WE INNOVATE TO SECURE YOUR BUSINESS

Analysis of the gas accounting algorithm of Cairo 1.0

Date	January 2, 2025
Version	1.0
Page count	18
Authors	CryptoExperts

Contents

1 Introduction

StarkWare is a company that develops scalability and privacy technologies for blockchain applications. Specifically, StarkWare develops a STARK-powered Layer-2 scalability engine that uses cryptographic proofs to attest to the validity of a batch of transactions. As part of StarkWare's technology suite, Cairo serves as a platform for generating STARK proofs for general computation, while Starknet is a decentralized Layer-2 network scaling Ethereum based on Cairo.

At the heart of this technology is the Cairo CPU [1], a full fledged STARK-friendly architecture tailored for proofs production and verification. This is a non-deterministic memory based machine that executes a Cairo bytecode, which is the binary compilation of CASM (Cairo Assembly). In order for the developers to be able to program in a high-level language, StarkWare developed Cairo 0, which is a (rather thin) programming layer above CASM bringing syntactic sugar and some abstractions. This programming language suffered from two issues: being very close to CASM means low-level understanding of the Cairo VM, and also leaving possible *runtime errors* of the program (e.g. through *false asserts*, invalid memory accesses violating immutable memory, etc.). This latter issue has a drastic implication regarding gas fees accounting: since a failing program cannot be proven, the sequencer cannot account for it and cannot charge the user for the "lost gas". This leaves the sequencer vulnerable to possible Denial of Services attacks from malicious actors sending failing programs to overload it. This is also frustrating for legitimate users as proofs for failing programs is expected. One would ideally seek for something similar to the "reverted transaction" feature of Ethereum.

In order to overcome these issues, StarkWare introduced Cairo 1.0, which provides a higher level language to programmers when compared to Cairo 0 with the introduction of Sierra (Safe InteRmediate RepresentAtion), an new intermediate language ensuring safety when translated to CASM. In StarkWare's context, safety means that all the possible execution paths of the compiled CASM can be proven, even the failing ones, which also enables a sound gas accounting on the program during its execution on the sequencer. This is a major step forward from Cairo 0 since no "unsafe" CASM can be produced.

The purpose of the current blog post is to dive into how Cairo 1.0 and Sierra allow to perform sound gas accounting. We will first quickly recall the main Sierra features. We will then introduce simple examples exposing why gas accounting can be complex to implement, specifically hitting the unsolvable halting problem in the more general case. Then, we show how both Sierra features, local gas wallet accounting algorithms as well as dedicated gas accounting libfuncs allow to overcome these challenges. Finally, we provide a full concrete example from Cairo 1.0 to CASM with the associated gas accounting.

2 Sierra, safe CASM And Cairo 1.0 to the rescue of Cairo 0

A gentle introduction to the motivations for Sierra as well as an under the hood overview are exposed in Nethermind's blog posts series [2, 3, 4].

In a nutshell, Sierra is a strongly typed language implementing linear typing: each variable must be consumed exactly once by the statements of a program. All the statements of a Sierra program are either invocations that cannot fail or return statements: by design, a Sierra program always terminates its execution with the return of a function, with no other possible execution branch. When translated to CASM using the dedicated compiler, all the Sierra statements translation blocks do not trigger runtime errors. This is by construction of how each statement to CASM translation is written: this requirement is checked by manual audits of the possible translations. The resulting safe CASM is ensured to be free of illegal or unallocated memory addresses dereferencing, possibly false assertions, multiple writes to the same memory cell (because of Cairo's write-once model), etc.

The building blocks of Sierra statements are called *libfuncs*: they are defined on strongly typed inputs with polymorphism as one could expect from a Rust-like high-level language (e.g. `store_temp<a>` is the libfunc associated to storing in memory the type `a`, for instance `a` can be `felt252`, `u8`, etc.).

A Sierra program can be dissected as a set of user functions, each function being made of statements. Every statement is either a *libfunc* call, or a *return* statement marking a user function return.

As presented on Figure 1, the Sequencer now only deals with Sierra code (while for Cairo 0, it was directly dealing with possibly unsafe CASM): the developer locally compiles the Cairo 1.0 code with the dedicated compiler, and then declares it by sending Sierra code. The Sequencer then compiles the Sierra code to safe CASM bytecode to deal with proofs production and execution (although execution could be performed at the Sierra level). Dealing with Sierra code allows the Sequencer to ensure soundness properties and gas accounting as we will present hereafter.

We will not provide much more details about Sierra and its syntax as they are not really needed for understanding the key concepts of gas accounting. However, the curious reader can refer to [2, 3, 4] for more insights on this topic.

As a very simple example, we provide on Listing 1 a basic Cairo 1.0 function `test()` that takes nothing as input and returns a `felt252`. On Listing 2, we show how this example is compiled to Sierra: as we can see, two libfuncs `const_as_immediate` (instantiated with the `Const<felt252,1>` type) and `store_temp` (instantiated with the `felt252` type) are used here, as well as a return statement returning the `felt` value. Finally, the resulting safe CASM compiled code is presented on Listing 3 where we can see that some Sierra abstractions are removed to produce the simplest affectation instruction.

```
fn test() -> felt252 {
    let a = 1;
    a
}
```

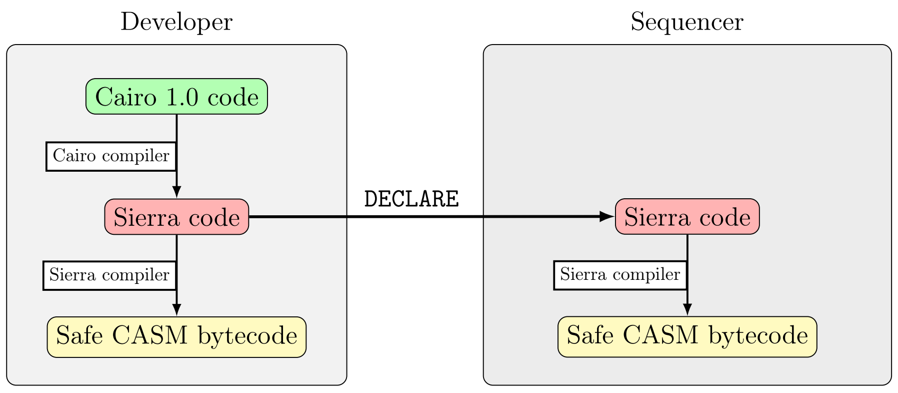


Figure 1: Overview of Cairo 1.0 and Sierra generation and running flows during contract `DECLARE` and user transaction

