Future Works in LLVM
Register Allocation
Talk Overview

1. Introduction
2. Upcoming Changes
3. PBQP
Register Allocation in LLVM

- PHI Elimination
- Two-Address
- Liveness
- Coalescing
- Allocator
- Rewriter
Register Allocation in LLVM

- PHI Elimination
- Two-Address
- Liveness
- Coalescing
- Allocator
- Rewriter

Lower PHI-instructions to copies
PHI Lowering

**BB1:**
\[ \%x1 = . \]

**BB2:**
\[ \%x2 = . \]

**BB3:**
\[ \%x3 = \text{phi} \left[ \%x1, \text{BB1} \right], \left[ \%x2, \text{BB2} \right] \]
PHI Lowering

BB1:
\[ \%x1 = . \]

BB2:
\[ \%x2 = . \]

BB3:
\[ \%x3 = \%xP \]
PHI Lowering

BB1:
\%x1 = .
\%xP = \%x1

BB2:
\%x2 = .
\%xP = \%x2

BB3:
\%x3 = \%xP
Register Allocation in LLVM

PHI Elimination ➔ Two-Address ➔ Liveness

Coalescing ➔ Allocator ➔ Rewriter

Lower PHI-instructions to copies
Register Allocation in LLVM

1. PHI Elimination
2. Two-Address
3. Liveness
4. Coalescing
5. Allocator
6. Rewriter

Lower Three-Address Instructions
Two-Address Instructions
Two-Address Instructions

\[ x_3 = x_2 + x_1 \]
Two-Address Instructions

\[ x_3 = x_2 + x_1 \]

\[ x_3 = x_2 
 x_3 += x_1 \]
Register Allocation in LLVM

Lower three-address instructions
Register Allocation in LLVM

- PHI Elimination
- Two-Address
- Liveness
- Coalescing
- Allocator
- Rewriter

Construct live intervals
Live Intervals

\begin{align*}
\text{BB:} \\
\%x_1 &= \ldots \\
\quad & \quad . \\
\quad & \quad . \\
\quad & \quad . \\
\%x_2 &= \%x_1 \\
\quad & \quad . \\
\quad & \quad . \\
\ldots &= \%x_2
\end{align*}
Live Intervals

BB:

\%x1 = \ldots \\
\ldots \\
\ldots \\
\%x2 = \%x1 \\
\ldots \\
\ldots \\
\ldots = \%x2
Live Intervals

BB:

\%x1 = \ldots

\ldots

\%x2 = \%x1

\ldots

\ldots = \%x2
Register Allocation in LLVM

- PHI Elimination
- Two-Address
- Liveness
- Coalescing
- Allocator
- Rewriter

Construct Live Intervals
Register Allocation in LLVM

- PHI Elimination
- Two-Address
- Liveness

- Coalescing
- Allocator
- Rewriter

Aggressively eliminate copies
Coalescing

BB:

\[
\begin{align*}
\%x1 &= \ldots \\
\vdots & \\
\%x2 &= \%x1 \\
\vdots & \\
\ldots &= \%x2
\end{align*}
\]
Coalescing

BB:

\( \%x1 = \ldots \)

\( \ldots = \%x1 \)
Register Allocation in LLVM

- PHI Elimination
- Two-Address
- Liveness
- Coalescing
- Allocator
- Rewriter

Aggressively eliminate copies
Register Allocation in LLVM

PHI Elimination → Two-Address → Liveness → Coalescing → Allocator → Rewriter

Apply register assignment
Register Allocation in LLVM

PHI Elimination → Two-Address → Liveness

Coalescing → Allocator → Rewriter
Improvements

- New and better optimizations
- New allocators
- Cleaner infrastructure
1. Optimizations
Rematerialization
Rematerialization

\[
vrl = \langle \text{expr} \rangle
. \quad // \quad \text{stuff}
\ldots = \text{vrl}
. \quad // \quad \text{more stuff}
\ldots = \text{vrl}
\]
Rematerialization

vr1  = <expr>

// stuff
.
...
= vr1

// more stuff
.
...
= vr1
Rematerialization

\[ [M] = \langle \text{expr} \rangle \]

\[ \cdot \quad // \quad \text{stuff} \]
\[ \ldots = [M] \]

\[ \cdot \quad // \quad \text{more stuff} \]
\[ \ldots = [M] \]
Rematerialization

vr1 = <expr>

// stuff
.

