INTEGRATING OVERLAPPING SYSTEMS – ALTAIR® FLOWTRACER™ AND BAZEL

Stuart Taylor and Nandish Gadgimath – Altair / December 8, 2022

{Fast, Correct, In Control} – Choose Three

Bazel and Altair® FlowTracer™ Integration Strategy

Overlapping systems can be challenging to integrate; both FlowTracer and the open-source Bazel build-and-test tool have build-system functionality. For performance, both are client-server. Both are capable of remote workflow execution, scheduling, and ordering. We can learn from previous FlowTracer integrations with Make-based flows and from prior large-scale FlowTracer deployments. In the FlowTracer/Make integrations we see FlowTracer running at a higher level with its jobs being calls to make. These can be aggregate (make all), or for more fine-grained control some targets might be broken out, resulting in separate FlowTracer jobs for each discrete make target.

Surviving Capabilities

With overlap between two systems, it's useful to decide which features of each should survive in the final combined system.

- Bazel strengths: As a build language itself, it mixes both declarative and imperative styles; it's more modern than Tcl, Perl, etc.; and it's more targeted at the build problem than pure Python.
- Bazel has claimed integration with GCP using remote execution, and being able to run tasks on a freshly spun-up instance allows us to consider scaling/cloud integration with lower effort.
- FlowTracer has a proven visualization methodology for flows in the physical design space. We want to carry this forward.
- FlowTracer leaves the designer in control of the flow. Jobs can be stopped, restarted, skipped, and run without perfect dependencies.

The Jobs

In most recent FlowTracer physical design implementations, a particular flow step is not represented by a single job but as a set of closely related jobs or tasks. The jobs are grouped into a set, and that set is the main object that is run/stopped/skipped.

We propose using Bazel to implement each step of the physical design sequence, with FlowTracer managing the overarching flow. To enable steps to run under designer control, each step is implemented as a separate Bazel workspace. Dependencies across Bazel steps are managed by FlowTracer and thus follow FlowTracer's more permissive constraints.

When running a Bazel step, FlowTracer may run the step locally or on a remote host, and the Bazel step itself may run in situ or make calls to other infrastructure (GCP). Bazel steps may be run in parallel without restrictions since they are essentially independent of each other. A scheduled or invalid job in FlowTracer will not have an active Bazel process. A running job in FlowTracer will have an active Bazel process. A done or failed job in FlowTracer should not have an active Bazel process, and special steps may be needed to shut down the Bazel process. This can be achieved either within the Bazel step itself (preferred) or via FlowTracer's agent (vovtasker) clean-up epilogue.

FlowTracer job control can send a complex sequence of signals to the job. While this could be the Bazel server process (assumed to be backgrounded and attached to the vovtasker process), it is probably more flexible to have a bash wrapper that translates the signals into specific Bazel commands. The wrapper can organize an orderly shutdown of the Bazel server, allowing Bazel to shut down remote processes too.

It is expected that code (a rule?) is available in each Bazel step that will, at runtime, (re)declare the inputs and outputs of the Bazel job. Conventionally, these are files in a POSIX file system, but this seems too limited in a cloud world where object stores for persistent data seem more natural. Work is ongoing, providing an easy user interface to arbitrary object stores, but the current FlowTracer data model supports non-POSIX objects already. While such a facility is not needed to run an object-store-based FlowTracer flow, it is useful because it can catch the case where an object is updated via a third party (perhaps a rogue Bazel step) outside of FlowTracer's direct knowledge. That is life doing real chip projects.

Configuration

A typical PnR flow on a DSM process node will run to 10-20k parameters, perhaps expressed as key-value pairs. Managing that number of parameters on a large project run over multiple sites is non-trivial, and most customers have their own scheme. FlowTracer is necessarily agnostic in this respect and has been proven to work with many proprietary schemes. It's reasonable to assume that either a proprietary scheme can be used here, or perhaps one directly implemented in Bazel. Some thought is needed for nonlinear flows, which are common in the early stages of design. Here we find that with a bad result from a certain step, we need to rerun that (and perhaps an earlier step) with a different parameter set. What is important is that regenerating the parameter set doesn't invalidate steps that are not semantically impacted by the parameter change. FlowTracer has facilities for squashing such unwanted rebuilds (change propagation control), but the underlying system needs to be amenable.

