fud2: A Framework for Making Build Tool Orchestrators

Jeremy Ku-Benjet
Cornell University
USA

Adrian Sampson
Cornell University
USA

ABSTRACT

Hardware toolchains are complicated. To manage with this complexity, hardware engineers have both invented hardware-specific build systems and repurposed software build tools for hardware. Sometimes, however, a full-fledged build system is too heavyweight: when dealing with small-scale projects, we see a need for a simpler tool that does not require writing build scripts. The need is analogous to a compiler driver in software compilation: a command-line tool that orchestrates various components to carry out a self-contained task. fud2 is an extensible compiler driver for hardware tooling. It is the first compiler driver framework we are aware of that provides a declarative way to integrate new backend tools. fud2 is the primary way to compile and run code in the Calyx ecosystem.

1 INTRODUCTION

Imagine you are writing a program in a high-level accelerator design language (ADL) such as Dahlia [8]. To run the program, you need to (1) compile the Dahlia source code to the Calyx intermediate language, (2) run the Calyx compiler to produce Verilog, (3) use an RTL simulator like Verilator to produce a simulation executable, and finally (4) execute the simulation on input data you specify.

The purpose of fud2[1] is to provide a compiler driver that can run all 4 steps in a single command. fud2 avoids hard-coding a fixed set of task pipelines; instead, it provides a framework that can incorporate arbitrary transformation steps. This extensibility means that fud2 resembles a build system, with the crucial difference that no scripting is required: a single, high-level command suffices to invoke chains of tools.

1.1 Overview

The fud2 framework revolves around two main concepts: *states* and *ops.* Informally, a state is a type of file and an op is a command that can transform files between states. Each op can have an arbitrary number of inputs and outputs, each labeled with a state. For example, states include Dahlia, Calyx, Verilog, simulation executables, and simulation results; ops include Dahlia compilation (whose input is the Dahlia state and output is the Calyx state) and Verilator compilation (whose input is Verilator and output is a simulation executable).

fud2 takes requests to transform files from initial states to final states. It computes a sequence of ops that can fulfill the request, called a *plan.* Figure 1 illustrates a plan for executing a Dahlia program. Finally, fud2 executes the plan by generating and running a Ninja file [1].

[1]https://github.com/calyxir/calyx/tree/main/fud2

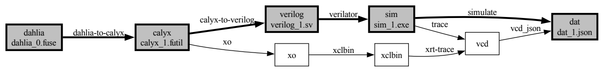

Figure 1: Some of the states and ops implemented in fud2, with a plan highlighted for executing a Dahlia program. Boxes are states and arrows are ops. Shaded ops indicate the states that are used in the plan. This plan uses Verilator simulation [11]. An alternate route (not shaded) can use AMD's XRT stack for simulation or on-FPGA execution [3].

The hard part is computing the plan. Most of this position paper focuses on that problem (Section 2). We also briefly discuss fud2's implementation and its context.

1.2 Alternatives

Classic software compiler drivers, such as the `clang` and `gcc` executables, also orchestrate a chain of tools for each job: the preprocessor, compiler, assembler, and linker. In fud2 terminology, they make a small number of predetermined plans involving these ops. A hardware compiler driver must manage a larger number of tools and potential plans. We also need this compiler driver to be extensible: for example, it should be possible to add new ADLs and new RTL simulators.

On the other hand, build tools (from Make to Bazel and Buck2) are extensible and designed to handle any number of build actions. However, the user must write a build script to specify the specific series of actions for a certain build. When there are alternative routes to the same outcome (such as the Verilator and AMD alternatives in Figure 1), the build scripts resolves this ambiguity. fud2's goal is to simplify the interface by automatically deriving a plan and involving the user only when they need to pick a specific alternative to a step in the derived plan.

Existing hardware tool orchestrators such as SiliconCompiler [9] and Hammer [6] are closer to build tools than to compiler drivers; they use explicit scripting and are appropriate for large projects. mflowgen [4] in particular has constructs similar to fud2's ops, except that they accept and return files instead of states. Developers use these constructs to form dependency graphs, which mflowgen lowers to a lower-level build system. An important difference in fud2 is that ops are *typed:* states describe what kinds of files are legal inputs and outputs for a given op. The typed representation allows fud2 to automatically construct plans.

```
defop calyx_to_verilog(calyx_prog: calyx_state) >> verilog_prog: verilog_state {
 // Retrieve variables from a configuration file to get binary locations.
 let calyx_base = config("calyx.base");
 let calyx_exe = config_or("calyx.exe", `${calyx_base}/target/debug/calyx`);
 let args = config_or("calyx.args", "");
 shell(`${calyx_exe} -l ${calyx_base} -b verilog ${args} ${calyx_prog} > ${verilog_prog}`);
}
```

Figure 2: Definition of the "calyx-to-verilog" op using Rhai.

