Pseudo-Array Causalization in the New Backend

Karim Abdelhak, Bernhard Bachmann

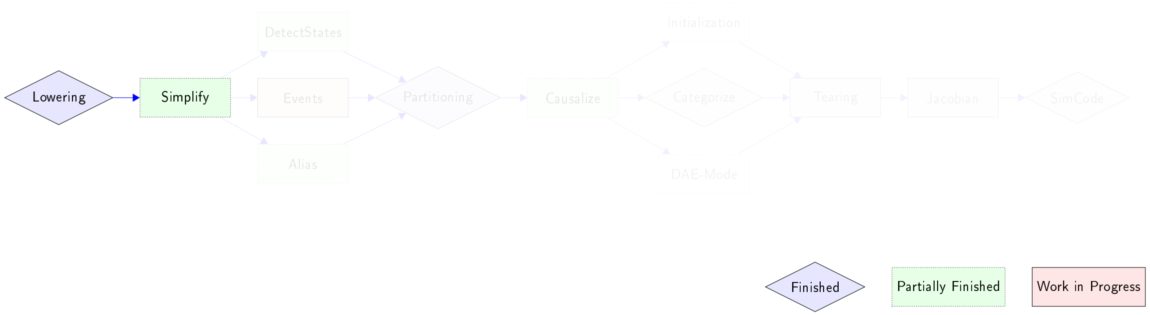
University of Applied Sciences Bielefeld
Bielefeld, Germany

