From Bottlenecks to Breakthroughs: Optimizing zkEVM Provers

Leo Jeong
Linea, Consensys

Devcon 7
Nov 13, 2024

Linea

- Intro
  - Journey of Optimizing zkEVM Provers in Production
  - Why is zkEVM prover efficiency important?
  - Characteristics of zkEVM provers and common bottlenecks
  - Scope of the talk: what will and won't be covered

- Optimizing zkEVM Provers
  - Diagnostics and Bottleneck Analysis
  - CPU Optimization
  - Memory Optimization
  - Others

- Conclusion
  - Results
  - Lessons learned: Dos and Don'ts
  - Future work

Linea

Intro Journey of Optimizing zkEVM Provers in Production

Initial challenges and problems

- Linea mainnet alpha launch (Aug 16th, 2023)
- Prover was slow
- We were spending $$$ for Provers

Linea

Intro Journey of Optimizing zkEVM Provers in Production

What we did

- 1 year of optimization in production
- Reduced the prover runtime by 75% (40 to 10 mins)

Linea

Intro Why is zkEVM prover efficiency important?

Cost

- One of the biggest cost contributors
- Prover optimization → directly reduces operational expenses

Performance Impact (prover runtime)

- Faster verification and finalization
- Improved throughput of zk-rollups (e.g., TPS).
- Better user experience

Scalability

- Supports more transactions
- Handles more complex proofs

Linea

Intro Characteristic of zkEVM Provers and common bottlenecks

Complexity

- Inherently complex due to the full compatibility required with EVM
- Each zkEVM project has its own design

High resource consumption

- CPU and Memory-intensive due to large-scale cryptographic calculations and extensive data structures

---

Common bottlenecks and issues

CPU usage / parallelization

Memory overhead

I/O bottlenecks

Linea

Intro Scope of the talk: what will and won't be covered

What won't be covered

- Linea-specific optimizations
- Detailed cryptographic theory behind zkEVM prover
- Hardware accelerated optimizations (GPU, FPGA, etc.)

What will be covered

- Quick wins and strategies for any zkEVM project
- CPU/memory optimizations without architectural changes or significant modifications

Linea

- Intro
  - Journey of Optimizing zkEVM Provers in Production
  - Why is zkEVM prover efficiency important?
  - Characteristics of zkEVM provers and common bottlenecks
  - Scope of the talk: what will and won't be covered

- Optimizing zkEVM Provers
  - Diagnostics and Bottleneck Analysis
  - CPU Optimization
  - Memory Optimization
  - Others