Listing 1: A simple Cairo 1.0 example

```
// Sierra first section: types definitions
type felt252 = felt252 [storable: true, drop: true, dup: true, zero_sized: false];
type Const<felt252, 1> = Const<felt252, 1> [storable: false, drop: false, dup:
        ↪ false, zero_sized: false];

// Sierra second section: (typed) libfuncs used in the
libfunc const_as_immediate<Const<felt252, 1>> = const_as_immediate<Const<felt252,
        ↪ 1>>;
libfunc store_temp<felt252> = store_temp<felt252>;

// Sierra third section: the functions and statements (with their ids)
// Statements indices are in comment on the right
const_as_immediate<Const<felt252, 1>>() -> ([0]); // 0
store_temp<felt252>([0]) -> ([0]); // 1
return([0]); // 2

// Sierra fourth section: the functions names and statements ids
test::test::test@0() -> (felt252);
```

Listing 2: Sierra compilation of example in Listing 1

```
[ap + 0] = 1, ap++; // Sierra statement #1 translation
ret; // Sierra statement #2 translation
```

Listing 3: CASM compilation of example in Listing 2

3 Sierra and gas accounting challenges

3.1 Sierra libfuncs gas costs and branching costs

Since each Sierra libfunc has a known translation to safe CASM, it is possible to fully evaluate the cost in gas for this libfunc. The rationale is to count the number of "steps" (i.e. roughly `ap` and `fp` affectations in CASM), as well as builtins costs (e.g. `RangeCheck`). The "cost" of a libfunc will be the sum of the costs of all its CASM instructions. There are some subtleties related to how builtins other than `RangeChecks` are handled, and how the cost is scaled depending on the type cast of the libfunc instantiation, but for the purpose of conciseness, we will not detail this here as these are not crucial for understanding how gas accounting works at high-level. For now, let us keep in mind that each CASM step is rated by convention as 100 gas units.

There is a catch, though. Most of the Sierra libfuncs have only one possible execution path in the generated CASM, i.e. the execution flow is linear and there is only one possible next statement. For these libfuncs, called non-branching, the gas cost is straightforward to evaluate as the sum of the CASM execution flow. However, some of the libfuncs can have multiple possible execution paths. These branching libfuncs are, for instance, dealing with conditional execution, statements that might "fail", enumerations values extraction, and so on. For those libfuncs, there are multiple possible gas costs: one for each execution path. These costs are called *branching costs*: Figure 2 illustrates this.

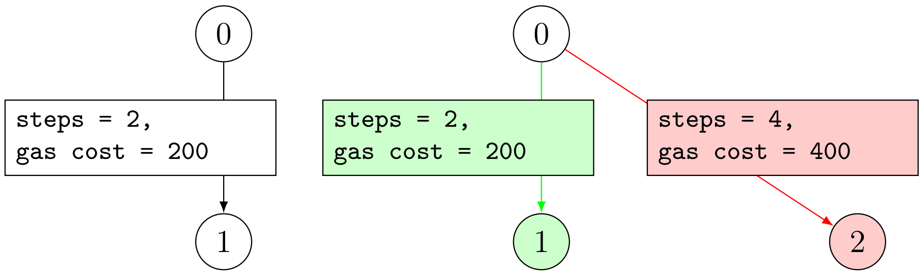


Figure 2: Illustration of a non-branching (on the left) and a branching (on the right) libfunc. The non-branching libfunc always takes 200 gas units (for 2 steps). The branching libfuncs has two execution paths yielding two possible costs: either 200 gas units, or 400 gas units.

3.2 A simple approach of gas cost evaluation of a Sierra program

3.2.1 Sierra programs and graphs

All Sierra programs can be represented as directed graphs where each node is a Sierra statement (i.e. a libfunc invocation or a `return` statement), and two nodes are connected with a directed edge following the execution flow. User functions calls use a specific libfunc `function_call`. The directed graph can be seen as the concatenation of the *control flow graph* and the *call graph* of the Sierra program.

By construction, `return` statements do not have children. Non-branching libfuncs have only one child, while branching libfuncs have at least two children. Functions calls have two children (the next statement in the current caller function and the entry point of the called function). Nodes numbering can be any ordered set: by convention, the linear numbering in the order of appearance of statements in the Sierra program is chosen.

We provide on Figure 3 an example of such a representation of a simple Sierra program made of two functions `Test1` and `Test2`: please note that this program does nothing interesting, it is exhibited for the sake of the example. Following the types declarations from lines 1 to 4 and `libfuncs` aliases declaration from line 6 to 27, we have the list of statements from line 29 to 72, from statement 0 to statement 42. The statements numbers and the number of steps they take are put in comment just after the semicolon `';'` (no explicit step means 0). We represent on the left of Figure 3 the graph representation of this program as explained above. The regular branching nodes are represented by ◯, the functions entry-points are ◇, and the `return` statements by ▭. We can see an example of branching statement for ① for instance, where `felt252_is_zero(n){fallthrough() 9(n)};` either jumps to statement ⑨ if the `felt252` variable `n` is zero, or falls through to statement ② if it is non-zero. The portions of the graph that include statements whose numbers are linearly incremented without branching are somehow self-explanatory, and it is possible to simplify the graph representation as shown on Figure 4 (we also represent the cumulative number of steps of the collapsed nodes on the edges).

