Hammer: Enabling Reusable Physical Design

Edward Wang*, Adam Izraelevitz*, Colin Schmidt*, Borivoje Nikolić*
Elad Alon*, Jonathan Bachrach*

***Abstract*—A lack of reusable physical design methodologies in ASIC design contributes to high design effort and costs. We posit that traditional flows suffer from a lack of separation between logical design, physical design, technology, and tool concerns, preventing effective re-use. We introduce Hammer, a physical design generator which remedies these problems via separation of concerns. In addition, Hammer addresses the needs of users to achieve practical re-usability using the principles of incremental adoption, system evolution, and abstraction with modularity. We elucidate how the design of Hammer separates concerns to enable usability, and walk through examples of how the principles of Hammer facilitate and contribute to open-source re-use and sharing in the domain of physical design.**

I. Introduction

Emerging application domains such as the Internet of Things, machine learning, and autonomous vehicles are increasing the demand for computing more than ever. The slowdown of technology scaling combined with the limits of general-purpose architectures means that we need specialized architectures in order to achieve further gains in efficiency and performance [1][2], yet specialization is costly due to high non-recurring engineering (NRE) costs.

A major component of NRE costs in silicon product design involves physical implementation and IP[1] integration. Physical design is the process of mapping the logical description of a design to a physical placement and routing. The class of algorithms underlying automatic place-and-route is generally considered to be NP-complete [3] [4] - in other words, no silver bullet or single algorithm/heuristic suffices. As a result, designer intervention is required to solve a multitude of difficult problems[2].

Solving these physical design problems involves many different *concerns*. We define a concern as a set of related code and data that collectively addresses a specific purpose. We identify four key concerns in an automated physical design flow[3]:

- Tools - settings and APIs required to solve a particular physical design problem in a given tool (e.g. how to apply an output load constraint in a synthesis tool like Yosys [5]).
- Technology Process/PDK - information concerning a semiconductor process node (e.g. standard cells, design rules, etc).
- Logical Design - consists of register-transfer level (RTL) descriptions of registers and combinational logic, module hierarchies, and inter-module connections. While the logical design can be specified with plain Verilog, we can leverage re-usable logical design/RTL generators like Chisel [6] to further augment the benefits of re-usable physical design.
- Physical Design - information concerning the physical implementation of the logical design. This includes floorplanning, clocking (e.g. clock tree synthesis), and power concerns.

Traditional flows exemplified by [7] [8] typically involve manual customization of tool vendor provided reference scripts for every chip project. Customization consists of the manual insertion of relevant technology, RTL, and physical design information directly into the scripts, generally tied to the specific project at hand. There are a number of problems with traditional flows that make it hard to re-use these scripts across different projects.

Firstly, these flows are often expressed directly in vendor-specific APIs that are not interoperable. Changing tool vendors becomes an intensive effort that entails rewriting physical design scripts using different APIs, despite any conceptual similarities between the APIs e.g. [9]. This contributes to both high NRE costs and an increased risk of experimenting with new CAD tools. Reduced appetite for experimentation and increased NRE costs make open source adoption even more difficult.[4].

Secondly, experienced designers accumulate large bodies of specific knowledge (e.g. design tips for a technology, CAD tool quirks [11], or logical design understanding). This knowledge is hard to re-use effectively in a traditional flow, since it must be manually encoded in every new chip project. The lack of programmatic re-use increases the risk of re-implementing previously known solutions.

Finally, while some re-usability is possible in the TCL (Tool Control Language) language[5] itself or through string/macro processing tools like Perl or the Unix sed utility, these approaches are cumbersome as the macro processing language has no awareness of the underlying physical design concepts, giving rise to ad-hoc approaches which make it difficult to achieve programmatic re-use. Furthermore, the TCL language

*Electrical Engineering and Computer Sciences, University of California, Berkeley. edwardw@eecs.berkeley.edu

[1]Intellectual property (IP) in chip design refers to a soft or hard macro block ready to be dropped into designs/layouts.