- Conclusion
  - Results
  - Lessons learned: Dos and Don'ts
  - Future work

Linea

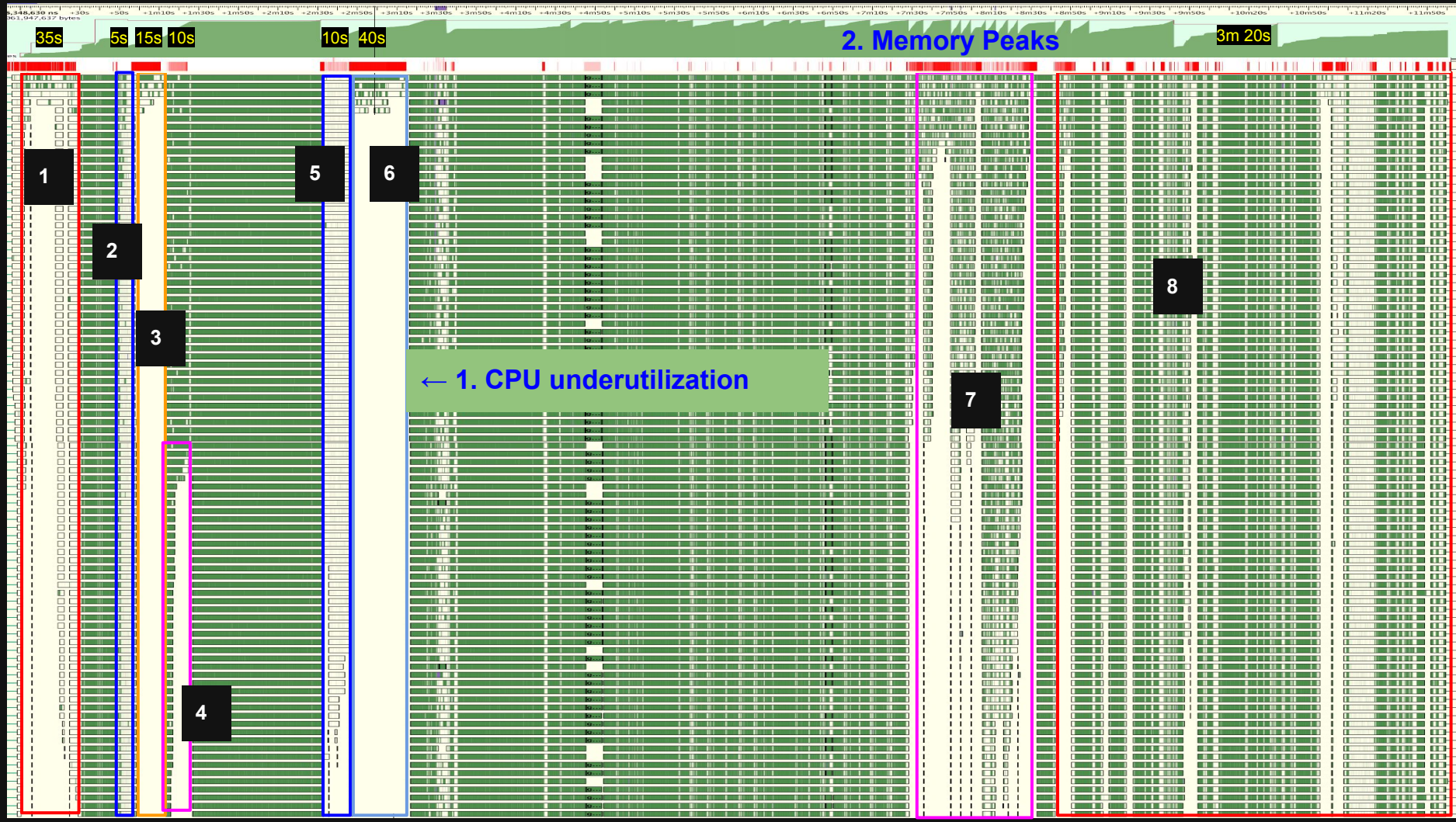
Optimizing zkEVM Provers: Diagnostics and Bottleneck Analysis

Mesure before you optimize

- “Premature Optimization Is the Root of All Evil”
  by Donald Knuth, The Art of Computer Programming

CPU and memory profiling

- Visualize CPU and memory usage
- Understand your code
- Built-in or 3rd-party profiling tools
  - golang: https://gotraceui.dev/

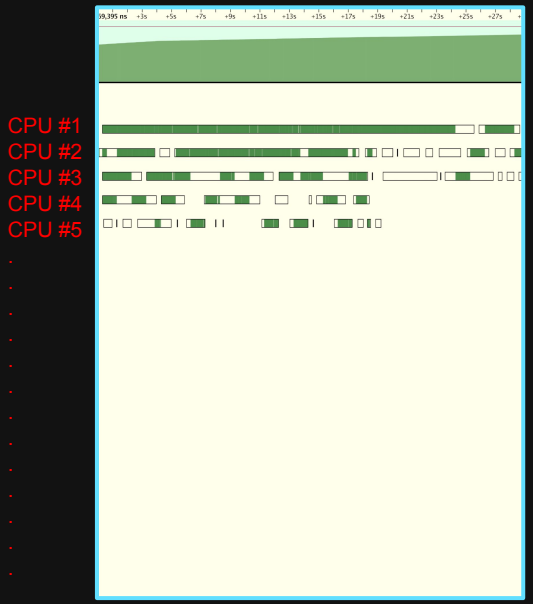

35s
5s
15s
10s
10s
40s
2. Memory Peaks
3m 20s
1
2
3
4
5
6
7
8
← 1. CPU underutilization

