hsu – Altair



Introduction

Physical design flows that push the power, performance, and area (PPA) envelope need to be fully repeatable and tuned for every aspect of the design block so an organization can retain the knowledge and recipe for future use.

Altair® FlowTracer™ is a rare example of a flow manager that can have its flow redefined even as the flow is running. A common case is where the designer sees some bad results from a design step and wants to attempt an alternate approach. We could stop the current flow, make some changes and then restart the flow; for gross configuration errors this is reasonable, but for minor parameter changes we lose the results from our current state — poor as they might be — for some speculative gain. Alternatively, we could start an entirely new flow with a modified parameter set, and while we'd still have our original flow running, we'd be repeating the earlier steps in this new flow instance. Ideally, we shouldn't stop the current flow (because we may still get good results) nor should we restart a new flow instance from the beginning (because we would needlessly repeat steps with good results). With FlowTracer, we avoid these problems by branching the flow just before the problematic step by cloning that step and its dependencies. This duplicates the flow from the problematic step (down-cone) and retains the results from the previous steps (up-cone). We may do this repeatedly, both at the same point in the flow and in other places — we can branch from a branch. Our previously linear trunk flow may grow into a densely branched tree. Some of these branches may start to yield bad results that are irrecoverable for even the best optimization tools; to avoid wasted computation and disk space, we can prune these branches and allocate precious resources to the winning recipes.

The branch-and-prune strategy is a proven approach to drive PPA, used by leading chip design teams. Let's consider a few enabling techniques and requirements for success.

Reproducibility and Designer Control

There was a strong development push in the early 1990s for CAD frameworks. They all failed commercially. Why? They were constraining the design process and getting in the way of designer ingenuity and productivity. Chip design is hard enough, and no one wants tooling that gets in the way. A central tenet of a successful FlowTracer deployment is that it leaves the designer in control. Successful flows allow the designer to comprehend what is going on and how to modify the behavior when desired. However, with all this flexibility, how do we retain our primary goal of reproducibility? In the classic mode of chip design, the designer's work output was the physical layout for the block; in the new model the primary output is the recipe for the block, and the block layout is almost a side effect of running the recipe upon the primary input data. To get reproducibility, we must capture all the flow tweaks the designer added. Complicating matters is that we may have a large, semi-automated configuration system; physical design flows often have more than 10,000 variables, perhaps sourced from multiple files owned by different teams. Designer tweaks may occur as primary inputs to the configuration system, and we assume the configuration is rebuilt or the designer may insert the change at the point of use by the tool, which is essentially an override to the main configuration system. Both approaches need to be acknowledged.

Visibility

Most complex physical design tools have a run control file, often supporting the Tcl language as input syntax. While Tcl support allows the input file to be fully programmatic — perhaps a long series of procedure definitions with the final line being the equivalent of calling the "main" procedure — we lose a lot of visibility into what's going on. Another approach is to unroll much of that code into a series of one-line instructions; we can still have the programmatic aspect, but we can keep this in a template and just expose the fully expanded steps. Any suitable templating scheme or code generator can be used, so long as it can interface well with the configuration builder.

Now we build the configuration and then apply it to each of the flow steps' templates, creating one run control (RC) file for each step. A key element of this approach is that changing the configuration results in a change to the RC file, if — and only if — the change was relevant to the RC file. So, if we change a parameter for the clock tree synthesis (CTS) step and that parameter is only used in the CTS step, then even if we rebuild all the RC files, only the CTS file will have changed. Conversely, if our motivation is a change to CTS but that same parameter is also used in routing — and they need to be consistent — our approach will guarantee the change will be propagated to both RC files. FlowTracer can leverage this behavior to enable modifications to the flow while it's running.

Barriers

Our desired behavior is for the designer to make the necessary change to the configuration system, rerun the config build, and have only the steps that are impacted by the change be invalidated. However, when we rerun the configuration system, the typical behavior is to update every RC file — the timestamps will change — and consequently all steps will be invalidated. FlowTracer brings a useful device, the barrier, to address the problem; we can hold back timestamp-only changes and let through impactful updates. An MD5 checksum property or similar is typically used to assess whether to propagate the change. Other approaches are also possible.