FH Bielefeld
University of
Applied Sciences

1. Overview

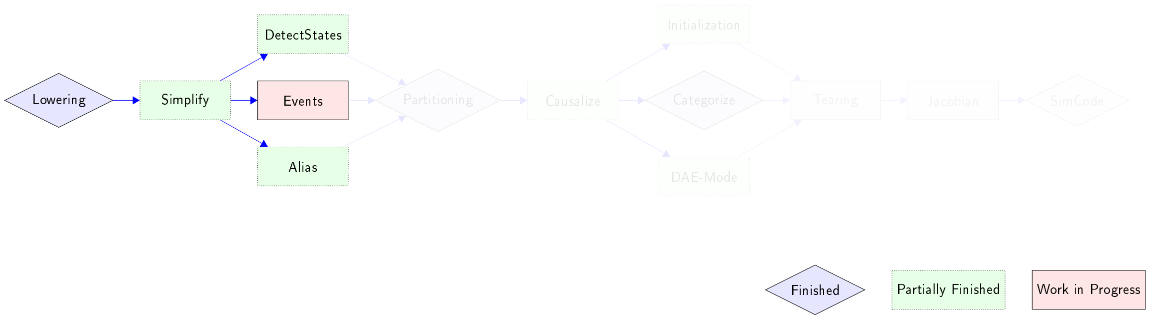
Backend Modules

Status on Array-Handling



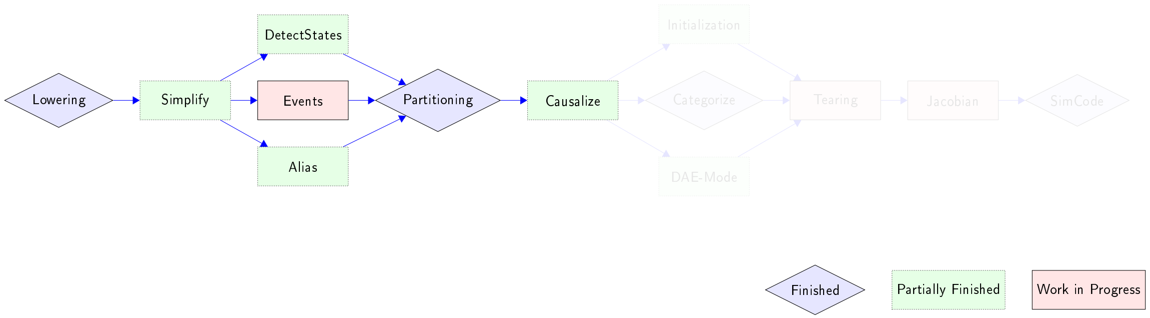
Backend Modules

Status on Array-Handling



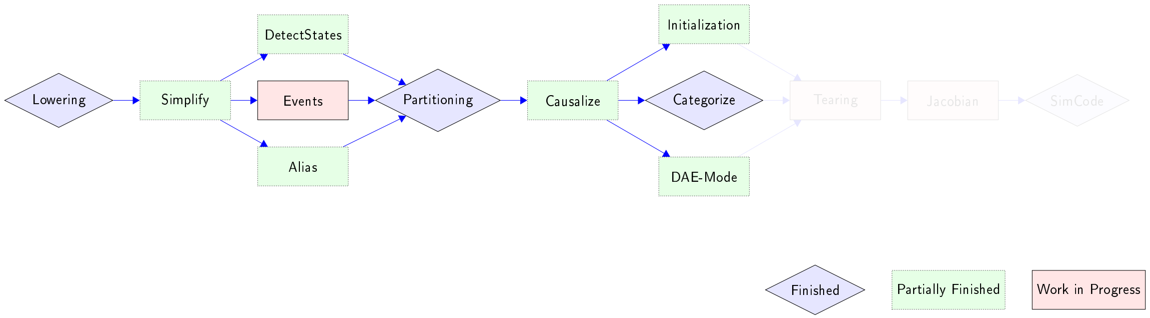
Backend Modules

Status on Array-Handling



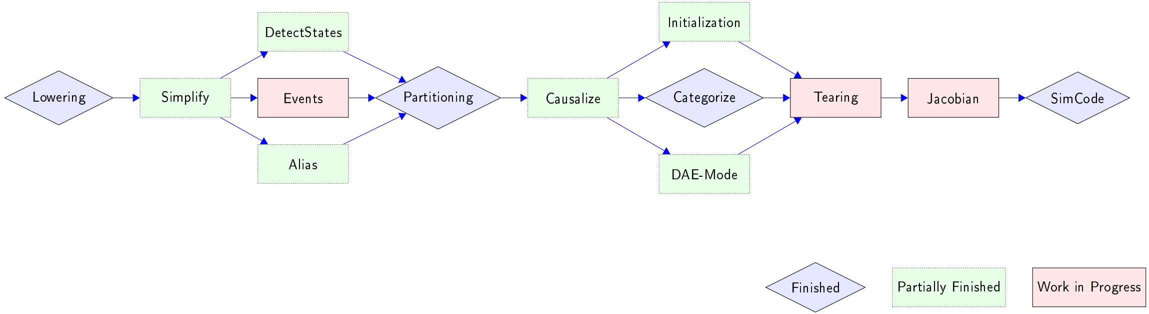
Backend Modules

Status on Array-Handling



Backend Modules

Status on Array-Handling



2. Pseudo-Array Causalization

Pseudo-Array Causalization

Algorithm Description

1. Create index mapping `scalar <-> array` for both variables and equations.
2. Create causalize modes for array equations, to recover array structure after causalization.
3. Create scalar Adjacency Matrix, do Matching and Sorting as if the system was scalarized.
4. Create buckets and fill them with the pseudo scalar equations. This collects all equations that belong to the same array equation and get causalized in the same way.
5. Create the BLT blocks with recovered array structures.
6. Slice the equations according to the bucket information (SimCode).

Pseudo-Array Causalization

Algorithm Description

1. Create index mapping `scalar <-> array` for both variables and equations.
2. Create causalize modes for array equations, to recover array structure after causalization.
3. Create scalar Adjacency Matrix, do Matching and Sorting as if the system was scalarized.
4. Create buckets and fill them with the pseudo scalar equations. This collects all equations that belong to the same array equation and get causalized in the same way.
5. Create the BLT blocks with recovered array structures.
6. Slice the equations according to the bucket information (SimCode).

Pseudo-Array Causalization

Algorithm Description

1. Create index mapping `scalar <-> array` for both variables and equations.
2. Create causalize modes for array equations, to recover array structure after causalization.
3. Create scalar Adjacency Matrix, do Matching and Sorting as if the system was scalarized.
4. Create buckets and fill them with the pseudo scalar equations. This collects all equations that belong to the same array equation and get causalized in the same way.
5. Create the BLT blocks with recovered array structures.
6. Slice the equations according to the bucket information (SimCode).

Pseudo-Array Causalization

Algorithm Description

1. Create index mapping `scalar <-> array` for both variables and equations.
2. Create causalize modes for array equations, to recover array structure after causalization.
3. Create scalar Adjacency Matrix, do Matching and Sorting as if the system was scalarized.
4. Create buckets and fill them with the pseudo scalar equations. This collects all equations that belong to the same array equation and get causalized in the same way.
5. Create the BLT blocks with recovered array structures.
6. Slice the equations according to the bucket information (SimCode).

