Invited: A Distributed Approach to Silicon Compilation

Andreas Olofsson, William Ransohoff, and Noah Moroze
Zero ASIC, Cambridge, MA, USA
{andreas,will,noah}@zeroasic.com

ABSTRACT

Hardware specialization for the long tail of future energy constrained edge applications will require reducing design costs by orders of magnitude. In this work, we take a distributed approach to hardware compilation, with the goal of creating infrastructure that scales to thousands of developers and millions of servers. Technical contributions in this work include (i) a standardized hardware build system manifest, (ii) a light-weight flowgraph based programming model, (iii) a client/server execution model, and (iv) a provenance tracking system for distributed development. These ideas have been reduced to practice in SiliconCompiler, an open source build system that demonstrates an order of magnitude compilation speed up on multiple designs and PDKs compared to single threaded build systems.

1 INTRODUCTION

With Dennard scaling long gone and Moore's Law stagnating, many believe that hardware specialization may be our best option for extending the current trajectory of exponential year over year improvements in electronics [5]. This path is currently blocked by the high costs of chip design, which can exceed $100M and require hundreds of person-years per project. Long gone are the days of the "tall thin designer" that could claim expertise in all aspects of design. Today's advanced chip designs are made of large teams with specialized skills in circuits, floorplanning, packaging, EDA tools, DTCO, STA, DFT, DFM, LEC, EMI, SI, CTS, ESD, OCV, and CDC. Extreme post-Moore hardware specialization will require abstracting away all of these skills while reducing the cost and time barriers to the level of software.

A core challenge in modern SoC design is the numerical complexity of optimizing billions of inter-connected physical devices for power, cost, area, and speed while ensuring that the circuit operates reliably under all possible temperature, voltage, and process conditions. Tackling these types of NP-hard physical design optimization challenges has been a core focus of the semiconductor industry over the last 50 years. Due to this complexity, the effort required to create hardware designs can be many orders of magnitude higher than software. Closing this enormous gap will require breakthroughs in distributed algorithms, programming frameworks, and runtimes that enable efficient use of distributed warehouse-scale compute infrastructure throughout the hardware compilation pipeline.

A second fundamental challenge of hardware compilation comes from the complexity of optimizing for physical goals and scenarios that can't be easily described mathematically. As an example, the fields of robotics and autonomous driving have shown that taking humans out of the loop gets exponentially harder with increasing quality requirements. Developing an "intelligent" hardware compiler that can completely replicate the millions of engineering hours spent yearly on physical design of SoCs is a once in a generation grand challenge that requires advances in compilers, algorithms, models, design, high performance computing, and machine learning. The scope and breadth of these challenges suggests the need for a global collaborative effort similar to what we have seen within Linux, GCC, LLVM, the human genome project, and Python.

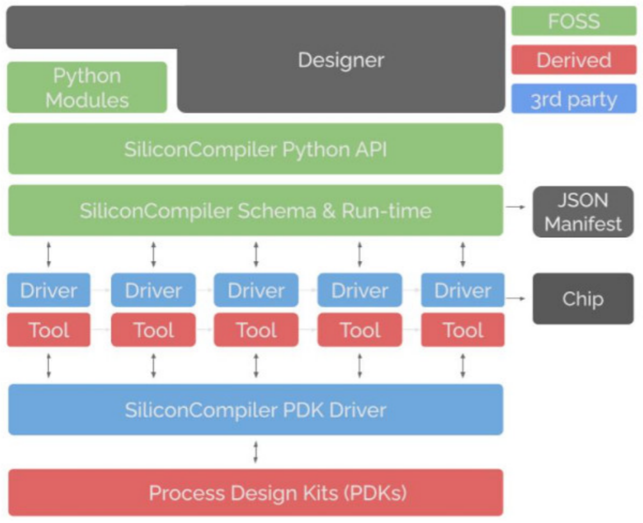
Designer
FOSS
Derived
3rd party
Python Modules
SiliconCompiler Python API
SiliconCompiler Schema & Run-time
JSON Manifest
Driver
Driver
Driver
Driver
Driver
Tool
Tool
Tool
Tool
Tool
Chip
SiliconCompiler PDK Driver
Process Design Kits (PDKs)

Figure 1: SiliconCompiler Architecture

The SiliconCompiler project illustrated in Fig. 1 is an open source build system that aims to address these fundamental challenges and facilitate the development and deployment of fully automated hardware compilers. It achieves this through key design decisions that enable distributed development and distributed execution of compilation flows.

