Sound WCET Analysis, Explanation of the Method and of the Results

Reinhard Wilhelm
Informatik
Saarland University
Saarbrücken, Germany
ORCID ID 0000-0002-5599-7560

Jan Reineke
Informatik
Saarland University
Saarbrücken, Germany
ORCID ID 0000-0002-3459-2214

***Abstract*—Sound WCET analysis computes reliable upper bounds to all execution times of a program as required by a schedulability analysis. It is a complex method, composed of many component methods. Software developers and certification authorities are interested in understanding the derivation of the results in order to develop trust in their correctness. The results cannot be understood without a basic understanding of the methods. We argue that essentially one of the component methods, Microarchitectural Analysis, is critical for understanding and accepting the results. In this article we therefore explain Microarchitectural Analysis and show how its provides local explanations for the overall result.**

In addition, we show that progress monotonicity is a sufficient condition for timing compositionality and that it increases explainability.

***Index Terms*—WCET analysis, real-time, instruction execution times, timing compositionality**

I. Introduction

Sound WCET analysis computes reliable upper bounds to all execution times of a program. The problems of WCET analysis are caused by performance-enhancing features of microarchitectures. They introduce a large variability of execution times of instructions. This article explains the principle behind sound solutions of the WCET problem. This principle is to prove the absence of *timing accidents*, i.e., events during the execution of an instruction execution that increase the execution time compared to the fastest execution. These proofs are based on the determination of invariants about the set of potential execution states at each program point. Such invariants can be computed by a fixed-point iteration by Abstract Interpretation. The particular ingredients of the employed instances of abstract interpretations are explained and connected to fundamental insights into the timing-predictability of execution platforms. We show how the computed invariants are essential for explaining the results of WCET analysis.

II. WCET Analysis—the Problem

WCET analysis can be seen as the search for the longest path through the control-flow graph of a program. The nodes are the instructions of the program, annotated with their execution times. This task was relatively easy in the (good old) times of instructions with constant execution times [1], [2]. Structural induction over the structure of a program was used to compute global upper bounds on all execution times of instructions.

Unfortunately, in modern high-performance processors the execution times of instructions vary widely. This is caused by their dependence on the execution state of performance-enhancing features such as caches, pipelines, and all kinds of speculation. The core of the WCET–analysis problem for modern high-performance processors thus is, how to safely bound the execution times of individual instructions in a program. This is done by the analysis component called *Microarchitectural Analysis* in the picture of the tool structure in Fig. 1. The determination of a safe upper bound on the execution time of the whole program and of the path on which this bound is determined is called *Global Bounds Analysis* in Fig. 1. This analysis explores all paths in a state space, spanned by the program and the architecture. It would be desirable to cut down the size of this space. However, as we will see later, a ghost that haunts WCET researchers is the existence of *Timing Anomalies* [3], [4], namely that cheaper continuations of execution paths may lead to globally more expensive paths and vice versa. For microarchitectures exhibiting timing anomalies, it would be incorrect to explore only worst-case transitions. The full graph needs to be explored. We will come back to timing anomalies in the context of timing predictability.

III. WCET-Analysis—A Solution

The different execution times result from different *execution states* in which an instruction may be executed. A memory access is fast if the accessed memory block is in the cache, it is slow if it has to be fetched from memory, and it is even slower if the memory load is blocked on the bus. We call cache misses, pipeline stalls, bus collisions, and mis-speculations *Timing Accidents* and the associated extra cycles *Timing Penalties*. The search through the annotated graph could safely assume a cache hit if it were known, for example as results of a static cache analysis, that the accessed memory block were in the cache each time execution reaches that program point. So, the solution consists in computing invariants at each program point that safely describe the execution states, i.e., the occupancy of the machine resources.

We now argue that all but one component analysis from Fig. 1 are nor critical for understanding sound WCET analysis.