Pseudo-Array Causalization

Algorithm Description

1. Create index mapping `scalar <-> array` for both variables and equations.
2. Create causalize modes for array equations, to recover array structure after causalization.
3. Create scalar Adjacency Matrix, do Matching and Sorting as if the system was scalarized.
4. Create buckets and fill them with the pseudo scalar equations. This collects all equations that belong to the same array equation and get causalized in the same way.
5. Create the BLT blocks with recovered array structures.
6. Slice the equations according to the bucket information (SimCode).

Pseudo-Array Causalization

Algorithm Description

1. Create index mapping `scalar <-> array` for both variables and equations.
2. Create causalize modes for array equations, to recover array structure after causalization.
3. Create scalar Adjacency Matrix, do Matching and Sorting as if the system was scalarized.
4. Create buckets and fill them with the pseudo scalar equations. This collects all equations that belong to the same array equation and get causalized in the same way.
5. Create the BLT blocks with recovered array structures.
6. Slice the equations according to the bucket information (SimCode).

Pseudo-Array Causalization

Algorithm Description

1. Create index mapping `scalar <-> array` for both variables and equations.
2. Create causalize modes for array equations, to recover array structure after causalization.
3. Create scalar Adjacency Matrix, do Matching and Sorting as if the system was scalarized.
4. Create buckets and fill them with the pseudo scalar equations. This collects all equations that belong to the same array equation and get causalized in the same way.
5. Create the BLT blocks with recovered array structures.
6. Slice the equations according to the bucket information (SimCode).

Structures in the New Backend

Index Mapping

```
record MAPPING
  "scalar <-> array index mapping for variables and equations"
  array<Integer> eqn_StA;
  array<Integer> var_StA;
  array<tuple<Integer,Integer>> eqn_AtS;
  array<tuple<Integer,Integer>> var_AtS;
end MAPPING;
```

Changes compared to the current Backend

1. Permanently store mapping information.
2. Also create a mapping for variables.
3. Save index lists as ranges (always consecutive).
4. Use universal utility functions to create scalar indices.

Structures in the New Backend

Index Mapping

```
record MAPPING
  "scalar <-> array index mapping for variables and equations"
  array<Integer> eqn_StA;
  array<Integer> var_StA;
  array<tuple<Integer,Integer>> eqn_AtS;
  array<tuple<Integer,Integer>> var_AtS;
end MAPPING;
```

Changes compared to the current Backend

1. Permanently store mapping information.
2. Also create a mapping for variables.
3. Save index lists as ranges (always consecutive).
4. Use universal utility functions to create scalar indices.

Structures in the New Backend

Index Mapping

```
record MAPPING
  "scalar <-> array index mapping for variables and equations"
  array<Integer> eqn_StA;
  array<Integer> var_StA;
  array<tuple<Integer,Integer>> eqn_AtS;
  array<tuple<Integer,Integer>> var_AtS;
end MAPPING;
```

Changes compared to the current Backend

1. Permanently store mapping information.
2. Also create a mapping for variables.
3. Save index lists as ranges (always consecutive).
4. Use universal utility functions to create scalar indices.

Structures in the New Backend

Index Mapping

```
record MAPPING
  "scalar <-> array index mapping for variables and equations"
  array<Integer> eqn_StA;
  array<Integer> var_StA;
  array<tuple<Integer,Integer>> eqn_AtS;
  array<tuple<Integer,Integer>> var_AtS;
end MAPPING;
```

Changes compared to the current Backend

1. Permanently store mapping information.
2. Also create a mapping for variables.
3. Save index lists as ranges (always consecutive).
4. Use universal utility functions to create scalar indices.

Structures in the New Backend

Index Mapping

```
record MAPPING
  "scalar <-> array index mapping for variables and equations"
  array<Integer> eqn_StA;
  array<Integer> var_StA;
  array<tuple<Integer,Integer>> eqn_AtS;
  array<tuple<Integer,Integer>> var_AtS;
end MAPPING;
```

Changes compared to the current Backend

1. Permanently store mapping information.
2. Also create a mapping for variables.
3. Save index lists as ranges (always consecutive).
4. Use universal utility functions to create scalar indices.