2 RELATED WORK

Since the beginning of Moore's Law, the design industry has worked tirelessly to ensure that designer productivity kept pace with transistor density doubling every two years. One of the earliest attempts at automating SoC design was the original "silicon compiler" which automated the assembly of parametrized building blocks through customized software [6]. These early compilers were created on a per domain basis and were too limited in scope to gain widespread adoption. Still, many ideas from these "compilers" remain relevant today, as shown by the DARPA-funded Epiphany-V design which demonstrated assembly and tapeout of a 4.5B transistor chip in 16nm by a single designer on a single server [9].

The dominant SoC design process over the last 40 years has been based on mapping (synthesis) of general purpose hardware design languages (Verilog/VHDL) into standard cell netlists, and

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

then placing and routing these mapped netlists using fully automated algorithms. These RTL-to-GDS type compilers have served us well as demonstrated by the thousands of silicon miracles taped out every year. The inconvenient truth, however, is that modern RTL based flows are not really true compilers because they require a significant level of manual work and optimization for each design. An equivalent in software would be having a C-compiler that requires the user to write a portion of the program in assembly for every architecture and to manually fix 1000's of errors in the final binary executable on every compilation cycle.

To address challenges in design efficiency, DARPA launched the IDEA research program in 2017 with the ambitious goal of creating a set of “no human in the loop 24 hour turnaround layout generators for System-On-Chips, System-In-Packages, and Printed Circuit Boards” [10]. Open source physical design packages that are direct results from the IDEA program include OpenROAD (digital layout), Align and Magical (analog layout), FaSOC (mixed signal), and ACT (asynchronous design) [1–4].

To make all of these tools interoperate effectively, a compilation framework is needed for data exchange per design customization. Commonly used names for these frameworks include: reference flows, reference methodologies, build systems, and CAD flows. Until recently, all industry frameworks were proprietary and were generally based on a combination of makefiles, TCL, and various scripts.

In recent years, a number of open source frameworks have been developed to address the lack of shared infrastructure for small design groups, students, and hobbyists. The Hammer project demonstrated that physical design costs can be greatly reduced by introducing common IRs and separation of concerns between technology, EDA tools, and design [12]. The OpenLane project integrated Yosys (synthesis), OpenROAD (place and route), Magic (LVS/DRC), the Skywater 130nm open source PDK, and project specific TCL reference flows to demonstrate the first completely automated fully open source tapeout flow [11].

For the goal of realizing a general purpose hardware compiler, some significant challenges remain. Build systems must support warehouse-scale computing to facilitate performant compilation, and the framework must enable work to be easily distributed amongst many developers.

3 APPROACH

3.1 Distributed development

The Internet as we know it would not exist without open standards like TCP/IP and HTML. For autonomous hardware compilation to become a reality, we need similar open standards in place for the hardware development community. The industry currently lacks an open standard for PDK, EDA, and IP abstraction, making it impractical to automate the process of translating thousands of designs to 100's of PDKs using 100's of individual executables. A well specified standard has the potential to reduce the N-squared translation problem to a 2N order problem [7].

SiliconCompiler addresses this using a standardized schema for storing and communicating configuration and runtime results of a hardware compilation. The schema is organized as a human readable nested dictionary with parameter leaf cells containing data. Primitive data types supported for leaf cell values include string, float, integer, file/directory, and boolean. Data types can also be

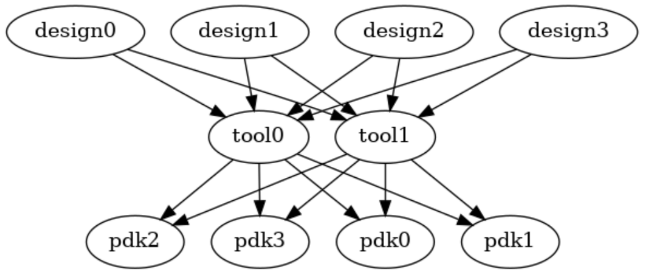

Figure 2: Format Translation Challenge

specified as dynamic lists or fixed-length tuples of these primitive types. In addition to the data and type field, each parameter includes a set of requirements, configuration, and documentation fields (scope, require, lock, switch, shorthelp, example, help). All file-type parameters also include a set of additional fields for provenance traceability (filehash, date, copy, author, hash algorithm).

The schema was designed with the following goals: (i) make the official schema implementation platform independent, (ii) implement a reference executable that can read and write the official JSON schema format to reduce ambiguity, (iii) make every parameter standalone to simplify adding extensions, (iv) include documentation and examples inside the parameter definition to make the specification self documenting and self testing, and (v) use reserved keywords to enable dynamic dictionary extensions where appropriate (e.g. library names, tool names). In total, the schema standard contains approximately 350 dynamic parameters, split into: source/options (100), EDA tool setup (25), execution flowgraph (6), PDK setup (50), ASIC methodology (20), library setup (80), provenance tracking (16), packaging (45), design checklists (10), and runtime metrics (35).

