Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea triennale in Informatica

Performance,ottimizzazioni e best-practice nei contratti Solidity di Ethereum

Relatore:
Chiar.mo Prof.
Cosimo Laneve

Presentata da:
Milos Costantini

II Sessione
Anno Accademico 2017-2018

Indice

Capitolo 1

Introduzione

1.1 Il registro distribuito o "blockchain"

Nata nel 2009 in seguito all'invenzione del bitcoin, negli ultimi anni si è sempre più diffusa la buzzword "blockchain"; non solo in ambito informatico ma anche finanziario e politico. Per blockchain si intende una continua e crescente lista di record (detti blocchi), collegati tra di loro e messi in sicurezza mediante tecniche crittografiche. Una blockchain viene utilizzata come un registro pubblico distribuito, capace di registrare transazioni tra due o più parti, in modo permanente e facilmente verificabile. Come tutte le grandi invenzioni non sono frutto di un solo "colpo di genio", ma scaturiscono da uno storico di tentativi formato da un insieme di tecnologie create, fallite, dimenticate ed infine rinate. Se si parla di blockchain, è impossibile iniziare senza parlare dell'origine del Bitcoin.

1.2 Prima del bitcoin

Come ogni tecnologia che utilizza bit per rappresentare valori che possono essere scambiati per beni o servizi, le valute digitali sono strettamente legate all'evoluzione della crittografia [1]. Infatti, negli anni ottanta, quando la crittografia iniziò a diffondersi molti ricercatori provarono ad utilizzarla per implementare monete digitali. Nonostante funzionassero, questi primi esperimenti erano centralizzati, quindi sensibili a leggi governative ed attacchi hacker. Come le banche, per registrare le transazioni utilizzavano una camera di compensazione centrale, acquisendone tutte le debolezze. Queste valute emergenti vennero in breve tempo sommerse da contenziosi e scomparvero, non senza crolli devastanti.

1.3 Il Bitcoin

Nel 2009, sotto lo pseudonimo di *Satoshi Nakamoto* venne pubblicato un articolo scientifico chiamato "Bitcoin: an electronic peer to peer cash system." [2]. Nakamoto ha combinato varie invenzioni scoperte precedentemente quali B-money e

HashCash per arrivare alla creazione di un sistema di contante elettronico completamente decentralizzato e indipendente da qualsiasi autorità centrale per l'emissione di moneta o per la validazione delle transazioni. L'invenzione chiave è stata quella di utilizzare un algoritmo *proof-of-work*, simile a quello inventato da HashCash ma utilizzandolo per validare le transazioni. Il network bitcoin prese vita nel 2009, basato sull'omonimo paper, in seguito fu aggiornato da molti programmatori diffondendosi senza sosta, arrivando sino ad oggi con un marketcap di circa 200 miliardi di dollari e con una forte community attiva nello sviluppo.

1.4 Funzionamento di una blockchain e il ruolo del miner.

Questo testo non ha intenzione di spiegare dettagliatamente il funzionamento di una blokchain, verranno introdotti i concetti base per parlare di smart-contract, ovvero una delle principali tecnologie che si basano su blockchain. Analizzare i dettagli tecnici (o politici) sui vari algoritmi di consenso (Proof-of-*) non è importante ai fini di questa tesi. I dati della struttura dati blockchain sono un'ordinata, back-linked list di blocchi di transazioni. In sostanza, è un una lista(catena/chain) di record sempre crescente, immutabile e permanente.Nel suo paper Satoshi Nakamoto propose una soluzione al problema dei generali bizantini,che consiste nel concordare su un corso di azioni scambiandosi informazioni su un network non affidabile e potenzialmente compromesso, rivoluzionando così i database distribuiti. La soluzione di Nakamoto prevede l'utilizzo del proof-of-work (inventato da Hash Cash per evitare DDoS) per raggiungere un consenso senza autorità centrale. Viene introdotto quindi il ruolo del miner, un partecipante attivo al network (bitcoin o altro) che ha il ruolo di " verificatore " delle transazioni che avvengono nella rete. A differenza di quanto si possa pensare il miner non crea monete ma vengono rilasciate dal protocollo con queste dinamiche:
Ogni x minuti inizia una gara computazionale. I Miners, che raccolgono le transazioni che avvengono nel mondo in blocchi, partecipano a questa gara. Il vincitore della gara ha il diritto di scrivere in modo atomico (aggiungere il blocco alla blockchain) sul registro distribuito. Come incentivo a questa attività di validazione e garanzia il protocollo premia il miner con un block reward e con le fees contenute nel blocco. Questo sistema garantisce la sicurezza e l'integrità dei dati. Un attacco DDoS sarebbe troppo dispendioso, se un attaccante provasse a far minare blocchi vuoti gli costerebbe miliardi di USD solamente rallentare la rete per pochi minuti. Anche il problema principale delle valute digitali, ovvero il double-spending è reso impossibile da questo sistema. Il database distribuito viene aggiornato da un solo miner alla volta, e in caso di spit i miners seguono sempre il branch con più potenza computazionale (hash rate dato che si risolvono puzzle hash). Sostanzialmente è l'incentivo economico a garantire la sicurezza della rete, maggiore è l'hash rate, più è difficile e meno conveniente avere comportamenti malevoli verso di essa. Durante il proof-of-work, si integra l'hash del blocco precedente e lo si linka a quello nuovo. Provare a modificare una transazione equivale a modificare anche tutti i blocchi precedenti, ovvero hackerare nello stesso istante il 50+1% dei nodi sparsi nel mondo.

Ai miners conviene comportarsi onestamente, accettando solo transazioni eseguite secondo le regole.
La tecnologia blockchain dopo il 2009 ha iniziato ad avere sempre più utilizzi (sebbene ad oggi ancora sperimentali) come sistemi di votazione, registri general purpose, computazione distribuita, social network ecc. In questo testo tratteremo Ethereum, la prima e più famosa piattaforma di Dapps e smart-contracts, analizzandone il comportamento a basso livello e le possibili ottimizzazioni sulle performance.

Capitolo 2

Il network Ethereum

2.1 Gli smart-contracts

Proposto per la prima volta nel 1996 da Nick Szabo [3], uno smart-contract è un protocollo pensato per facilitare,verificare e far rispettare la negoziazione o l'esecuzione di un contratto. Szabo ipotizzava che attraverso protocolli crittografici e sistemi di firma digitale si potesse costruire un sistema di contratti legali più raffinato di quello tradizionale, automatizzando le clausole. Con la nascita del bitcoin, quasi per caso (o forse no, alcuni ipotizzano che Szabo fosse stato parte del team Satoshi Nakamoto) questo tipo di protocollo trovò terreno fertile per evolversi. Nelle implementazioni odierne, gli smart-contracts sono principalmente utilizzati come computazione general-purpose eseguita su una blockchain e non è necessariamente legato al concetto tradizionale di contratto. Nakamoto diede al Bitcoin features interessanti che non erano originariamente descritte nel whitepaper: Oltre che poter ricevere bitcoin ad una chiave pubblica (address) e poterli spendere tramite una firma digitale, Nakamoto diede agli utenti la possibilità di scrivere scripts che si comportano come firme digitali e chiavi pubbliche dinamiche. Gli script sviluppati sulla rete Bitcoin sono però Turing incompleti.

2.2 Ethereum e le DApps

Nato come critica da parte dello studente Vitalik Buterin al linguaggio di scripting Turing incompleto del bitcoin, venne per la prima volta descritto nel 2013 come una soluzione allo sviluppo di applicazioni decentralizzate [4]. In breve tempo assieme a Gavin Wood, nell'aprile 2014 cofondarono Ethereum e in seguito alla pubblicazione del Yellow Paper [5], contenente le specifiche dettagliate, implementarono il client di Ethereum. Dopo aver raccolto con successo 18 milioni di dollari nella ICO, nel 2015 fu lanciato il network Ethereum a cui molti sviluppatori e miners si sono uniti.
Il network sta guadagnando sempre più popolarità, principalmente per il fatto che permette a sviluppatori di scrivere Dapp e smart-contrant con un linguaggio Turing completo sfruttando la tecnologia blockchain. Questo nuovo paradigma apre le porte ad infinite opportunità. Dato che la blokchain in sé, è spesso definita "secure by design", ora che è possibile sfruttarla per lo sviluppo di applicazioni l'attenzione

si sposta su tematiche che riguardano il costo delle operazioni e la scalabilità, la sicurezza del codice e dei modi per implementare "patch".
Ethereum è un network peer–to-peer dove ogni peer salva la stessa copia del registro distribuito ed esegue una Ethereum Virtual Machine per mantenere e modificare il proprio stato. Utilizza un algoritmo di proof-of-work come il bitcoin (per ora), quindi per aggiungere un nuovo blocco è necessaria l'approvazione di tutto il network. Il consenso è raggiunto incentivando i miners ad accettare la catena più lunga, in cambio di una remunerazione sotto forma di token di rete: "ether". Questo modello rappresenta una novità nel mondo informatico e, anche se ci sono alcune somiglianze, si discosta dal modello client-server e peer-to-peer classico. Data la sua natura distribuita e sicurezza crittografica, può comportarsi come "thirt-party", creando un sistema trust-less e non attaccabile da terze parti. La sua cryptovaluta: *ether*, attualmente è la seconda cryptovaluta più capitalizzata al mondo. Ethereum è composto da diversi elementi chiave:

1. Ethereum blockchain network e protocollo contenente SWARM e Whisper
2. Nodi che eseguono il client ethereum sempre aggiornato
3. Gas
4. Web-3 interface
5. Ethereum Virtual Machine
6. Smart Contracts

L'insieme di queste tecnologie permette lo sviluppo di applicazioni decentralizzate dette "DApps" che secondo il *dapp white paper* [6] devono avere le seguenti caratteristiche:

- Essere completamente open-source e operare autonomamente, senza nessuna entità che possiede la maggior parte della moneta, ogni cambiamento al protocollo dev'essere approvato da tutti gli utenti
- Ogni operazione deve essere crittografata e salvata su una blockchain pubblica
- Deve utilizzare il bitcoin o una cryptovaluta nativa della propria blockchain, necessaria per l'accesso alla DApp e data ai miners come incentivo
- L'applicazione deve generare tokens/currencies mediante un algoritmo di crittografia standardizzato che faccia da da proof-of-value. (Ethereum e bitcoin usano POW)

A differenza di una applicazione tradizionale in esecuzione su un server centralizzato, ha il proprio codice back-end in esecuzione su un network peer-to-peer decentralizzato(la blockchain). Proprio come la controparte tradizionale il frontend può essere scritto in qualsiasi linguaggio di programmazione ed esegue chiamate al

lato server. Come vedremo in seguito, la memorizzazione di dati su blockchain è estremamente costosa e in continuo aumento. Basti pensare che nel 2016 memorizzare 1GB sulla blockchain costava circa *76000 dollari* (calcolato con il prezzo del gas all'epoca) mentre oggi supera il *milione di dollari*.
Per questo una DApp basa soltanto la parte logica sulla blockchain, utilizzando le transazioni in modo simile alle richieste http e memorizzando la parte frontend con metodi di storage decentralizzato come IPFS o SWARM ed utilizzando un protocollo per la comunicazione tra DApps chiamato Whisper. Le applicazioni sono pressoché infinite, dallo IoT all'intelligenza artificiale. I vantaggi di questo approccio sono palesi: resistenza alla censura, dei server attivi 24 ore al giorno senza bisogno di manutenzione, resistenza ad attacchi hacker e molto altro. Di contro richiede un approccio diverso rispetto all'architettura client-server introducendo nuovi problemi di sicurezza e dinamiche mai viste sino ad ora come il *gas*. Se si definisce una app tradizionale *app=frontend+server* allora
DaPP=frontend+smart-contracts

Creare una DApp su Ethereum porta il vantaggio di non dover creare la propria blockchain ma potersi appoggiare a quella di Ethereum, assimilando cosi la protezione data dal Proof-Of-Work e risparmiando risorse che sarebbero necessarie a creare una propria blockchain. Inoltre offre uno standard token chiamato ERC20 che permette di essere una moneta a sé stante ma aderendo allo standard di Ethereum, compatibile quindi con tutti i contratti ed applicativi wallet che lo supportano. La differenza tra un token e una currency è che il primo non possiede una blockchain ad esso dedicata ma si appoggia ad un'altra, nel caso dei token con standard ERC20, quella di Ethereum, la cui currency ha anch'essa questo standard.

2.3 I linguaggi di smart-contract, le piattaforme e i problemi

Gli smart-contract sono astrazioni ad alto livello, compilate in bytecode per una macchina virtuale e memorizzate su una blockchain. Anche se la documentazione riguardo la scrittura di smart-contract è ancora limitata e manca di pubblicazioni importanti, si sta sviluppando attorno ad essa una forte community, che contribuisce attivamente allo sviluppo dei linguaggi, dei compilatori, dei framework e delle macchine virtuali. Essendo un nuovo modo di pensare lo sviluppo di software non ci sono ancora linee guida chiare e complete riguardo la sicurezza e l'ottimizzazione delle performance. Del resto, l'idea di sviluppare software su blockchain è un concetto che, solo da pochi anni è possibile. Esistono molte piattaforme di smart-contract in fase di sviluppo che permetteranno la scrittura di smart-contract in linguaggi di programmazione diffusi e tradizionali. Anche Ethereum lo permette, generalmente si tratta di linguaggi di programmazione noti, adattati per essere compilati in bytecode eseguibile dall' Ethereum Virtual Machine. Per questa tesi verrà preso in considerazione solo Solidity per i motivi che vedremo in seguito, ma tra i più famosi troviamo:

- Viper: Studiato per essere simile a Python, è tra quelli con più sviluppo

- LLL: "Lisp-Like Language" Di basso livello, che permette accesso diretto a memoria
- Mutan: Basato su Go, ma deprecato
- Serpent: Linguaggio di basso livello, predecessore di viper. Ora è deprecato
- Simplicity: pensato appositamente per sviluppo su blockchain. Linguaggio funzionale senza loop o ricorsione. Attualmente in sviluppo

Ci sono molte piattaforme di smart-contract attive e in via di sviluppo. Alcune delle più interessanti sono:

- EOS: Ha un architettura blockchain studiata per permettere uno scaling orizzontale e verticale delle DApp grazie a costrutti simili ad un sistema operativo. Compila in WASM e fornirà sistemi di accounting, autenticazione, database, scheduling tra CPU e comunicazione asincrona, promettendo milioni di transazioni al secondo. Attualmente è l'alternativa ad Ethereum più interessante
- TEZOS: Complementare ad EOS, sacrifica le performance per garantire estrema sicurezza grazie ad un linguaggio funzionale ed una verifica formale obbligatoria. La testnet è avviata ma stanno avendo problemi legali
- NEO: Chiamata anche l'Ethereum cinese, permette di scrivere smart-contract in C ma anche in VB, .Net e F e in futuro JAVA, C, C++, GO, Python e Javascript
- LISK: Basata su javascript, permette di scrivere smart-contract con node.js e utilizza side-chains per gestire lo scaling
- Bitcoin: Molti pensano che nella creazione di smart-contract la Turing incompletezza sia una feature. RSK è in testnet e permette di sviluppare smart-contract su bitcoin offrendo anche scaling tramite soluzioni off-chain

In questo studio utilizzeremo la piattaforma Ethereum poiché la più diffusa e quella con uno sviluppo maggiore. Abbiamo introdotto le possibilità offerte da Ethereum ma non è tutto oro quel che luccica. Ethereum in particolare ha avuto non pochi problemi, principalmente riguardanti la sicurezza e la scalabilità. Dal punto di vista della sicurezza non sono mancati di certo i problemi, il caso più eclatante fu quello del *The DAO*, uno smart-contract che operava come fondo di investimenti autonomo e decentralizzato. Un partecipante al fondo riuscì a svuotare l'address principale, sfruttando l'assenza di controllo di reentrancy da parte di Solidity, rendendo inaccessibili 60 milioni di dollari che all'epoca erano una percentuale altissima degli ether in circolazione. Da questo scaturì un harkfork fulmineo che fu molto criticato per i modi in cui venne effettuato; ritenuta da molti una decisione troppo centralizzata si concluse con lo spit della catena in ETH e ETC. Non fu né il primo né l'ultimo dei bug di ethereum, tanto da portare a molte pubblicazioni riguardo la sicurezza degli smart-contracts [7] [8].

Il problema della scalabilità è un problema diffuso nelle blockchain. Esiste in quanto c'è un limite alle transazioni inseribili in un blocco e un tempo di verifica dato dall'algoritmo di consenso. Aumentare la grandezza del blocco ed utilizzare algoritmi di consenso meno dispendiosi e più veloci come il PoS non rappresenta una soluzione definitiva; ci sono molti progetti che promettono un abbattimento dei costi ed un aumento esponenziale della velocità delle transazioni offrendo transazioni off-chain, ma ad ora non sono ancora utilizzabili[9].
Un effetto collaterale del problema della scalabilità è il tema centrale di questa tesi, ovvero la necessità di best-practice per ridurre le operazioni dispendiose di gas, che in Ethereum si traduce in un ottimizzazione delle performance per le ragioni che vedremo in seguito. Per poter analizzare le performance in Ethereum andremmo a definire il gas e dei costrutti fondamentali della macchina virtuale (principalmente sullo storage), per poterne poi studiare le ottimizzazioni a basso livello, ovvero a tempo di compilazione. Tra i linguaggi proposti utilizzeremo Solidity, illustrandone prima un sotto insieme composto dai costrutti di base che torneranno utili nell'analisi delle best-practice riguardo l'ottimizzazione del gas.