Linea

Optimizing zkEVM Provers: Diagnostics and Bottleneck Analysis

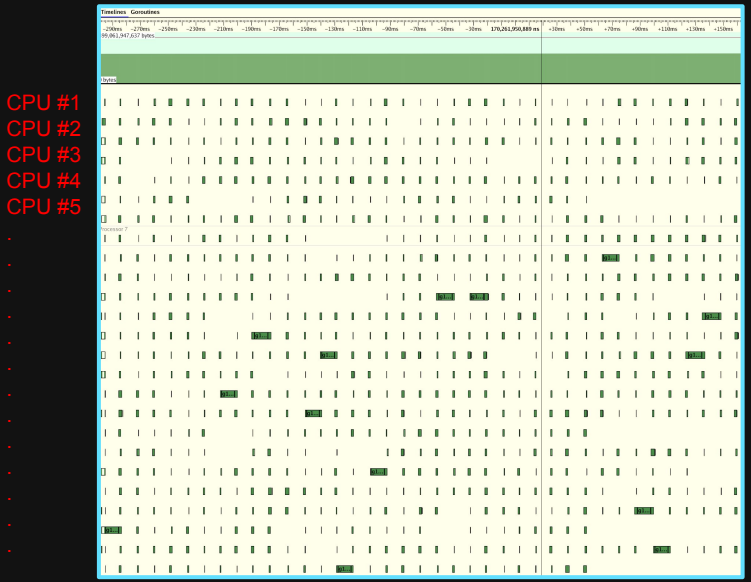
Poor Parallelization:

- Case 1, 3, 4, 6 7 and 8



CPU Underutilization:

- Case 2, 5, 7 and 8



Linea

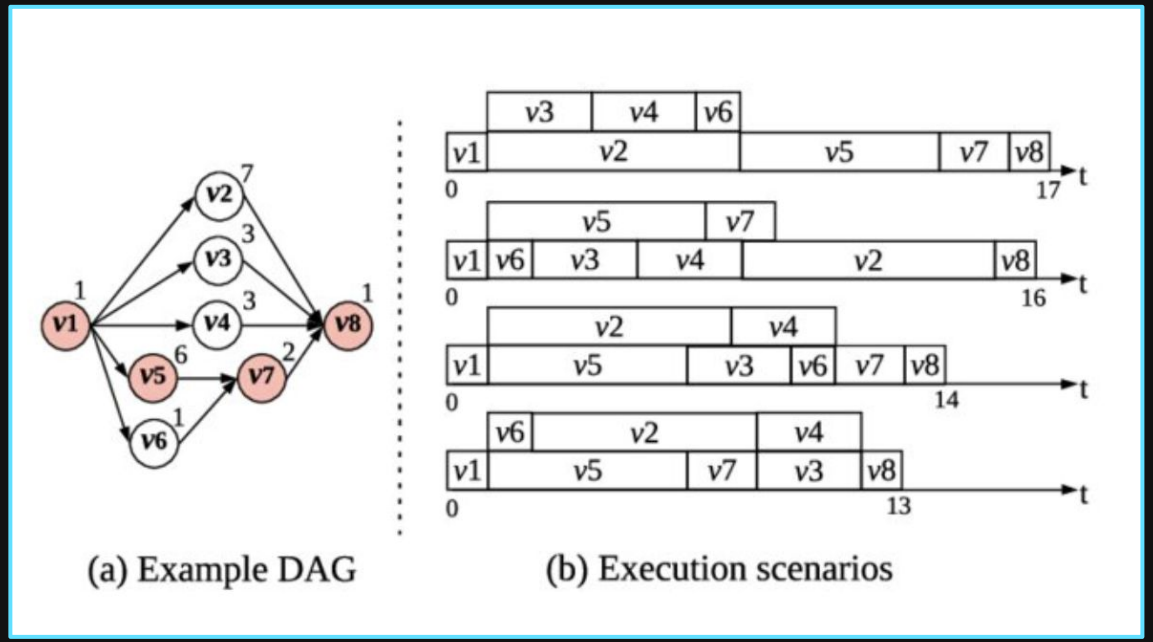
Optimizing zkEVM Provers: CPU Optimization