3.2.2 Sierra functions and programs worst execution gas consumption

Now given the graph representation of a Sierra program, it is possible to compute the *worst execution path* in terms of gas consumption, which is equivalent to the number of CASM steps[1]. This quantity is the minimum amount of gas needed for the Sierra program execution in the worst case. In a function, an execution path is the set of directed edges from the entry-point to one of the possible return values, split across the branches of the control flow graph. Thanks to Sierra properties, any possible execution must be one of such paths. Then, the gas consumption cost of a given path is simply the *sum* of the gas costs of each statement along this path. Finally, the worst execution path of a function is simply the maximum value of the costs of all the possible paths. Regarding the specific case of function calls, the cost of the calling statement is simply the worst execution cost of the called function. Finally, the worst execution gas cost of a Sierra program corresponds to the worst execution cost of the top level function of this program.

Informally, computing the execution path gas cost can be algorithmically performed with graph traversing: from a `return` statement of a function representing a possible execution path, roll back to the entry-point of the function by accumulating the gas cost of each statement. The gas cost of this path will be represented by the sum of all these costs. When encountering a function call, either the worst execution cost of this function is already computed: then use it in the cumulative sum. Or this cost is not yet computed, and apply the algorithm first to the called function to get it (i.e. we traverse the call graph of the program from the leaf functions to the top level ones). When applied to the simple example of Figure 3 with the steps represented on Figure 4, the worst gas cost of `Test1`

[1]Things are actually a bit more subtle because of possible adjustments that are not explained in this post for the sake of clarity and conciseness.

is 8 steps, i.e. 800 gas units, following the path $\langle 0 \rangle \rightarrow \textcircled{1} \rightarrow \textcircled{10} \rightarrow \boxed{29}$. The `Test2` function has a worst gas cost of 4 steps, i.e. 400 gas units, with an equivalent cost of the two possible execution paths $\langle 30 \rangle \rightarrow \textcircled{31} \rightarrow \boxed{36}$ and $\langle 30 \rangle \rightarrow \textcircled{37} \rightarrow \boxed{42}$.

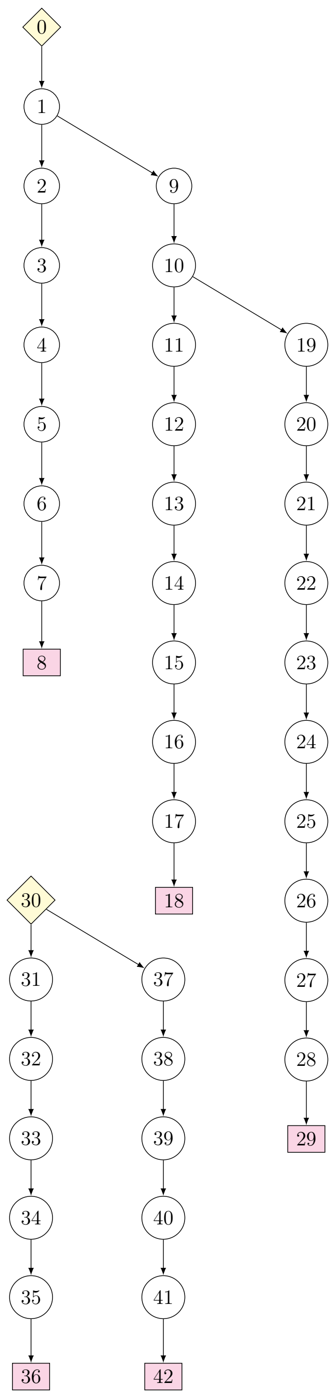
3.3 Hitting the halting problem

We could conclude from the simple approach presented in Section 3.2 that evaluating worst case gas cost of a Sierra program (and hence a Cairo 1.0 program through compilation to Sierra) is a simple matter. However, we have neglected on purpose the most challenging cases: *using loops and recursion.*

Sierra offers conditional and unconditional branching libfuncs, allowing to have loops whose output condition might be unknown at compilation time (i.e. only known at execution time), possibly yielding infinite execution time. Similarly, recursive functions calls can be used in a Sierra program, and the recursion stopping condition might also be unknown at compilation time. In such situations, the very simple graph traversal algorithm to compute worst execution costs of Sierra programs cannot be applied since *cycles* will actually appear in the graph representing the program. Such cycles prevent the construction of a path rolling back from `return` statements to root entry-point nodes, since at the nodes representing the conditional branching of the cycle, we cannot decide *how many times* these cycles are taken. All-in-all, the worst execution gas cost with a cycle will be represented by a fixed cost taking into account one cycle execution, plus a certain *unknown* number of cycles costs.

Let us take a very simple graph example to illustrate the issue. We represent on Figure 5 a graph representing a fictitious Sierra program containing a loop through the usage of an unconditional `jump` at node $\textcircled{18}$ in the function with entry-point at node $\langle 10 \rangle$. When applying the algorithm to compute worst case execution cost to this graph from `return` statement $\boxed{25}$, we do not known which path to choose to continue the graph traversal when arriving at the branching node $\textcircled{12}$: we can either go to $\langle 10 \rangle$, or go to $\textcircled{18}$ depending on a predicate we cannot statically guess in the most general case. This loop prevents the computation of the gas cost of function at $\langle 10 \rangle$, and by extension (through the call graph of the program) the gas cost of the caller function at $\langle 0 \rangle$.

