GSA to HDL: Towards principled generation of dynamically scheduled circuits

Aditya Rajagopal[1], Diederik Adriaan Vink[*2], Jianyi Cheng[*2], Yann Herklotz[*2]

* Imperial College London, UK

ABSTRACT

High-level synthesis (HLS) refers to the automatic translation of a software program written in a high-level language into a hardware design. Modern HLS tools have moved away from the traditional approach of static (compile time) scheduling of operations to generating dynamic circuits that schedule operations at run time. Such circuits trade-off area utilisation for increased dynamism and throughput. However, existing lowering flows in dynamically scheduled HLS tools rely on conservative assumptions on their input program due to both the intermediate representations (IR) utilised as well as the lack of formal specifications on the translation into hardware. These assumptions cause suboptimal hardware performance. In this work, we lift these assumptions by proposing a new and efficient abstraction for hardware mapping; namely *h-GSA*, an extension of the Gated Single Static Assignment (GSA) IR. Using this abstraction, we propose a lowering flow that transforms *GSA* into *h-GSA* and maps *h-GSA* into dynamically scheduled hardware circuits. We compare the schedules generated by our approach to those by the state-of-the-art dynamic-scheduling HLS tool, Dynamatic, and illustrate the potential performance improvement from hardware mapping using the proposed abstraction.

KEYWORDS: Gated Static Single Assignment; High-Level Synthesis; Dynamic Scheduling

1 Introduction and Related Works

High Level Synthesis (HLS) tools convert a specification of a hardware circuit provided in a high level programming language such as C/C++ into a hardware description language (HDL) that can then be synthesised into the desired circuit. Input programs in high level languages are often described with sequential semantics and considerable control-flow as CPUs are highly optimised to execute such programs. The challenge with HLS lies in successfully converting such a description into highly parallel circuits. To do so, most commercial tools such as Vivado HLS by AMD and Catapult HLS by Siemens utilise static scheduling, i.e. the clock cycle at which each operation executes is determined at compile time rather than runtime. Static scheduling, which is sensitive to the latency of the operators, generates circuits with predictable execution times, low resource consumption and high performance for input programs with regular control-flow. However, in programs with irregular control-flow, this

---

[1]E-mail: adityarajagopal0@outlook.com
[2]E-mail: {dav114,jc9016,ymh15}@ic.ac.uk

approach can result in low throughput as static scheduling relies on worst-case assumptions of operation latencies in the absence of runtime information.

One solution to this issue is to generate circuits that execute purely based on dataflow, i.e. an operation is only executed when all its inputs contain valid data from previous operations. Such a circuit would execute in a manner that is insensitive to the latency of the components that constitute it and would be able to execute each operation as early as possible. Working towards generating pure dataflow circuits, [6] introduced Dynamatic, a dynamically scheduled HLS flow which bases its foundations on latency insensitive circuit design [1, 2]. They demonstrate up to 2.5x improvement in execution time and 4x improvement in throughput on programs with irregular control-flow when compared to statically scheduled HLS at the cost of up to 5x more resources. Dynamatic uses a lowering flow that transforms a control-dataflow (CDFG) [3] representation of the program (in LLVM IR) directly to a dataflow circuit by mapping IR constructs to predefined elastic modules. However, this process requires significant static analysis [3] to prevent superfluous control-flow dependencies that limit the amount of parallelism in the generated hardware.

To avoid the pitfalls of mapping a CDFG to a dataflow circuit as well as the limitations of static analysis from the lack of runtime information, [4, 9] propose transforming LLVM IR into new IRs that abstract dataflow into the representation before lowering to a predefined set of elastic components. [4] propose to use Gated Static Single Assignment [8] (GSA) while [9] propose the *handshake* dialect in the MLIR ecosystem. However, [4] neither provide formal semantics of the transformation nor the resulting GSA, which makes it hard to reason about correctness of lowering. The *handshake* IR restricts the ability to perform further static analysis as it relies explicitly on runtime information. Instead, we propose a novel version of the GSA IR, *h-GSA*, that aims to alleviates both these issues. The proposed lowering flow first uses Compcert [7] to perform a formally verified conversion of an SSA based IR to GSA [5], then transforms this into *h-GSA*, a version of GSA that is amenable to elastic circuit generation, and maps *h-GSA* into a predefined set of elastic components.

The following sections demonstrate the increased dynamism that our methodology can generate over Dynamatic and introduces the formal semantics of GSA nodes and invariants that hold when transforming GSA to *h-GSA*. Finally, we discuss the various future directions of research we envision this work will enable.