Flow Definition

FlowTracer doesn't know anything about chip design; it needs to be told. Intrinsically, neither does Bazel. A combination of Bazel rules and configuration parameters, however, do know; they represent the knowledge of the methodologist and the designer's intent. However, FlowTracer controls the overall job sequencing, so how can we tell FlowTracer about the intent in a scalable manner? A common approach is to use a custom program, often called a job sequencer. This contains the know-how of the methodologists and uses the designer-intended configuration parameters. While proven successful, this approach doesn't seem appropriate here. We're using the isolated Bazel workspaces to embed the methodology, so also putting the methodology into the job sequencer seems like duplication. It may be possible to share a common set of rules across the disparate workspaces and the central sequencer, but it seems tricky. Is there another approach? We think so — that of using a depend-only or prototyping mode.

Assuming we have an initial configuration that is unique for this flow instance — minimally a pointer to the initial data and chip project parameters — we can now run each Bazel step in a depend-only mode. The assumption here is that each Bazel step has an understanding that when run in a depend-only mode, it merely runs a rule that establishes what its inputs and outputs will be based on configuration parameters. For example, a CTS step might define its inputs as coming from the placement step and some clock constraints. In some cases, a step may not be required for a given flow, so the step just exits without declaring inputs or outputs.

With all steps run in this manner, FlowTracer can assemble an initial view of the flow. This step is also used to prove that the flow is realizable. Flow configuration and methodology errors can be exposed, including the common case of missing outputs and the rarer case of two jobs writing to the same file. Conceptually the process is analogous to "make -n" but with ramifications. Some care is needed for multiple instances of the same step; this often shows up for timing corners where there might be dozens of different STA

runs, each for a different corner, but a common master template step. In this case an early part of the configuration flow will need to create a Bazel workspace using a common rules file. We've seen symlinks used here, and a naming scheme to emphasis the template-instance nature of these steps.

A Simple Example

We can define our flow steps and other key parameters using environment variables:

```
#!/bin/sh -f
echo "Setting Workspace & Design Info"
export WS="/Users/taylor/Downloads/BZL+FT"
export VOV_FDL_ONLY=1
export DESIGN_NAME="TSMC40NM"
export FLOW_NAME="apr_fc"
stages=("import_design" "floorplan" "placement" "place_opt" "CTS" "CTS_opt" "route" "route_opt"
"chip_finish" "signoff")
```

Our flow steps may consist of shell scripts, Bazel rules, and a handful of FlowTracer directives. Here's an example of a shell-centric step:

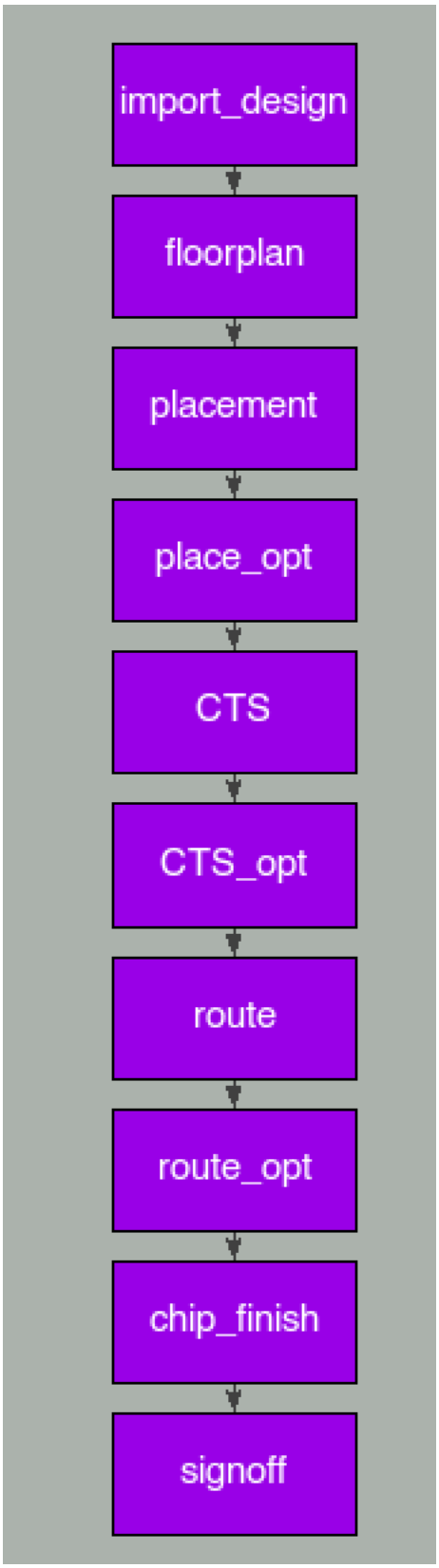
```
#!/bin/sh
########## Part 1: The FlowTracer Section.
VovFdl N CTS
VovFdl T vov bazel run //apr_fc/CTS:CTS_stage
VovResources "RAM/10"
VovInput /Users/taylor/Downloads/BZL+FT/apr_fc/CTS/WORKSPACE
/Users/taylor/Downloads/BZL+FT/apr_fc/CTS/BUILD /Users/taylor/Downloads/BZL+FT/INPUTS/CTS.input
VovInput /Users/taylor/Downloads/BZL+FT/OUTPUTS/place_opt.output
# some conditional IO
if [ $?USE_MP_CTS ]; then
  VovInput /Users/taylor/Downloads/BZL+FT/clock_grid_possible_hookup
     VovOutput /Users/taylor/Downloads/BZL+FT/OUTPUTS/clock_grid_actual_hookup
endif
VovOutput /Users/taylor/Downloads/BZL+FT/OUTPUTS/CTS.output
# if we're prototyping the flow we exit on the next line
if ( $?VOV_FDL_ONLY ) exit
########### Part 2: now do the job for real..
DIR="$WS/OUTPUTS"
if [ ! -d "$DIR" ]; then
mkdir "$DIR"
fi
if [ $USE_MP_CTS ];then
   cts -mpcts -roots clock_grid_possible_hookup \
         -i $WS/INPUTS/CTS.input -o $WS/OUTPUTS/CTS.output
else
    cts -i $WS/INPUTS/CTS.input -o $WS/OUTPUTS/CTS.output
endif
echo "CTS is done"
```

When VOV_FDL_ONLY is set we get the prototyping behavior; when run, the standard output will contain records for describing the flow. We get to specify how the script will be run (VovFDL T); in this case we're expecting to be called via Bazel. We can also specify resource requirements that can be picked up by Bazel's dispatcher or a batch system. Most importantly, we get to define our inputs and outputs. The orange text shows how conditional I/O can be controlled. FlowTracer uses these I/O directives to stitch a flow together. This is just for one step; in practice we will run each step as defined in the environment variable stages, in prototyping mode, and collect all the output together. We can use Bazel to do this:

```
bazel run //:gen_fdl (This generates fdl files)
```