The issue raised here with gas cost accounting is equivalent to the *static analysis of programs* general problem, which hits the computer theory fundamental *halting problem*!



```
type felt252 = felt252;
type GasBuiltin = GasBuiltin;
type RangeCheck = RangeCheck;
type NonZeroInt = NonZero<felt252>;

libfunc branch_align = branch_align;
libfunc felt252_add = felt252_add;
libfunc felt252_const_0 = felt252_const<0>;
libfunc felt252_const_1 = felt252_const<1>;
libfunc felt252_const_minus_1 = felt252_const<-1>;
libfunc felt252_drop = drop<felt252>;
libfunc felt252_non_zero_drop = drop<NonZeroInt>;
libfunc felt252_dup = dup<felt252>;
libfunc felt252_is_zero = felt252_is_zero;
libfunc felt252_sub_1 = felt252_sub_const<1>;
libfunc felt252_unwrap_non_zero = unwrap_non_zero<felt252>;
libfunc withdraw_gas = withdraw_gas;
libfunc jump = jump;
libfunc redeposit_gas = redeposit_gas;
libfunc rename_felt252 = rename<felt252>;
libfunc revoke_ap_tracking = revoke_ap_tracking;
libfunc store_temp_felt252 = store_temp<felt252>;
libfunc store_temp_gb = store_temp<GasBuiltin>;
libfunc store_temp_rc = store_temp<RangeCheck>;
libfunc dup_felt = dup<felt252>;
libfunc drop_felt = drop<felt252>;
libfunc drop_nz_felt = drop<NonZeroInt>;

revoke_ap_tracking() -> (); // 0
felt252_is_zero(n) { fallthrough() 9(n) }; // 1 Step = 1
branch_align() -> (); // 2
store_temp_rc(rc) -> (rc); // 3 Step = 1
store_temp_gb(gb) -> (gb); // 4 Step = 1
felt252_const_1() -> (one); // 5
store_temp_felt252(one) -> (one); // 6 Step = 1
drop_felt(z) -> (); // 7
return(rc, gb, one); // 8
branch_align() -> (); // 9
felt252_is_zero(z) { fallthrough() 19(z) }; // 10 Step = 1
branch_align() -> (); // 11
store_temp_rc(rc) -> (rc); // 12 Step = 1
store_temp_rc(rc) -> (rc); // 13 Step = 1
store_temp_gb(gb) -> (gb); // 14 Step = 1
felt252_const_1() -> (one); // 15
store_temp_felt252(one) -> (one); // 16 Step = 1
drop_nz_felt(n) -> (); // 17
return(rc, gb, one); // 18
branch_align() -> (); // 19
felt252_unwrap_non_zero(z) -> (z); // 20
felt252_unwrap_non_zero(n) -> (n); // 21
drop_felt(n) -> (); // 22
store_temp_rc(rc) -> (rc); // 23 Step = 1
store_temp_rc(rc) -> (rc); // 24 Step = 1
store_temp_rc(rc) -> (rc); // 25 Step = 1
store_temp_rc(rc) -> (rc); // 26 Step = 1
store_temp_gb(gb) -> (gb); // 27 Step = 1
store_temp_felt252(z) -> (z); // 28 Step = 1
return(rc, gb, z); // 29
// Test 2
felt252_is_zero(n) { fallthrough() 37(n) }; // 30 Step = 1
branch_align() -> (); // 31
store_temp_rc(rc) -> (rc); // 32 Step = 1
store_temp_gb(gb) -> (gb); // 33 Step = 1
felt252_const_1() -> (one); // 34
store_temp_felt252(one) -> (one); // 35 Step = 1
return(rc, gb, one); // 36
branch_align() -> (); // 37
store_temp_rc(rc) -> (rc); // 38 Step = 1
store_temp_gb(gb) -> (gb); // 39 Step = 1
felt252_unwrap_non_zero(n) -> (n); // 40
store_temp_felt252(n) -> (n); // 41 Step = 1
return(rc, gb, n); // 42

Test1@0(rc: RangeCheck, gb: GasBuiltin, n: felt252, z: felt252) -> (RangeCheck, GasBuiltin, felt252);
Test2@30(rc: RangeCheck, gb: GasBuiltin, n: felt252) -> (RangeCheck, GasBuiltin, felt252);
```

Figure 3: The graph representation of an example Sierra code with two functions `Test1` and `Test2`. 0 is the entry-point of `Test1`, and 30 is the entry-point of `Test2`. The nodes 8, 18 and 29 are return statements of `Test1`, 36 and 42 are return statements of `Test2`.

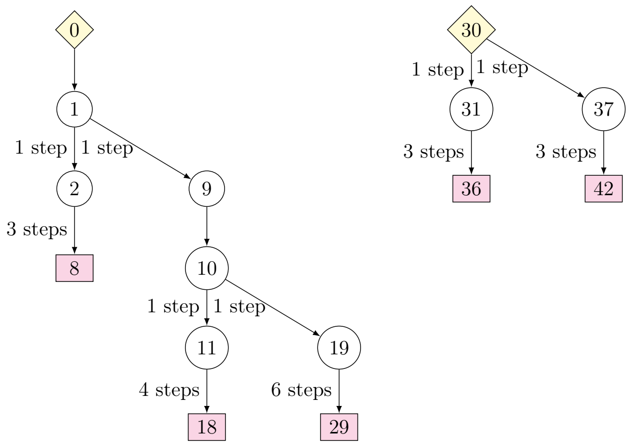


Figure 4: Simplified graphs for the Sierra code example provided in Figure 3, with collapsed number of steps in each branch.

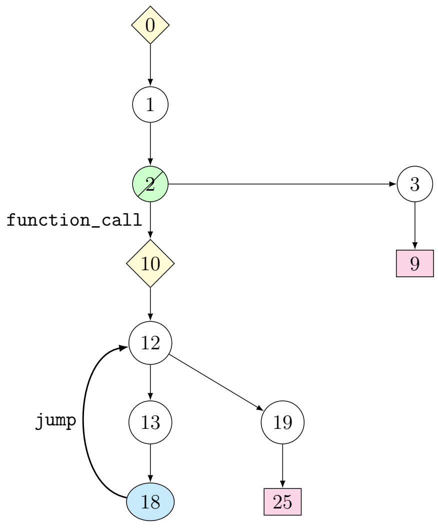


Figure 5: Simple example of a graph containing a function call at statement 2 and an unconditional `jump` at statement 18. This represents two functions, one with entry-point at statement 0 that calls another one with entry-point at statement 10. The first function has a "linear" statement execution from 0 to its return at 9, i.e. no branch except for the function call at 2. The second function has two branches with a two-branches statement at 12, and a loop in one of the branch through 18.

4 Trying to overcome the halting problem

4.1 A gas dedicated libfunc: `withdraw_gas`

Now that we have seen that *static analysis* is a dead end in the most general case, one could think that a solution using *dynamic analysis* would be the way to go. Given the user gas balance that is denoted `GC` (for Gas Counter), a very simple approach to try to solve the issue (summarized in [5]) would be at each Sierra statement (inside the libfunc's safe CASM code) to check if `GC` is large enough to execute the current statement, and if yes reduce `GC` by the dynamic gas cost of the libfunc (depending on the branch if this is a branching libfunc). Although such an approach would be valid, it has the drastic drawback of *consuming steps* for handling `GC` (as well as reducing the runtime performance): this means that the gas measurement code itself will heavily impact the gas consumption of the original user code, which is not acceptable. There is no reason to charge the user for elements that are unrelated to his Cairo 1.0 program!