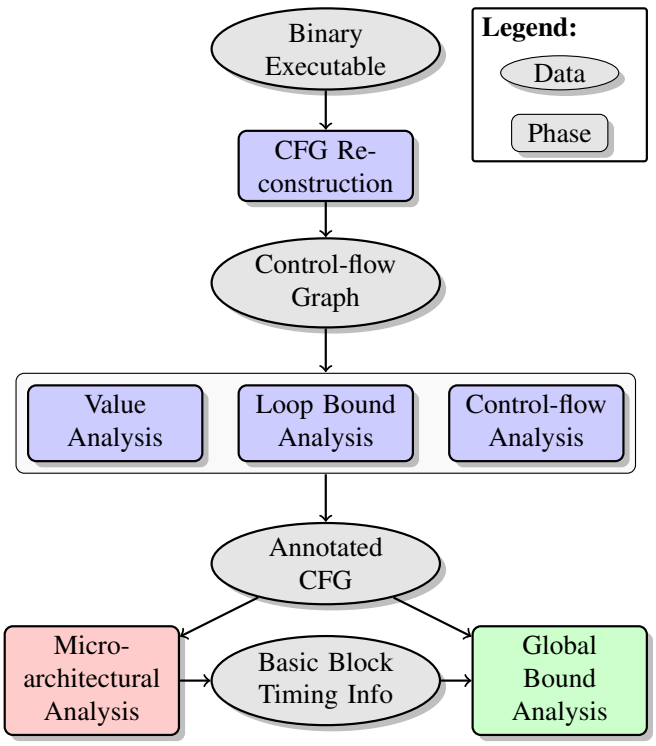


Fig. 1. Architecture of WCET tools using static analysis based on an abstraction of the execution platform such as AbsInt's aiT tool [5].

The *Value Analysis*, shown in Fig. 1, computes enclosing intervals for the values in program variables and machine registers. The results restrict potential memory accesses to enable a precise data-cache analysis and support *Loop Bounds Analysis*. The *Control-Flow Analysis* determines facts about the control flow, like infeasible control flow paths, that cannot contribute to overall execution times. In case of doubt these computed approximations can be inspected and compared with the user's expectations. The *Global Bounds Analysis* exhibits the control-flow paths through the program, augmented by the paths through the architecture, together with annotations of the costs. The overall graph shows all cycle-wise evolutions of the computation. The direct or indirect algorithm, used to compute this graph, can be taken from textbooks and proved correct. In case of doubt the annotated graph can be manually checked.

The *Microarchitectural Analysis* in this solution to the WCET-Analysis problem [6] consists in determining at each program point an invariant that describes a safe approximation of the set of execution states that are possible when execution reaches this program point. Based on these invariants our WCET analysis proves safety properties of the following kind: *A certain timing accident cannot happen when the instruction at this program point is executed.* These approximations are *safe* in the sense that any timing accident excluded based on them will indeed never happen. Each safety property proved in this way allows WCET analysis to reduce the execution-time bound of the instruction by the associated timing penalty.

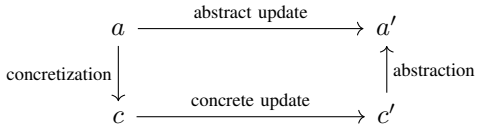
The invariants are computed by an instance of *Abstract Interpretation* [7], algorithmically by a fixed-point iteration over the control-flow graph of the program. Abstract interpretation is a semantics-based static analysis. Unlike most of the instances of abstract interpretations used in practice or published, abstraction interpretation used in WCET analysis needs to include the semantics of the underlying execution platform. Its core, therefore, is an abstraction of the underlying microarchitecture. This abstraction is structured into different components according to the structure of the architecture. Abstractions of several architectural components are described in Subsection III-A and Section IV.

A. Execution-state Invariants as Means to Explain the Result of WCET-Analysis

The execution-state invariants and the transitions between invariants at consecutive program points are the means to show local correctness to a user or certification authorities. Execution-state invariants describe possible occupancies of machine resources, i.e. what is in the caches, how far have instructions progressed through the pipeline, and which conflicts may they encounter in attempting to access pipeline units, which part of the bus bandwidth is occupied by running transfers. Microarchitectural execution states are structured as a collection of components according to the overall structure of the machine resources. The occupancy of different components is represented in different abstract domains. Understanding the invariants means being able to read the abstract states.

In Abstract–Interpretation terminology, *concretization* is the function that returns the set of concrete states represented by an abstract state. The abstract cache states, for instance, describe sets of concrete cache contents, together with replacement information. In the case of an LRU cache the latter are upper bounds on the ages of the memory blocks guaranteed to be in the cache. In the case of non–LRU caches this information may be complex. Abstract pipeline states of most current architectures are easier to understand since they are collections of concrete pipeline states.

