Bifold

A collision-resistant, splitable RNG for structured parallelism

Stefan Karpinski

https://JuliaHub.com

RNG algorithm history

version	algorithm	seeding	location	size
≤ 1.5	Mersenne Twister	naive	global	2496 bytes
= 1.6	Mersenne Twister	SHA256	global	2496 bytes
≥ 1.7	Xoshiro256++	SHA256	task local	32 bytes

Xoshiro256's compact size enables task-local RNG state

- Reproducible multithreaded RNG sequences (seed & task tree shape)

Works great, except...

In Julia 1.7-1.9:

```
julia> begin
           Random.seed!(0)
           println(repr(rand(UInt64)))
           println(repr(rand(UInt64)))
       end
0x67dbeba77c5b608f
0x118c381a04770c92
```

Works great, except...

In Julia 1.7-1.9:

```
julia> begin
           Random.seed!(0)
           println(repr(rand(UInt64)))
           @async nothing # <= this can't matter, right?
           println(repr(rand(UInt64)))
       end
0x67dbeba77c5b608f
0x1dc7a124563dedbd # <= different value!
```

Why does this happen?

When forking a task, child's RNG needs to be seeded

- Can't just copy parent state
- Parent & child would produce same RNG values

Child is seeded by sampling from the parent RNG (in 1.7-1.9)

- This modifies the parent RNG, changing its RNG sequence
  - (uses RNG four times—once per word of Xoshiro256 state)

Surely someone has worked on this...

DotMix (2012): *Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms*

- Charles Leiserson, Tao Schardl, Jim Sukha
- for MIT Cilk parallel runtime

SplitMix (2014): *Fast Splittable Pseudorandom Number Generators*

- Guy Steele Jr, Doug Lea, Christine Flood
- for Oracle's Java JDK8

DotMix

Concept: “pedigree” vector of a task

- Root task has pedigree $\langle\rangle$
- If parent has pedigree $\langle k_1, k_2, \ldots, k_{d-1}\rangle$
- Its children are at depth d in the task tree
- The k_dth child has pedigree $\langle k_1, k_2, \ldots, k_{d-1}, k_d\rangle$

Every prefix of a task’s pedigree is the pedigree of an ancestor

DotMix

Core idea:

- Compute a dot product of a task's pedigree with random weights
- Can prove dot product collisions have probability $1/2^{64}$
- Apply bijective, non-linear "finalizer" based on MurmurHash
- Finalized value is used to seed a main RNG (per-task)

DotMix: details

The dot product of a pedigree vector looks like this:

$$\chi\langle k_1, \ldots, k_d \rangle = \sum_{i=1}^{d} w_i k_i \pmod{p}$$

- p is a prime modulus
  - necessary for proof of collision resistance
  - complicates the implementation a fair bit
  - they use $p = 2^{64} - 59$

DotMix: proof

Suppose two different tasks have the same χ value:

$$\sum_{i=1}^{d} w_i k_i = \sum_{i=1}^{d} w_i k'_i \pmod{p}$$

Let j be some coordinate where $k_j \neq k'_j$

$$w_j(k_j - k'_j) = \delta \pmod{p}$$

Unlikely: only one w_j value satsifies this (needs primality)

SplitMix

So funny story...

Authors spend *a lot of time* on an optimized version of DotMix

- I thought that this optimized version was SplitMix (it's not)
- Then the paper just throws up its hands and does something else

In my defense, they spend the first *12 out of 20* pages on DotMix

What SplitMix actually is

```
advance(s::UInt64) = s += γ # <= very simple RNG transition

function gen_value(s::UInt64)
    s ⊻= s >> 33; s *= 0xff51afd7ed558ccd
    s ⊻= s >> 33; s *= 0xc4ceb9fe1a85ec53
    s ⊻= s >> 33
end
```

- Task state: s is the main RNG state; γ is a per-task constant
- RNG core is very weak, relies entirely on output function

SplitMix: splitting

Similar to what we're doing in Julia 1.7-1.9

- Sample parent's RNG to seed child state

With a clever addition: SplitMix is parameterized by per-task γ

- As long as γ values are different, s collisions are fine
- Child γ derived from parent's s value on task fork
  - (this has to be done somewhat carefully)

Auxiliary RNG or not?

DotMix is explicitly intended as an *auxiliary RNG*

- Used to seed main RNG on task fork, not to generate samples

SplitMix can be used as main RNG *and* to fork children

- But if you do that, then forking changes the parent RNG stream
    - (what we’re trying to avoid)

Auxiliary RNG or not?

"Proof" that we need auxiliary RNG state:

- Requirement: forking children must not change main RNG state
- But *something* must change or every child task would be identical
- That "something" is the auxiliary state. QED.

SplitMix as auxiliary RNG?

If we used SplitMix for this, it would add 128 bits of aux RNG state

- s is 64 bits, γ is 64 bits — 128 bits total

We don't need SplitMix's ability to generate *and* split

- We should use all auxiliary bits for splitting, not generation

DotMix does this — and it has collision resistance proof

- So, let's try using DotMix...