We've allowed designers to make a configuration change using whatever tools or mini-flow they used to initially create the configuration. They can do this while the flow is running, and only those steps that are impacted get invalidated. The configuration tool can ease the task of managing many parameters. Being able to use it both a priori and concurrently with a running flow is a big win, and collecting the recipe is much easier too.

Snippets

Sometimes parameters just aren't enough. Designers need to be able to inject bespoke code into the RC file. One approach we've seen used successfully is to make sourcing calls to a series of customization files. These files may be empty, but when populated, the code will be called as the RC file is evaluated. When the recipe for the run is being assembled for publishing, these files need to be packed along with configuration parameters. This approach is a win for the designer too, which is why the technique is successful. On starting a new run, perhaps with updated design files, the recipe, along with the snippet files, gets copied into a new FlowTracer session run area. The designer's last best recipe is now the starting point for the new run. No need to worry about stashing the snippets away and recopying them into the new run — it's all handled automatically.

This approach can be abused. For example, the first snippet file (the calling point is normally at the start of the RC file) might just call a completely custom sequence and then exit — so the designer can avoid the prescribed recipe completely, but at least the new recipe is captured. We should make it easy for a designer to do what they want, and we can later analyze why they're ignoring our starting recipe. Ultimately, this leads to "top recipes" or "best-known methods," based on capturing designer ingenuity.

Finding Best-Known Methods

The best recipe for a design block may depend on many factors including the technology node, standard cell height, tool version, tool chain, power grid, etc. Some of these will be fixed for a given project and we'll need to vary others to find good recipes. We'll want to change the parameters for various flow steps but also decide which flow steps we want. Some blocks may benefit from extra placement optimization or certain clock strategies. These choices will impact flow topology — the number and type of flow steps required.

While both step parameters and flow topology can be set using a comprehensive configuration system, sometimes this can be overwhelming for designers in charge of exploration. Perhaps there are 30 discrete flow steps, 10 different master flow recipes, and 10,000 separate parameters to be chosen across 50 design blocks. We've occasionally heard the request for a schematic approach with a graphical user interface (UI) to help with flow definition. While we're skeptical about the benefits of a true schematic system, which is slow and tedious to edit, there may be some benefit to staging configurations in a more structured manner.

Leverage FlowTracer UI for Configuration

If we can leverage the FlowTracer UI, we don't need to exit and return to the tool – everything happens in place, easing the burden on the designer. We may forgo some of the slickness of a schematic tool, but we avoid semantic gaps that are nearly inevitable with a side-by-side approach.

Scheme 1: Let's start with a blank sheet and go to a freshly created but empty set, Flows:master.

Right-click and get a menu item for "Add Flow." This assumes the technology node has already been defined, and if it hasn't, perhaps our starting point is "Choose Technology," and then we get to choose a flow with "Add Flow." This is highly dependent on the CAD master framework. You'll need to choose something that works for you. The key is that you're given the ability to add a placeholder for the flow, and that the placeholder is just a job (transition) represented by one of FlowTracer's rectangles. It will be purple because it hasn't been scheduled yet.

Next, we need to choose a master flow recipe. Select the job and, using a context-sensitive menu, choose one of several master flows — these are the best-known methods. This will set a property on the job or populate a file to act as an input to the job. Now we need to define which blocks will use this flow. Select "Add Blocks" and a pop-up will show each block available to the user; this will likely be provided by the initialization step and keyed off some master database. The list of selected blocks will now be added to the input file for the job. It can be the same file as the master flow selection, but it's likely to be easier to have a different file.

We may have some blocks that we know require a different recipe. No problem. We add another job with "Add Flow" and configure as appropriate, matching the block to the flow.

Let's assume we have two add-flow jobs (perhaps called LV and HP), the first with blocks A, B, and C, the second with blocks D and E. We'll schedule and run the jobs just as we would for ordinary steps.

The result will be five discrete flows in the same FlowTracer session, resulting in five sets within the initial Flows:master set. We'll have run the flow configuration step separately for each of the five sets and we'll have built a flow for each block/master flow combination.