In addition to the concretization of the abstract cache states, the transitions between invariants need to be explained. In principle, one could imagine that all component states described in an invariant would be concretized, then the concrete transformation would be applied to all of the concrete ones, and then the resulting set of concrete states would be abstracted back into abstract states forming the new invariant as illustrated by the following diagram:



More realistically, the developer of the microarchitectural analysis determines a conservative approximation of the composition of these three functions and explains the result to the interested public. The abstract update needs to satisfy the following *local correctness condition*: Given a set of

concrete caches states c described by abstract cache state a. The transition from all the cache states in c yields concrete cache states collected in a set c'. The application of the abstract transition relation to a must yield an abstract cache state a' that contains at least all the concrete cache states in c'. This is captured by the following diagram:

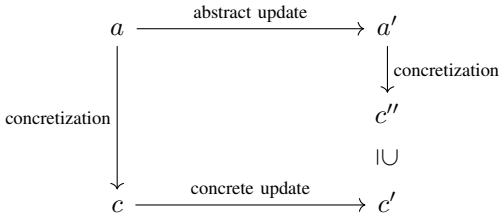


Fig. 2 shows an example of an abstract transition in the case of caches. This figure should be read as follows: Transitions from one *concrete cache state* to a successor state happen under a memory access. In the lower part of Fig. 2 transitions from four concrete cache states under an access to memory block C lead to two potential successor states. The initial four concrete cache states are described by (can be abstracted to) the *abstract cache state* pictured above them. This is an abstract cache state that contains all memory blocks contained in all of these concrete cache states with an age at least as large as that of all their ages in the concrete cache states. The transition from the abstract cache state leads to an abstract cache state that describes the two resulting concrete cache states in the same way. We also say that the concretization of the resulting abstract cache state contains the two concrete cache states.

In this particular case, the abstraction is *exact*, i.e., the resulting abstract cache states represent the two potential successor states, and no more. In general, the cache abstraction from the example, and other abstractions, may introduce additional spurious concrete states, which may result in a loss of precision, but correctness remains guaranteed by overapproximating any set of reachable concrete states.

The *local correctness condition* we have just explained is the basis to explain the *global correctness* of the analysis. If each abstract transition is locally correct, then it can be shown that the concretization of the reachable set of abstract states computed by fixed-point iteration is guaranteed to contain all reachable concrete states.

Again, abstract transitions on non–LRU abstract cache states may be complex. However, there is no way to give a simple explanation of a complex mechanism.

IV. Microarchitectural Abstractions

Attempting to determine the real worst-case execution time by exhaustive exploration of the space of all paths is a hopeless endeavor for all but trivial programs. Not only the space of all paths through the control-flow graphs needs to be explored, but also the much larger space of paths through the sets of execution states of the underlying microarchitecture.

This latter space can be reduced by abstraction, considering *abstract execution states* instead of concrete execution states.

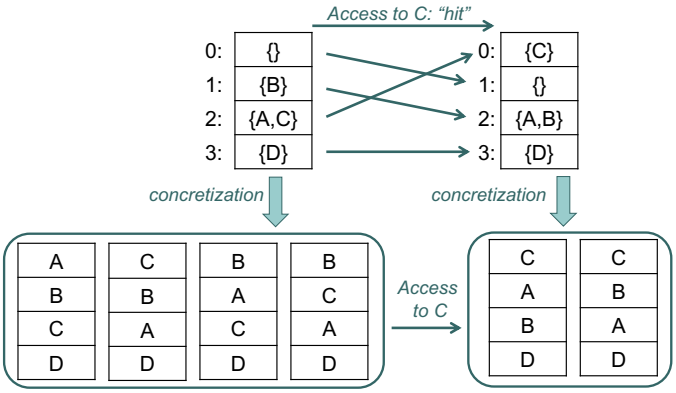


Fig. 2. Transition from one abstract LRU cache state to a successor abstract cache state and their concretizations

The different components of the execution platform are abstracted as to make the space exploration feasible, but on the other hand keep enough information for the above mentioned exclusion of timing accidents. It turns out that different types of components need different types of abstractions.

A. Abstractions of State–Dependent Resources

The essential property of *state-dependent resources* as far as timing of instruction execution is concerned is that their state influences instruction execution times, that is, the execution of instructions takes different number of cycles depending on the state of the resource. For example, an instruction or an operand fetch takes different number of cycles depending on the state of the instruction or data cache. The access to a memory bank takes a different number of cycles depending on whether the bank is open or closed. The next instruction to be fetched depends on the state of the branch predictor. Static cache analysis has been the mother of all WCET analyses. Compact abstractions of cache states with efficient updates, such as the one shown in Fig. IV, have been identified for caches with LRU-replacement policy [8].

B. Abstractions of Bandwidth Resources

We call microarchitectural components such as buses and other interconnects *bandwidth resources* since they offer limited resources in time to different competing actors. Contention is resolved based on some protocol, sometimes based on the state of the resource. In contrast to storage resources, where the actual state of a resource can still be influenced by long-passed state changes, this state usually results from recent actions, i.e., whether two cores are currently competing for access or not. Abstract states in the static analysis of bandwidth resources record everything needed to predict guaranteed access to the resource, such as all potential states of the access protocol, all maximal delay times caused by running accesses, and the minimal available bandwidth of the resources.