Capitolo 3

Performance: Gas, Solidity e l'Ethereum Virtual Machine

3.1 Solidity

Solidity è un linguaggio di programmazione orientato ai contratti, creato esclusivamente per lo sviluppo di smart-contract su Ethereum. Oltre ad essere quello più utilizzato in assoluto è anche l'unico ad essere supportato ufficialmente. La motivazione di sviluppare un linguaggio ad-hoc è data dal fatto che è stato studiato per lo specifico impiego all'interno degli smart-contract. Sono un tipo di architettura sostanzialmente nuova, dove un approccio ottimizzato sul basso livello è ancora richiesto, per via del costo di gas. Ricorda gli esordi dell'informatica, con architetture nuove e da costruire. Si sarebbero potuti utilizzare anche altri linguaggi, come ad esempio il C, ma adattare un compilatore in questo senso sarebbe stato più costoso che non sviluppare un linguaggio ex-novo.
Solidity è un linguaggio tipato staticamente, supporta l'ereditarietà, librerie e tipi di dato complessi definiti dall'utente.

3.1.1 Contratti

Un contratto dal punto di vista di Solidity è una collezione di codice (le funzioni) e dati (il suo stato) che risiedono in uno specifico indirizzo sulla blockchain di Ethereum. Sono simili alle classi in linguaggi orientati ad oggetti. Ogni contratto contiene dichiarazioni di variabili di stato, funzioni, modificatori di funzioni, strutture dati ed eventi.

Primo esempio della documentazione ufficiale Solidity [10], utile a capirne il funzionamento di base:

```
1
2  pragma solidity ^0.4.0;
3
4          contract SimpleStorage {
5      uint storedData;
6
7      function set(uint x) {
8          storedData = x;
9      }
10
11     function get() constant returns (uint) {
12         return storedData;
13     }
14 }
```

In un contratto .sol le funzioni sono le unitá di codice eseguibile, invocabili tramite transazioni. Le chiamate possono avvenire dall'interno o dall'esterno e hanno diversi gradi di visibilita. Nell'esempio la funzione set è un metodo setter, assegna cioè un valore storedData, modifica quindi lo stato della blockchain memorizzandolo nello storage; la funzione get invece e come una funzione read-only.

3.1.2 Tipi

Solidity è un linguaggio tipato staticamente, ogni variabile (variabile o globale) deve essere dichiara a tempo di compilazione. Fornisce tipi *elementari* che possono essere combinati per formarne di complessi:

- *Booleani:* !, , ||, ==, ! =
- *Operatori logici*
- *Interi:* int/uint con o senza segno di dimensioni variabili da 8 a 256bit int8 uint8 int250 uint256
- *Operatori di confronto e aritmetici:*<=, =>,==,!=, +,-,/,*,

INDIRIZZI
Contengono 20byte di dati e sono la base per tutti i contratti, su di essi si possono fare operazioni di confronto oltre che quelle di gestione:

- *balance:* interroga sul contenuto di un indirizzo
- *transfer:* manda eth ad un determinato indirizzo, ha un gas limit di 2300 ma modificabile
- *send:* come transfer ma in caso di fallimento non lancia un eccezione ma ritorna false, ha un gas-limit 2300 non modificabile (rischio reentrancy se non si fanno check per il valore di ritorno)

3.1.3 Strutture Dati

Per la nostra analisi ricoprono una grande importanza le strutture dati, in quanto nella maggior parte dei casi fanno uso dello storage.

ARRAYS
Gli array possono essere di grandezza fissata a tempo di compilazione o dinamici. Per gli array nello storage il tipo è arbitrario, mentre gli array tenuti in memoria non possono essere mappings. Inoltre gli array a grandezza dinamica possono esistere solo nello storage perché memorizzati con (chiave-valore) mentre in memoria, per ragioni di architettura della EVM che vedremo in seguito, ciò è impossibile. Dato un array di lunghezza fissata n e tipo T lo scriviamo come T[n] es. uint[3]; Hanno due membri: *lenght* che memorizza la lunghezza e *push* per gli array dinamici, serve per fare un append di un elemento alla fine

STRUCTS
Sono forniti dei metodi per definire nuovi tipi sotto forma di structs, ad esempio in un contratto per delle votazioni:

```
1  contract Ballot {
2          // singolo votante
3      struct Voter {
4          uint weight; //peso per delegazione
5          bool voted;  // if true, la persona ha gia votato
6          address delegate; // indirizzo persona delelgata
7          uint vote;   // index della proposta di voto
8      }
9
```

Oltre che tutti i tipi di dati possono anche ma anche array e mappings.

MAPPINGS
Il tipo Mappings è dichiarato come mapping(chiave/valore). Il tipo chiave può essere qualsiasi tipo a parte un mapping mentre il valore può essere qualsiasi tipo. I mappings possono essere visti come tabelle Hash inizializzate virtualmente in modo tale che ogni possibile chiave esista e sia mappata ad un valore (alla creazione tutti 0). Il dato non è salvato dal mapping ma contiene solo il suo keccak256 hash che viene usato per trovarne il valore. È possibile settarle a public cosicché Solidity crei un getter.

3.2 Il Gas e l Ethereum Virtual Machine

Come abbiamo visto, Solidity offre molti costrutti ad alto livello. È proprio per questo che spesso è difficile capire cosa sta succedendo in fase di esecuzione. Per capirlo, la documentazione ufficiale di Solidity non è sufficiente e ad ora non copre alcuni aspetti importanti del comportamento a livello bytecode delle funzioni. Capirlo, più che in altri linguaggi/piattaforme, è molto importante sia dal punto di vista delle performance che dal punto di vista della sicurezza. Dato che ad ogni operazione assembly è associato un costo monetario reale, ottimizzazioni in fase di compilazione ci permettono un abbattimento dei costi che rappresenta un fattore critico in Ethereum. Dopo aver introdotto il gas e la sua relazione con la EVM, andremo ad analizzare semplici ottimizzazioni possibili su dei contratti Solidity compilati in assembly.

3.2.1 Il Gas

Ogni operazione che può essere effetuata da una transazione o da un contratto su Ethereum costa gas. Per gas intendiamo un'unità di misura speciale utilizzata per pagare la computazione effettuata dalla rete. Misura quanto lavoro deve fare una macchina virtuale per eseguire una serie di istruzioni; ad esempio, per calcolare un hash crittografico Kekkac256 richiederà 30 gas ad ogni hash calcolato, più 6 gas ogni 256 bit di dati sui cui è stata eseguita la funzione hash. Le operazioni più dispendiose in termini di risorse costeranno più gas. In questo modo viene incentivato un certo tipo di programmazione, disincentivando le operazioni costose che andrebbero a pesare sulla rete Ethereum.
Il gas è importante perché assicura che un appropriata fee venga pagata al network quando avviene una transazione. Richiedendo che una transazione paghi per ogni operazione che esegue, assicuriamo che il network non venga rallentato da lavoro intensivo per cui Ethereum non è stato progettato. (Ad es. memorizzare il frontend, delle immagini, un film...) Nel bitcoin la fee di transazione è esclusivamente basata sui kilobyte. Data la complessità maggiore di Ethereum (scripting intensivo) e la presenza della EVM, è meglio commisurare la fee alle risorse computazioni utilizzate per eseguire un contratto piuttosto che dai kilobyte della transazione. Anche se è un unità misurabile non esiste nessun token per il gas , viene pagato in ether, ed esiste solamente nel contesto del calcolo delle risorse usate dalla virtual machine; non è possibile infatti possedere gas. Questa decisione è stata presa perché il prezzo di ether è ritenuto troppo volatile, si è preferito quindi un unità più stabile, facendo decidere il prezzo ai miners, aggiornandolo quando necessario e separando quindi le due logiche. Nel momento in cui effettuiamo una transazione (che sia un invio di denaro o una esplicita computazione) ci viene addebitata una certa quantità di ether, che sarà utilizzata per comprare il gas usato nella computazione. Ogni operazione nella EVM ha un costo in gas ma come abbiamo detto anche il gas ha un costo in ether. In ogni transazione che andiamo ad eseguire specifichiamo il *gas price* che intendiamo pagare a singola unità di gas ed il *gas limit*, ovvero la massima quantità di gas che vogliamo comprare. A fine computazione avremo:

$\text{TotalGasUsed} \times gaspricepaid = totalfee$

Per capire quanto fondamentale sia la conoscenza del funzionamento del gas nello sviluppo di smart-contract (o anche solo di utilizzo di rete) basta pensare agli scenari possibili:

- Se il prezzo in gas che abbiamo impostato è troppo basso nessun miner vorrà includere la mia transazione nella blockchain
- Se fornisco una quantità insufficiente di gas e la mia transazione utilizza molte più operazioni di quanto pensassi allora verrà lanciata l'eccezione "out-of-gas", la mia transazione fallirà ma il gas sarà comunque perso perchè pagato al miner per il suo lavoro.
- Se fornisco un prezzo troppo alto, avrò una priorità alta e il miner mi includerà subito nel blocco, ma andrò a pagare troppo per poche operazioni.
- Se invece fornisco un prezzo del gas adeguato ed un alto numero di gas verrò rimborsato del gas in eccesso. E' quindi importante capire la situazione e dare un giusto trade-off tra gas price e gas limit.