The choice of an ASCII JSON file for the design database over an optimized binary implementation implies a larger memory and disk footprint, but since the manifest doesn't actually contain any physical design data (only pointers), this is a non-issue. The file sizes of gzip-compressed SiliconCompiler manifests range from a few KB for a small design in FreePDK45 to a few hundred KB at an advanced FinFet node. Compared to chip design physical data which ranges from GB to TB, the overhead of the SiliconCompiler manifest is negligible.

While the SiliconCompiler JSON schema exactly defines the standard for compilation configuration and tracking, it is too low level to be programmed directly. Specialized data markup languages like YAML, JSON, and XML are not equipped with the constructs and abstractions needed for productive development of complex projects. To raise the level of abstraction, we developed an object oriented Python API and programming model to interface with the schema. The API includes 6 low level core functions for interfacing directly with the manifest (`set/get/add/getkeys/getdict/valid`) and 30+ higher level functions for configuring compilation runs and inspecting results after compilation.

To enable encapsulation and reuse of configuration, SiliconCompiler supports several categories of dynamic modules that contain setup code for PDKs, libraries, flows, and targets. Targets are a special category that themselves load flows, libraries, and a PDK that could be used together for multiple designs. The configuration performed by these modules can be accessed using a set of "load" functions: `load_pdk`, `load_lib`, `load_flow`, and `load_target`.

```
import siliconcompiler                              # import python package

def main():
    chip = siliconcompiler.Chip()                   # create chip object
    chip.set('source', 'heartbeat.v')               # define list of source files
    chip.set('design', 'heartbeat')                 # set top module
    chip.set('constraint', 'heartbeat.sdc')         # set constraints file
    chip.load_target('freepdk45_demo')              # load predefined target
    chip.run()                                      # run compilation
    chip.summary()                                  # print results summary
    chip.show()                                     # show layout file
```

Figure 3: Example illustrating the basics of compiling RTL to GDS using the SiliconCompiler Python API.

The combination of an open standard and dynamic modules is crucial for distributed development. These modules allow us to provide an open reference implementation without having to hard-code technology specific information, which is often encumbered by NDAs. They also allow reuse and distributed development within an organization, where different engineers could be in charge of writing setup modules for PDKs, libraries, and flows, which could then be reused in any number of combinations in any number of design projects. Looking forward, PDK, IP, and EDA vendors could even ship SiliconCompiler setup modules to enable easy integration by end-users, distributing the design effort and ultimately eliminating the inefficiencies of repeated work.

3.2 Distributed execution

Along with enabling distributed development, SiliconCompiler needs to enable transparent distributed execution to tackle the computational complexity of advanced designs. Compiling a design with SiliconCompiler entails writing a build script, such as the one in Fig. 3, that includes a minimum of three distinct steps: (i) Create an instantiation of the `Chip` class that encapsulates the design build process for the lifetime of the program. (ii) Configure the compilation parameters by setting values in the schema using the `set()/add()/load_*` access methods. (iii) Call the `Chip run()` method to start compilation.

A single atomic `run()` call has many benefits in terms of implementation flexibility, and it is enabled by our schema, which includes all information required to describe what executable steps the compilation entails. The backbone of the execution model is the flowgraph, which consists of a set of connected nodes and edges. A node is an executable tool performing some task, and an edge is the connection between those tasks. SiliconCompiler defines a *task* as an atomic combination of a step and an index, where a *step* is defined as a discrete function performed within the compilation flow such as synthesis, linting, placement, routing, etc, and an *index* is defined as variant of a step operating on identical source data with some compilation options changed. The ability to parallelize steps via indexing enables automation and parallelization of design parameter sweeps. Fig. 4 shows an example of a flowgraph.

Since all of this information is encapsulated in the schema, the operations performed by `run()` can execute anywhere. SiliconCompiler supports a remote client/server execution model where the flowgraph can be transparently executed on a server, and it supports the Slurm job scheduler for distributing execution of individual nodes in the flowgraph across multiple machines. The SiliconCompiler runtime automatically resolves dependencies and handles synchronization to ensure tasks are scheduled properly.