Now we want to do some configuration for just block B, which uses the LV master recipe. Select the set for flow B, then right-click and bring up "Edit Config," which will invoke the configuration editor for this flow instance. Edit the flow variables as desired. Save and rerun the config. Now we have a custom flow for B.

We then want to change the MpCTS step for blocks B and D. Click into the set for B, find the MpCTS step, right-click and bring up "Edit Config." Change the config as desired. Save and rebuild. Only the MpCTS step for block B will be changed. Go back to the top and repeat the process for block D's MpCTS.

Now we have the five flows with their custom mods:

LV-A, LV-B♥-MpCTS♦, LV-C, HP-D-MpCTS♣, HP-E

Now suppose we have 100 blocks and want to change the MpCTS for all of them. That's 100 edits, and that's not a good experience. Instead, what we can do — in addition to creating the block set view (those five sets) — we can create a completely separate organization of flow steps, perhaps in the set Design:Steps. Here we'll have sets for each flow step type, and each set will contain all the instances of that flow step. That might be one for each block, or fewer if one of the master flows doesn't have the MpCTS step. Now we select the MpCTS containing set (e.g., Design:Steps:MpCTS), right-click and select "Edit Config." The changes we make here will propagate to all MpCTS instances.

We've just configured our flow in a context-sensitive manner. We've leveraged the configuration tool's ability to guide the user to the right variable, and we've leveraged FlowTracer to give context to where changes are being made. For non-programmers, this may be easier than dealing with dense configuration parameters. Because we've ensured our overall scheme can be run while the flow is running, we can do this reconfiguration operation after the flow has already started. If we're careful with our file naming and branching strategy, we can combine this configuration approach with branching too.

No schematic editor required.

Scheme 2: Let's start with a blank sheet and go to a freshly created but empty set, Flows:master.

Right-click and get a menu item for "Add Flow." This assumes the technology node has already been defined, and if it hasn't, perhaps our starting point is "Choose Technology," and then we get to choose a flow with "Add Flow." This is highly dependent on the CAD master framework. You'll need to choose something that works for you. The key is that you're given the ability to add a placeholder for the flow, and that the placeholder is just a job (transition) represented by one of FlowTracer's rectangles. It will be purple because it hasn't been scheduled yet. This is the same as Scheme 1 at this point.

Next, we need to choose a master flow recipe. Select the job and, using a context-sensitive menu, choose one of several master flows — these are the best-known methods. This will set a property on the job or populate a file to act as an input to the job. It should set the job name to be Config_$flowname. Run the job. It will create a set, Flows:Master:$flowname, that will be visible in the current set. This set will contain prototype flow (e.g., Config -> ProtoFloorplan -> ProtoPlace -> ProtoCTS). These jobs act as a template and will be used to configure the real flow. If we want to do special configuration of the place step, we'll select ProtoPlace and right-click to select config, invoking the configuration editor and allowing us to customize the step. We may rename a step, so if we don't like ProtoCTS, we can rename it ProtoMpCTS. Perhaps we don't want to run RouteOpt, so we skip (FT autoflow) ProtoRouteOpt.

Getting fancier, we can have an operator that inserts a step. You'll need some fancy footwork to add the transition as a descendant to that which is currently selected and rewire the previous descendant. We hit play (run these steps). The first one errors. We haven't specified a block list. Select job edit and add the block names to the command line.

Now hit play again. This time each job runs in sequence; the job could be a no-op or it could check that it is compatible with its ancestor (e.g., don't allow route before place). This enforces a semantic consistency check.

The very last step in this proto flow does the real work. Perhaps we have a step InstantiateFlow that this builds the real flow, creates navigation sets, and potentially automatically triggers a trace (run/play) of this real flow. We may add (duplicate) sets (e.g., Flows:Master:$flowname:$blockname and Flows:Blocks:$blockname). That way we have a representation of the real flow where it should be (Flows:Blocks) and also where we're currently viewing Flows:Master:$flowname:$blockname. That ensures the user gets to see the flows produced where they're currently looking and not somewhere else — a better UI experience.