Hence, one must find a way to provably and soundly charge the user with the correct amount of gas without significantly adding artificial steps to the original code. StarkWare Cairo developers team have found a clever way of *mixing static analysis and lightweight dynamic code analysis* to achieve this using gas dedicated libfuncs added during Cairo to Sierra compilation. The rationale is the following:

- Static analysis for the local wallet `LW`: without cycles, the worst case execution gas cost of a Sierra program is easy to compute using the algorithm previously presented in Section 3.2. Hence, a first step consists in transforming the graph representing the Sierra program to an *acyclic graph* by "breaking the cycles": this is achieved by enforcing during compilation the presence of a dedicated libfunc called `withdraw_gas` along the cycle[2]. This is a branching libfunc that has a success and a failure branch: in the success branch, the user has enough gas to continue the execution, and in the failure branch the user has not enough gas for the success case (but there is enough to handle the usually more lightweight failure case). From a graph perspective, the cycle is "broken" by removing the edge corresponding to the success branch of `withdraw_gas`. The result of this static worst case computation is called the local wallet `LW`, and can be seen as the minimum amount of gas needed for the program executing at least one cycle in the worst execution path.

- Runtime code: the `withdraw_gas` libfunc safe CASM translation actually interacts with the user global gas counter `GC`. Following the same idea as the full dynamic check for gas previously presented, `GC` is checked against a value that will ensure enough gas for the worst execution path from this node to any `return` statement. If the check is OK, we take the success branch and remove the proper amount from `GC`. If the check is not OK, we take the failure branch without touching `GC`, and in this branch we should be (by static analysis) ensured that we have enough gas. Actually, gas withdrawal can be seen as a stop to the pit for a refuel that will inject enough gas boost in the local wallet for possibly one more cycle. Using this strategy, each time a cycle is taken we are ensured to have enough gas or fail, hence overcoming the halting problem at a very low cost in terms of additional steps!

[2]This can be ensured through the compiler that easily detects cycles in the Sierra program graph equivalent, and ensures that at least one `withdraw_gas` is placed in one node along this path.

4.2 Solving the gas constraints and interactions between LW and GC

From the previous description of how gas withdrawal somehow solves the gas accounting challenges, there is a missing piece regarding the amount of gas that must be checked at `withdraw_gas` nodes for possible gas refuel. This is in fact resolved through linear constraints deduced from the graph analysis.

Let us demonstrate how this works on the example presented in Figure 5. We will denote by $\Gamma(i)$ the value of the local wallet at node i, i.e. the worst gas cost from this node to any of the possible paths leading to `return` (but *not following* the `withdraw_gas` success branch). We will also denote by $\gamma_{i \to j}$ the gas cost of a transition from a libfunc at node i to its successor j (as we have seen, for branching libfuncs the gas cost might not be the same in each branch). As a matter of fact, $\Gamma(i)$ is the maximum amount among the $\Gamma(j) + \gamma_{i \to j}$ for all the possible children j of i. Also, by construction, Γ is zero at the `return` statements (since the local wallet is computed by rolling back from `return` to entry-point).

We represent on Figure 6 the example with fictitious values for the sake of the explanation: let us assume without loss of generality that the node 12 is the `withdraw_gas` invocation in the cycle, with a success branch $12 \to 13$ and a failure branch $12 \to 19$. We have represented the gas reinjection by ε_{12}: this is the unknown variable value to be found. As we have previously stated, all the $\Gamma(i)$ for the local wallet values have been computed here as worst execution cost from i while removing the success edges of `with-draw_gas` by breaking the cycles. The value of ε_{12} will be found by balancing the local wallet on each side of the removed edge.

First, we can see that there is only one path to a return statement from 13: $13 \to 18 \to 12 \to 19 \to \boxed{25}$, yielding $\Gamma(13) = \gamma_{13\to18} + \gamma_{18\to12} + \gamma_{12\to19} + \gamma_{19\to25} = 800$. When considering the missing edge $12 \to 13$, *we must ensure the equation* $\Gamma(12) + \varepsilon_{12} = \Gamma(13) + \gamma_{12\to13}$ (note the addition of the gas boost ε_{12} here). This translates to $\varepsilon_{12} = \Gamma(13) + \gamma_{12\to13} - \Gamma(12) = 800 + 270 - 500 = 570$. Consequently, at the `withdraw_-gas` statement, we need to inject 570 gas units from the global gas counter `GC` into the local wallet to ensure at least one cycle execution (in the worst case). One can notice that solving such equations can produce a negative value for ε at the `withdraw_gas` statement, which will usually mean that the failure branch takes actually more gas than the success one, and that the local wallet *contains enough gas* so no gas should be pulled from `GC`: this can only happen when this `withdraw_gas` is not part of a cycle[3], which will occur when explicitly calling this libfunc when not needed.

Solving gas flow constraints from the global counter to the local wallet (and vice-versa, see below) is one of the purposes of the Sierra compiler, which makes use of various graph traversals to achieve this in linear time. The constants values computed by solving these constraints are then injected in the safe CASM generated for the libfunc, with a test to check if `GC` has enough gas or not for `withdraw_gas`.

[3]Or else, the failure branch will be one of the branches of the nodes of the cycle maximizing the cost, so its cost cannot be less.

Other gas related libfuncs exist, notably **`redeposit_gas`** as it does the reverse operation of **`withdraw_gas`**: it redeposits the leftover gas into `GC`. Indeed, when taking branches that are not the "worst case", the local wallet will be overestimated (as the maximum cost of all the possible branches), and in the end an amount of gas can be given back to the user in these branches. Besides this, coupon related libfuncs have been recently introduced to deal with precharging the cost of functions before calling them (this can be useful when "heavy operations" are expected to be called in the failure branch of a **`withdraw_gas`**). We will not enter into more details about these libfuncs in this post as our purpose is mostly to provide a high-level explanation on gas accounting, and withdrawal is the main operation for this. Also, while **`withdraw_gas`** is enforced at Sierra compilation time whenever there is a cycle (or a compilation error will be triggered), redepositing gas and coupons[4] libfuncs must be called willingly by the programmer as of the current version of the Cairo compiler.