Leveraging DAG (Directed Acyclic Graph) for Better Parallelization

v

module

dependency

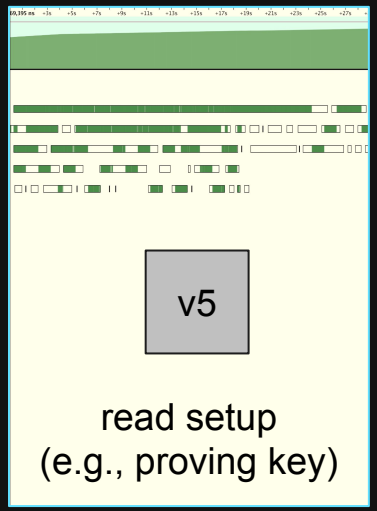


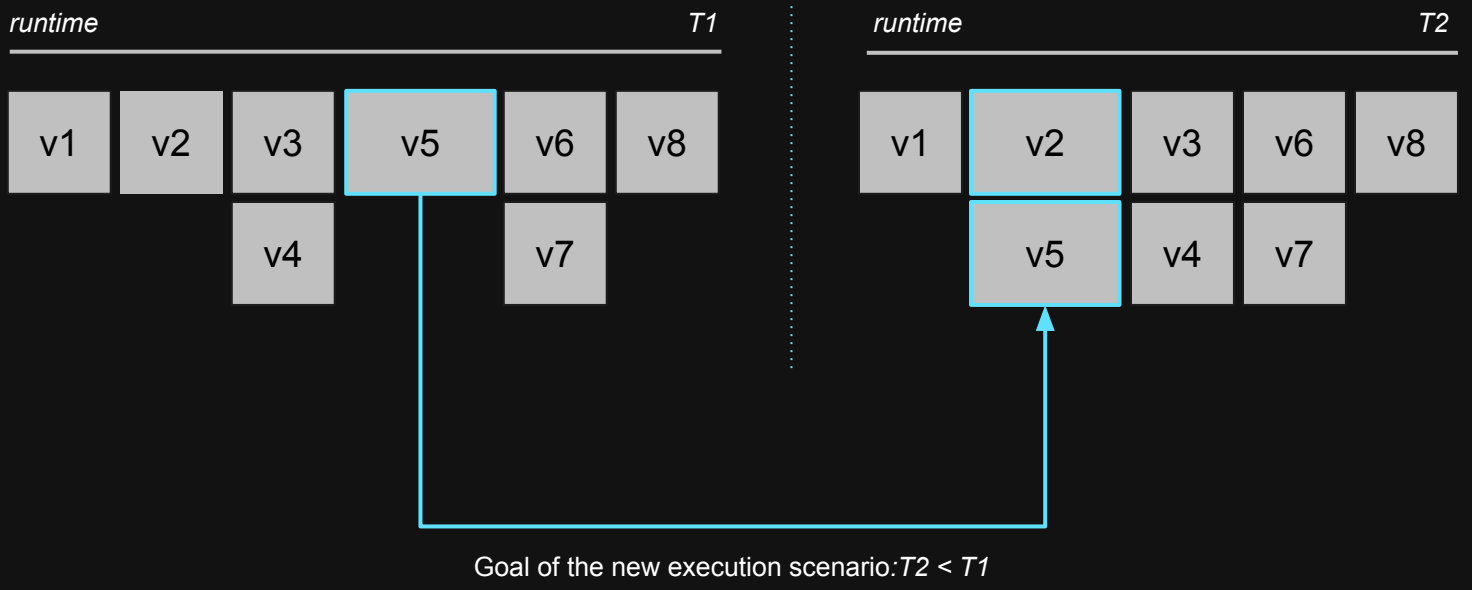
Reference: DAG Scheduling and Analysis on Multiprocessor Systems: Exploitation of Parallelism and Dependency, Shuai Zhao, et al.