[2]Examples include floorplanning, power domain, pad frame insertion, filler cell insertion, clock tree synthesis, and more.

[3]A physical design flow that uses logic synthesis and automatic place-and-route, guided by constraints with a digital-top i.e. analog designs are introduced to a RTL-top design as blackboxes and connected in RTL.

[4]In addition, despite the increased importance of open design and open hardware (motivated by the hardware security field [10]), lack of separation of concerns makes it difficult to share open source physical design libraries/generators due to proprietary tool APIs.

[5]Commonly used to interface with CAD tools.

has not kept up to date with advances in programming languages and software engineering since its inception in the 1980s, such as static type safety or functional programming abstractions which are useful in increasing robustness and re-usability [12] [13]. Language and API features, even if present, may be incompletely or inconsistently implemented by different tool vendors e.g. [14][6], limiting re-use.

Overall, the above problems limit our ability to separate concerns and achieve efficient, programmatic re-use across related chip projects, resulting in increased effort and NRE cost.

In addition, previous works which attempted to automate physical design that were locked into a monolithic framework e.g. [16] was limited in re-use as it was often a tedious engineering effort to extract or re-use part of the solution. Nonetheless, we recognize that increasing usability out-of-the-box can play a role in adoption [17]. To achieve this, we include a command-line interfaces, as well as examples and templates for new users to jump in and use Hammer.

Agile flows suffer from this lack of re-usability to an even greater extent [11]. Agile hardware design relies on obtaining post-physical design quality of results (QoR) including area, performance, and power earlier in the design cycle in order to provide feedback to the other parts of the design (architecture, RTL, etc.) to improve the overall quality of the design [11]. A traditional flow does not lend itself well to rapid changes to any of the four concerns, limiting the ease and speed at which new feedback can be provided.

In this paper, we introduce Hammer (Highly Agile Masks Made Easily from RTL). Hammer aims to reduce design effort by separating concerns, and empowers designers to modify aspects of one concern without invasive changes or tediously rewrites as is the case with traditional flows. This separation of concerns reduces the effort associated with experimenting with new tools, and as a byproduct, increases the likelihood of adoption of newer tools which may otherwise seem risky. We hope that the Hammer infrastructure will contribute to an ecosystem of open tools and designs that encourage re-use.

II. Hammer: A Platform for Reusable Physical Design

A. Overview

We present here the main components of Hammer; see Figure 1 for a diagram of the components discussed below.

The **Hammer IR**[7] is the primary standardized data exchange format of Hammer. The IR standardizes physical design constraints such as placement constraints and clock constraints. In addition, the Hammer IR also standardizes communication among and to Hammer plugins, including tool control (e.g. loading tools, etc) and configuration options (e.g. number of CPUs).

The **Hammer Tool Library** is a CAD tool abstraction layer which consists of APIs for performing various physical design tasks, including synthesis and place-and-route. Different plugins implement the same interface for interoperability. To use Hammer tool plugins, we pass physical design information and settings using the Hammer IR. Developers implement Hammer tool plugins as Python classes which consume Hammer IR and emit the appropriate TCL fragments to implement those features for a certain tool. TCL hooks allow expert users to bypass the abstraction by injecting TCL code directly into the generated flow, similar to how inline assembly allows injection of assembly into C [18].