Optimized DotMix (SplitMix paper)

Main optimization: incremental computation of dot product

```
cached_dot += weights[depth]
child_dot = cached_dot
```

Also improved:

- Cheaper non-linear, bijective finalizer
- Prime modulus of $p = 2^{64} + 13$ with some cleverness

Improving DotMix even further

Prime modulus arithmetic is slow and complicated

- *So much* effort is put into this in the DotMix & SplitMix papers
    - (lack of unsigned integers in JVM does not help)

Would be excellent if we could just use native arithmetic

Why do we need a prime modulus?

For the proof of collision resistance:

- So that $k_j - k'_j \neq 0$ is guaranteed to be invertible
- Recall: $w = (k_j - k'_j)^{-1}\delta \pmod p$

Why do we need a prime modulus?

But why are pedigree coordinates arbitrary integers?

- Because the task tree is n-ary

But forking tasks is binary: one task splits into two

- Can we assign pedigree with binary coordinates somehow?

IDs instead of pedigrees

Root ID:

- $\mathrm{root}_{\mathrm{id}} = 0$ (node ID – immutable)
- $\mathrm{root}_{\mathrm{ix}} = 0$ (fork index – mutable)

Child ID:

- $\mathrm{child}_{\mathrm{id}} = 2^{\mathrm{parent}_{\mathrm{ix}}} + \mathrm{parent}_{\mathrm{id}}$
- $\mathrm{child}_{\mathrm{ix}} = \mathrm{parent}_{\mathrm{ix}} + = 1$

Recovering pedigree

Can turn task IDs into binary pedigree vectors:

- k_i is the ith bit of the task ID

Very different tree shape

Collision proof revisited

With binary coordinates $k_j - k'_j$ is always ± 1

$$w_j(k_j - k'_j) = \delta \pmod{n}$$
$$w_j = (k_j - k'_j)\delta = \pm\delta \pmod{n}$$

- Modulus can be anything including $n = 2^{64}$
- No more prime modulus shenanigans!
- Yay, machine arithmetic

Simplified dot product

Incremental dot product computation becomes *very* simple:

```
child_dot = parent_dot + weights[fork_index]
```

That’s all:

- Get the random weight for the current “fork index”
- Add it to the parent’s dot product

Random weights

DotMix uses a pre-generated array of random weights

- Static — shared between all tasks
- 1024 random 64-bit values (8KiB of static data)
- If the task tree gets deeper than 1024, they recycle weights!

This all seems a bit nuts

- Can't we use a PRNG to generate weights?

Pseudorandom weights

That’s what we’ll do:

- Use 64-bits of aux RNG state to generate weights
- Output: same size as state
  - can’t have duplicates
  - beneficial in this case

PCG-RXS-M-XS-64 (PCG64) is arguably the best PRNG for this case

- LCG core + strong non-linear bijective output function

Julia 1.10: "DotMix++"

- Generate weights with PCG (adds 1 × 64-bit word)
- Accumulate dot product into main RNG state (no extra words!)

```
w = aux_rng # weight from previous LCG state (better ILP)
aux_rng = LCG_MULTIPLIER*aux_rng + 1 # advance LCG
w ⊻= w >> ((w >> 59) + 5)
w *= PCG_MULTIPLIER
w ⊻= w >> 43
main_rng += w # accumulate dot product into main RNG
```

Variations on a theme

But our main RNG has *four* 64-bit state registers, not just one...

- We compute four different “independent” weights
- Accumulate a different dot product into each register
- Improves collision resistance from $1/2^{64}$ to $1/2^{256}$

Win-win design:

- Avoid extra task state for dot product accumulation
- Massively increase collision resistance

Four PCG variations

```
w = aux_rng
aux_rng = LCG_MULTIPLIER*aux_rng + 1

for register = 1:4
    w += RANDOM_CONSTANT[register]
    w ⊻= w >> ((w >> 59) + 5)
    w *= PCG_MULTIPLIER[register]
    w ⊻= w >> 43

    main_rng[register] += w
end
```

Accumulating into main RNG

Main RNG registers used to accumulate dot products — is this ok?

- DotMix suggests “seeding” dot products with random initial values
- We’re effectively seeding with what main RNG state happens to be

Collision resistance proof can be made to work

- Even when main RNG use is interleaved with task forking
  - (key facts: RNG advance is bijective, δ doesn’t matter)

All Good?

Unfortunately not. In Feb 2024, foobar_lv2 pointed out:

- In Julia 1.10 there's an observable linear relationship between RNG outputs when four tasks are spawned in a certain way

Linear Relationship

On Julia 1.10 this function only produces *nine* different values:

```
using .Threads

macro fetch(ex) :(fetch(@spawn($(esc(ex))))) end

function taskCorrelatedXoshiros()
    r11, r10 = @fetch (@fetch(rand(UInt64)), rand(UInt64))
    r01, r00 = (@fetch(rand(UInt64)), rand(UInt64))
    (r01 + r10) - (r00 + r11)
end
```

Linear Relationship: Why?