Linea

Optimizing zkEVM Provers: CPU Optimization

Leveraging DAG (Directed Acyclic Graph) for Better Parallelization



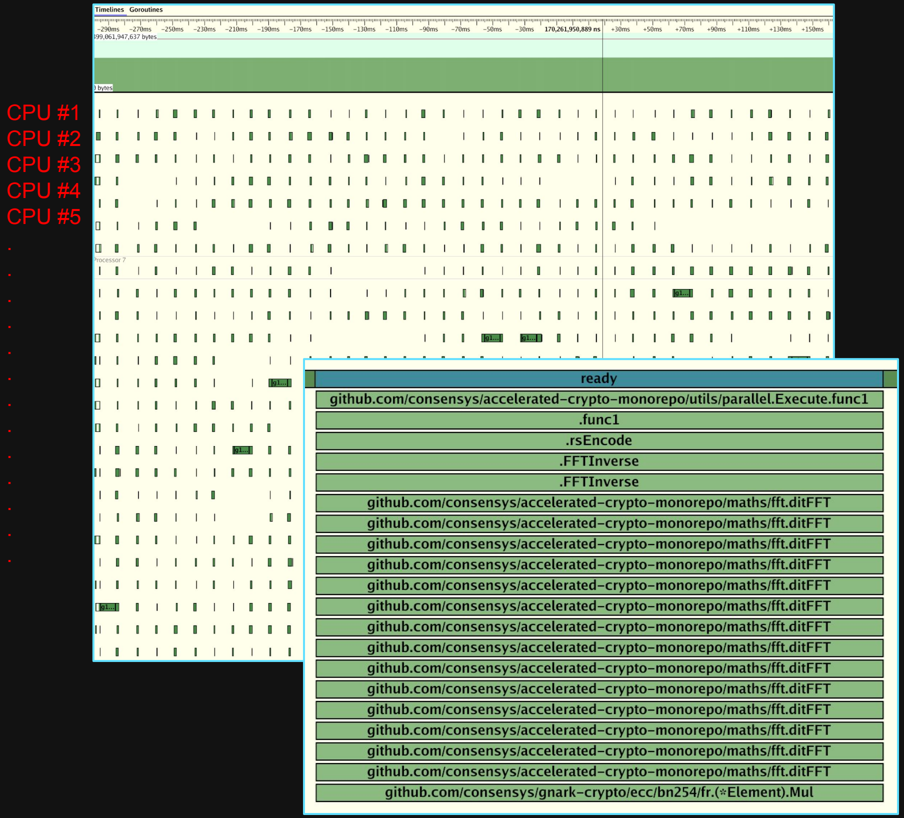


Linea

Optimizing zkEVM Provers: CPU Optimization

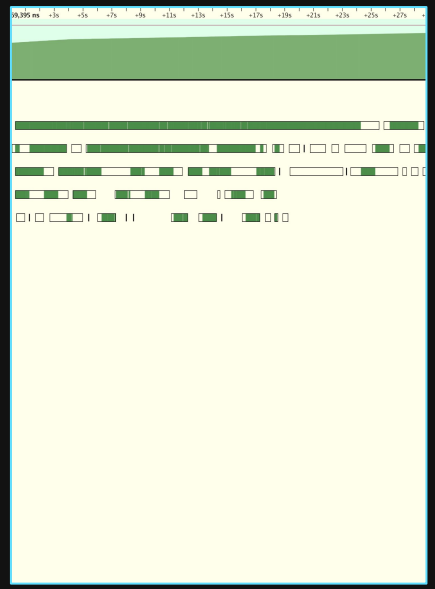
CPU Underutilization

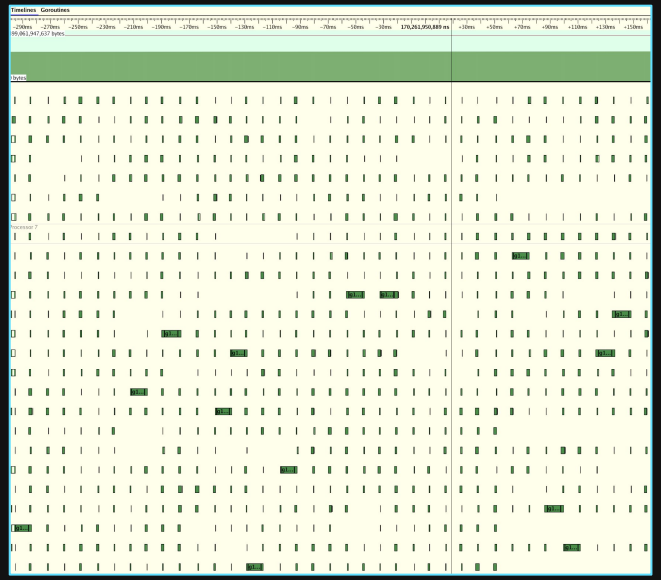
- Poor parallelization
- Synchronization
- Locks
- Blocking I/O operations (e.g., file reads, network requests)



Linea

Optimizing zkEVM Provers: CPU Optimization





Poor Parallelization

Underutilization

CPU #1
CPU #2
CPU #3
CPU #4
CPU #5

Linea

- Optimizing zkEVM Provers
  - Diagnostics and Bottleneck Analysis
    - CPU and Memory Profiling
    - Identifying bottlenecks
  - CPU Optimization
    - Poor Parallelization (DAG)
    - Underutilization
  - Memory Optimization
  - Others

Linea

Optimizing zkEVM Provers: Memory Optimization [peak: 700 GB]

Why is it important?