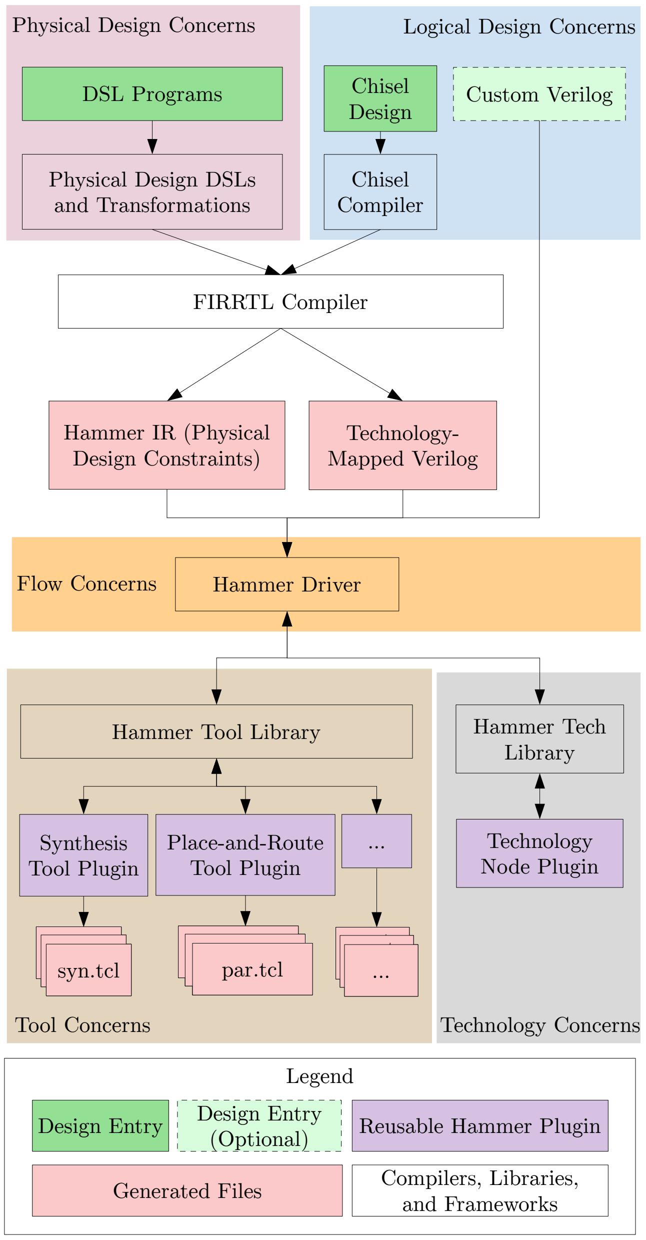

Fig. 1. The overall architecture of the Hammer methodology showing the points of design entry, compilers/libraries, generated files/formats, and Hammer plugins, as well as the interactions between them.

[6]We see a similar phenomenon in the inconsistent implementations of SystemVerilog across tool vendors [15].

[7]Intermediate representation

The **Hammer Tech Library** provides a standard data interchange format and corresponding Python library to encapsulate technology concerns. Hammer tool plugins are linked with a technology library so that they can perform actions involving technologies, like reading timing libraries or inserting filler cells. Creating a new technology plugin involves describing paths for components of the foundry-provided PDK, including standard cell libraries, timing databases, memories, layouts, and design rules. The interface can be bypassed by using TCL hooks to inject arbitrary TCL code that may include technology-specific references/functions.

Finally, technology and tool-independent physical design information is captured in **Annotations**. Annotations are metadata kept in sync with the logical design using a hardware compiler framework (HCF) like FIRRTL [12]. Users can generate annotations directly from a logical design generator like Chisel (e.g. retiming), or from **Physical Design DSLs** that are parameterized by the logical design (e.g. floorplanning constraints). Compiler transforms ingest annotations and emit the appropriate Hammer IR to implement the desired feature. Users can still explicitly pass physical design features without using annotations by writing Hammer IR or writing a TCL hook.

The **Hammer Driver**, Hammer's interface for flow concerns (concerns about scheduling builds, build dependencies, tracking/managing build outputs, etc), orchestrates the ingestion of inputs/outputs and loads/calls tools, all via programmatic Python and JSON APIs. This allows users to build their own customized flow solutions using Hammer while decoupling build/flow concerns from the other four concerns, making it possible to use a variety of build tools (e.g. shell scripts, bazel, Make, SCons).