Il gas è il meccanismo di base che rende le computazioni per il network necessariamente sicure, questo evitando la non terminazione di un programma rendendo impossibile i loop infiniti. Riguardo al discorso introdotto prima sulla scalabilità è importante notare che più cresce la rete Ethereum più si avranno transazioni causando un aumento di codice eseguito sulle EVM ed un aumento del prezzo del gas. Gli sviluppatori devono tenere ben presente questi costi, attualmente per determinate applicazioni la rete Ethereum è esageratamente costosa. Bisogna trovare il giusto bilanciamento tra operazioni on-chain e off-chain, almeno finché non saranno migliorate le capacità delle blockchain.

3.2.2 L'Ethereum Virtual Machine

L'EVM è il runtime enviroment per gli smart-contract in Ethereum, esegue smart-contracts da bytecode localizzato in un indirizzo a 160 bit situato nella blockchain. (chiamato anche "account"). EVM opera su "'pseudo registri" a 256bit, ma non si tratta di veri registri. Opera invece su uno stack crescente usato per passare parametri non solo a funzioni/istruzioni ma anche allo storage e per operazioni aritmetiche. Dalla documentazione ufficiale si suddividono gli account in due tipi che utilizzano lo stesso spazio di indirizzi.

- External accounts: Controllati da chiave pubblica/privata (umani quindi)
- Contract accounts: Controllati dal codice memorizzato nell'account stesso

Indipendentemente dal fatto che un account abbia memorizzato del codice, i due tipi vengono trattati nella stessa maniera dall'EVM. È Turing Completa, ha sia le primitive di un nastro di Turing (op per gestire la memoria e saltare a posizioni

arbitrarie del programma) sia quella di agent-based message passing, dove gli agenti possono avere del codice (op per chiamare o creare contratti, restituire valori). È stata studiata per essere eseguita da tutti i partecipanti del network contemporaneamente. Può leggere e scrivere sulla blockchain sia codice eseguibile (scripts) sia dati ed esegue codice solo quando riceve un messaggio verificato da una firma digitale(le transazioni).

L'EVM si può definire come una virtual machine orientata alla sicurezza, studiata per permettere di eseguire codice untrusted da parte di un global network di computers. Per ottenere questo impone determinate restrizioni:

- Ogni step di computazione eseguito da un programma in esecuzione deve essere pagato in anticipo. (Questo per prevenire attacchi DoSs)
- I programmi possono interagire tra di loro solamente trasmettendo un unico array di byte di grandezza arbitraria. Non possono avere accesso allo stato dell'altro programma.
- L'esecuzione è in una sandbox: un programma EVM può accedere e modificare il proprio stato e fare riferimento ad altri programmi, ma non altro.
- L'esecuzione di un programma è di tipo deterministico e produce lo stesso risultato per ogni "conforming implementation beginning" in un identico stato.

Queste restrizioni sono motivate dalla natura di state transition machine di ethereum. Per assicurarsi che tutti i programmi terminino,(halting problem) ad ogni operazione è assegnato un esplicito costo chiamato gas. L'esecuzione deve specificare un massimo di gas da assegnare all' esecuzione, così in caso si utilizzi più gas di quanto assegnato venga lanciata una eccezione OutOfGas. Nell'appendice G del "Yellow Paper" c'è lista degli opcodes con lo specifico costo in gas, tra cui i più importanti:

```
 1  step             1      Default amount of gas to pay for an execution cycle
 2  stop             0      Nothing paid for the SUICIDE operation
 3  sha3             20     Paid for a SHA3 operation
 4  sload            200    Paid for a SLOAD operation
 5  sstore           20000  Paid for a normal SSTORE operation
 6  balance          400    Paid for a BALANCE operation
 7  create           32000  Paid for a CREATE operation
 8  call             7      Paid for a CALL operation
 9  memory           1      Paid for every additional word when expanding memory
10  txdata           5      Paid for every byte of data or code for a transaction
11  transaction      500    Paid for every transaction
```

Quando in Ethereum si parla di performance, non ci si riferisce a m/s di computazione risparmiati, ma ad un uso ottimale del gas. Il compilatore ci può dare delle stime del gas consumato ma se lo smart-contract interagisce con la blockchain (timestamp, balance ,hash del blocco) non può prevederlo con esattezza, sta a noi dare un quindi upperbound impostando un gas limit. Se si prende in considerazione solo la documentazione Solidity, rimangono molti punti poco chiari

sul comportamento di determinate operazioni. In particolare sui tipi da usare ed il costo in gas di allocazioni, funzioni, e deployment. È utile capire come un linguaggio di alto livello come Solidity viene eseguito dalla EVM per una serie di ragioni. In primis, quando si scrive un contratto Solidity è bene sapere che non è l'ultima parola prima dell'esecuzione e non è l'unico linguaggio. L'ultima parola è sempre del bytecode, i nodi eseguono bytecode. In secondo luogo l'EVM si può definire un database engine, per capire come funzionano i linguaggi di programmazione che solo successivamente vengono compilati in EVM bisogna avere una conoscenza su come i dati vengono organizzati, modificati e scritti. In genere, in un linguaggio di programmazione, non è essenziale capire come i data types sono rappresentati a così basso livello. In Solidity o in qualsivoglia altro linguaggio di programmazzione che compila in EVM capirlo è essenziale perché tutta la computazione sara dominata dal costo (altissimo) degli accessi in memoria. (memoria "fisica" non virtuale) basti vedere:

- sstore costa 20000 gas, 5000x più costosa di un'operazione aritmetica di base
- sload costa 5000 gas, 1600x più costosa di un op aritmetica di base

Sottolineaiamo chep er costo non intendiamo qualche m/s di guadagno in performance ma un vero e proprio costo monetario. Il costo nell'usare ed eseguire smart-contract è dominato dalle operazioni di storage.

FUNZIONI EVM
Le principali funzioni/istruzioni EVM sono:

- Operazioni aritmetiche
- Operazioni logiche/di confronto
- SHA3
- Informazioni dell' ambiente
- Informazioni del blocco
- Su memoria, stack, storage e di flusso
- Push/Dup/Pop/Ex
- Operazioni di logging
- Operazioni di sistema

3.2.3 Gestione della memoria

LA MEMORIA VIRTUALE
La memoria secondaria è lineare e può essere indirizza a byte level, ma le letture sono limitate a 256bits (32bytes), mentre le scritture vanno dagli 8 ai 32 bytes. Quando viene allocata (è continua) un costo in gas dev'essere pagato, questo costo cresce in maniera quadratica. Le OP sono:

- MSTORE(dove,cosa) viene usata quando si aggiungono dati sullo stack

- MLOAD(dove) vengono letti.