- Memory is expensive
  - hpc6id.32xlarge (64 cores, 970 GB): 5.70 USD / Hour
  - hpc6a.xlarge (96 cores, 380 GB): 2.88 USD / Hour
- Higher peak memory usage → more expensive machines

Goal

- Flatten memory usage and eliminate memory peaks

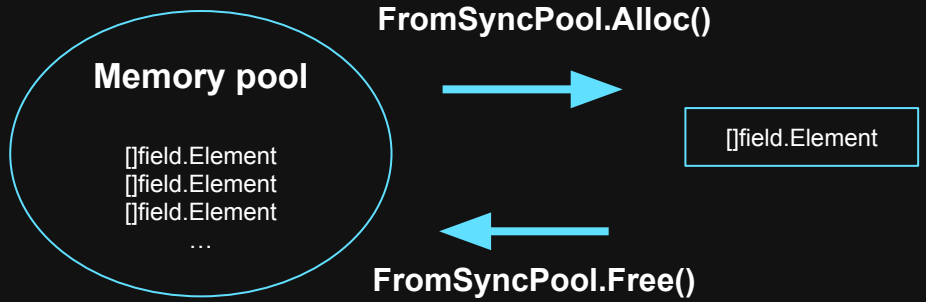
Reference : https://tip.golang.org/doc/gc-guide

Linea

Optimizing zkEVM Provers: Memory Optimization [700 -> 450 GB]

Memory pool and allocation

- Define a large memory pool of frequently used data structure
- Reduce the number of memory allocation
- Reduce GC




```go
type FromSyncPool struct {
    size int
    P    sync.Pool
}

func CreateFromSyncPool(size int) *FromSyncPool {
    return &FromSyncPool{
        size: size,
        P: sync.Pool{
            New: func() any {
                res := make([]field.Element, size)
                return &res
            },
        },
    }
}

func (p *FromSyncPool) Alloc() *[]field.Element {
    res := p.P.Get().(*[]field.Element)
    return res
}

func (p *FromSyncPool) Free(vec *[]field.Element) error {
    if len(*vec) != p.size {
        utils.Panic("expected size %v, expected %v", len(*vec), p.Size())
    }

    p.P.Put(vec)

    return nil
}
```

linea-monorepo/prover/maths/common/mempool/from_sync_pool.go

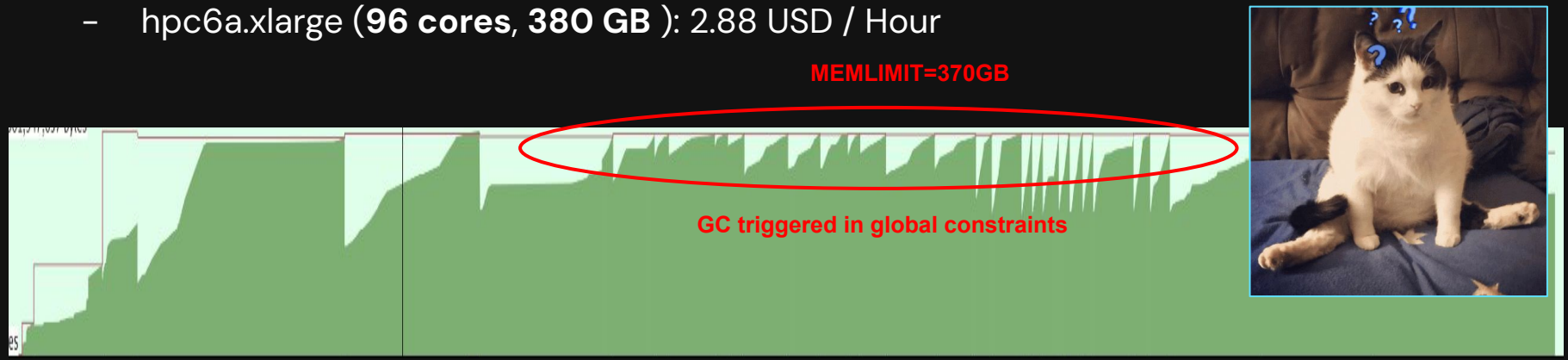
Linea

Optimizing zkEVM Provers: Memory Optimization [450 -> 370 GB]