B. Feature Realization in Hammer

Users have three options for realizing a physical design feature in a design: (1) direct re-use; (2) implement a re-usable solution (system evolution); (3) build a non-reusable one-time solution (incremental adoption).

1) Direct Reuse of Supported Features: For tool and technology concerns, re-use consists of leveraging a pre-existing plugin. With Hammer's abstractions, switching tools or technologies is a one-liner, whereas with a traditional flow, this same procedure would either require rewriting the flow essentially from scratch using a new API (for switching tools) or tedious manual editing of the flow (for switching technologies). Likewise, designers can take advantage of tool plugins for particular features (e.g. filler cell insertion) rather than relearning how to implement features in multiple tools.

To enable separating physical design from logical design, we provide DSLs that allow designers to express a subset of physical design concerns in a re-usable and programmatic way; for example, we support a floorplanning DSL that users can use to explore different floorplans without modifying the logical design generator. Alternatively, some types of physical design intent, like register retiming, can be expressed directly in the logical design generator since it is unlikely that different projects will want to change that.[8] In contrast, a traditional flow would require manual updates as it lacks the powerful programmatic APIs that enable the physical design to stay synchronized with the logical design.

2) Building Reusable Solutions: If a plugin for a new tool or technology does not exist, the one-time overhead of creating such a plugin can be amortized via re-use by future projects.

If a plugin exists for a given tool or technology but does not support some particular feature, we can add the feature in the plugin as well as a corresponding Hammer IR API to pass information to it. If appropriate, we can create annotations to generate the Hammer IR constraints in a way that keeps them in sync with the logical design.

3) One-time Non-reusable Solution: If we want to use a certain physical design feature (e.g. pad frames) but no such annotation/DSL or Hammer IR abstraction exists, we can still use this feature by using TCL hooks to write the relevant TCL commands directly. To facilitate system evolution, these TCL snippets can later be generalized into a Hammer IR API to amortize effort for future projects.

If a physical design constraint cannot be inferred from the logical design or expressed using a DSL[9], we can still re-use tool/technology information by writing Hammer IR manually. This shows that while we can take advantage of HCFs or logical design/RTL generators to achieve greater re-use and robustness, our methodology does not remove control from the designer.

C. Implementation Details

The Hammer Tool and Technology Abstractions, the Hammer Driver, the Hammer IR library, and all plugins are written in object-oriented, type-safe Python 3 [19]. We have tested Hammer at the time of writing with Python 3.4 and above. Physical design generators, DSLs, and annotations are implemented in Scala with the FIRRTL HCF [12], and we have implemented a floorplanning DSL using FIRRTL that generates Hammer IR for floorplanning. The Hammer IR format is represented as a key-value store in memory, and on disk as JSON/YAML, and is agnostic to the environment/languages used to generate it.

Hammer plugins are implemented as Python classes. The plugins extend/mix-in Hammer tool abstractions which are Python interfaces (via Python 3's Abstract Base Classes library [20]). The Hammer Tool abstraction includes functions for various aspects of writing tool plugins including interfaces with the Hammer IR, Hammer technology plugin, IP library filtering, and logging. The Hammer Tool abstraction also includes facilities for TCL hooks to allow users to inject/override/customize TCL generated for the tools and exercise step control (running only a sub-step of a particular

[8]DSLs or HCFs can enable us to change this separately from the logical design if necessary.

[9]While in theory all known physical design information could be expressed or generated from annotations/DSLs, this is unrealistic in practice since 1) the set of physical design features possible is always in flux; 2) the annotation/DSL might not support the feature in its entirety, so we want to still empower the user to use the underlying API (the Hammer IR).

task - for example, running only the routing step in a place-and-route flow).

At the time of writing, we have implemented synthesis and place-and-route tool plugins for two major CAD tool vendors (Cadence and Synopsys), in addition to technology plugins for an educational PDK, a 28nm fully-depleted SOI process, and a 16nm FinFET process.