Now when we push into these sets, we're dealing with real flow steps that we can further configure and parameterize.

The process can be repeated for a different flow and a different set of blocks. Care and attention to naming schemes is important to avoid collisions. It's perfectly acceptable to have a block being run through different flows, all within the same FlowTracer session, but separation at the filesystem level is important; recall that the triplet of ENV/DIRECTORY/COMMAND must be unique.

Let's recap. We can choose a recommended flow from a best-known methods list, and we can edit that flow, parameterizing some steps (flow instance scope), skipping some steps, and potentially adding more. We can then choose which blocks we want to apply this flow to and then we can build (unfold) the block-flow combination into real flows.

That's a lot of flexibility, and the context for configuration is guided by the proto-flow rendering. We have semantic checking during proto-flow to flow instantiation.

No schematic editor required.

Comparing Schemes 1 and 2: Scheme 2 is more elaborate but allows more flexibility. It's a two-step process that exposes the prototype flow data model. This, in turn, allows flow editing/configuration before it expands across the block list with the possibility of flow semantic checks before realization. Will end users be confused about the two related but very different job types (e.g., ProtoMpCts and MpCts)? One runs CTS for real, and the other does nothing of the sort. But they do edit the flow in folded space and can catch semantic errors before any real tools get invoked.

Scheme 3: Designers occasionally find a complete flow too restrictive. They want to focus on a certain subset of the flow, and they may run experiments that have a slim chance of success. Indeed, proving that a certain strategy is unprofitable is part of the design process. They, reasonably, don't want those results published, as would happen in the regular flow. They may want to run some steps iteratively, more like a mini engineering change order (ECO) flow; perhaps we end up with three PlaceOpt steps, one after another, with the directives (snippets) looking very much like an ECO. Designers capable of such techniques will do it anyway, regardless of whether

the formal flow supports it. This typically involves a lot of file copying and symlink magic. When they get it right, they get great results, though the recipe is off-menu. Often, though, they waste a lot of time copying and dealing with errors that result from coupling these manual steps into an automated flow. How can we make it easier to break free and get back into the flow for the remaining downstream steps? We introduce three branch primitives:

1. Branch and Disconnect
2. Iterate/Clone
3. Publish/Reconnect

In Branch and Disconnect we disconnect the down-cone from our branch point and then duplicate the branch point job. We may start with our flow trunk going from Place into Route; we then clone Place to Place* and disconnect Route from them both. We're now free to experiment in Place* or clone again to Place**.

With Iterate/Clone, we can take our Place* and feed it into a second Place step, Place*_1. This can, of course, be repeated and is analogous to ECO.

Eventually we end up with a set of Place steps that provide a winning recipe. We choose the winner by reconnecting the previously disconnected down-cone into the best Placement branch. The flow can now complete and the results and recipe be published.

Scheme 3 can be used in conjunction with other schemes.

No schematic editor, no messy copying — and the recipe is retained.

Fixing the Plane While Flying

Can reconfiguration occur during flow execution with either scheme? Yes. Flows can be reconfigured and regenerated while executing. Some things to consider: If a flow step that has already run successfully receives a meaningful reconfiguration change (passes the barrier), it will be invalidated. This is correct behavior; if the user doesn't like it and wants to keep the current results, the step will need to be skipped/autoflowed. If a step is reconfigured while it's running, it will run to completion, generate results based on the previous config, and report an error status (FAILED) because inputs were not stable during execution. This is correct behavior, and the user will have to either rerun or skip/force validate.

When running the configuration and building the flow there is typically a master set creation phase S flow { my job sequence }. If the flow generation recreates the same step, as is often the case, the step's current status is preserved. Jobs that are valid remain valid. This is desirable behavior. Make sure you understand the magic behind this master set creation. If, after reconfiguration, a job step is missing from this set, that job will be automatically deleted from FlowTracer (though the job's files remain). This is expected behavior and provides an automatic way for steps to be removed from a flow. If this occurs while the job is running, then we end up with some undefined behavior that needs to be characterized and a remedy suggested. If we cycle through a deletion and a re-add, then the step being re-added will start in an invalid state and consequently invalidate the down-cone. A manual Force Validate or similar is required to escape this mess, which is best avoided.