STACK

Non ha il concetto di registri, uno stack virtuale viene utilizzato per operazioni come parametri per il codice operativo. EVM usa valori a 256 bit per lo stack virtuale, ed è limitato superiormente, ha infatti un massimo di 1024 elementi e utilizza una politica LIFO. Dato che la EVM non ha registri tutte le chiamate a funzioni/istruzioni avvengono mediante stack. Ad esempio se eseguiamo 1+2

```
1 PUSH1 0x1 ==> {stack[0x0] = 0x1}
2 // viene caricato il valore 1 nella posizione 0 dello stack
3 PUSH2 0x2 ==> {stack[0x0] = 0x2, stack[0x1] = 0x1}
4 // essendo lifo nella posizione 0x0 viene caricato 2
5 // e nella posizione 0x1 rimane 1
6 ADD ==> {stack[0x0] = 0x3}
7 // Add prende i primi due numero sullo stack, li somma,
8 // li toglie dallo stack e memorizza la somma nella posizione 0x0
```

STORAGE

Lo storage è un tipo di memoria permanente con la forma chiave-valore (256 a 256 bit integer), ed è posseduto da ogni account. Oltre questo ogni account ha anche un "balance" che può venir modificato con le transazioni. Leggere e scrivere da questo tipo di memoria è l'operazione più costosa in assoluto. Un contratto non può ne leggere ne scrivere su nessun tipo di memoria eccetto la propria. Ad esempio

```
1 contract InviaDenaro{
2 mapping ( address -> uint) saldoUtente;
3 bool ritirato = false;
4    ....
5 }
```

Lo storage viene dichiarato nel contratto, nei tipi semplici è rappresentato da una variabile. Una volta dichiarata la variabile utilizzeremo le OP SSTORE e SLOAD per leggere e scrivere nello storage.

3.2.4 Compilazione e bytecode

Per ottimizare il codice e/o eseguire controlli di sicurezza esistono molti tool utili a riga di comando da utilizzare durante la compilazione:[11]

go get github.com/ebuchman/evm-tools/...

È un tool che contiene evm, evm-deploy e sopratutto *disam* usato per decompilare il bytecode in assembly rendendolo molto più leggibile, cosi da rendere più chiare le ottimizzazioni. Ad esempio da un "semplice" bytecode come: *6005600401*

Con *6005600401 — disasm*
Ottengo:

```
1   PUSH1 => 05
2   PUSH1 => 04
3   ADD
```

Le istruzioni possono quindi usare i valori salvati nello stack come argomenti, e rimetterli (push) come risultato. In questo caso due valori salvati sullo stack, e avviene un op ADD Lo stack sara':
Stack:[0] Storage:(0)
Primo PUSH
Stack:[5] Storage:()
Secondo PUSH
Stack [4 5] Storage:(0)
Operazione ADD
Stack [0] Storage:(0)

Utlizzando *evm –debug –code 6005600401* si può vedere lo stato attuale dello stack, gas, memoria e storage ad ogni step

Per verificare quanto le operazioni di storage dominano sulle performance basta seguire passo passo l'esecuzionione di uno smart-contract in bytecode:

```
1  // esempio.sol
2          pragma solidity 0.4.11;
3          contract E {
4                  uint256 x;
5                  function C() {
6                          x=1;
7                  }
8          }
```

Compilando con :

```
1  $ solc --bin --asm c1.sol
```

Abbiamo:

```
1  ======> esempio.sol:C <======
2  EVM assembly:
3
4  ..........
5  tag_2:
6  /* "esempio.sol":84:85 1 */
7  0x1
8  0x0
9  dup2
10 swap1
11 sstore
12 pop
13 ...........
```

Che verrà eseguito dalla Virtual Machine sotto forma di bytecode:

```
60606040523415600e57600080fd5b5b60016000819055505b5b
6036806026600039600f3006060040525b600080fd00a165627
a7a72305820af3193f6fd31031a0e0d2de1ad2c27352b1ce081b4
f3c92b5650ca4dd542bb770029
```

Dove l'istruzione di assegnamento è 6001600081905550. Come abbiamo detto EVM è una macchina a stack che esegue dall' alto verso il basso. Se decompiliamo con *disam*:

Tornando all' operazione di prima 6001600081905550

```
1  // 60 01     0x1: mette il valore 1 in cima allo stack
2  stack: [0x1]
3  // 60 00     0x0: mette 0 sullo stack (posizione)
4  stack: [0x0 0x1]
5  // 81     dup2: duplica il secondo elementi e lo mette sullo stack
6  stack: [0x1 0x0 0x1]
7  // 90     swap1: scambia di posizione i primi due elementi
8  stack: [0x0 0x1 0x1]
9  // 55     sstore: memorizza nella posizione 0x0 il valore 0x1
10 stack: [0x1]
11 store: { 0x0 => 0x1 }
12 // 50     pop: throw, elimina il primo elemento dello stack
13 stack: []
14 store: { 0x0 => 0x1 }
```

Fine, stack vuoto, elemento salvato sulla blockchain. Solidity ha deciso di salvare unit256 a nella posizione 0x0 in quanto non specificato altrimenti ma è possibile memorizzare in posizioni arbitrarie. Andando ad analizzare il costo delle operazioni assembly si capisce quanto le operazioni di storage influiscano sulle performance dominando il costo. dup,swap, push e pop costano in media da 2 a 7 gas mentre sstore 20000. Il funzionamento e l'influenza delle operazioni di storage si nota ancora meglio se aumentiamo le operazioni di sstore:

```
1
2  // es2.sol
3                  pragma solidity ^0.4.11;
4                  contract C {
5                          uint256 a;
6                          uint256 b;
7                          function C() {
8                                  a = 1;
9                                  b = 2;
10                 }
11                 }
```

Essendo la EVM è una macchina a 256bit, con inidirizzi da 32byte, se aggiungiamo una variabile si può notare che il codice solidity, compilato in assembly/bytecode memorizza le variabili in maniera sequenziale: 1 in 0x0 e 2 in 0x1. Semplificando l'assembly in pseudocodice viene:

```
1  a = 1      sstore(0x0, 0x1)
2  b = 2      sstore(0x1, 0x2)
```

Capitolo 4

Ottimizzazioni a tempo di compilazione

4.1 Celle di memoria e optimizer

Abbiamo introdotto il concetto del gas è visto come le operazioni di sstorage influiscano sulle performance. In questa sezione da studi di riferimento [12][13][14][15] analizzeremo come, perché e quando è possibile ottimizzare il codice a tempo di compilazione nelle operazioni di storage delle strutture dati introdotte precedentemente.

4.1.1 Accorpamento delle variabili

Sappiamo che ogni slot di memoria è di 32byte, se una variabile dovesse occupare ad es. 16 bytes sarebbe uno spreco utilizzare un intero slot dato che ogni operazione rappresenta un costo monetario reale. Di default Solidity si comporta bene, anziché sprecare due celle di memoria, impacchetta le due variabili da 16byte (dato che sono consecutive in memoria), e le memorizza in una sola cella.

```
1  // es2.sol
2  pragma solidity ^0.4.11;
3  contract C {
4  uint128 a;
5  uint128 b;
6  function C() {
7  a = 1;
8  b = 2;
9  }
10 }
```

da cui il codice assemby:

```
1  // a = 1
2  0x1
3  0x0
4  dup1
5  0x100
6  exp
7  dup2
8  sload
9  dup2
10 0xffffffffffffffffffffffffffffffff
11 mul
12 not
13 and
14 swap1
15 dup4
16 0xffffffffffffffffffffffffffffffff
17 and
18 mul
19 or
20 swap1
21 sstore
22 pop
23 // b = 2
24 ................
25 sstore
26 pop
```

Ora abbiamo salvato a nei primi 16bytes, mentre b risiede negli ultimi 16.

Il problema è che seppur le variabili vengano memorizzate in una sola cella, vengono comunque eseguite due operazioni sstore. Ne risulta comunque un risparmio in termini di gas. I 20000 gas sono utilizzati per la allocazione della cella di storage Dopo aver allocato lo spazio pagando 20000 gas, le scritture successive costano 'solo' 5000 gas perché andiamo ad operare su una cella di memoria di cui abbiamo già pagato 'l'affitto'.Le operazioni successive saranno quindi di aggiornamento e la spesa risulta essere 25k anziche 40k in GAS. Da questo comportamento si evince che il compilatore evm sia stato scritto appositamente per ridurre al minimo gli accesso allo storage, operazione enormemente costosa per il network Ethereum.