In addition, we have used Hammer to tape out an AI accelerator and multiprocessor SoC in 16nm, as well as a Bluetooth SoC in 28nm. We will discuss these results in future works.

III. Design Principles

In designing our system, we kept three main principles in mind with regards to two main needs in physical design: building a specific design and developing tools/methodologies for reducing effort in future projects.[10]

1) Incremental adoption. We observe that our users' main goal is taping out a chip, while ecosystem development is a secondary focus. For them, we aim to make it easy to leverage re-usable features to accelerate taping out but also allow them to create non-reusable solutions to get specific jobs done as needed.

2) System evolution. Only developing non-reusable, one-off solutions does not contribute to reducing effort in future tapeouts. Tool developers can help here, as their main goal is ecosystem development with taping out as a secondary focus. They will serve a key role in generalizing non-reusable solutions for future users.

3) Modularity + Abstraction = Clarity. As previously outlined, separating concerns is not a strength of traditional flows. Hence, we rely on modularity and abstraction as the mechanisms of separating concerns. In addition, to facilitate re-usability, we also aim to standardize data exchange among the different parts of the flow.

We define re-use to be programmatic re-use with minimal forking or copy-pasting[11]. Merely making scripts available without the appropriate programmatic APIs necessary for re-use leaves copy-paste or brittle string manipulation as the primary means of re-use. These methods of "re-use" still require intensive manual effort to change aspects of one concern, such as switching to another vendor's APIs.

As shown in Figure 2, we aim to build a system usable by designers from all backgrounds. In the long run, we hope these designers will contribute to the open-source environment by increasing the set of re-usable features and APIs supported by our system. The following use cases in Figure 2 could be described as follows:

Traditional physical designer: A traditional physical designer breaks the barriers of what is possible by increasing

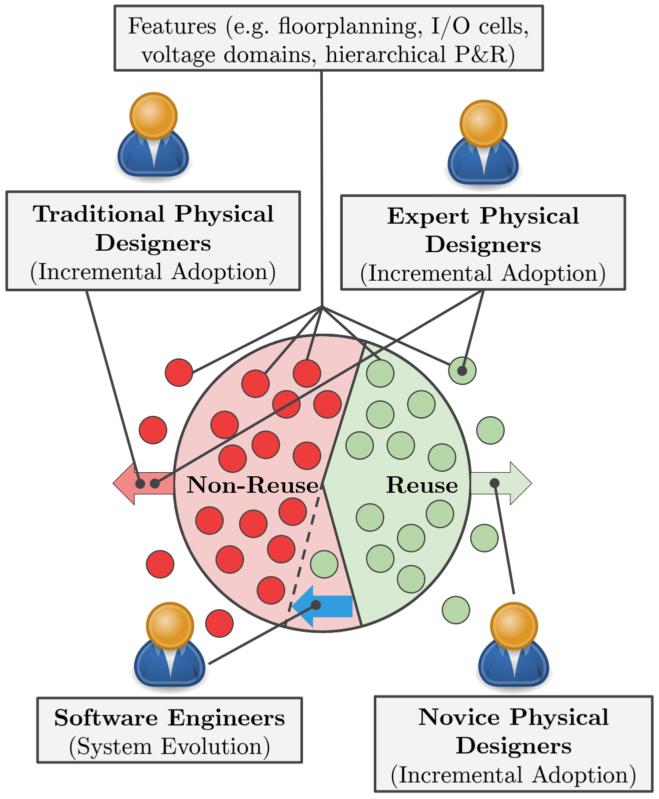
Features (e.g. floorplanning, I/O cells, voltage domains, hierarchical P&R)
Traditional Physical Designers (Incremental Adoption)
Expert Physical Designers (Incremental Adoption)
Non-Reuse
Reuse
Software Engineers (System Evolution)
Novice Physical Designers (Incremental Adoption)