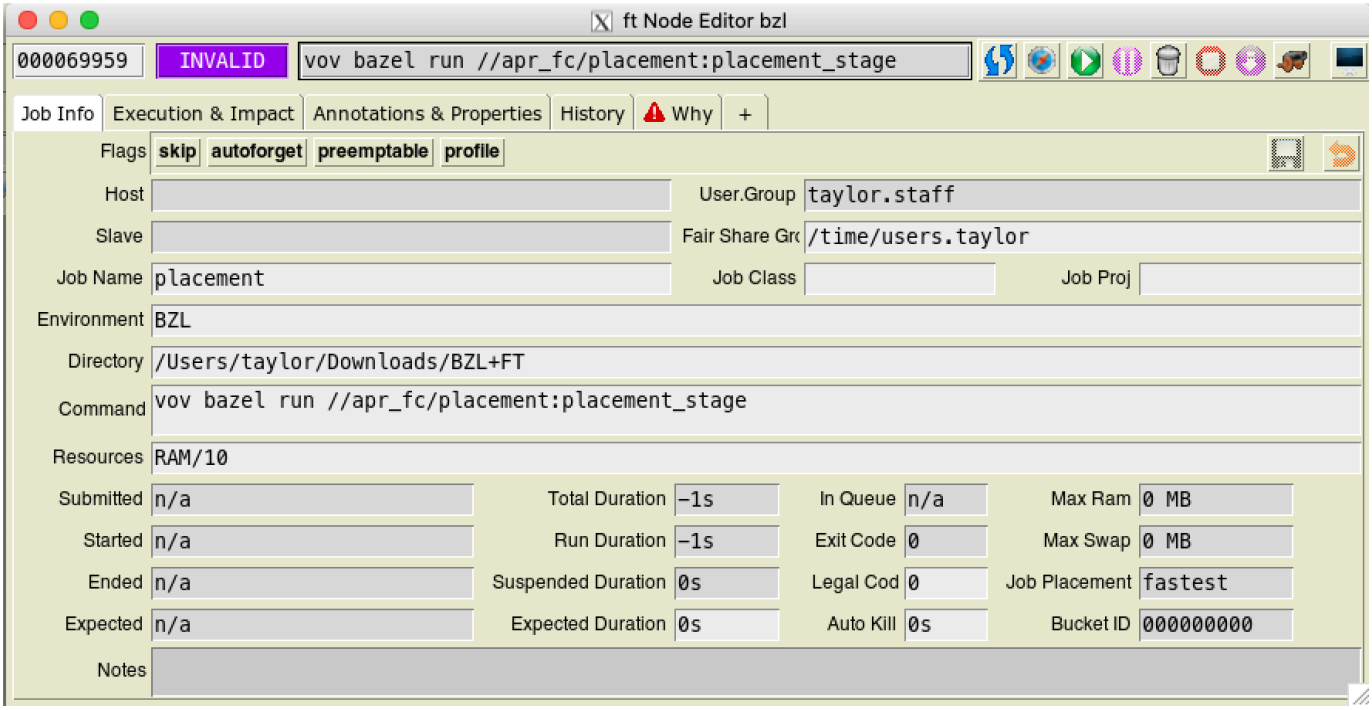
Now we have our flow description we can load it into FlowTracer and initiate the interactive GUI window vovconsole. Again, we can use Bazel as the entry point:

```
bazel run //:MergeAndBuild
```