Structures in the New Backend

Index Mapping

```
record MAPPING
  "scalar <-> array index mapping for variables and equations"
  array<Integer> eqn_StA;
  array<Integer> var_StA;
  array<tuple<Integer,Integer>> eqn_AtS;
  array<tuple<Integer,Integer>> var_AtS;
end MAPPING;
```

Changes compared to the current Backend

1. Permanently store mapping information.
2. Also create a mapping for variables.
3. Save index lists as ranges (always consecutive).
4. Use universal utility functions to create scalar indices.

Structures in the New Backend

Preparing the Slicing of Equations

```
record CAUSALIZE_MODES
  "array equation reconstruction information"
  array<array<Integer>> mode_to_var;
  array<array<ComponentRef>> mode_to_cref;
end CAUSALIZE_MODES;
```

Detecting the different ways an equation can be causalized.

1. Leave entries for scalar equations empty.
2. Create a mode index for each variable instance an array equation can be solved for.
3. Save the variable index to each mode index for each pseudo-scalarized equation (mode_to_var).
4. Save the variable instance to each mode index for each unscalarized equation (mode_to_cref).

Structures in the New Backend

Preparing the Slicing of Equations

```
record CAUSALIZE_MODES
  "array equation reconstruction information"
  array<array<Integer>> mode_to_var;
  array<array<ComponentRef>> mode_to_cref;
end CAUSALIZE_MODES;
```

Detecting the different ways an equation can be causalized.

1. Leave entries for scalar equations empty.
2. Create a mode index for each variable **instance** an array equation can be solved for.
3. Save the variable index to each mode index for each pseudo-scalarized equation (`mode_to_var`).
4. Save the variable **instance** to each mode index for each unscalarized equation (`mode_to_cref`).

Structures in the New Backend

Preparing the Slicing of Equations

```
record CAUSALIZE_MODES
  "array equation reconstruction information"
  array<array<Integer>> mode_to_var;
  array<array<ComponentRef>> mode_to_cref;
end CAUSALIZE_MODES;
```

Detecting the different ways an equation can be causalized.

1. Leave entries for scalar equations empty.
2. Create a mode index for each variable **instance** an array equation can be solved for.
3. Save the variable index to each mode index for each pseudo-scalarized equation (`mode_to_var`).
4. Save the variable **instance** to each mode index for each unscalarized equation (`mode_to_cref`).

Structures in the New Backend

Preparing the Slicing of Equations

```
record CAUSALIZE_MODES
  "array equation reconstruction information"
  array<array<Integer>> mode_to_var;
  array<array<ComponentRef>> mode_to_cref;
end CAUSALIZE_MODES;
```

Detecting the different ways an equation can be causalized.

1. Leave entries for scalar equations empty.
2. Create a mode index for each variable **instance** an array equation can be solved for.
3. Save the variable index to each mode index for each pseudo-scalarized equation (`mode_to_var`).
4. Save the variable **instance** to each mode index for each unscalarized equation (`mode_to_cref`).

Structures in the New Backend

Preparing the Slicing of Equations

```
record CAUSALIZE_MODES
  "array equation reconstruction information"
  array<array<Integer>> mode_to_var;
  array<array<ComponentRef>> mode_to_cref;
end CAUSALIZE_MODES;
```

Detecting the different ways an equation can be causalized.

1. Leave entries for scalar equations empty.
2. Create a mode index for each variable **instance** an array equation can be solved for.
3. Save the variable index to each mode index for each pseudo-scalarized equation (`mode_to_var`).
4. Save the variable **instance** to each mode index for each unscalarized equation (`mode_to_cref`).

Structures in the New Backend

Preparing the Slicing of Equations

```
record CAUSALIZE_MODES
  "array equation reconstruction information"
  array<array<Integer>> mode_to_var;
  array<array<ComponentRef>> mode_to_cref;
end CAUSALIZE_MODES;
```

Detecting the different ways an equation can be causalized.

1. Leave entries for scalar equations empty.
2. Create a mode index for each variable **instance** an array equation can be solved for.
3. Save the variable index to each mode index for each pseudo-scalarized equation (`mode_to_var`).
4. Save the variable **instance** to each mode index for each unscalarized equation (`mode_to_cref`).