4.1.2 Packing delle variabili

Per risparmiare ancora più gas un'ottimizzazione possibile offerta dal compilatore ufficiale è quella in incapsulare le due variabili grazie ad operazioni sulla memoria virtuale e memorizzarle utilizzando una sola operazione di sstore. Nel memorizzare di due variabili consecutive si comporta abbastanza bene:

Il compilatore di solidity supporta il flag –optimize:

```
1 $ solc --bin --asm --optimize es2.sol
```

Avremo:

```
tag_2:
/* "es2.sol":95:96 a */
0x0
/* "es2.sol":95:100 a = 1 */
dup1
sload
/* "es2.sol":108:113 b = 2 */
0x2000000000000000000000000000000000000000
not(sub(exp(0x2, 0x80), 0x1))
/* "es2.sol":95:100 a = 1 */
swap1
swap2
and
/* "es2.sol":99:100 1 */
0x1
/* "es2.sol":95:100 a = 1 */
or
sub(exp(0x2, 0x80), 0x1)
/* "es2.sol":108:113 b = 2 */
and
or
swap1
sstore
```

Questa e' la parte che interessa a noi, organizzata sotto il costruttore segnato come tag_2.*Ilbytecodegeneratorisultarisultaessere* :

```
Bytecode:
60008054700200000000000000000000000000000000006001608060020a
0319909116600117600160806002 0a0316179055
```

Disassemblando il bytecode con -asm si può vedere come si possa utilizzare una sola *sstore* per salvare due variabili; (x+y) <= 32byte:

```
1
2  60 00 // push 0x0
3  80 // dup1
4  54 // sload
5  // push17 push i prossimi 17bytes come se fossero un numero da 32 byte
6  70 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
7  /* not(sub(exp(0x2, 0x80), 0x1)) */
8  60 01 // push 0x1
9  60 80 // push 0x80 (32)
10 60 02 // push 0x80 (2)
11 0a // exp
12 03 // sub
13 19 // not
14 90 // swap1
15 91 // swap2
16 16 // and
17 60 01 // push 0x1
18 17 // or
19 /* sub(exp(0x2, 0x80), 0x1) */
20 60 01 // push 0x1
21 60 80 // push 0x80
22 60 02 // push 0x02
23 0a // exp
24 03 // sub
25 16 // and
26 17 // or
27 90 // swap1
28  55 // sstore
```

Il codice è piuttosto compilcato e non ci interessa nei dettagli. Basta notare che l'optimizer ci permette di sfruttare di più la memoria virtuale (meno costosa) ed di utilizzare una sola sstore per entrambe le variabili, permettendo così un ottimizzazione sulle performance. Verrà quindi utilizzata una sola sstore e loperazione di ottimizzazione si è conclusa con successo.

Interessante notare come salva le due variabili in una cella. La EVM è una macchina big endian quindi:

```
con 0x1 (0x01 in bytecode) alloca i byte da 0 a 16
16:32 0x00000000000000000000000000000000
00:16 0x00000000000000000000000000000001

con 0x2 (0x2000000000000000000000000000000000 in byte code) l'altra parte

16:32 0x00000000000000000000000000000002
00:16 0x00000000000000000000000000000000

con not(sub(exp(0x2, 0x80), 0x1))
// Bitmask per i 16 byte superiori
16:32 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
00:16 0x00000000000000000000000000000000

Uguale poi per i byte inferiori, effettua degli shuffle di bits
e si ritrova nella forma desiderata, con in memoria virtuale
due variabili da 16byte impacchettate in una da 32 pronte
da salvare con una sola
operazione di storage

16:32 0x00000000000000000000000000000002
00:16 0x00000000000000000000000000000001
```

0x20000000 è direttamente allegato nel bytecode, anche se si potrebbe calcolare da exp(0x2, 0x81), che risulta molto più leggibile.
Il fatto è che 0x2000000000 è molto più economico in termini di gas. Per ogni valore non zero di una transazione il costo in gas è 68, mentre per ogni 0 è 4. Quindi 340 contro 196, L' ottimizzatore qui ha funzionato.

Dato che lo storage nel EVM è ad accesso diretto sia in lettura che scrittura. Se prendiamo una variabile dichiarata (quindi prenotando spazio) ma inizializzata (0x0 di default) non paghiamo gas)

```

pragma solidity ^0.4.11;
contract C {
    uint256 a;
    function C() {
      a = a + 1;
    }
}
```

```
$ solc --bin --asm --optimize c-zero-value.sol
```

```
 1  tag_2:
 2    // sload(0x0) returning 0x0
 3    0x0
 4    dup1
 5    sload           // sload fa riferimento ad una posizione non inizializzata.
 6    // a + 1; where a == 0
 7    0x1
 8    add
 9    // sstore(0x0, a + 1)
10    swap1
11    sstore
```

Sapendo che il secondo tag è il costruttore è che, non avendo inizializzato a, il suo valore è zero possiamo sostituire la parte sload (dove carica dallo storage il dato su cui sommare 1) e sostituirlo con push 0x0, risparmiando 5000 unità di GAS, sload è infatti una delle operazioni più costose dato che opera sullo storage.

4.2 Strutture dati fisse e tags

L'ottimizzatore come abbiamo visto riesce ad impacchettare le variabili sotto ad un unico tag e risparmiare gas sulle operazioni di sstore. Ma in un codice con più funzioni, dove l'assembly risultante è inevitabilmente raggruppato tra diversi tag l'ottimizzatore fallisce:

```
1  pragma solidity ^0.4.11;
2  contract tags {
3      uint64 a;
4      uint64 b;
5      uint64 c;
6      uint64 d;
7
8      function AB() {
9        setAB();
10       setCD();
11     }
12
13     function setAB() internal {
14       a = 0xaaaa;
15       b = 0xbbbb;
16     }
17     function setCD() internal {
18       c = 0xcccc;
19       d = 0xdddd;
20     }
21 }
```

```
1 $ solc --bin --asm --optimize packEtags.sol
```

in pseudo codice abbiamo una suddivisione in più tag dato i continui jump:

```
1  // Constructor function
2  tag_2:
3    // ...
4    // call setAB() by jumping to tag_5
5    jump
6  tag_4:
7    // ...
8    // call setCD() by jumping to tag_7
9    jump
10 // function setAB()
11 tag_5:
12   // Bit-shuffle and set a, b
13   // ...
14   sstore
15 tag_9:
16   jump  // return to caller of setAB()
17 // function setCD()
18 tag_7:
19   // Bit-shuffle and set c, d
20   // ...
21   sstore
22 tag_10:
23   jump  // return to caller of setCD()
```

Dall'esempio si evince come non ci si possa sempre fidare delle ottimizzazioni del compilatore. Ci si aspetterebbe che 4 variabili da 64 bit vengano impacchettate e salvate in una cella da 32 byte con una sola operazione di sstore. Questo non avviene perché il compilatore non riesce ad ottimizzare tra tags diversi ma solo sotto lo stesso tag, ovvero dove ci sono operazioni di jump. L'ottimizzazione è comunque utile in questo caso dato che ci fa risparmiare 2 sstore ma ad oggi non funziona come ci si potrebbe aspettare.

4.2.1 Fixed-Size Array

Lo storage negli array a dimensione fissa funzionano in modo analogo a quello delle altre strutture dati. Il codice assembly che ne deriva però è diverso in quanto ci sono dei controlli sui limiti nell' accesso agli array.

```
1 pragma solidity ^0.4.11;
2
3 contract C {
4     uint256[6] array;
5     function C() {
6       array[5] = 0xC0FEFE;
7     }
8 }
```

```
1 $ solc --bin --asm c-static-array.sol
```

```
1   tag_2:
2     0xc0fefe
3       [0xc0fefe]
4     0x5
5       [0x5 0xc0fefe]
6     dup1
7     /* array bound checking code (if 5<6)*/
8     0x6
9       [0x6 0x5 0xc0fefe]
10    dup2
11      [0x5 0x6 0x5 0xc0fefe]
12    lt
13      [0x1 0x5 0xc0fefe]
14    // bound_check_ok = 1 (TRUE)
15    // if(bound_check_ok) { goto tag5 } else { invalid }
16    tag_5
17      [tag_5 0x1 0x5 0xc0fefe]
18    jumpi
19      // Test condition is true. Will goto tag_5.
20      // And `jumpi` consumes two items from stack.
21      [0x5 0xc0fefe]
22    invalid
23  // Array access is valid. Do it.
24  // stack: [0x5 0xc0fefe]
25  tag_5:
26    sstore
27      []
28      storage: { 0x5 => 0xc0fefe }
```

Come si può vedere effettua un controllo ad ogni assegnamento. Questo "spezzetta" il codice in tanti tag quante le operazioni di assegnamento, rendendo impossibili le operazioni di ottimizzazione:

```
1  pragma solidity ^0.4.11;
2  contract C {
3      uint64[4] array;
4      function C() {
5        array[0] = 0x0;
6        array[1] = 0x1111;
7        array[2] = 0x2222;
8        array[3] = 0x3333;
9      }
10 }
```

```
1   solc --bin --asm --optimize c.sol | grep -E '(sstore|sload)'
2    sload
3    sstore
4    sload
5    sstore
6    sload
7    sstore
8    sload
9    sstore
```

Infatti abbiamo 4 operazioni di SSTORE per 4 variabili, i controlli sui limiti degli array a lunghezza fissa non permettono quindi l'uso dell' ottimizzatore. In ogni caso le operazioni sono svolte sulla stessa cella di memoria, quindi una volta inizializzata con 20000 GAS, le operazioni successive sono comunque meno costose: 5000 GAS ad operazione. In questo modo avremo una spesa di 35000 contro gli 80000 se non avessimo dichiarato le variabili come uint64 e i 20000 di una piena ottimizzazione.