2   Motivating Example

This section motivates the use of GSA as in intermediate representation for HLS by comparing the execution schedules generated by our tool vs. Dynamatic for a simple motivating example (Figure 1a). Figure 1b shows the schedule generated by Dynamatic. As a consequence of lowering directly from a CDFG (LLVM IR), Dynamatic generates pure dataflow circuits within basic blocks, but requires basic blocks to start sequentially. This means L2, which modifies variable b needs to start after L1 (which increments a with g(j)) has started, even though there is no data-dependency between a being incremented and b being incremented. This results in reduced task level parallelism, where the second iteration of L1, shown in green in the schedule, only starts on cycle 8.

On the other hand, Figure 1c shows the schedule that is achieved by using *h-GSA*. This achieves the optimal schedule for this example. As there are no basic-blocks in GSA, there are
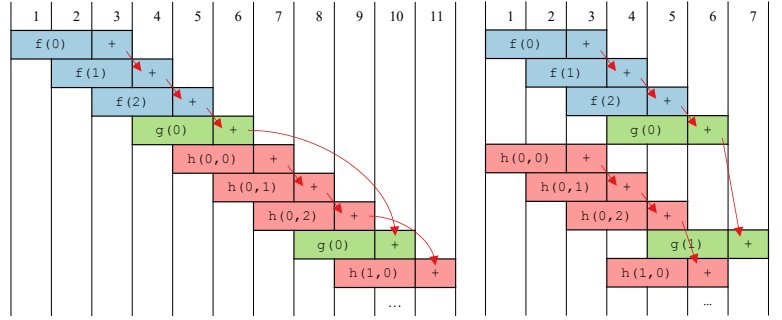
---

[3]dataflow within basic blocks and control-flow between basic blocks

```c
int foo() {
  int i, j, k;
  int a = 3, b = 5;
  L0: for (i=0; i<3; i++)
    a += f(i);
  L1: for (j=0; j<4; j++) {
    a += g(j);
    L2: for (k=0; k<3; k++)
      b += h(j, k); }
  return a + b; }
```


(a) Motivating example.   (b) Pipeline schedule by Dynamatic [6].   (c) Pipeline schedule by our work.

Figure 1: The red arrows mean the data dependence in the program. Existing dynamic scheduling restricts the loop iteration to *start* sequentially, although they can compute in parallel. Our work safely enables out-of-order loop computations by mapping GSA to hardware.

no restrictions on when instructions start based on control-flow dependencies. Instead, only data-dependencies are taken into account, leading to all operations executing as soon as their dependencies are met. In this example, lowering via h-GSA allows L0 and L2 to start at the same time. Due to the write-after-write dependency between assign a in L0 and L1, L1 needs to wait for L0 to finish before it can start, which is shown by the green loop waiting for the blue loop to finish (Figure 1c). However, contrary to the schedule provided by the Dynamatic circuit, L2 (red) can start executing at the first cycle, because there is no data-dependency between L0 and L2. This additional dynamism is a feature of GSA because it provides a purely dataflow centric view of the whole program using the primitive functions that are further described in section 3.

3   IR Level Formal Semantics and Hardware Mapping

This section will describe the notable differences between *h-GSA* and the version of GSA [8] formalised in CompCert [5]. Starting from such a formal description means that there is a well-understood starting point on which we can base the translation from GSA into dynamically scheduled circuits. GSA is an extension to SSA, defining two types of gating ϕ-instructions, namely μ-instructions at loop headers and γ-instructions at the exit of conditional statements. Additional nodes called η-instructions are also added at loop exits. These nodes can then be assigned different semantics based on their usual control-flow (software) semantics, and in *h-GSA*, dataflow (hardware) semantics, which are shown in Table 1.
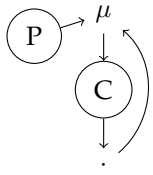
Our work performs a lowering from this GSA representation to a netlist of pre-defined components. Each operation in the software program is translated into a hardware component and connected using the handshake interface [6]. The arithmetic and logic operations used by our work are the same as the ones in Dynamatic, however, we add additional hardware modules which implement the hardware semantics of the GSA-specific IR constructs.