Memlimit

- Soft-limit of memory peak by triggering GC when the memory usage hits MEMLIMIT (*MEMLIMIT*=370GB)

- ~~hpc6id.32xlarge (64 cores, 970 GB): 5.70 USD / Hour~~
- hpc6a.xlarge (**96 cores**, **380 GB**): 2.88 USD / Hour



Linea

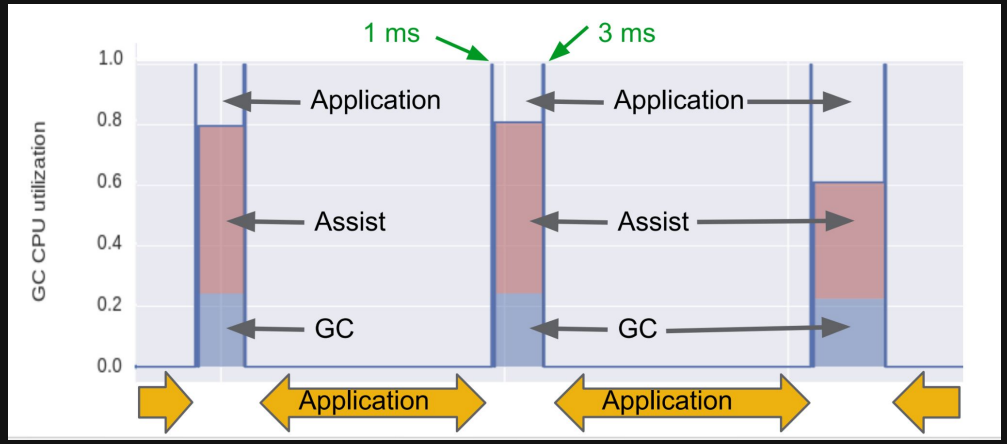
Optimizing zkEVM Provers: Memory Optimization [370 GB]

Garbage-collected language

- Go, Java, Python, ...

It's convenient, but...

- Non-deterministic GC is a problem
- What if it happens during a critical task?



reference: https://go.dev/

Linea

Memory Optimization [370 GB] + Manual GC

Fine-grained garbage collection

- Make sure no GC is triggered during memory and CPU-intensive tasks
- Manage non-deterministic GCs for critical areas of your code

Manual GC

- Run manual GC before critical tasks
- Less context switching overhead
  → less GC pause time
- Significant runtime improvement

```go
if ctx.DomainSize >= GC_DOMAIN_SIZE {
    runtime.GC()
}

go func() {
    // Compute once the FFT of the natural columns
    parallel.ExecuteChunky(len(ctx.AllInvolvedRoots), func(start, stop int) {
        for k := start; k < stop; k++ {
            pol := ctx.AllInvolvedRoots[k]
            name := pol.GetColID()

            var witness sv.SmartVector
            lockRun.Lock()
            witness, isNatural := run.Columns.TryGet(name)
            lockRun.Unlock()

            if !isNatural {
                witness = pol.GetColAssignment(run)
            }
            witness = sv.FFTInverse(witness, fft.DIF, false, 0, 0, nil)

            lock.Lock()
            coeffs[name] = witness
            lock.Unlock()
        }
    })
    wg.Done()
}()
```

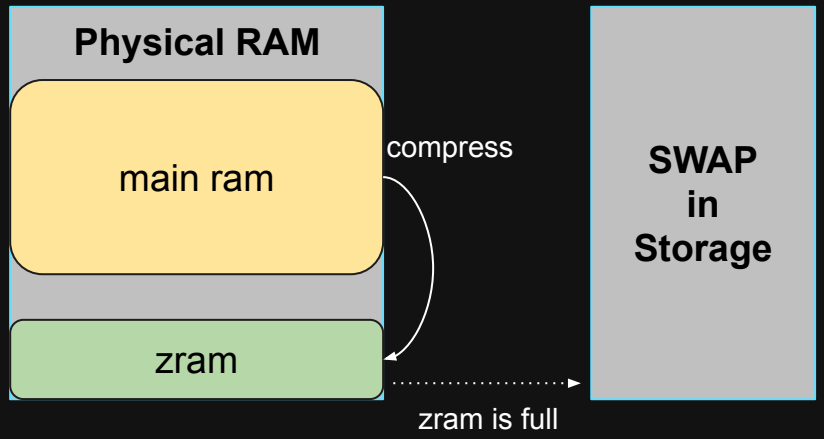
linea-monorepo/prover/protocol/compiler/globalcs/quotient.go