2 FINDING PLANS

2.1 Ops and States

We now define states, ops, and plans precisely. We assume two opaque sets of names: ID is the set of identifiers (which name states and ops), and FILENAME to be the set of all possible filenames. Let $\mathcal{M}(A)$ denote the set of all multisets containing elements of A.

A configuration for fud2 defines all the available states and ops:

- A *state* is simply a name, $s \in \text{ID}$. Let STATES be the set of states in the configuration.
- An *op* is a tagged pair of input to output state multisets, $(t, I, O) \in \text{ID} \times \mathcal{M}(\text{STATES}) \times \mathcal{M}(\text{STATES})$. Call the set of all ops for a fud2 configuration OPS.

To request a plan, the user specifies the input files' states, the desired output files' states, and a (possibly empty) set of ops that must be included in the resulting plan. The latter component lets users optionally resolve ambiguities: for example, they might request that a simulation plan go *through* the Verilator op instead of the XRT ops.

Formally, a *request* is a tuple $r = (R_{\text{through}}, R_{\text{in}}, R_{\text{out}}) \in \mathcal{P}(\text{OPS}) \times \mathcal{M}(\text{STATES}) \times \mathcal{M}(\text{STATES})$. R_{through} is the set of required ops; R_{in} and R_{out} are the states of the input and output files.

In response, fud2 creates a plan to fulfill the request. A *plan* $P \subseteq \text{OPS}$ is a selection of the available ops in the configuration. A plan fulfills a request iff both of these conditions are met:

(1) The required ops are present, i.e., $R_{\text{through}} \subseteq P$.
(2) The ops suffice to transform the inputs to the outputs. Formally, let $F(X) = X \cup \bigcup_{(t,I,O) \in P, I \in X} O$. This operator computes the set of states that that the plan P can produce given inputs in the states X. We require that R_{out} is within the reflexive transitive closure of F over STATES applied to the input states: i.e., $R_{\text{out}} \subseteq \bigcup_{i=0}^{\infty} F^i(R_{\text{in}})$, where $F^0(X) = X$.

2.2 Finding a Plan

Requirement (2) above suggests how to find plans: repeatedly apply F to a "created states" set X, starting with R_{in}. If F adds states to X using an op o, add that op to P. We can iterate this process to a fixed point, at which point no more ops can possibly run using the inputs in X. Finally, we check the two criteria above: if they are satisfied, P is a valid plan; otherwise, fud2 reports failure. Algorithm 1 illustrates this process in more detail.

However, this algorithm is an over-approximation: it can include more ops than necessary. There are two ways for this to happen:

(1) If two ops provide the same available input states, Algorithm 1 will include both.

Algorithm 1 Constructing an initial plan, P.

```
$X \leftarrow R_{\text{in}}$
loop
    modified $\leftarrow$ false
    for op $= (t, I, O) \in$ OPS do
        if $I \subseteq X \land \text{op} \notin P$ then
            $X \leftarrow X \cup O$
            $P \leftarrow P \cup \{\text{op}\}$
            modified $\leftarrow$ true
    if $\neg$modified then
        break
```

Algorithm 2 Pruning P to produce the final plan.

```
loop
    modified $\leftarrow$ false
    for op $= (t, I, O) \in P$ do
        if $\text{op} \in P \land \forall o \in O, o \notin R_{\text{out}} \cup \{I' \mid (t', I', O') \in R\} \lor$
          $\exists \text{op}' = (t', I', O') \in P, \text{op}' \neq \text{op}, o \in O'$ then
            $P \leftarrow P - \{\text{op}\}$
            modified $\leftarrow$ true
    if $\neg$modified then
        break
```

(2) If an op's outputs are unused, Algorithm 1 will still include it.

These ops are unnecessary to fulfill the plan (unless they are required by R_{through}).

The next step in fud2's planner is to prune unnecessary ops. To address (1) all but one of the ops with identical output states are removed. The preserved op is chosen arbitrarily. To address (2), an op with all its outputs unused will be removed. Algorithm 2 illustrates this pruning process.

2.3 Assigning Filenames to States

To run a plan, fud2 must generate filenames for all the intermediate states in the plan: i.e., for every $o \in P$, we need to determine the filenames for each of o's inputs and outputs so we can execute o's commands. If an op's input state is in R_{in} or its output state is in R_{out}, then its filename comes from the user's original request. fud2 generates fresh filenames for each other output state for every op in P. Then, it assigns each op's input filenames to match one of these generated output files for the appropriate state.

For that final step, if there are multiple output files in the same state, fud2 makes an arbitrary choice. This arbitrariness is unavoidable without prompting the user for more information: if two ops output the same state (or an op outputs a state in R_{in}) used by an input, fud2 does not know which file should be given to the input. As users of fud2 have no control over file naming, this can cause some plans to be unobtainable using Algorithm 1 and Algorithm 2.