4.3 Strutture dati dinamiche

4.3.1 Mappings

La EVM si può pensare come un database a chiave-valore con slot da 32byte. Sui mappings non è possibile effettuare ottimizzazioni, in quanto ogni record in memoria va ad occupare esattamente 32byte, se memorizziamo più di 32byte semplicemente collega i due hash. Ad esempio se volessi salvare 40byte andrei ad utilizzare 2 sstore ed occupare due celle di storage.

```
1 pragma solidity ^0.4.11;
2 contract map {
3     mapping(uint256 => uint256) ogg;
4
5     function map() {
6       ogg[0xB0EEFF] = 0x32;
7     }
8 }
```

In questo esempio di usa solo uno sstore ma per calcolare il kekkac256 della chiave saranno necessarie delle operazioni in memoria. La mancanza di possibilità di ottimizzare lo storage e le istruzioni in memoria virtuale aggiuntive rendono i mapping meno convenienti quando si utilizzano dati piccoli.

4.3.2 Array Dinamici

In solidity gli array dinamici hanno una struttura praticamente identica ai mappings, ma con delle features in più, che li rendono invevitabilmente più costosi:

length per contare il numero di oggetti memorizzato

Bound-checking controlli sugli accessi

Ottimizzazioni per byte e stringe per rendere gli array piccoli più efficenti

Ma a differenza dei mappings il compilatore incapsulando le variabili come visto prima:

```
1
2 pragma solidity 0.4.11
3 contract C {
4     uint128[] s;
5
6     function C() {
7         s.length = 4;
8         s[0] = 0xAA;
9         s[1] = 0xBB;
10        s[2] = 0xCC;
11        s[3] = 0xDD;
12    }
13 }
```

Se proviamo a memorizzare 4 oggetti in un vettore da 128 occuperemo 2 celle di memoria, più una per il campo lenght, che verrà memorizzato nella prima posizione.

```
1 key: 0x0000000000000000000000000000000000000000000000000000000000000000
2 value: 0x0000000000000000000000000000000000000000000000000000000000000004
3
4 key: 0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563
5 value: 0x000000000000000000000000000000bb000000000000000000000000000000aa
6
7 key: 0x290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e564
8 value: 0x000000000000000000000000000000dd000000000000000000000000000000cc
```

Purtroppo, come per gli array a lunghezza fissa non sono possibili altre ottimizzazioni sul numero di *SSTORE* utilizzate per via dei bound-cheking che suddividono il codice assembly in tags, rendendo impossibile un eventuale accorpamento da parte dell'ottimizzatore.

Dagli studi presi in considerazione si possono tracciare delle linee, utili ad avere un approccio a basso livello nella programmazione di smart-contract. Il compilatore solidity quando può ottimizza il codice, e lo fa per risparmiare storage. Purtroppo quando il controllo passa ad un altra funzione o quando entra in gioco il bound-checking degli array il compilatore fallisce ad ottimizzare tra tags diversi. Il problema principale però risiede nel fatto che non è consigliabile utilizzare l'ottimizzatore indiscriminatamente, dato che aggiunge un layer di complessita' in più, esponendoci a rischi di bug, anche piuttosto gravi.[16][17][18] Se si volesse utilizzare l'ottimizzatore i miei consigli sono i seguenti:

1. Avere una profonda conoscenza della EVM e dell'ottimizzatore
2. Tenere aggiornato all' ultima versione il compilatore
3. Prima di effettuare un deploy del codice controllare tutte le ISSUE riguardo l'ottimizzatore nel github di Ethereum.

Capitolo 5

Gas e best-practice in Solidity

Da un'analisi effettuata sugli smart-contract presenti sulla blockchain Ethereum è risultato che dall' 80% al 93% degli smart-contract sono sotto-ottimizzati se confrontati con dei pattern di cattivo codice, ed utilizzano molto più gas di quanto sia necessario.[19] Questo spreco può avvenire in fase di deploy di un contratto o in fase di esecuzione da parte di un chiamante (o in entrambi i casi). In questo capitolo tracceremo le best-practices analizzando i pattern di codice inutilmente costoso. Eventualmente delle ottimizzazioni potranno essere implementate in un futuro compilatore, ma come abbiamo visto nel capitolo precedente c'è molta strada da fare e potrebbe essere un rischio per la sicurezza. Tuttavia ciò ci permetterà, con l'analisi fatta nel capitolo precedente, di avere una visione generale sul codice da evitare e conoscere per quanto riguarda l'ottimizzazione delle performance nello sviluppo di smart-contract da una prospettiva ad alto livello. Analizzeremo i patterns suddivisi in due categorie, quelli relativi al codice inutile o non adeguato e quelli relativi ai loops.

5.1 Useless and non adeguate code

5.1.1 External vs Public

Si tratta di un costo relativo all'esecuzione. Quando esponiamo le funzioni del nostro smart-contract a chiamate esterne dobbiamo stare attenti a che tipo di visibilità dargli [20], dalla documentazione ufficiale di Solidity di può leggere che:

External functions are sometimes more efficient when they receive large arrays of data

Ma non specifica altri dettagli su come questo sia possibile e che risparmio effettivo ci può dare. Le funzioni in Solidity oltre la visibilità di default, *public*, possono avere visibilità *external*. Prendendo un semplice programma solidity:

```
1  pragma solidity^0.4.12;
2
3  contract Test {
4      function test(uint[20] a) public returns (uint){
5          return a[10]*2;
6      }
7
8      function test2(uint[20] a) external returns (uint){
9          return a[10]*2;
10     }
11 }
```

Compilandolo sulla testnet si può analizzare il costo del gas per chiamata di funzione:

- La funzione public usa 496 gas
- La funzione external usa 261 gas

In questo caso quindi, external usa quasi la metà del gas. La differenza sta nel fatto che in Solidity una funzione public può, a differenza di external, essere chiamata internamente. Nelle funzioni public gli argomenti dell'array vengono immediatamente copiati in memoria, mentre con external si possono leggere direttamente da calldata. Nel capitolo precedente abbiamo visto i costi delle operazioni assembly, sappiamo quindi che le operazioni sulla memoria hanno un costo maggiore rispetto a quelle di stack ed aritmetiche. Le funzioni public dato che possono essere chiamate internamente funzionano in maniera completamente diversa da quelle *external*, sono eseguite via *jumps* e gli argomenti dell'array vengono passati internamente con puntatori alla memoria. Quindi quando un compilatore genera il codice proveniente da una chiamata interna, la funzione si aspetta che gli argomenti siano in memoria. Nelle funzioni *external* invece non c'è bisogno di permettere chiamate interne, permettendo di leggere direttamente dal *calldata*. Una best-practice possibile è utilizzare *external* quando si ha la certezza che la funzione venga solo ed esclusivamente chiamata dall'esterno del contratto, mentre *pubblic* quando venga chiamata anche dall'interno.

5.1.2 Codice morto

Si tratta di un costo relativo alla fase di deploy di un contratto. Quando andiamo ad effettuare un deploy sulla blockchain di un contratto Solidity andiamo a pagare il gas per la creazione di un contratto, che comprende tutte le righe di codice del nostro programma, incluso il codice morto che non verrà mai eseguito. Il dead-code non viene rimosso dal compilatore, generando così uno spreco di soldi. Ad esempio:

```
1 function deadc(uint x) {
2         if (x>5) {
3                 if (x*x=25){
4                         ..............}}
5 }
```

Il codice dopo la terza riga non verrà mai valutato a causa del predicato (x*x¡20) che risulterà sempre falso.

5.1.3 Predicato opaco

Un predicato opaco è un predicato (un'espressione valutabile "true" o "false") dove il risultato è conosciuto a priori. Ad esempio:

```
1 function predicate (uint x) {
2         if (x>5){
3                 if (x>1){
4                         ..........
5 }}}
```

Il secondo predicato è sempre vero ma ogni volta che viene chiamata la funzione viene comunque eseguito. Se si rimuovesse dal codice risparmierebbe gas sia il creatore del contratto in fase di deploy, perché occuperebbe meno righe di codice, sia chi chiama la funzione in quanto non verrebbero eseguite operazioni inutili.