Linea

Optimizing zkEVM Provers: Memory Optimization [No OOM]

ZRAM (In-memory swap)

- To handle extensive memory demands and avoid Out-of-memory (OOM)
- Available since linux kernel version 3.14
- Massive integration tests with TBs of memory usage
  - *vm.swappiness = 1 to 10*
  - *lz4 compression algorithm* (https://github.com/lz4/lz4)



750 GB physical memory + 1.5 TB ZRAM In-memory Swap Space

Linea

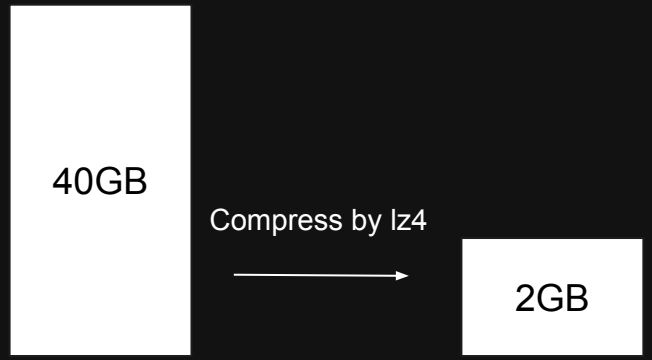
- Optimizing zkEVM Provers
  - Diagnostics and Bottleneck Analysis
    - CPU and Memory Profiling
    - Identifying bottlenecks
  - CPU Optimization
    - Poor Parallelization (DAG)
    - Underutilization
  - Memory Optimization
    - Memory Pooling
    - Memlimit
    - Fine-grained Garbage Collection (Manual GC)
    - ZRAM
  - Others

Linea

Optimizing zkEVM Provers: Others

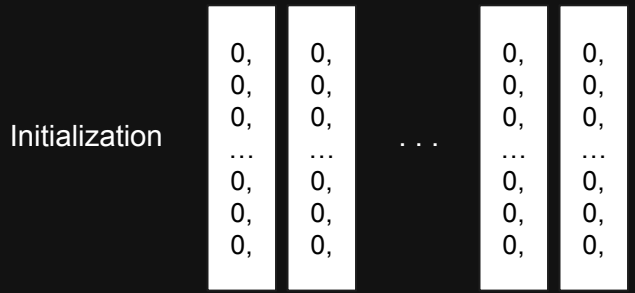
Data Compression

- The transaction traces
  → show high compression ratio
- 40GB -> 2GB
- Use it when you have I/O bottlenecks



Zeroization

- Time taken for initializing your data structure
- Auto-zeroization, optimized libraries for zeroization, etc.



Linea

- Intro
  - Journey of Optimizing zkEVM Provers in Production
  - Why is zkEVM efficiency important?
  - Characteristics of zkEVM provers and common bottlenecks
  - Scope of the talk: what will and won't be covered

- Optimizing zkEVM Provers
  - Diagnostics and Bottleneck Analysis
  - CPU Optimization
  - Memory Optimization
  - Others

- Conclusion
  - Results
  - Lessons learned: Dos and Don'ts
  - Future work

Linea

Conclusion

Results: Happy, Happy, Happy

- Reduced the zkEVM prover runtime by 75%
- Saved operating cost
- Improved Prover throughput
  - Faster finality
  - Higher TPS
- (Ironically) We understood what we did and what we are doing

Linea

Conclusion

Lessons learned: Dos and Don'ts for other zkEVM projects

- Don’t pre-optimize your code (premature optimization)
- Make optimization plans based on measurement
- Do it with a global view to set priorities

Future work

- Distributed zkEVM prover
  - different modules on different machines
- Hardware accelerated zkEVM prover

Linea

Optimization is never a one-time task.

Measure, Analyze, Optimize and Repeat.

https://github.com/Consensys/linea-monorepo

THANK YOU

leo.jeong@consensys.net