Recovering Array Structures

Bucket

```
record PSEUDO_BUCKET_KEY
  Integer eqn_arr_idx;
  Integer mode;
end PSEUDO_BUCKET_KEY;

record PSEUDO_BUCKET_VALUE
  ComponentRef cref_to_solve;
  list<Integer> eqn_scal_indices;
end PSEUDO_BUCKET_VALUE;

record PSEUDO_BUCKET
  "stores information about array equation subsets that get solved in
     the same way"
  UnorderedMap<PseudoBucketKey, PseudoBucketValue> bucket;
  array<Boolean> marks;
end PSEUDO_BUCKET;
```

Recovering Array Structures

Bucket

Collecting Buckets

1. Traverse all strong components after scalar matching and sorting.
2. For each nontrivial strong component of *size* > 1 traverse all scalar equation indices.
3. Use the matching to get the matched scalar variable v for each scalar equation s.
4. Use the *mapping* and scalar equation s to get array equation a.
5. Use the *mode_to_var* and matched variable v to get the causalization mode m.
6. Use the *mode_to_cref* and mode m to get the cref c.
7. If the $key = (a, m)$ exists, add the scalar index s to the list of *value*, otherwise create $value = (c, \{s\})$ and save it for that key.

Recovering Array Structures

Bucket

Collecting Buckets

1. Traverse all strong components after scalar matching and sorting.
2. For each nontrivial strong component of $size > 1$ traverse all scalar equation indices.
3. Use the matching to get the matched scalar variable v for each scalar equation s.
4. Use the *mapping* and scalar equation s to get array equation a.
5. Use the *mode_to_var* and matched variable v to get the causalization mode m.
6. Use the *mode_to_cref* and mode m to get the cref c.
7. If the $key = (a, m)$ exists, add the scalar index s to the list of *value*, otherwise create $value = (c, \{s\})$ and save it for that key.

Recovering Array Structures

Bucket

Collecting Buckets

1. Traverse all strong components after scalar matching and sorting.
2. For each nontrivial strong component of *size* > 1 traverse all scalar equation indices.
3. Use the matching to get the matched scalar variable v for each scalar equation s.
4. Use the *mapping* and scalar equation s to get array equation a.
5. Use the *mode_to_var* and matched variable v to get the causalization mode m.
6. Use the *mode_to_cref* and mode m to get the cref c.
7. If the $key = (a, m)$ exists, add the scalar index s to the list of $value$, otherwise create $value = (c, \{s\})$ and save it for that key.

Recovering Array Structures

Bucket

Collecting Buckets

1. Traverse all strong components after scalar matching and sorting.
2. For each nontrivial strong component of *size* > 1 traverse all scalar equation indices.
3. Use the matching to get the matched scalar variable v for each scalar equation s.
4. Use the *mapping* and scalar equation s to get array equation a.
5. Use the *mode_to_var* and matched variable v to get the causalization mode m.
6. Use the *mode_to_cref* and mode m to get the cref c.
7. If the $key = (a, m)$ exists, add the scalar index s to the list of *value*, otherwise create $value = (c, \{s\})$ and save it for that key.

Recovering Array Structures

Bucket

Collecting Buckets

1. Traverse all strong components after scalar matching and sorting.
2. For each nontrivial strong component of $size > 1$ traverse all scalar equation indices.
3. Use the matching to get the matched scalar variable v for each scalar equation s.
4. Use the *mapping* and scalar equation s to get array equation a.
5. Use the $mode\_to\_var$ and matched variable v to get the causalization mode m.
6. Use the $mode\_to\_cref$ and mode m to get the cref c.
7. If the $key = (a, m)$ exists, add the scalar index s to the list of $value$, otherwise create $value = (c, \{s\})$ and save it for that key.

Recovering Array Structures

Bucket

Collecting Buckets

1. Traverse all strong components after scalar matching and sorting.
2. For each nontrivial strong component of $size > 1$ traverse all scalar equation indices.
3. Use the matching to get the matched scalar variable v for each scalar equation s.
4. Use the *mapping* and scalar equation s to get array equation a.
5. Use the *mode_to_var* and matched variable v to get the causalization mode m.
6. Use the *mode_to_cref* and mode m to get the cref c.
7. If the $key = (a, m)$ exists, add the scalar index s to the list of *value*, otherwise create $value = (c, \{s\})$ and save it for that key.