// more stuff
.

... = vr1

... = vr1
Rematerialization

// stuff
... = <expr>

// more stuff
... = <expr>
Splitting
Splitting
Splitting
Splitting
Splitting

Spills introduced in to other loops

Spill
2. New Allocators
New Allocators
New Allocators

- “Linear Scan” is not, in fact, linear
New Allocators

- “Linear Scan” is not, in fact, linear
- We want something faster
New Allocators

- “Linear Scan” is not, in fact, linear
- We want something faster
- Priority coloring?
New Allocators

- “Linear Scan” is not, in fact, linear
- We want something faster
- Priority coloring?
- *Linear* Scan?
New Allocators

• “Linear Scan” is not, in fact, linear
• We want something faster
• Priority coloring?
• Linear Scan?
• Need to tidy the infrastructure
3. Cleaner Infrastructure
Currently...
Currently...

Liveness Analysis
Currently...

Liveness Analysis → Allocator
Currently...

- Liveness Analysis
- Virtual Register Map
- Allocator
Currently...

Liveness Analysis → Virtual Register Map → Allocator
Currently...

- Virtual Register Map
- Liveness Analysis
- Allocator

The diagram shows the flow between Liveness Analysis, Virtual Register Map, and Allocator.
Currently...

- Liveness Analysis
- Virtual Register Map
- Allocator

Diagram showing the flow between Liveness Analysis, Virtual Register Map, and Allocator.
Currently...

- Virtual Register Map
- Liveness Analysis
- Allocator
- Rewriter
Rewrite In Place
Rewrite In Place

Liveness Analysis
Rewrite In Place

- Live Intervals
- Liveness Analysis
Rewrite In Place

- Live Intervals
- Liveness Analysis
- Allocator
Rewrite In Place

Live Intervals

Liveness Analysis

Allocator

Modify code in-place
Rewrite In Place

Live intervals kept up-to-date

Live Intervals

Liveness Analysis

Allocator

Modify code in-place
Rewrite In Place

Live intervals kept up-to-date

Live Intervals

Liveness Analysis

Allocator

Live intervals remain valid post-alloc

Modify code in-place
Improvements

• New and better optimizations
• New allocators
• Cleaner infrastructure
Upcoming Changes
Upcoming Changes

• Live index renumbering
• Improved splitting
• Better def/kill tracking for values
Live Indexes
Live Indexes

BB:

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2

\%
\x2\ = \%
x1
add \%
x3, \%
x2
Live Indexes

BB:

\[ \%x2 = \%x1 \]
\[ \text{add } \%x3, \%x2 \]

1
2
3
4
5
6
Live Indexes

**BB:**

\[
\begin{align*}
\%x2 &= \%x1 \\
\%x3 &= \ldots \\
\text{add} \quad \%x3, \quad \%x2
\end{align*}
\]
Live Indexes

BB:

\%
x2 = \%
x1
\%
x3 = \ldots
add \%
x3, \%
x2

1
2
3
4
5
6

?
Live Indexes

BB:

\[ \%x2 = \%x1 \]
\[ \%x3 = \ldots \]
\[ \text{add} \ \%x3, \ \%x2 \]

[Column of P_i symbols]
Live Indexes

BB:

\[ \%x2 = \%x1 \]
\[ \%x3 = \ldots \]
\[ \text{add } \%x3, \%x2 \]

\[ P_1 < P_2 < P_3 < P_4 < P_5 < P_6 \]
Live Indexes

BB:

... %x2 = %x1
% x3 = ...
add %x3, %x2

P₁ < P₂ < P₃ < P₄ < P₅ < P₆
BB:

\[ \%x2 = \%x1 \]
\[ \%x3 = \ldots \]
\[ \text{add } \%x3, \%x2 \]

\[ P_1 < P_2 < P_3 < P_7 < P_4 < P_5 < P_6 \]
Live Indexes

• unsigned ➔ LiveIndex

• Index Renumbering
Improved Splitting
Improved Splitting

• Break multi-value intervals into component values.
Improved Splitting

• Break multi-value intervals into component values.