Fig. 2. The colored bubbles represent individual physical design features. The circle represents the set of all physical design features expressible or implementable in Hammer (regardless of re-usability). The red/left-hand-side portion of the circle represents non-reusable features developed for a specific project, while the green/right-hand-side portion of the circle represents features re-usable across multiple projects. The red arrow represents the implementation of a non-reusable feature, while the green arrow represents the implementation of a re-usable feature. Finally, the blue arrow represents the transformation of a non-reusable feature into a re-usable feature. The different users in this diagram are discussed in the body of the text.

the set of possible features or solving a problem for a specific chip project, albeit not necessarily in a re-usable manner. The traditional physical designer is not necessarily well acquainted with software engineering but may be an specialist in a CAD tool, circuit design, or microfabrication.

Expert physical designer: An expert physical designer wants to leverage as many re-usable features as possible but also tape out/get the job done. He or she uses the re-usable features to save time and effort and creates as many non-reusable solutions as needed to finish the job. The expert physical designer is well versed in integrated circuits but is also reasonably skilled in using software engineering techniques to most effectively leverage the re-usable features.

Software engineer: Tool developers are software engineers who are frustrated by inefficiencies in physical design flows in the past. They create re-usable features to generalize non-reusable solutions used to accomplish one-off tasks.

Novice physical designer: A computer architect, for example, is interested in obtaining more realistic area and power

[10]Traditionally-organized ASIC design teams feature stratified silos between architects, RTL engineering, physical design, and verification. This leads to physical design teams focusing on taping out instances, with little to no focus on re-usable tools and methodologies [11].

[11]Examples of acceptable copy-pasting/forking include copying from lightweight examples or templates that mostly call out to re-usable libraries. We explicitly aim to keep these examples/templates as minimal as possible in order to avoid replicating the problem with existing flows.

numbers than is possible with high-level simulators with minimal time investment. He or she is able to easily leverage the reusable features in Hammer to get what he or she needs without the intensive effort associated with using the tools directly. A re-usable system lowers barriers for students and professionals new to ASIC design.

IV. Future Directions

1) ASIC Support: ASICs require packaging and PCBs for system integration [21]. Future work includes extending Hammer to support generating packaging and PCB information, such as footprints, packaging, or pin assignment. In addition, firmware co-generation, perhaps via DSLs like Halide [22], will also reduce NRE costs as software is a major component of chip product NREs [23].

2) Simulation/Testing/Verification: Additional Hammer plugins for SMT-based formal verification and signoff tools like DRC/LVS checking will further reduce development/maintenance effort. Hammer integration for gate-level/post-synthesis/post-place-and-route simulation (e.g. gate-level simulation) will also reduce overall design effort. Future targets for automation include design-for-test (DFT), built-in self-test (BIST), analog self-test, and self-calibration.

3) Analog/Mixed-Signal: For mixed-signal generators, further integration with BAG [24] and Chisel [6] at a generator level would allow analog and digital parameters to be optimized in concert.

4) Alternative Targets: Hammer's core infrastructure is target agnostic, opening the door to targeting FPGAs or even alternative architectures like coarse-grain reconfigurable arrays (CGRAs). In addition, integration with a simulation framework like FireSim [25] could allow seamless test-based verification of an ASIC design.

5) Useful Libraries: Hammer would enable the development of open source physical design libraries for many commonly used blocks in physical design including embedded FPGA fabrics, crossbars, and register files by allowing them to be shared without licensing or non-disclosure agreement (NDA) encumberment.

6) Generator Methodology: Generator education is a step towards increased Hammer adoption by teaching novice designers how to design using generators, and teaching veteran designers how to generalize existing designs into generators.

Automatically updating the generator given manual changes to the generated product (back-annotation) would also increase adoption of Hammer, particularly among traditional physical designers owing to their lesser familiarity with software engineering.