Recovering Array Structures

Bucket

Collecting Buckets

1. Traverse all strong components after scalar matching and sorting.
2. For each nontrivial strong component of *size* > 1 traverse all scalar equation indices.
3. Use the matching to get the matched scalar variable v for each scalar equation s.
4. Use the *mapping* and scalar equation s to get array equation a.
5. Use the *mode_to_var* and matched variable v to get the causalization mode m.
6. Use the *mode_to_cref* and mode m to get the cref c.
7. If the *key* $= (a, m)$ exists, add the scalar index s to the list of *value*, otherwise create *value* $= (c, \{s\})$ and save it for that key.

Recovering Array Structures

Bucket

Collecting Buckets

1. Traverse all strong components after scalar matching and sorting.
2. For each nontrivial strong component of *size* > 1 traverse all scalar equation indices.
3. Use the matching to get the matched scalar variable v for each scalar equation s.
4. Use the *mapping* and scalar equation s to get array equation a.
5. Use the *mode_to_var* and matched variable v to get the causalization mode m.
6. Use the *mode_to_cref* and mode m to get the cref c.
7. If the *key* $= (a, m)$ exists, add the scalar index s to the list of *value*, otherwise create *value* $= (c, \{s\})$ and save it for that key.

Recovering Array Structures

Slicing

Slicing Operations

1. Order the nested loop iterators properly.
2. Reduce the start and stop of the ranges according to the sliced subsets.
3. Fix the step of the ranges according to the sliced subsets.
4. Solve the body for the specified variable **instance** (cref).

```
BLOCK 3: Sliced Equation
________________________________________
### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {i in 1:4, j in 1:2} loop
        [----]   [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

Recovering Array Structures

Slicing

Slicing Operations

1. Order the nested loop iterators properly.
2. Reduce the `start` and `stop` of the ranges according to the sliced subsets.
3. Fix the `step` of the ranges according to the sliced subsets.
4. Solve the body for the specified variable **instance** (cref).

```
BLOCK 3: Sliced Equation
________________________________________
### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {i in 1:4, j in 1:2} loop
        [----]   [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

Recovering Array Structures

Slicing

Slicing Operations

1. Order the nested loop iterators properly.
2. Reduce the start and stop of the ranges according to the sliced subsets.
3. Fix the step of the ranges according to the sliced subsets.
4. Solve the body for the specified variable **instance** (cref).

```
BLOCK 3: Sliced Equation
-----------------------------------------
### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {j in 1:2, i in 1:4} loop
        [----]   [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

Recovering Array Structures

Slicing

Slicing Operations

1. Order the nested loop iterators properly.
2. Reduce the `start` and `stop` of the ranges according to the sliced subsets.
3. Fix the `step` of the ranges according to the sliced subsets.
4. Solve the body for the specified variable **instance** (cref).

```
BLOCK 3: Sliced Equation
________________________________________
### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {j in 1:2, i in 1:4} loop
        [----]   [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

Recovering Array Structures

Slicing

Slicing Operations

1. Order the nested loop iterators properly.
2. Reduce the `start` and `stop` of the ranges according to the sliced subsets.
3. Fix the `step` of the ranges according to the sliced subsets.
4. Solve the body for the specified variable **instance** (cref).

```
BLOCK 3: Sliced Equation
________________________________________
### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {j in 1:2, i in 1:2} loop
        [----]   [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

Recovering Array Structures

Slicing

Slicing Operations

1. Order the nested loop iterators properly.
2. Reduce the `start` and `stop` of the ranges according to the sliced subsets.
3. Fix the `step` of the ranges according to the sliced subsets.
4. Solve the body for the specified variable **instance** (cref).

```
BLOCK 3: Sliced Equation
_______________________________________
### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {j in 1:2, i in 1:2} loop
        [----]   [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

Recovering Array Structures

Slicing

Slicing Operations

1. Order the nested loop iterators properly.
2. Reduce the `start` and `stop` of the ranges according to the sliced subsets.
3. Fix the `step` of the ranges according to the sliced subsets.
4. Solve the body for the specified variable **instance** (cref).

```
BLOCK 3: Sliced Equation
________________________________________
### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {j in 1:2, i in 2:(-1):1} loop
        [----]   [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

Recovering Array Structures

Slicing

Slicing Operations

1. Order the nested loop iterators properly.
2. Reduce the `start` and `stop` of the ranges according to the sliced subsets.
3. Fix the `step` of the ranges according to the sliced subsets.
4. Solve the body for the specified variable **instance** (cref).