5.2 Bad loops

5.2.1 Risultato costante

Se prendiamo un contratto dove un loop restituisce sempre lo stesso risultato:

```
1 function const() returns( uint ){
2         uint sum = 0;
3         for(uinti=1; i<=10; i++) {
4                 sum += i; }
5                 return sum;
6 }
```

Ogni volta che una transazione in entrata chiederà l'esecuzione della funzione, il loop verrà eseguito ogni volta inutilmente, sprecando gas. Neanche in questo caso il compilatore ottimizza il codice. Il loop andrebbe eliminato e la funzione dovrebbe semplicemente restituire il risultato dell'operazione. (55 in questo caso)

La stessa cosa vale per funzioni dove una parte di computazione è sempre costante:

```
1 uint x=1;
2 uint y=2;
3 function const2( uint k ){
4         uint sum = 0;
5         for(uinti=1; i<=k; i++) {
6         sum = sum + x + y; }
7         }
```

La somma di x e y sarà sempre costante (3), un compilatore potrebbe effettuare anticipatamente il calcolo e sostituirlo alla somma.

5.2.2 Operazioni costose nei loop

Come abbiamo visto nel capitolo precedente, le operazioni di lettura/scrittura dello storage influiscono negativamente sulle performance, essendo le più costose in termini di gas. Operazioni costose come queste nei loop rappresentano un enorme spreco di gas poiché ripetute più volte in un'unica invocazione.

Nel contratto:

```
1 Contract c {
2         uint sum=0;
3         function deadc(uint x) {
4                 for (uint i=0; i<x; i++){
5                 sum=sum+1; }
6                 }
7         }
```

Ad ogni iterazione verrà richiesto un'operazione *SLOAD* per caricare *sum* nello stack, e una *SSTORE* per memorizzarlo nello storage (le operazioni intermedie non ci interessano in quanto poco dispendiose di gas). Una soluzione può essere quella di utilizzare nel loop una variabile temporanea *tmp* caricata nello stack e solo finito il loop assegnare
sum=tmp

Verrebbero eseguite 2 operazioni sullo storage invece che le 2x dell'esempio.

5.2.3 Loop accorpabili

Quando possibile, unire due loop consecutivi in uno ci permette di risparmiare gas, ad esempio:

```
1 function fusion(uint x){
2         uint m=0;
3         uint v=0;
4         for(uint i=0; i<x; i++) {
5                 m += i; }
6         for(uint j=0; j<x; j++) {
7                 v -= j; }}
```

Unendo i due cicli for non solo si ottiene un risparmio sulle operazioni di storage (o memoria se decidiamo di utilizzare una variabile temporanea come visto prima), si riducono salti condizionali e operazioni di comparazione, ma anche diminuisce la lunghezza dei bytecode generato:

```
1 for(uint i=0; i<x; i++) {
2         m += i;
3         v -= i;
4 }
```

In futuro il compilatore potrebbe fare ottimizzazioni di questo genere in automatico, ma attualmente non ci sono EIP a riguardo e le community di Ethereum in genere è contraria a modifiche di questo

Capitolo 6

Conclusione

Essendo un campo completamente nuovo, la programmazione di smart-contract introduce dinamiche nuove ed ancora poco esplorate. Se da un lato le possibilità aperte dalla tecnologia blockchain affascinano per la portata globale che possono avere, dall'altro ci si rende conto che attualmente la tecnologia deve sciogliere molti nodi. Problemi di governance, sicurezza e scalabilità ne minano l'adozione di massa e dinamiche complesse come la gestione del costo in gas delle operazioni possono spaventare gli sviluppatori che si affacciano a questo mondo. La documentazione infatti è ancora poca, non è un caso che la maggior parte dei problemi di sicurezza e gestione sbagliata del gas derivano da un approccio alla programmazione di smart-contract troppo "tradizionale", che non tiene conto delle nuove dinamiche: la necessità di un codice minimale (in quanto costoso) e di una conoscenza della tecnologia sottostante, che va dal funzionamento della blockchain alle dinamiche dei costi delle operazioni in bytecode. Tuttavia ritengo che ci si stia muovendo nella direzione corretta, le community del mondo blockchain sono molto attive, nascono progetti a vista d'occhio per poter migliorare sicurezza e performance. Stanno nascendo le prime soluzioni di scaling off-chain [], nuovi framework, compilatori, linguaggi, virtual machine e piattaforme che promettono di diminuire i costi del gas (e più in generale delle fee), avere transazioni veloci e sicure proponendosi di fatto come il back-bone di un nuovo internet. Se si riuscirà in questa impresa ci saranno profondi cambiamenti nel mondo, la blockchain è una tecnologia impossibile da censurare, libera e decentralizzata che basa il suo stesso funzionamento su regole sociali e incentivi monetari, creando così dei piccoli sistemi economici con modelli di governance ad-hoc. Ad oggi però un approccio allo sviluppo su questa tecnologia non solo richiede una profonda conoscenza dei funzionamenti a basso livello ma anche la necessità di essere estremamente aggiornati su bug, aggiornamenti e best-practice. Superati questi ostacoli si aprono infinite possibilità che ci permettono di sperimentare e rendere accessibile a tutto il mondo questa tecnologia potenzialmente rivoluzionaria.

Bibliografia

[1] - Andreas M. Antonopoulos. *Mastering Bitcoin 2nd edition*. Giugno 2017. https://github.com/bitcoinbook/bitcoinbook .

[2] - Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system* (Bitcoin whitepaper). 31 Ottobre 2008.

[3] - Nick Szabo. *Smart Contracts: Building Blocks for Digital Markets*. 1996.
http://www.fon.hum.uva.nl

[4] - Ethereum Foundation. *Ethereum white paper*. 6 Aprile 2014.
https://github.com/ethereum/wiki/wiki/White-Paper.

[5] - Dr. Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger* (aggiornato dopo la revisione EIP-150). 8 Luglio 2017 (6 Aprile 2014 la prima versione). https://ethereum.github.io/yellowpaper/paper.pdf

[6] - David Johnston, Sam Onat Yilmaz, Jeremy Kandah, Nikos Bentenitis, Farzad Hashemi, Ron Gross, Shawn Wilkinson and Steven Mason. *The General Theory of Decentralized Applications, Dapps*. Feb 2015. https://github.com/DavidJohnstonCEO/DecentralizedApplications.

[7] - Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli. *A survey of attacks on Ethereum smart contracts*. 22 Aprile 2017

[8] - Michele Emiliani. *Sicurezza, attacchi e best practice nei contratti Solidity di Ethereum* Ottobre 2017. http://amslaurea.unibo.it/14504/

[9] - Joseph Poon, Vitalik Buterin. *Plasma: Scalable Autonomous Smart Contracts*. August 11, 2017. https://plasma.io/plasma.pdf

[10] - Documentazione Solidity. *Solidity Documentation*. 2017. https://solidity.readthedocs.io/en/develop/.

[11] - Go-Ethereum. *Go Implementation of Ethereum protocol*. 2016. https://github.com/ethereum/go-ethereum

[12] - ebuchman *Notes on the EVM*. 2016. https://github.com/CoinCulture/evm-tools/blob/master/analysis/guide.md

[13] - Howard *Diving Into Ethereum Virtual Machine*. 2017. https://medium.com/@hayeah/diving-into-the-ethereum-vm-6e8d5d2f3c30

[14] - Howard *How fixed-length data types are represented*. 2017. https://medium.com/@hayeah/diving-into-the-ethereum-vm-part-2-storage-layout-bc5349cb11b7

[15] - pirapira *Awesome Ethereu Virtual Machine*. 2017. https://github.com/pirapira/awesome-ethereum-virtual-machine

[16] - Martin Swende *Solidity optimizer Bug*. 3 Maggio 2017. https://blog.ethereum.org/2017/05/03/solidity-optimizer-bug/

[17] - chriseth issue1265 *Critical: Optimizer bug for multi-dimensional arrays*. 2016. https://github.com/ethereum/solidity/issues/1265

[18] - issue333 emphBug in solidity optimizer. Storage changes are not persisted. 2017. https://github.com/ethereum/solidity/issues/333

[19] - Ting Chen, Xiaoqi Li, Xiapu Luo, Xiaosong Zhang. *Under-optimized smart contracts devour your money*. 2017. http://ieeexplore.ieee.org/document/7884650/

[20] - MediumDotCom. *Public vs External Functions in Solidity*.2017. https://ether.direct/2017/07/29/public-vs-external-functions-in-solidity/

Ringraziamenti

Ringrazio il professor Cosimo Laneve per avermi dato l'opportunità ed una ragione per approfondire un argomento per il quale ho una forte passione, ringrazio la mia famiglia e i miei amici per il sostegno e la pazienza.