Each compilation task is configured by schema values, and upon completion the schema is written back to disk with results from

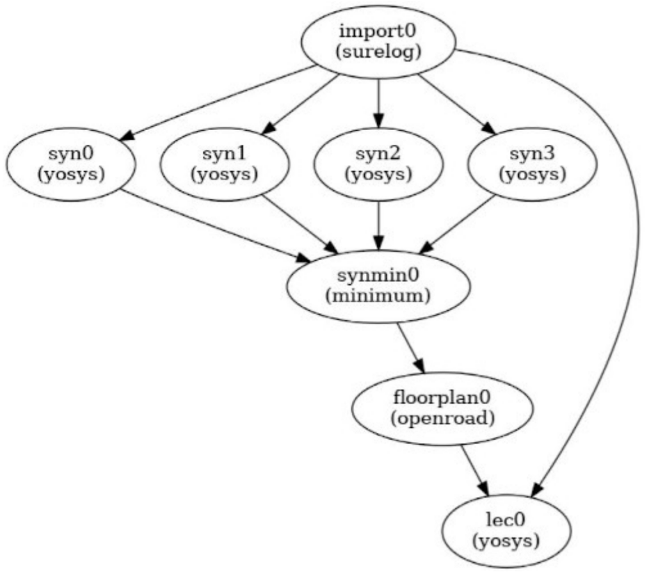

Figure 4: Simple flowgraph with four parallel synthesis tasks.

```
source sc_manifest.tcl
set sc_option [dict get sc_cfg eda my_tool variable . . . ]
run_command -option $sc_option
```

Figure 5: Schema EDA Tool Interface Example

the task. The per-design schema based configuration and recording database is called a manifest.

The JSON manifest enables a simple implementation, since it can be easily serialized and sent to many machines in a distributed system to give them complete information about the build. The SiliconCompiler runtime relies on the manifest's file pointers to bundle required files and pass them to each machine. In order to save bandwidth, the schema supports selectively determining which files will be copied to machines versus assumed to exist. For example, if each machine in a cluster has access to large PDK files they can be marked as `copy=False`, but smaller design-specific files can be marked as `copy=True`, since they will be constantly changing as the designer makes updates on their client machine. Each file-type schema parameter has a sensible default that can be overriden.

All executables called during compilation are directly controlled by the manifest file. In the case of EDA tools, which are generally controlled by TCL, this turns out to be an excellent choice. Advanced physical design flows can include 10-50K lines of highly optimized proprietary TCL code that cannot easily be generated directly from Python. A more practical approach is to write the SiliconCompiler standard manifest as a large TCL dictionary and then to create a thin driver that assigns dictionary variables to the various private reference flow variables. This driver approach has already been demonstrated successfully with the open source OpenROAD tools and several state of the art proprietary commercial reference flows.

By encapsulating all options required to fully describe a build, SiliconCompiler's schema enables a distributed execution model where the schema can be serialized, sent to multiple machines in a distributed cluster, and used to directly drive EDA tools. Wrapping this entire execution in an atomic `run()` function that can be implemented by multiple backends enables SiliconCompiler builds to

Table 1: SiliconCompiler Project Statistics

	Metric
PDK drivers	Open (3), Proprietary (2)
Tool drivers	Open (18), Proprietary (20)
Documentation	319 pages
Code Size (LOC)	17K Python, 2K TCL
CI Tests	122
Contributors	8
Popularity	6.9K downloads, 413 Github stars

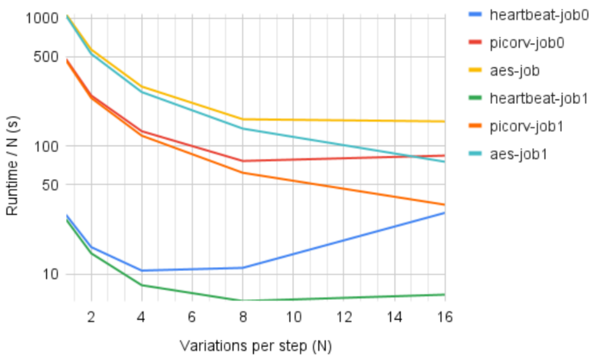

Figure 6: Compiler Scaling Results

transparently scale from a single client machine to a remote server to a many-node datacenter. This is crucial for enabling performant compilation of complex designs without increasing complexity from the perspective of the designer.

4 PROJECT STATUS

The SiliconCompiler project was released as an open source repository under an Apache 2.0 license at on December 3, 2021. It can be found at https://github.com/siliconcompiler. Table 1 shows key metrics for the project as of March 15, 2022. Open source tools supported by the project so far include: bambu, bluespec, chisel, ghdl, icarus, klayout, magic, netgen, nextpnr, openroad, surelog, sv2v, verilator, vpr, and yosys.