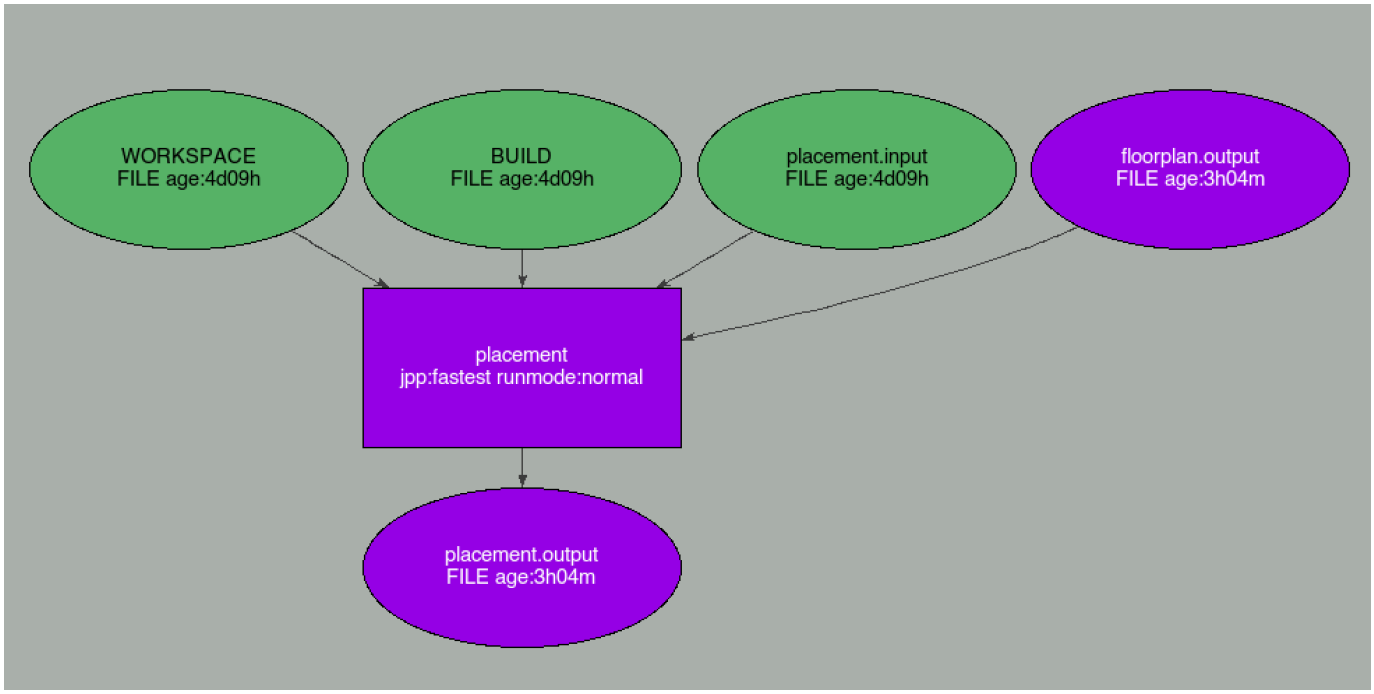


FlowTracer has created the jobs and the assembled the job sequence by hooking up the I/O.

Let's look at the placement job:

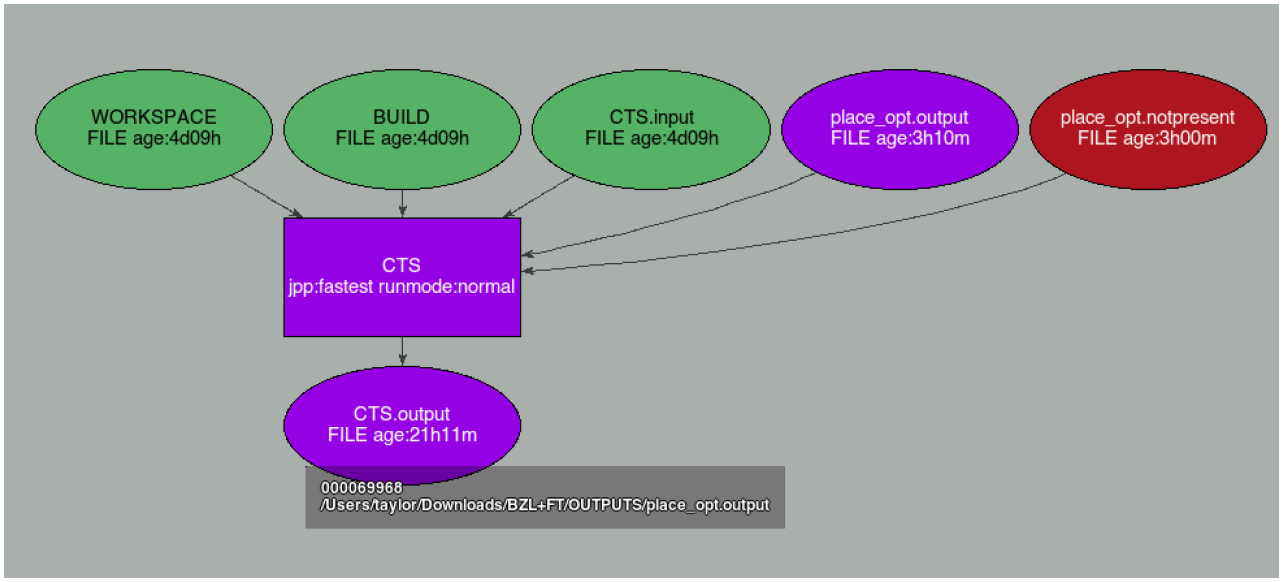


We can see that the job is a call to Bazel with a separate workspace (placement) and rule placement_stage. Selecting just that job and turning on the file view option we can see that job's inputs and outputs. Some may look familiar to Bazel experts.