Bandwidth resources are particularly challenging when they are shared, as e.g. buses are in multi-core processors, due to the large number of possible interactions between processes running on different cores. A promising approach to efficiently

analyze shared bandwidth resources is compositional analysis, which we discuss in the next section.

C. Abstractions of Progress Resources

Pipelining overlaps the execution of multiple instructions to improve performance. From an analysis point of view pipelining is challenging as it prohibits decomposing the problem into the analysis of one instruction at a time. Flowing through the pipeline, each instruction gradually progresses through the resource. We thus regard pipelines as *progress resources*. Abstract pipeline states in static pipeline analysis record the minimum progress instructions under execution can be guaranteed to have made at a program point.

Efficient abstractions for pipeline states have been particularly hard to find. Out-of-order pipelines required using expensive power-set domains [9], [10]. This means that the pipeline analysis computed (often very large) sets of pipeline states at each program point instead of small descriptions of such sets like in the case of caches. Out-of-order pipelines also unavoidably came with timing anomalies, which we will discuss further in the following section. In contrast to common belief, even in-order pipelines can exhibit timing anomalies as shown in [11].

V. Timing Predictability

The notion of *Timing Predictability* has been around even before the WCET problem became exciting [12]. Varying instruction execution times were not a problem at this time. The method described in this paper was first instantiated for two different microarchitectures used by Airbus [6]. It became immediately clear that these two microarchitectures, a Motorola Coldfire and a PowerPC 655 had different *Timing Predictabilities* [13].

Thiele and Wilhelm [14] published a first recommendation for the design of timing-predictable microarchitectures based on empirical evidence. The dissertation of Jan Reineke was the first to present a formal notion of timing predictability, namely that of cache replacement strategies [15], [16].

A. Timing Anomalies and Timing Compositionality

Microarchitectures exhibiting timing anomalies force sound WCET analyses to explore inherently larger search spaces. On the other hand, microarchitectures that are provably free from anomalies lead to more scalable WCET analyses and more explainable results, as the analysis and its explanation can focus on a smaller set of states. A natural question thus is how to systematically construct microarchitectures that are free from timing anomalies.

Similarly, multi-core platforms with shared resources offer a severe challenge for WCET analysis since different interferences on the shared resources, in general, lead to different timing behaviors. Explicitly exploring all interleavings of accesses to shared resources leads to an enormous increase in the size of the search space [17], [18]. A promising approach to scalably analyze WCET in multi-core systems is compositional analysis [19], [20], where the timing contributions of shared resources are analyzed separately and then composed. In such an approach, the "base" WCET of each task is analyzed assuming the task is run in isolation. Then, the amount of interference on each shared resource is bounded separately and added to the base WCET to obtain a bound on the execution time under contention. We argue that compositional analyses are naturally more explainable than integrated analyses as each individual analysis is less complex. Unfortunately, such an approach requires timing compositionality [21], the ability to soundly compose contributions from multiple resources. Complex processors have been shown to violate timing compositionality.

It turns out that timing compositionality and freedom from timing anomalies are strongly related. In fact, Hahn and Reineke [22] showed that *progress monotonicity* is a sufficient condition for both properties. More precisely, consider two microarchitectural states a and b in which every instruction exhibits more progress towards retirement in b than it does in a, we say that $a \leq b$. A microarchitecture exhibits *progress monotonicity* if for any pair of states $a \leq b$, the immediate successor states $s(a)$ and $s(b)$ maintain the ordering, i.e., $s(a) \leq s(b)$. In [22] they also demonstrate how to construct a processor that provably satisfies progress monotonicity and thus facilitates fast and explainable WCET analysis.

Computer architects are mainly concerned with increasing the average-case performance and ignore the requirements of the embedded real-time domain. The most notable exception is the Kalray many-core processor which is designed to avoid interferences and make sound and precise WCET analysis possible [23].

VI. Explainability Challenges

The correctness of the architectural abstraction is the critical point for customers and certification authorities when it comes to the soundness of a WCET–analysis tool. Formal verification of the soundness is infeasible in the absence of formal models of the underlying execution platforms. Sophisticated testing strategies have been used to convince one of the correctness [24].