Meta Flows

Once we can capture a recipe reliably, we can replay it at any time. We can use it as a baseline and vary certain parameters to look for further improvements. This can be done outside a normal project cycle; we can search for better recipes 24/7 and make use of spare compute cycles and available software licenses. The variation of parameters can, of course, be controlled by various algorithms; choose your favorite from GA, ML, PSO, Convex, etc. But let's not forget that running a chip design flow is expensive, so while we may have available resources, they're far from infinite. We want to run promising experiments, not futile ones.

We've already identified two key strategies — branching and pruning — that a designer can use to get a better result efficiently. Recall that branching enables us to experiment further on a known good result (e.g., multiple CTS runs from a good placement without rerunning the placement or its antecedents). We now need to think how this process can be automated. We'll assume that there are some values for each step that are known to be variable and where variation can give rise to different PPA results. Our search algorithm (parameter sweep) can introspect FlowTracer's graph and identify promising places to branch the flow with a new parameter

set. It can then choose the best options, according to its search strategy, and branch the flow from there. While it will likely take the usual metrics (congestion, utilization, negative slack) into consideration it can also assess the time it took to get to the current state. For example, a placement job result that took 10x longer than normal for a minor reduction in utilization may not be a promising place to run further experiments. FlowTracer can provide not only per-job metrics but also how long it took to get to a certain point in the flow.

This leads to the idea of automated pruning of the flow graph. In a deep search of the design space, we may have hundreds of alternatives (parameter choices) for each step. We expect that the master algorithm will have chosen the most promising one, but inevitably some poor choices can result. We can prune those branches, typically using a periodic job also known as a watchdog. This job runs every few minutes and introspects the flow graph, looking for branches that are unpromising. When it detects a "bad" branch — perhaps based on total (up-cone) runtime, current runtime, or design metrics — it can curtail that branch by stopping the running job or de-scheduling that job's down-cone.

Branch Reconvergence

We're often asked if FlowTracer can support the join of two or more up-cones. For example, we may have three different placement strategies for implementing power optimization and can choose the one that works the best (lowest power subject to other constraints), then connect that into the down-cone (routing) job. Yes, FlowTracer can do this — but we think you shouldn't.

If the up-cones have short runtimes and are reliable, then a simple join can work. Just define the last job in each up-cone as the antecedent to the joining job. The AD declaration works well for this. Then the join job gets to choose which input to use based on selection criteria. In practice, though, physical design steps are not reliable, nor are they short, and the run time is highly variable. We can handle the failed job case using the JOBSTATUS database, which allows job completion to be the criterion for dispatch of the join job rather than successful completion (VALID). However, we don't have an efficient scheme for handling wide-ranging runtimes of inputs to the join job; we must wait for the slowest job before we can decide. It's often the case that the slowest job yields the worst result, so we end up waiting for results that are rarely selected.

Our preferred approach to this problem is to avoid joins. Instead have each choice of placement strategies feed into separate and complete routing down-cones. This will allow the first to complete branch strategy (and possibly best) to get into placement immediately. You'll get results sooner. For the other branches, we wait for each one to complete the placement phase, then we analyze the results (in a watchdog), and if the results are better than the previous branch, we de-schedule/force stop that branch and allow the new (better) one to continue. The watchdog allows us to "vote" on the best approach as new results come in, but we're always pushing forwards as aggressively as we can. This seems like a better overall approach.

Meta Flow Execution

We need something at a higher level to manage design flows and their branches. This will be another FlowTracer instance. Jobs in this instance will be invocations of subordinate FlowTracer sessions that contain the actual flows. This meta-flow instance is where the subordinate flows are initiated, either in response to some optimization goal or a change in primary inputs (collaterals); it's effectively an automated design engine. We anticipate one such engine instance per block, and it should form the locus for optimization metrics on that block, but there's no reason why multiple blocks couldn't be optimized from a single instance.