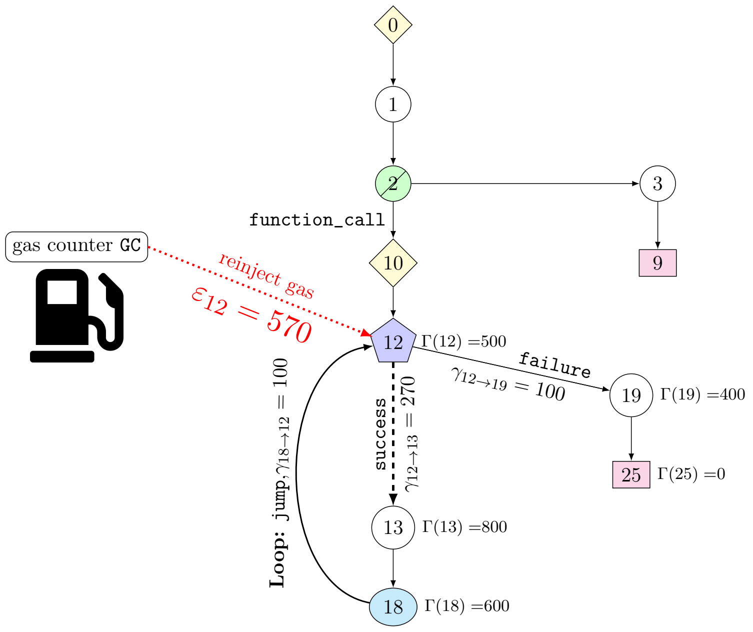


Figure 6: Reinjecting gas in the local wallet from the gas counter to compensate for cycles

4.3 How is the gas accounting problem solved: the big picture

Now that we have explained how gas accounting can be performed by a mix of static analysis (with the local wallet) and runtime updates (of the global gas counter), let us recall the key features that allow to solve our original challenge: bring a sound and provable gas accounting mechanism to Cairo 1.0, ensuring that no Denial of Service can be performed

[4]Also, coupons are still an experimental feature.

on the sequencer (i.e. the sequencer does not spend gas without retribution) by malicious actors, and that the end-user is happy with a proof of his concrete gas consumption for the contracts he executes:

- First of all, the compilation of the Cairo 1.0 code to the Sierra intermediate language allows to remove any source of runtime error since each libfunc is translated to safe CASM. A Sierra program is hence ensured to execute from its entry point to one of the `return` statements. This is useful for a static analysis of the graph representing the program, allowing to compute the local wallet `LW` of worst case execution gas cost of the program (this can be done using the fact that each libfunc to safe CASM translation has an associated known gas cost).
- When cycles are present in the Cairo 1.0 code (e.g. using loops or recursion), the Cairo to Sierra compiler injects a `withdraw_gas` statements in the cycle. This allows, using graph traversal and constraints solving, to dynamically reinject (at runtime) the proper amount of gas (statically computed) whenever cycles are executed again.
- The Sierra program is the one sent to the sequencer, and the sequencer using the Sierra to safe CASM compiler is the one performing the previous computations. Notably, the presence of `withdraw_gas` statements along cycles in the Sierra program is enforced here to avoid possible attacks at the Sierra level.
- After the Sierra to CASM compilation, the local wallet is known to the runner so that an initial amount of user gas can be used from the global gas counter. Then, when a cycle is reached, more gas is (possibly) dynamically withdrawn from the global user counter. Two situations are possible during this withdrawal attempt: either enough gas is present in the global counter `GC`, in which case it is decremented with the proper amount in CASM. Or the user does not have enough gas, and the failure branch is taken with an assertion that `GC` is not large enough.

5 Putting it all together: a concrete Cairo 1.0 example

In this section, we will show all the concepts presented above in action using a concrete Cairo 1.0 example, from the Cairo code to the produced safe CASM, with the details of gas accounting. In order to cover the gas withdrawal concepts, we will use the recursive Fibonacci implementation example from `cairo/examples/fib.cairo`. The simple Fibonacci computation code is presented on Listing 4, and we can see there is only one recursive function `fib`. The compilation to Sierra, using the `cairo-compile` program, is presented on Figure 7. The resulting safe CASM, compiled from Sierra using the `sierra-compile` program, is presented on Listing 5 (with, for each CASM statement, the corresponding Sierra source statement).

5.1 Analysis of Sierra gas accounting

We present on Figure 7 the Sierra code along with its (simplified) graph representation. We have omitted for the sake of clarity the declaration part of the Sierra code. In the following analysis, we will not detail each libfunc as we are more interested in gas accounting than in how Sierra libfuncs work.

As we can see, a notable thing to notice is the *presence of a* `withdraw_gas` *statement* 1 in line 6: the Cairo compiler detected the cycle induced by the recursion and transparently produced this withdrawal. Beyond that, the program contains three branching paths (three `return` statement) and the recursive function call in statement $\textcircled{\cancel{26}}$. We have put in the comments of the Sierra code the number of steps `RangeChecks` (rc) taken by each statement (no explicit value means 0). For the record, 1 step takes 100 gas units and one range check takes 70 gas units. These yield the gas transitions in the simplified graph presented on the edges of Figure 8. For instance, the cost of the success branch of the gas withdrawing statement 1 is 3 steps and 1 range check, i.e. 370 gas units, while the cost of the failing branch is 4 steps and 1 range check, i.e. 470 gas units, and so on.

As previously presented, in order to compute the local wallet value at each statement, we break the cycle by removing the success branch $1 \to \textcircled{2}$. When doing so, the local wallet $\Gamma(i)$ at statement i is computed as the worst gas cost among all the possible branches in the children, while $\Gamma = 0$ for return statements. For instance, coming from $\boxed{42}$, we have 8 steps from $\textcircled{28}$, meaning $\Gamma(28) = 800$ gas units. Regarding the branching statements, e.g. $\textcircled{5}$, the worst branch is used for computing Γ ($\textcircled{5} \to \textcircled{15}$ in this case). Finally, the specific case of the function call $\textcircled{\cancel{26}} \to 0$ takes into account the worst execution path from here, the only path here being $\textcircled{\cancel{26}} \to 0 \to 1 \to \textcircled{28} \to \boxed{42}$, yielding $\Gamma(26) = 1470$.