For example, a plan may have two ops $\text{op}_1, \text{op}_2 \in R_{\text{through}}$ with the same outputs and a third op_3 whose inputs are the outputs of op_1 and op_2. Filename assignment may choose to only use outputs from op_1 as inputs to op_3 against the user's wishes. And the user has no recourse.

This is rare in practice as plans tend to be sparse graphs. We found one case arbitrary file naming caused problem for Calyx: extending fud2 to support iteratively applying an op to it's inputs. Given an op whose input states equal its output states, a user may want to apply this op to an input file, creating an output, and then apply that same op to that output. In terms of the above example, $\text{op}_1 = \text{op}_2 = \text{op}_3$. As a solution, fud2 allows users to export and edit plans to fix incorrect file naming, manually resolving the ambiguity.

2.4 Time Complexity

Algorithms 1 and 2 dominate the time complexity. Their outer loops can be bound by $|\textsc{ops}|$ iterations as there are only $|\textsc{ops}|$ ops to add or remove from P. The inner checks are applied to every op and require iterating over an op's states to check set membership in and update X. This leads to a complexity of $O\left(|\textsc{ops}|^2\,|\textsc{states}|\right)$. As both $|\textsc{ops}|$ and $|\textsc{states}|$ are small in practice (less than 100, in the case of Calyx's fud2 configuration), planning is fast in practice.

3 IMPLEMENTATION

Ops are specified using a syntax extension[2] to Rhai[2]. Rhai supports modules, functions, and scripting which can be used to share behavior between ops.

Many ops allow for users to set *config* variables used by ops. These are analogous to variables used to configure a build in a build system like Make[10]. For example, Figure 2 uses the `calyx.exe` variable to select a version of the Calyx compiler. These variables can passed on the command line or collected into a TOML[12] file.

Plans are not directly run by fud2 but instead lowered to Ninja[1] build files and executed using Ninja. This allows fud2 to boast parallel and incremental jobs. Plan finding is implemented in Rust.

4 FUTURE WORK

Sometimes an op may want to send information, for example a compiler flag, to a future op. Currently passing information between ops requires creating a new state which is an input to a new op, effectively encoding the data in the state name. That can lead to many similar ops and states. By *tagging* states with data, it may be possible to alleviate this. If ops can change the states they return based on this data, it can allow for more complicated dependency models akin to Shake[7] or Pluto[5].

There is no built in way to resolve incorrect file naming: the user must export and manually edit the generated plan. Instead fud2 could detect when multiple states could be used as an input and prompt the user to choose one.

Users may not want to install the dependencies for all ops, but as plans are generated automatically and can change chaotically in response to user requests, it can be hard to know what to install. Adding dependency tracking to fud2's op specification could alleviate this.

[2]https://docs.calyxir.org/running-calyx/fud2/high-level-rhai.html

REFERENCES

[1] 2017. *Ninja build system homepage.* https://ninja-build.org
[2] 2021. *Rhai homepage.* https://rhai.rs/
[3] AMD. 2025. Xilinx Runtime (XRT) 2025.2. https://xilinx.github.io/XRT/2025.2/html/index.html.
[4] Alex Carsello, James Thomas, Ankita Nayak, Po-Han Chen, Mark Horowitz, Priyanka Raina, and Christopher Torng. 2022. mflowgen: a modular flow generator and ecosystem for community-driven physical design: invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1339–1342. https://doi.org/10.1145/3489517.3530633
[5] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A sound and optimal incremental build system with dynamic dependencies. *SIGPLAN Not.* 50, 10 (Oct. 2015), 89–106. https://doi.org/10.1145/2858965.2814316
[6] Harrison Liew, Daniel Grubb, John Wright, Colin Schmidt, Nayiri Krzysztofowicz, Adam Izraelevitz, Edward Wang, Krste Asanović, Jonathan Bachrach, and Borivoje Nikolić. 2022. Hammer: a modular and reusable physical design flow tool: invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1335–1338. https://doi.org/10.1145/3489517.3530672
[7] Neil Mitchell. 2012. Shake before building: replacing make with haskell. *SIGPLAN Not.* 47, 9 (Sept. 2012), 55–66. https://doi.org/10.1145/2398856.2364538
[8] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712
[9] Andreas Olofsson, William Ransohoff, and Noah Moroze. 2022. A Distributed Approach to Silicon Compilation: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California). 1343–1346.
[10] Paul Smith. 2026. Make. https://www.gnu.org/software/make/.
[11] Wilson Snyder. 2026. Verilator. https://www.veripool.org/verilator/.
[12] Martin Tournoij, Pradyun Gedam, and Tom Preston-Werner. 2026. TOML.