The ideal would be a formal or at least semi–formal derivation of the abstract model from an available architecture description in a HW description language like VHDL or Verilog. Marc Schlickling and Markus Pister [25] have attempted this approach, but were bogged down with too many detail problems as tool support for analysis and verification of HW description languages was under–developed at the time of their attempt. The growing ecosystem of RISC-V [26] open-source cores and systems-on-chip along with open-source synthesis toolchains such as Yosys [27] make such an approach appear more viable today.

In this context, it would even be conceivable to generate two tools from the Verilog code describing the underlying hardware:

1) An abstract hardware model used within WCET analysis. Based on this model the WCET analysis would

generate "WCET certificates" explaining the WCET bound.

2) A WCET certificate checker relying purely on the concrete hardware model to confirm the correctness of the WCET analysis result. This would reduce the trusted compute base by eliminating the need to trust the correctness of the abstract model.

ACKNOWLEDGEMENTS

We appreciate the comments of the reviewers who helped us to improve the paper. This work has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101020415).

REFERENCES

[1] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Trans. Software Eng.*, vol. 15, no. 7, pp. 875–889, 1989.

[2] P. P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real Time Syst.*, vol. 1, no. 2, pp. 159–176, 1989.

[3] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *RTSS*, pp. 12–21, 1999.

[4] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies," in *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[5] AbsInt Angewandte Informatik GmbH, "aiT WCET Analyzers." https://www.absint.com/ait/.

[6] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *EMSOFT*, vol. 2211 of *LNCS*, pp. 469 – 485, 2001.

[7] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, (New York, NY, USA), pp. 238–252, ACM Press, 1977.

[8] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems.," *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.

[9] S. Thesing, *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[10] M. Langenbach, S. Thesing, and R. Heckmann, "Pipeline modeling for timing analysis," in *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings* (M. V. Hermenegildo and G. Puebla, eds.), vol. 2477 of *Lecture Notes in Computer Science*, pp. 294–309, Springer, 2002.

[11] S. Hahn, J. Reineke, and R. Wilhelm, "Toward compact abstractions for processor pipelines," in *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, pp. 205–220, 2015.

[12] J. A. Stankovic and K. Ramamritham, "Editorial: What is predictability for real-time systems?," *Real Time Syst.*, vol. 2, no. 4, pp. 247–254, 1990.

[13] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *IEEE Proceedings on Real-Time Systems*, vol. 91, no. 7, pp. 1038–1054, 2003.

[14] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.

[15] J. Reineke, *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. PhD thesis, Saarland University, 2009.

[16] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, 2007.

[17] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Haupenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm, "Impact of resource sharing on performance and performance prediction: A survey," in *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings* (P. R. D'Argenio and H. C. Melgratti, eds.), vol. 8052 of *Lecture Notes in Computer Science*, pp. 25–43, Springer, 2013.

[18] T. Kelter and P. Marwedel, "Parallelism analysis: Precise WCET values for complex multi-core systems," *Sci. Comput. Program.*, vol. 133, pp. 175–193, 2017.

[19] S. Altmeyer, R. I. Davis, L. S. Indrusiak, C. Maiza, V. Nélis, and J. Reineke, "A generic and compositional framework for multicore response time analysis," in *RTNS*, pp. 129–138, 2015.

[20] R. I. Davis, S. Altmeyer, L. S. Indrusiak, C. Maiza, V. Nélis, and J. Reineke, "An extensible framework for multicore response time analysis," *Real Time Syst.*, vol. 54, no. 3, pp. 607–661, 2018.

[21] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: definition and challenges," *SIGBED Rev.*, vol. 12, no. 1, pp. 28–36, 2015.

[22] S. Hahn and J. Reineke, "Design and analysis of SIC: a provably timing-predictable pipelined processor core," *Real Time Syst.*, vol. 56, no. 2, pp. 207–245, 2020.

[23] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014* (G. Fettweis and W. Nebel, eds.), pp. 1–6, European Design and Automation Association, 2014.

[24] R. Wilhelm, M. Pister, G. Gebhard, and D. Kästner, "Testing implementation soundness of a WCET analysis tool," in *A Journey of Embedded and Cyber-Physical Systems - Essays Dedicated to Peter Marwedel on the Occasion of His 70th Birthday* (J. Chen, ed.), pp. 5–17, Springer, 2021.

[25] M. Schlickling and M. Pister, "Semi-automatic derivation of timing models for WCET analysis," in *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems, LCTES 2010, Stockholm, Sweden, April 13-15, 2010* (J. Lee and B. R. Childers, eds.), pp. 67–76, ACM, 2010.

[26] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Tech. Rep. UCB/EECS-2014-146, Aug 2014.

[27] C. Wolf, "Yosys open synthesis suite." https://yosyshq.net/yosys/.