7) Broader Applications: Reducing design effort could open ASIC design to a broader community of open hardware enthusiasts just as Arduino did for microcontrollers [26].[12] Likewise, ASICs' dense integration makes applications like microrobots, biomedical sensors [28], and augmented/virtual reality more feasible.

[12]Multi-project wafer-based ASIC fabrication can be as low as $10,000 in the small quantities needed for hobbyists [27].

V. Conclusion

Lack of re-use in physical design flows caused by poor separation of concerns leads to high designer effort and high NRE costs in chip projects. In this paper, we have presented Hammer, a physical design generator to enable re-usability in physical design and lower NREs in chip design. We show that Hammer effectively separates tool, technology, logical design, and physical design concerns. Our system is equipped for success with Incremental Adoption, System Evolution, Modularity/Abstraction as principles. We have realistically evaluated Hammer with real tapeouts and will discuss these results in future works. Our work is open source (BSD-licensed) and available on the web: https://github.com/ucb-bar/hammer.

Finally, technological progress is unlocked when tools are improved, making it progressively easier for designers to efficiently build systems of ever increasing scale and complexity. Increased demand for computing at large requires continued improvements (power, area, cost) in design efficiency in order for ASIC design to keep up with demand. Hence, we believe the time is ripe to invest in new methodologies and tools to tackle high NRE costs in chip design.

Acknowledgments

The authors would like to thank David Biancolin, Dayeol Lee, Donggyu Kim, Eric Chang, Jack Koenig, Jim Lawson, Kristofer Pister, Nathan Pemberton, and Richard Lin for providing constructive criticism of the manuscript. Research partially funded by DARPA CRAFT HR0011-16-C-0052 and HR0011-12-2-0016; Intel Science and Technology Center for Agile Design; and ADEPT Lab industrial sponsors and affiliates Intel, Google, Siemens and SK Hynix. The authors also would like to acknowledge the sponsors, students, and faculty of the Berkeley Wireless Research Center. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or policy of any sponsors.

References

[1] O. Shacham, O. Azizi, M. Wachs, S. Richardson, and M. Horowitz, "Rethinking Digital Design: Why Design Must Change," *IEEE Micro*, vol. 30, no. 6, pp. 9–24, Nov. 2010. [Online]. Available: https://ieeexplore.ieee.org/document/5567087/

[2] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*. Saint-Malo, France: ACM Press, 2010. [Online]. Available: https://dl.acm.org/citation.cfm?id=1815968

[3] K. Shahookar and P. Mazumder, "VLSI cell placement techniques," *ACM Computing Surveys*, vol. 23, no. 2, pp. 143–220, Jun. 1991. [Online]. Available: http://portal.acm.org/citation.cfm?doid=103724.103725

[4] M. R. Cramer and J. V. Leeuwen, "Wire Routing is NP-Complete," Department of Computer Science, University of Utrecht, Tech. Rep. RUU-CS-82-4, Feb 1982. [Online]. Available: http://web.archive.org/web/20180504191319/https://dspace.library.uu.nl/bitstream/handle/1874/16302/kramer_82_wire_routing.pdf

[5] C. Wolf and J. Glaser, "Yosys - A Free Verilog Synthesis Suite," in *Austrochip 2013*, Linz, Austria, Oct. 2013. [Online]. Available: http://www.clifford.at/yosys/files/yosys-austrochip2013.pdf

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, *Chisel: constructing hardware in a Scala embedded language*. ACM, 2012, pp. 1216–1225.

[7] OpenPiton, "OpenPiton-ZC706," https://github.com/s117/OpenPiton-ZC706/tree/zc706_port/piton/tools/synopsys.

[8] IIT Madras, "SHAKTI C-Class Core," https://bitbucket.org/casl/c-class/src/b0c2897482ac2a88a0b9949064528cd4ba982f36/asic_flow/.

[9] D. Allen, "Power Formats: You Can Have It Your Way," http://web.archive.org/web/20180908021042/https://www.electronicdesign.com/products/power-formats-you-can-have-it-your-way, Mar. 2008.