Interesting modifications had to be performed to the GSA program produced by Comp-CertGSA before we could generate functional hardware from it. Even though the transformation to GSA is formally verified, the hardware requires stronger properties about the structure of the GSA to execute correctly. Firstly, additional guarantees are needed about registers not appearing after they have been gated. Secondly, each μ instruction has to be paired with
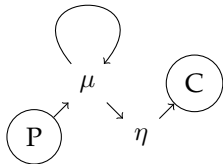
Table 1: Semantics of GSA functions in both hardware and software. These functions do not perform computation and only forward inputs to outputs when specific conditions are met.

Instruction	Software Semantics	Hardware Semantics
$d \leftarrow \mu(r_0, r_i)$	Behave like a ϕ-instruction from SSA and inspect the incoming control-flow edge to choose the register to assign to d.	Stateful component accepting inputs from r_0 the first time, then accepting inputs from r_i until it is reset.
$d \leftarrow \gamma(\overrightarrow{(p_i, r_i)})$	Make a local choice of which register r_i to pick based on the evaluation of its predicate p_i.	Component that selects the r_i based on the evaluation of p_i.
$d \leftarrow \eta(p, r)$	Only allow execution of the semantics to proceed when p evaluates to true.	Component stalls until p is true. It also resets its corresponding μ instruction to accept new inputs.

at least an η instruction so that it is reset upon loop exit. Finally, particular patterns need additional GSA nodes (described below), where the arrows represent a control-flow graph, μ-η pairs define entry and exit of loops, and P and C are producers and consumers of values.

When a producer is outside of the loop that contains its consumer, then μ nodes needs to be generated for it. This is not the case in software because the value assigned by the producer will always be accessible by reading its register, but in a dynamic circuit it needs to be passed around the loop (μ generates the same value at each new loop iteration).

One also needs to guarantee that any consumers that are separated from their producer by a loop have an appropriate μ and η pair. This rule in particular is quite hardware specific, because through control-flow execution of the program this pair would not be required, as the value of the producer P remains unchanged, and can be read by the consumer. In dynamic hardware it is needed to continually feed values through the circuit.

4 Conclusion and Future works

In conclusion, using GSA as an IR for a dynamically scheduled HLS tool demonstrates promise towards improving upon Dynamatic. In addition to that, building upon precisely specified semantics reduces the gap towards a verified translation from software to hardware, making it possible to build correct-by-construction dynamic circuits. It is currently difficult to verify the correctness of a dynamic circuit specifically because it is latency-insensitive, leading to a notion of correctness which specifies that "eventually" the outputs should be consistent. There are many topics that still need to be addressed however, such as efficient handling of memory and automatic buffering of circuits. These are features that Dynamatic already supports, and would be needed to carry out a fair performance comparison.

References

[1] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. "Theory of latency-insensitive design". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.9 (Sept. 2001). Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 1059–1076. ISSN: 1937-4151. DOI: 10.1109/43.945302.

[2] L.P. Carloni et al. "A methodology for correct-by-construction latency insensitive design". In: *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*. ISSN: 1092-3152. Nov. 1999, pp. 309–315. DOI: 10.1109/ICCAD.1999.810667.

[3] Jianyi Cheng et al. "Parallelising Control Flow in Dynamic-Scheduling High-Level Synthesis". In: *ACM Trans. Reconfigurable Technol. Syst.* (May 2023). Just Accepted. ISSN: 1936-7406. DOI: 10.1145/3599973.

[4] Ayatallah Elakhras et al. "Unleashing Parallelism in Elastic Circuits with Faster Token Delivery". In: *32nd International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2022, pp. 253–261. DOI: 10.1109/FPL57034.2022.00046.

[5] Yann Herklotz, Delphine Demange, and Sandrine Blazy. "Mechanised Semantics for Gated Static Single Assignment". en. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Boston MA USA: ACM, Jan. 2023, pp. 182–196. ISBN: 9798400700262. DOI: 10.1145/3573105.3575681.

[6] Lana Josipović, Radhika Ghosal, and Paolo Ienne. "Dynamically Scheduled High-level Synthesis". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey CALIFORNIA USA: ACM, Feb. 2018, pp. 127–136. ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174264.

[7] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814.

[8] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages". In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. PLDI '90. New York, NY, USA: Association for Computing Machinery, June 1990, pp. 257–271. ISBN: 978-0-89791-364-5. DOI: 10.1145/93542.93578.

[9] Morten Borup Petersen. "A Dynamically Scheduled HLS Flow in MLIR". MA thesis. EPFL, 2022. URL: https://infoscience.epfl.ch/record/292189.