The results in Fig. 6, demonstrate the performance scalability of the programming mode and runtime while compiling RTL examples into GDS at the skywater130 node using the Yosys, OpenROAD, and Klayout tools. The measurements were taken on a 10 core (20 threads) Intel® Core™ i9 CPU @ 3.70GHz × 20, 64GB of RAM. Job0 runs n independent tasks for syn, place, cts, and route stages with minimization steps to select the best result. The same goes for Job1, except for the route stage which is run with a single set of options to reduce run time.

5 FUTURE DIRECTIONS

1) **Package management:** A core component of modern software development is a method for effective packaging and distribution of known good code. Examples include Cargo (Rust), Pip (Python), npm (Javascript), dpkg (Debian), and Docker (applications). Since our schema already includes the necessary infrastructure for recording all information about the source code and configuration switches needed for compilation, the process of encapsulating the information as a JSON package is straightforward.

2) **Education:** There is currently a severe shortage of SoC physical designers in the US and at the current graduation rate, this gap will never get filled. Barriers to effective workforce development include: (i) information sharing restrictions imposed by NDAs/EULAs, (ii) high complexity of setting up physical design infrastructure, and (iii) impedance mismatches between the students' expectations (Python, Stackoverflow, Google, Github) and current industry reality (TCL). A low cost (or free) cloud based physical design platform could remove many barriers to effective workforce development.

3) **Language Support:** SiliconCompiler already supports a number of high level front end hardware design frameworks, including: Bambu (HLS), Migen (Python), Chisel, and Bluespec. Given the low barrier to driver development, we expect the list of supported DSLs to grow rapidly as the community expands.

4) **Manual Design:** SiliconCompiler has so far only been tested with automated compilation flows. Human centric design flows such as analog/mixed signal, package, and board design could also benefit from the standardized build system for simulation, IP packaging, archiving, and manufacturing tape outs.

5) **Machine Learning:** SiliconCompiler natively supports Machine Learning thanks to the Python API and standardized metrics schema, but additional effort is needed to develop data sets, ML use cases, and API enhancements to support advanced machine learning concepts [8].

6 CONCLUSION

In this paper we have presented the SiliconCompiler project, an open source build system aiming to reduce the barrier to creating fully automated hardware compilers.

REFERENCES

[1] Tutu Ajayi, Vidya A Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, et al. 2019. Toward an open-source digital flow: First learnings from the openroad project. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–4.

[2] Tutu Ajayi, Sumanth Kamineni, Yaswanth K Cherivirala, Morteza Fayazi, Kyumin Kwon, Mehdi Saligane, Shourya Gupta, Chien-Hen Chen, Dennis Sylvester, David Blaauw, et al. 2020. An open-source framework for autonomous SoC design with analog block generation. In *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*. IEEE, 141–146.

[3] Samira Ataei, Wenmian Hua, Yihang Yang, Rajit Manohar, Yi-Shan Lu, Jiayuan He, Sepideh Maleki, and Keshav Pingali. 2021. An Open-Source EDA Flow for Asynchronous Logic. *IEEE Design & Test* 38, 2 (2021), 27–37.

[4] Tonmoy Dhar, Kishor Kunal, Yaguang Li, Meghna Madhusudan, Jitesh Poojary, Arvind K Sharma, Wenbin Xu, Steven M Burns, Ramesh Harjani, Jiang Hu, et al. 2020. ALIGN: A system for automating analog layout. *IEEE Design & Test* 38, 2 (2020), 8–18.

[5] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[6] Dave Johannsen. 1979. Bristle blocks: A silicon compiler. In *16th Design Automation Conference*. IEEE, 310–313.

[7] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.

[8] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. 2021. A graph placement methodology for fast chip design. *Nature* 594, 7862 (2021), 207–212.

[9] Andreas Olofsson. 2016. Epiphany-v: A 1024 processor 64-bit risc system-on-chip. *arXiv preprint arXiv:1610.01832* (2016).

[10] Andreas Olofsson. 2018. Silicon Compilers - Version 2.0. Keynote, http://www.ispd.cc/slides/2018/k2.pdf.

[11] Mohamed Shalan and Tim Edwards. 2020. Building OpenLANE: a 130nm openroad-based tapeout-proven flow. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–6.

[12] Edward Wang, Adam Izraelevitz, Colin Schmidt, Borivoje Nikolic, Elad Alon, and Jonathan Bachrach. 2018. Hammer: Enabling reusable physical design. In *Workshop on Open-Source EDA Technology (WOSET)*.