• Each value gets a 2nd chance at allocation.
Improved Splitting

• Break multi-value intervals into component values.
• Each value gets a 2nd chance at allocation.
• ... but not a 3rd.
Improved Splitting

• Break multi-value intervals into component values.

• Each value gets a 2nd chance at allocation.

• ... but not a 3rd.

• 13% reduction in static memory references on test case (a pathological SSE kernel).
Better Def/Kill Tracking
Better Def/Kill Tracking

For Values

• Defined by a PHI - Track the def block.
Better Def/Kill Tracking

For Values

• Defined by a PHI - Track the def block.
• Killed by a PHI - Track the appropriate predecessor.
Future Work

• Better value def/kill tracking
• LiveIndex renumbering
• Improved splitting
• Rewrite-in-place
• New allocators
PBQP

Partitioned Boolean Quadratic Problems

- Discrete optimization problems
- NP-complete
- Subclass solvable in linear time
Irregular Architectures

- Multiple register classes.
- Register aliasing.
- Register pairing.
- ...

PBQP Example
PBQP Example
PBQP Example
PBQP Example
PBQP Example
PBQP Example

Solution \([3,2,1]\):
PBQP Example

Solution \([3, 2, 1]\):
PBQP Example

Solution [3,2,1]:
PBQP Example

Solution [3,2,1]:
PBQP Example

Solution [3,2,1]:

Node Costs: 6+0+9 = 15
PBQP Example

Solution $[3,2,1]$:

Node Costs: $6+0+9 = 15$
PBQP Example

Solution [3,2,1]:

Node Costs: 6+0+9 = 15
PBQP Example

Solution [3,2,1]:

Node Costs: 6+0+9 = 15
PBQP Example

Solution [3,2,1]:

Node Costs:  6+0+9 = 15
Edge Costs:  2+6+9 = 17
PBQP Example

Solution [3,2,1]:

Node Costs: 6+0+9 = 15
Edge Costs: 2+6+9 = 17
Total: 32
PBQP Example

Node Costs: 6+0+9 = 15
Edge Costs: 2+6+9 = 17
Total: 32

Solution [3,2,1]:

Solution [1,2,3]: 19
PBQP Example

For Register Allocation:

Nodes represent virtual registers.
Options reflect storage locations.

Option costs:
Typically zero cost for registers, spill cost estimate for stack slot.

Edge costs:
Depends on the constraint.
Example 1

Interference on a Regular Architecture

X

Y
Example 1

Interference on a Regular Architecture

X

Y

Sp
r0
r1

4
0
0
Example 1

Interference on a Regular Architecture

X

Sp
4
0
0
r0
r1

Y

Sp
7
0
0
r0
r1
Example 1

Interference on a Regular Architecture

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sp</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>r0</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>r1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sp</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>r0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>r1</td>
<td>0</td>
<td>∞</td>
</tr>
</tbody>
</table>
Example 1

Interference on a Regular Architecture

\[
\begin{array}{cccc}
\text{Sp} & 4 & 0 & 0 \\
r0 & 0 & 0 & \infty \\
r1 & 0 & 0 & \infty \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{Sp} & 7 & 0 & 0 \\
r0 & 0 & 0 & \infty \\
r1 & 0 & 0 & \infty \\
\end{array}
\]
Example 1

Interference on a Regular Architecture

Cost \((Sp, Sp) = 11\)
Example 1

Interference on a Regular Architecture

Cost \((r0, Sp) = 7\)
Example 1

Interference on a Regular Architecture

\[
\begin{array}{c|c|c}
\text{Sp} & 4 & \text{Sp} \\
\text{r0} & 0 & 0 \\
\text{r1} & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
0 & 0 & 0 \\
0 & \infty & 0 \\
0 & 0 & \infty \\
\end{array}
\]

\[
\begin{array}{c|c|c}
7 & 0 & \text{Sp} \\
0 & 0 & \text{r0} \\
0 & 0 & \text{r1} \\
\end{array}
\]

Cost \((\text{Sp}, \text{r0}) = 4 \)
Example 1

Interference on a Regular Architecture

\[
\begin{array}{ccc}
\text{Sp} & 4 & 0 \\
r0 & 0 & 0 \\
r1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{ccc}
& 0 & 0 \\
0 & 0 & \infty \\
\infty & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{ccc}
& 7 & 0 \\
r0 & 0 & 0 \\
r1 & 0 & 0 \\
\end{array}
\]