Dashboard

A reasonable form for reporting will be a table with columns representing the flow steps and rows representing the flow instance or branch. Moving left to right on a row represents progress in a mostly linear flow. We get most of this data free from the Streaming Data Service but not all; the collation and representation problem will likely be eased if each job has, as properties, a branch identifier (some tag) and the job ID from which the branch was done. This will allow us to sort results by branch and correctly offset the column. For example, we may have many MpCTS job instances in a given FlowTracer session (one for each branch or trunk), and we sort into rows based on branch ID so that the cell next to left of MpCTS is Placement, and to the right perhaps Routing, and all are on the same branch. The correct column is selected by level field on the SDS record. Level is the depth in the flow, and it increments by two because the graph is bipartite (job-file-job-file…).

Automated Monitoring of Running Flows

FlowTracer is a client/server model. There can be many clients, and as such we can build monitoring tasks orthogonal to flow execution. Assuming we want to do this at the meta-flow level and in the meta-flow FlowTracer instance, we'll need to connect from the context of one FlowTracer session to another. This is feasible with the current vovsh rpc API, but special care is needed to assert the host/port name combination and future-proof the right tool version (maybe we have some legacy flows running with an older version of FlowTracer at some point). The utility vovboot can help with some of these environment issues.

Alternatively, we may want to leverage the newer REST API. For this to be done safely we'll need to manage passwords and use the TLS layer in HTTP (HTTPS). The use of a role account/faceless user with read-only permissions may be helpful. For TLS we need certificates. While FlowTracer can self-sign on boot-up, modern browsers are not cooperative. A more robust strategy might involve a master wildcard certificate for some useful subdomain (e.g., *.silicon.acme.com) would work for all instances in the DNS domain of silicon for the Acme company. The IT security team will be instrumental in enabling such a strategy. A more elaborate approach may involve automated signing through some chain of trust. This is feasible but complex, and will again require IT to provide such a trusted service.

Realizing a More Effective EDA Environment

Semiconductor companies face daunting challenges related to chip design and verification. These challenges include increased competition; larger, more complex designs; time-to-market pressures; and limited budgets for infrastructure and tools. Improvements in processor speed have slowed, causing organizations to look for new ways to improve efficiency.

Workload management is a crucial area for optimization. Even marginal improvements in workload throughput and resource utilization can drive significant productivity improvements. Altair can help organizations improve efficiency in six valuable ways:

Monitor, measure, and optimize workloads. Administrators can improve sharing policies and express resource requirements more precisely by monitoring license and resource allocations with Altair® Monitor™. Improved monitoring leads to better utilization and helps ensure critical project deadlines are met.

Implement a high-throughput scheduler. By leveraging Altair® Accelerator™ and its high-throughput, event-based scheduler, design teams realize reduced scheduling latency and faster job turnaround, leading to better productivity. By running jobs faster and more efficiently, they also gain simulation capacity leading to higher quality, more thoroughly tested products.

Employ license-first scheduling. EDA tool licenses are the most valuable commodity in most verification environments. However, maximizing license utilization is hard. Even sophisticated organizations may achieve only 50-70% license utilization. By using Monitor and Altair® Allocator™ with advanced license matching techniques, sites can dramatically improve license utilization and throughput — in some cases achieving 90% utilization or higher.

Map and optimize design flows and simulations. While multi-step workflows are common in EDA environments, many schedulers support only rudimentary job dependency management. FlowTracer captures flows, provides visualization, identifies inherent parallelism opportunities to optimize resource usage, and enables collaboration around workflow execution.

Improve hardware emulation efficiency. Most EDA firms use workload management tools, but the most expensive hardware assets in the data center, hardware emulators, are usually managed manually. Altair® Hero™ brings advanced scheduling and resource sharing to leading hardware emulation platforms. This provides greater flexibility and control and enables organizations to get more productivity from their hardware emulation investments.

Leverage the cloud to augment on-premises resources. While using cloud resources during busy periods sounds like a good idea, the devil is in the details. Accelerator, Monitor, and Allocator provide a complete solution for hybrid cloud deployments. Organizations can easily tap their choice of clouds to improve capacity with efficient cloud auto-scaling and policy-based software license allocation.

Learn more about FlowTracer and more industry-leading EDA workload management solutions.