Now that the local wallet values have been computed at each statement, one must solve the constraints to find the gas boost ε_1 pulled from `GC` during gas withdrawal. As presented in Section 4.2, this is done by solving $\Gamma(1) + \varepsilon_1 = \Gamma(2) + \gamma_{1\to 2} = \Gamma(2) + 370$, i.e. $\varepsilon_1 = 2170 + 370 - 1270 = 1270$ in our case.

5.2 Analysis of the translation to safe CASM

We can observe on the safe CASM compiled code on Listing 5 that the value $\varepsilon_1 = 1270$ is injected in the generated assembly. The code reads as follows:

- First of all, a *hint* stores in `[ap + 0]` the result of checking if 1270 is $\leq$ than the global gas counter `GC` (that is stored in `[fp - 6]`).
- If the previous comparison is `false`, this means that *there is enough gas* and we jump to the relative instruction 7 at line 6 (this is the success branch). The instruction removes 1270 from the gas counter `[fp - 6]`, and stores this in the temporary variable `[ap + 0]`. Then, this value is range checked (using the `RangeCheck` builtin stored in `[fp - 7]`). We recall that a range check on a value v asserts that $0 \leq v < 2^{128}$. In our case, we assert that $0 \leq \texttt{GC} - 1270 < 2^{128}$.

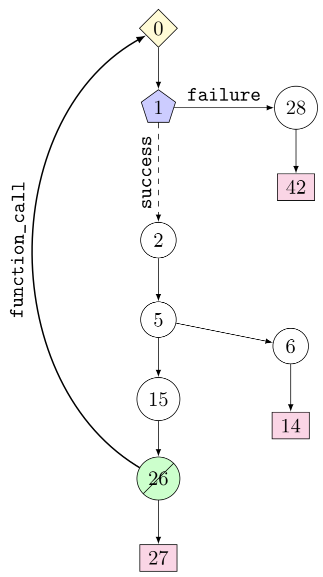
- If the previous comparison is `true`, this means that *there is not enough gas* and we follow the failure branch at line 3. The instruction there stores in temporary variable `[ap + 0]` the value $\texttt{GC} + 2^{128} - 1270$ (note that we have $2^{128} - 1270 = 340282366920938463463374607431768210186$, hence the large encoded value in the assembly). Then, at line 5 a range check asserts that $0 \leq \texttt{GC} + 2^{128} - 1270 < 2^{128}$, i.e. $\texttt{GC} < 1270$. Finally, at line 5 we jump to relative 26, that corresponds to line 24, which corresponds to Sierra statement 32 (the first statement after $\textcircled{28}$ that is not an untranslated Sierra abstraction).

```
// Calculates fib...
pub fn fib(a: felt252, b: felt252, n: felt252) -> felt252 {
    match n {
        0 => a,
        _ => fib(b, a + b, n - 1),
    }
}
```

Listing 4: Recursive Fibonacci Cairo 1.0 example

```
1  %{ memory[ap + 0] = 1270 <= memory[fp + -6] %}
2  jmp rel 7 if [ap + 0] != 0, ap++; // 0, Sierra statement #1
3  [ap + 0] = [fp + -6] + 340282366920938463463374607431768210186, ap++; // 2, Sierra
           statement #1
4  [ap + -1] = [[fp + -7] + 0]; // 4, Sierra statement #1
5  jmp rel 26; // 5, Sierra statement #1
6  [fp + -6] = [ap + 0] + 1270, ap++; // 7, Sierra statement #1
7  [ap + -1] = [[fp + -7] + 0]; // 9, Sierra statement #1
8  [ap + 0] = [fp + -7] + 1, ap++; // 10, Sierra statement #4
9  jmp rel 10 if [fp + -3] != 0; // 12, Sierra statement #5
10 [ap + 0] = [ap + -1], ap++; // 14, Sierra statement #11
11 [ap + 0] = [ap + -3], ap++; // 15, Sierra statement #12
12 [ap + 0] = 0, ap++; // 16, Sierra statement #13
13 [ap + 0] = 0, ap++; // 18, Sierra statement #13
14 [ap + 0] = [fp + -5], ap++; // 20, Sierra statement #13
15 ret; // 21, Sierra statement #14
16 [ap + 0] = [ap + -1], ap++; // 22, Sierra statement #21
17 [ap + 0] = [ap + -3], ap++; // 23, Sierra statement #22
18 [ap + 0] = [fp + -4], ap++; // 24, Sierra statement #23
19 [ap + 0] = [fp + -5] + [fp + -4], ap++; // 25, Sierra statement #24
20 [fp + -3] = [ap + 0] + 1, ap++; // 26, Sierra statement #25
21 call rel -28; // 28, Sierra statement #26
22 ret; // 30, Sierra statement #27
23 %{ memory[ap + 0] = segments.add() %}
24 ap += 1; // 31, Sierra statement #32
25 [ap + 0] = 375233589013918064796019, ap++; // 33, Sierra statement #34
26 [ap + -1] = [[ap + -2] + 0]; // 35, Sierra statement #35
27 [ap + 0] = [fp + -7] + 1, ap++; // 36, Sierra statement #39
28 [ap + 0] = [fp + -6], ap++; // 38, Sierra statement #40
29 [ap + 0] = 1, ap++; // 39, Sierra statement #41
30 [ap + 0] = [ap + -5], ap++; // 41, Sierra statement #41
31 [ap + 0] = [ap + -6] + 1, ap++; // 42, Sierra statement #41
32 ret; // 44, Sierra statement #42
```