[10] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation." in *USENIX Security Symposium*, 2016, pp. 857–874.

[11] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevic, P.-F. Chiu, R. Avizienis, B. Richards, J. Bachrach, D. Patterson, E. Alon, B. Nikolic, and K. Asanovic, "An Agile Approach to Building RISC-V Microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, Mar. 2016.

[12] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations." IEEE, Nov. 2017, pp. 209–216. [Online]. Available: http://ieeexplore.ieee.org/document/8203780/

[13] M. Endler, "Why Type Systems Matter," Jul. 2017. [Online]. Available: https://matthias-endler.de/2017/why-type-systems-matter/

[14] Stack Overflow User, "Collections to list in tcl," http://web.archive.org/web/20180904064416/https://stackoverflow.com/questions/17864980/collections-to-list-in-tcl, Jul. 2013.

[15] Krste Asanović, "Re: Coding style of Rocket a barrier to adoption of Rocket," https://groups.google.com/a/groups.riscv.org/d/msg/hw-dev/MSoVGfmFywY/KieW-uiYCQAJ, May 2017.

[16] P. Dabbelt, "PLSI: A portable VLSI flow," Master's thesis, EECS Department, University of California, Berkeley, 2017.

[17] Philip Guo, "The PhD grind: Year four: Reboot," http://pgbovine.net/PhD-memoir-year4.htm, 2012.

[18] Free Software Foundation, "How to Use Inline Assembly Language in C Code," https://gcc.gnu.org/onlinedocs/gcc-8.1.0/gcc/Using-Assembly-Language-with-C.html, 2018. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-8.1.0/gcc/Using-Assembly-Language-with-C.html

[19] Mypy project, "mypy - optional static typing for Python," http://mypy-lang.org/.

[20] Python Software Foundation, "ABC - Abstract Base Classes," http://docs.python.org/3/library/abc.html.

[21] J. Bachrach, D. Biancolin, A. Buchan, D. W. Haldane, and R. Lin, "JITPCB," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Daejeon, South Korea: IEEE, Oct. 2016. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7759349/

[22] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: decoupling algorithms from schedules for high-performance image processing," *Communications of the ACM*, vol. 61, no. 1, pp. 106–115, Dec. 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3176926.3150211

[23] International Business Strategies, Inc., "Semiconductor industry from 2015 to 2025," http://web.archive.org/web/20180903040121/http://media.wix.com/ugd/c69ef5_c12a90cb23f948a3b6b2a6bd3352d3fb.pdf?dn=SEMI-080415.pdf.

[24] J. Crossley, A. Puggelli, H.-P. Le, B. Yang, R. Nancollas, K. Jung, L. Kong, N. Narevsky, Y. Lu, N. Sutardja, E. J. An, A. L. Sangiovanni-Vincentelli, and E. Alon, "BAG: A designer-oriented integrated framework for the development of AMS circuit generators," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 74–81. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561843

[25] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. Los Angeles, CA: IEEE, Jun. 2018. [Online]. Available: https://sagark.org/assets/pubs/firesim-isca2018.pdf

[26] C. Severance, "Massimo Banzi: Building Arduino," *Computer*, vol. 47, no. 1, pp. 11–12, Jan. 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6750433

[27] Electronics Stack Exchange User, "How much does it cost to have a custom ASIC made," http://web.archive.org/web/20180908014028/https://electronics.stackexchange.com/questions/7042/how-much-does-it-cost-to-have-a-custom-asic-made, Nov. 2010.

[28] R. S. Zoll, C. B. Schindler, T. L. Massey, D. S. Drew, M. M. Maharbiz, and K. S. Pister, "MEMS-Actuated Carbon Fiber Microelectrode for Neural Recording," in *EMBS Micro Nanotechnol. Med. Conf.*, 2018, p. 6.