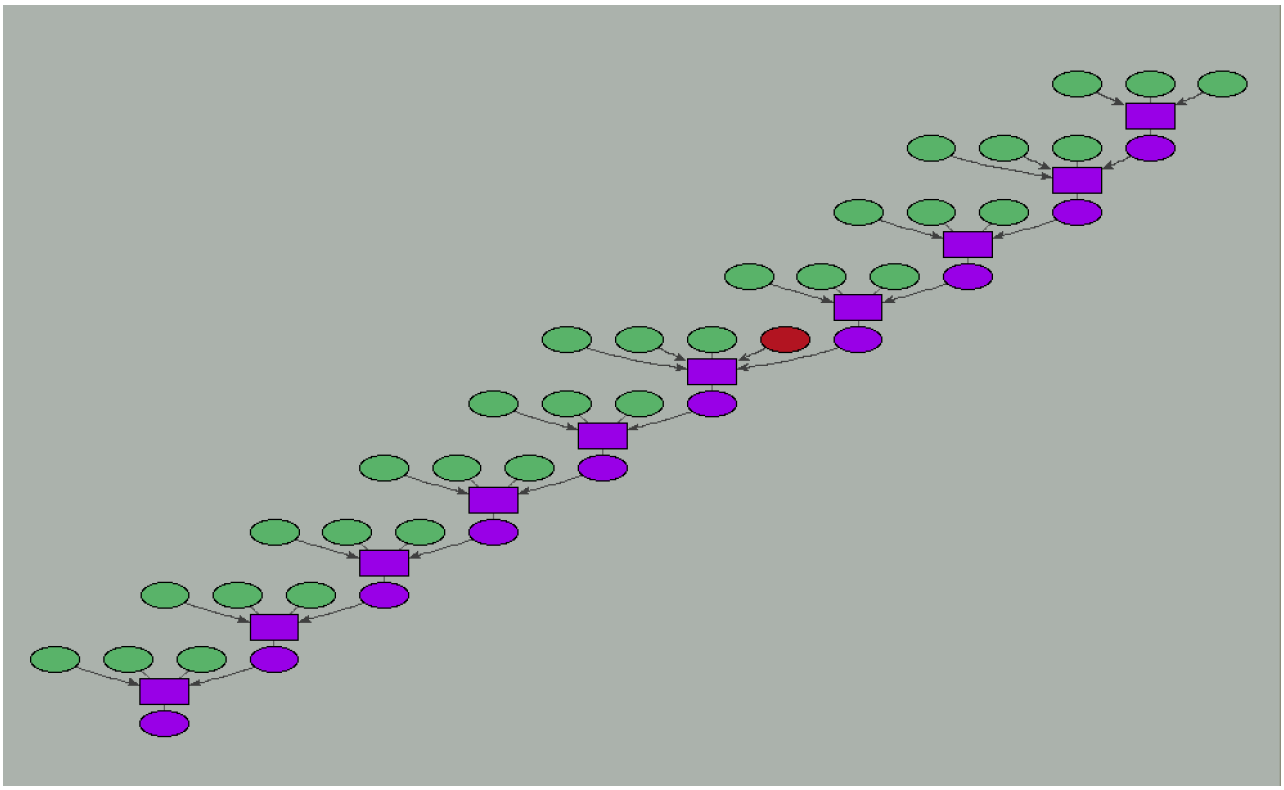


Next, we can look at a typical fault with flow definitions. We often have many configuration variables, and these have interdependencies and impact the flow data. When an inconsistent set of parameters is chosen, we can end up with missing files or jobs that both write to the same location. With build-as-you-go systems (e.g., Make, Bazel, and many others), you only see these problems after the flow is run; it runs up to the point of the problem, then errors out. With a quick software build that isn't a huge problem, but if a step like placement takes 14 hours and there's a problem with the collaterals for CTS, we'll only find out perhaps a day into the flow.

In contrast, FlowTracer assembles the entire flow and we can see errors before setting out. Here's a broken CTS step:



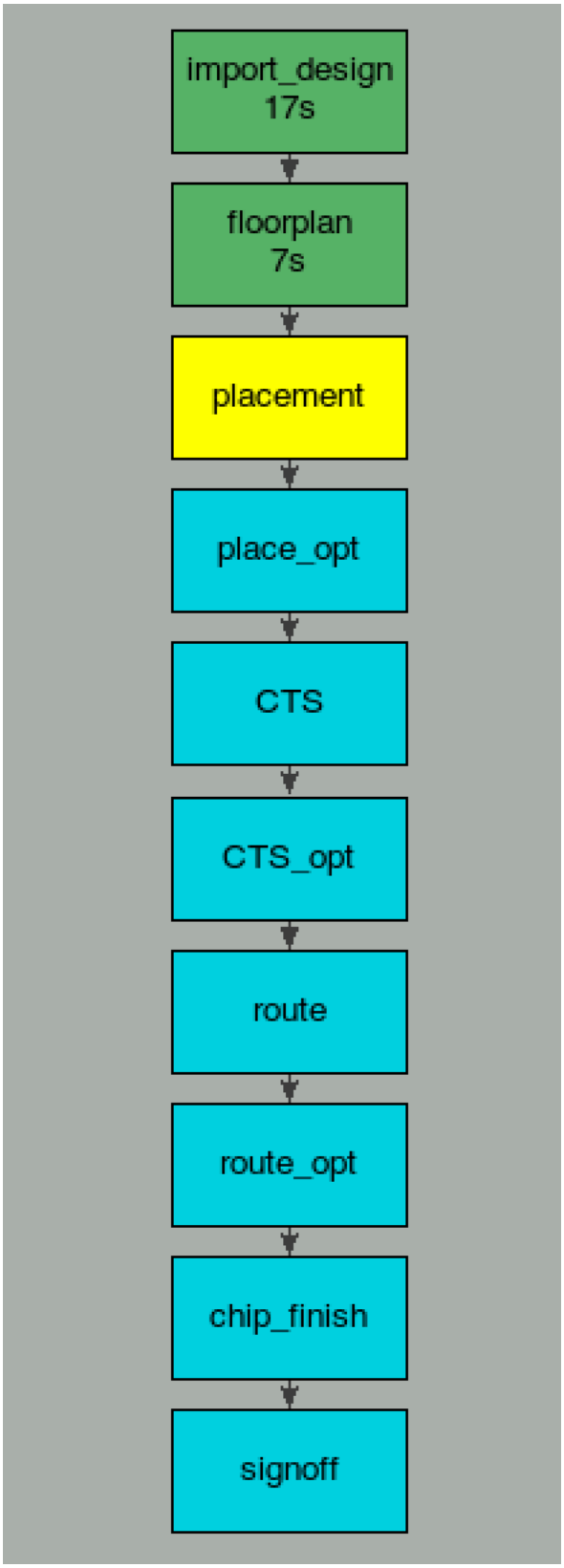
The missing file is shown in red (top right). We can also view the entire graph:



Or just files on their own:

Even with a few thousand files, it's easy to see missing ones (though you'll need to hover over to get the text and the actual filename). This enables users to do a quick triage of their flow before anything gets launched for real. Many engineering days are saved. It's entirely possible to write programmatic consistency checks too, and have them be part of the Bazel load-flow rules.

Multiple writes to a single file are also detected (black rectangle for the job not shown here), and this check is performed both at flow build and runtime. This kind of error in build-as-you-go (make) is very hard to detect, and in practice several customers have discovered bugs in their make flows when porting part of them to FlowTracer. Once the flow definition problems are resolved, we can kick off the flow. Green is done, yellow is running, and cyan is queued.


import_design
17s
floorplan
7s
placement
place_opt
CTS
CTS_opt
route
route_opt
chip_finish
signoff

A Complete Suite of EDA Workload Management Solutions

Altair provides a complete suite of EDA workload management solutions to help design firms implement the six optimizations described above. Customers can start using Accelerator for high-throughput EDA workload scheduling and gradually introduce additional workload management capabilities as requirements evolve.

- Altair Accelerator high-performance job scheduler
- Altair Monitor for license monitoring and management
- Altair® Allocator™ for multi-site license allocation
- Altair Accelerator Plus high-performance hierarchical scheduler
- Altair® Hero™ high-performance scheduler for hardware emulation
- Altair FlowTracer to develop and execute design flows

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 budget 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:

1. **Monitor, measure, and optimize workloads.** Administrators can improve sharing policies and express resource requirements more precisely by monitoring license and resource allocations with Monitor. Improved monitoring leads to better utilization and helps ensure that critical project deadlines are met.
2. **Implement a high-throughput scheduler.** By leveraging 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 and more thoroughly tested products.
3. **Employ license-first scheduling.** EDA tool licenses are the most valuable commodity in most verification environments, and maximizing license utilization is hard. Even sophisticated organizations may achieve only 50-70% license utilization. By using Monitor and Allocator with advanced license matching techniques, sites can dramatically improve license utilization and throughput, in some cases achieving 90% utilization or higher.
4. **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.
5. **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. 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.
6. **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

To learn more about FlowTracer and other industry-leading solutions for EDA workload management, visit https://www.altair.com/flowtracer/.