Cost \ (r0, \ r1) = 0
Example 1

Interference on a Regular Architecture

\[
\begin{array}{c|c|c|c|c|c|c}
& S_p & r_0 & r_1 & S_p & r_0 & r_1 \\
\hline
S_p & 4 & 0 & 0 & 7 & 0 & 0 \\
r_0 & 0 & 0 \infty 0 & 0 & 0 & 0 \\
r_1 & 0 & 0 & 0 \infty 0 & 0 & 0 \\
\end{array}
\]

\[\text{Cost} \ (r_0, \ r_1) = 0 \]
Example 1

Interference on a Regular Architecture

Cost \((r0, r1) = 0\)
Example 1

Interference on a Regular Architecture

Cost \( (r_0, r_1) = 0 \)
Example 1

Interference on a Regular Architecture

Cost \((r_0, r_0) = \infty\)
Example 1

Interference on a Regular Architecture

\[
\begin{array}{ccc}
\text{Sp} & 4 & 0 & 7 \\
r0 & 0 & 0 \infty & 0 \\
r1 & 0 & 0 & 0
\end{array}
\]

\[\text{Cost} \ (r0, \ r0) = \infty \]

\[\times\]
Example 1

Interference on a Regular Architecture

\[
\begin{array}{c|c|c}
0 & 0 & 0 \\
\hline
0 & \infty & 0 \\
\hline
0 & 0 & \infty \\
\end{array}
\]
Example 2

Interference on an Irregular Architecture

<table>
<thead>
<tr>
<th></th>
<th>Sp</th>
<th>AL</th>
<th>AH</th>
<th>BL</th>
<th>CL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sp</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AX</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BX</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>
Example 3

Coalescing

<table>
<thead>
<tr>
<th></th>
<th>Sp</th>
<th>AX</th>
<th>BX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sp</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AX</td>
<td>0</td>
<td>$-c$</td>
<td>0</td>
</tr>
<tr>
<td>BX</td>
<td>0</td>
<td>0</td>
<td>$-c$</td>
</tr>
</tbody>
</table>
Example 4

Register Pairing \((R_i, R_{i+1})\)

<table>
<thead>
<tr>
<th></th>
<th>Sp</th>
<th>s0</th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sp</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s0</td>
<td>0</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>s1</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>s2</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>
The PBQP Allocator

\[ \text{regalloc} \rightarrow \text{pbqp} \]
\[ \text{solution} = \text{solve pbqp} \]
\[ \text{solution} \rightarrow \text{allocation} \]
The PBQP Allocator

\[
\text{regalloc} \rightarrow \text{pbqp} \\
\text{solution} = \text{solve pbqp} \\
\text{solution} \rightarrow \text{allocation}
\]
The PBQP Allocator

\[
\text{regalloc} \rightarrow \text{pbqp} \\
\text{solution} = \text{solve pbqp} \\
\text{solution} \rightarrow \text{allocation}
\]
The PBQP Allocator

\[\text{regalloc} \rightarrow \text{pbqp}\]
\[\text{solution} = \text{solve pbqp}\]
\[\text{solution} \rightarrow \text{allocation}\]
How does it work?

Solver uses a graph reduction algorithm.

Reduce problem to the empty graph with reduction rules, then reconstruct it.
## PBQP

### PROS

- Ideal for Irregularity
- Very Simple
- Reasonable quality
<table>
<thead>
<tr>
<th>PROS</th>
<th>CONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Ideal for Irregularity</td>
<td>• Slooo0000ow</td>
</tr>
<tr>
<td>• Very Simple</td>
<td></td>
</tr>
<tr>
<td>• Reasonable quality</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROS</th>
<th>CONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Ideal for Irregularity</td>
<td>• Sloooooooow</td>
</tr>
<tr>
<td>• Very Simple</td>
<td></td>
</tr>
<tr>
<td>• Reasonable quality</td>
<td></td>
</tr>
<tr>
<td>• Perfect opportunity</td>
<td></td>
</tr>
<tr>
<td>for a coffee</td>
<td></td>
</tr>
</tbody>
</table>
• Improved Optimizations.
• New Allocators.
• Cleaner Architecture.
• PBQP.
the end.