Task diagram:

$$\begin{array}{ccc} \text{task}_{00} & \xrightarrow{+w_1} & \text{task}_{10} \\ \big\downarrow{\scriptstyle +w_2} & & \big\downarrow{\scriptstyle +w_2} \\ \text{task}_{01} & & \text{task}_{11} \end{array}$$

Linear Relationship: Why?

Relationships:

$$\begin{aligned}
\text{dot}_{10} &= \text{dot}_{00} + w_1 \\
\text{dot}_{01} &= \text{dot}_{00} + w_2 \\
\text{dot}_{11} &= \text{dot}_{10} + w_2 = \text{dot}_{00} + w_1 + w2
\end{aligned}$$

Therefore:

$$\text{dot}_{01} + \text{dot}_{10} = 2\,\text{dot}_{00} + w_1 + w_2 = \text{dot}_{00} + \text{dot}_{11}$$

Is DotMix broken? (No)

Core dot product computation in DotMix has this issue (inherently)

- DotMix applies a non-linear bijective finalizer to the dot product

The paper kind of glosses over this

- Probably bc they are professionals and this is very obvious to them
- I totally failed to realize how important this was
- [Me, an idiot]: *The weights are random, that’s good enough, right?*

How To Fix?

DotMix applies a non-linear finalizer that destroys linear relationships

- Can we do the same?

Yes, but we'd have to accumulate dot product *outside* of main RNG

- Increases every task size by the size of the accumulator
- Even one accumulator adds 8 bytes (64 bits)
- Four independent accumulators adds 32 bytes

Non-linear “dot products”?

Dot products inherently produce these problematic linear relationships

- Do we need a dot product? Could we use something non-linear?
- What is absoutely necessary to make the collision proof work?

Generalizing +

In our simplified version, the dot product is incrementally computed:

```
child_dot = parent_dot + weights[fork_index]
```

We can replace + with any doubly bijective reducer, β:

```
child_fld = β(parent_fld, weights[fork_index])
```

Accumulate = left-fold by β over active weights (bits in task ID)

Generalized proof

Can we prove collision resistance with interleaved main RNG usage?

Suppose two different tasks have the same reduction value

- Can rewind through indices where task ID bits are equal
  - because $s \mapsto \beta(s, w)$ is bijective
- Can also rewind through matching usages of main RNG
  - because main RNG advance function is bijective

Generalized proof

Reduces to one of two possible situations...

- Both tasks are forked from parents with different weights:
  - $\beta(s_1, w_1) = \beta(s_2, w_2)$ where $w_1 \neq w_2$
- One task is forked from its parent, other just used main RNG:
  - $\beta(s_1, w_1) = \alpha(s_2)$ where α is the main RNG transition

Generalized proof

Either way we have $\beta(s, w) = c$ for one of the tasks

- Only one value of w hits this value of c – unlikely to happen
  - because $w \mapsto \beta(s, w)$ is bijective

Probability of $1/2^{64}$ for each register of the main RNG

- Probability of $(1/2^{64})^4 = 1/2^{256}$ over all four registers

Result: collisions are practically impossible

“Bifold”: task forking in Julia 1.11+

```
w = aux_rng
aux_rng = LCG_MULTIPLIER*aux_rng + 1

for register = 1:4
    s = main_rng[register]
    w ⊻= RANDOM_CONSTANT[register]
    s += (2s + 1)*w # <= doubly bijective multiplication
    s ⊻= s >> ((s >> 59) + 5)
    s *= PCG_MULTIPLIER[register]
    s ⊻= s >> 43
    main_rng[register] = s
end
```

Bifold design notes

- Uses LCG state directly as weight rather than PCG64
  - Too weak for general RNG but fine for this use case
- Xor common weight with different constant per Xoshiro256 register
- Combine register and weight via “doubly bijective multiply”
  - $\mathrm{bimul}(s, w) = s + (2s+1)w = (2s+1)(2w+1) \div 2 \pmod{2^{64}}$
- Finalize with per-register variant of PCG64 non-linear output
- Accumulate into main RNG state
  - unsafe for DotMix (linear), safe for Bifold (very non-linear)

Summary

We wanted task forking *not* to affect the main RNG

- Need to add *some* auxiliary RNG state to each task

Possible algorithms:

- SplitMix — 2×64-bit words: state + gamma
- DotMix — 2×64-bit words: state + accumulator
- Bifold — 1×64-bit word: state (LCG)

Summary

Bifold = DotMix modified to being almost unrecognizable

- Fork operation is fast and simple
- “Dot product” = reduce by non-linear doubly bijective op
- Can safely accumulate into main RNG state
- Can safely interleave main RNG usage
- Collision probability is $1/2^{256}$ — effectively impossible