Listing 5: Recursive Fibonacci (Sierra code on Figure 7) compilation to CASM



```
// Types and libfuncs declarations removed for clarity
...

disable_ap_tracking() -> (); // 0
// Following withdraw_gas: [success = 3 steps, 1 rc], [failure = 4 steps, 1 rc]
withdraw_gas([0], [1]) { fallthrough([5], [6]) 28([7], [8]) }; // 1

branch_align() -> (); // 2
dup<felt252>([4]) -> ([4], [9]); // 3
store_temp<RangeCheck>([5]) -> ([5]); // 4, 1 step
// Following felt252_is_zero: 1 step for success and failure
felt252_is_zero([9]) { fallthrough() 15([10]) }; // 5

branch_align() -> (); // 6
drop<felt252>([4]) -> (); // 7
drop<felt252>([3]) -> (); // 8
struct_construct<Tuple<felt252>>([2]) -> ([11]); // 9
enum_init<core::panics::PanicResult::<(core::felt252,)>, 0>([11]) -> ([12]); // 10
store_temp<RangeCheck>([5]) -> ([5]); // 11, 1 step
store_temp<GasBuiltin>([6]) -> ([6]); // 12, 1 step
store_temp<core::panics::PanicResult::<(core::felt252,)>>([12]) -> ([12]); // 13, 3 steps
return([5], [6], [12]); // 14

branch_align() -> (); // 15
drop<NonZero<felt252>>([10]) -> (); // 16
dup<felt252>([3]) -> ([3], [13]); // 17
felt252_add([2], [13]) -> ([14]); // 18, 1 step
const_as_immediate<Const<felt252, 1>>() -> ([15]); // 19
felt252_sub([4], [15]) -> ([16]); // 20, 1 step
store_temp<RangeCheck>([5]) -> ([5]); // 21, 1 step
store_temp<GasBuiltin>([6]) -> ([6]); // 22, 1 step
store_temp<felt252>([3]) -> ([3]); // 23, 1 step
store_temp<felt252>([14]) -> ([14]); // 24, 1 step
store_temp<felt252>([16]) -> ([16]); // 25, 1 step
function_call<user@fib::fib::fib>([5], [6], [3], [14], [16]) -> ([17], [18], [19]); // 26, 2 steps for call
return([17], [18], [19]); // 27

branch_align() -> (); // 28
drop<felt252>([3]) -> (); // 29
drop<felt252>([4]) -> (); // 30
drop<felt252>([2]) -> (); // 31
array_new<felt252>() -> ([20]); // 32, 1 step
const_as_immediate<Const<felt252, 375233589013918064796019>>() -> ([21]); // 33
store_temp<felt252>([21]) -> ([21]); // 34, 1 step
array_append<felt252>([20], [21]) -> ([22]); // 35, 1 step
struct_construct<core::panics::Panic>() -> ([23]); // 36
struct_construct<Tuple<core::panics::Panic, Array<felt252>>>([23], [22]) -> ([24]); // 37
enum_init<core::panics::PanicResult::<(core::felt252,)>, 1>([24]) -> ([25]); // 38
store_temp<RangeCheck>([7]) -> ([7]); // 39, 1 step
store_temp<GasBuiltin>([8]) -> ([8]); // 40, 1 step
store_temp<core::panics::PanicResult::<(core::felt252,)>>([25]) -> ([25]); // 41, 3 steps
return([7], [8], [25]); // 42

fib::fib::fib@0([0]: RangeCheck, [1]: GasBuiltin, [2]: felt252, [3]: felt252, [4]: felt252) -> (RangeCheck, GasBuiltin, core::panics::PanicResult::<(core::felt252,)>);
```

Figure 7: The Sierra compiled code and graph representation of the Cairo 1.0 code of Listing 4.

References

[1] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Report 2021/1063, 2021. https://ia.cr/2021/1063.

[2] Mathieu. Under the hood of Cairo 1.0: Exploring Sierra [Part 1], May 2023. https://www.nethermind.io/blog/under-the-hood-of-cairo-1-0-exploring-sierra-part-1.

[3] Mathieu. Under the hood of Cairo 1.0: Exploring Sierra [Part 2], April 2023. https://www.nethermind.io/blog/

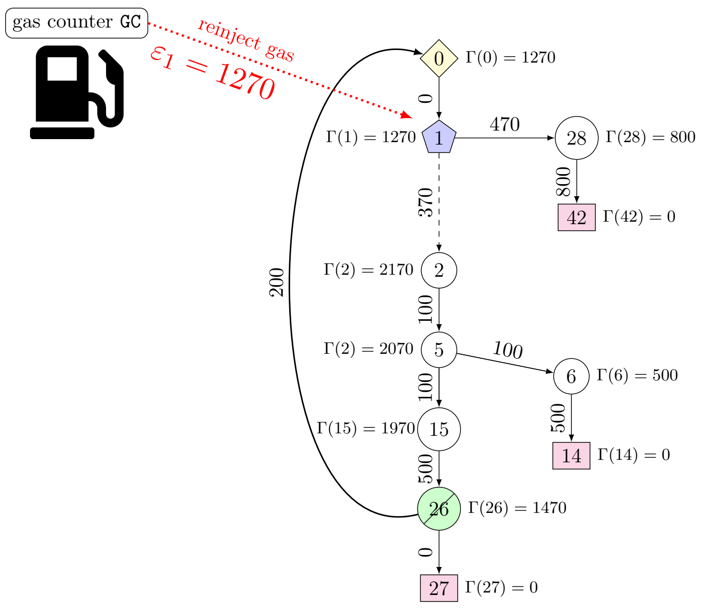


Figure 8: Local wallet values Γ computation at each node for the Sierra Fibonacci function of Figure 7, and computation of the gas withdrawal constraint at `withdraw_gas` node 1, leading to gas reinjection of 1270 gas units.

`under-the-hood-of-cairo-1-0-exploring-sierra-part-2`.

[4] Mathieu. Under the hood of Cairo 1.0: Exploring Sierra [Part 3], Marsh 2023. `https://www.nethermind.io/blog/under-the-hood-of-cairo-1-0-exploring-sierra-part-3`.

[5] Ori Ziv. Not Stopping at the Halting Problem. StarkWare Sessions 23, 02 2023. `https://www.youtube.com/watch?v=wYxUedcdVZ4`.