```
BLOCK 3: Sliced Equation
----------------------------------------
### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {j in 1:2, i in 2:(-1):1} loop
        [----]   [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

Recovering Array Structures

Slicing

Slicing Operations

1. Order the nested loop iterators properly.
2. Reduce the `start` and `stop` of the ranges according to the sliced subsets.
3. Fix the `step` of the ranges according to the sliced subsets.
4. Solve the body for the specified variable **instance** (cref).

```
BLOCK 3: Sliced Equation
__________________________________________

### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {j in 1:2, i in 2:(-1):1} loop
        [----]   [SCAL] (1) x[i, j] = x[i + 1, j] + y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

3. Exemplary Model

Examplary Model

Modelica Model

```
model exemplary
  Real x[5,2];
  Real y[2];
equation
  for i in 1:4, j in 1:2 loop
    x[i+1,j] = x[i,j] - y[j];
  end for;
  for j in 1:2 loop
    y[j] = j*sin(time);
    x[3,j] = j*cos(time);
  end for;
end exemplary;
```

Expected Results

1. y and $x[3,:]$ can be solved first, ordering does not matter.
2. $x[i+1,j]$ for $i >= 3$ has to be solved after 1) in ascending order for i.
3. $x[i,j]$ for $i < 3$ has to be solved after 1) in descending order for i.
4. Ordering for j does not matter. Nesting ordering of i and j does not matter.

Examplary Model

Modelica Model

```
model exemplary
  Real x[5,2];
  Real y[2];
equation
  for i in 1:4, j in 1:2 loop
    x[i+1,j] = x[i,j] - y[j];
  end for;
  for j in 1:2 loop
    y[j] = j*sin(time);
    x[3,j] = j*cos(time);
  end for;
end exemplary;
```

Expected Results

1. y and $x[3,:]$ can be solved first, ordering does not matter.
2. $x[i+1,j]$ for $i >= 3$ has to be solved after 1) in ascending order for i.
3. $x[i,j]$ for $i < 3$ has to be solved after 1) in descending order for i.
4. Ordering for j does not matter. Nesting ordering of i and j does not matter.

Examplary Model

Modelica Model

```
model exemplary
  Real x[5,2];
  Real y[2];
equation
  for i in 1:4, j in 1:2 loop
    x[i+1,j] = x[i,j] - y[j];
  end for;
  for j in 1:2 loop
    y[j] = j*sin(time);
    x[3,j] = j*cos(time);
  end for;
end exemplary;
```

Expected Results

1. y and $x[3,:]$ can be solved first, ordering does not matter.
2. $x[i+1,j]$ for $i >= 3$ has to be solved after 1) in ascending order for i.
3. $x[i,j]$ for $i < 3$ has to be solved after 1) in descending order for i.
4. Ordering for j does not matter. Nesting ordering of i and j does not matter.

Examplary Model

Modelica Model

```
model exemplary
  Real x[5,2];
  Real y[2];
equation
  for i in 1:4, j in 1:2 loop
    x[i+1,j] = x[i,j] - y[j];
  end for;
  for j in 1:2 loop
    y[j] = j*sin(time);
    x[3,j] = j*cos(time);
  end for;
end exemplary;
```

Expected Results

1. y and $x[3,:]$ can be solved first, ordering does not matter.
2. $x[i+1,j]$ for $i >= 3$ has to be solved after 1) in ascending order for i.
3. $x[i,j]$ for $i < 3$ has to be solved after 1) in descending order for i.
4. Ordering for j does not matter. Nesting ordering of i and j does not matter.

Examplary Model

Modelica Model

```
model exemplary
  Real x[5,2];
  Real y[2];
equation
  for i in 1:4, j in 1:2 loop
    x[i+1,j] = x[i,j] - y[j];
  end for;
  for j in 1:2 loop
    y[j] = j*sin(time);
    x[3,j] = j*cos(time);
  end for;
end exemplary;
```

Expected Results

1. y and $x[3,:]$ can be solved first, ordering does not matter.
2. $x[i+1,j]$ for $i >= 3$ has to be solved after 1) in ascending order for i.
3. $x[i,j]$ for $i < 3$ has to be solved after 1) in descending order for i.
4. Ordering for j does not matter. Nesting ordering of i and j does not matter.

Examplary Model

Modelica Model

```
model exemplary
  Real x[5,2];
  Real y[2];
equation
  for i in 1:4, j in 1:2 loop
    x[i+1,j] = x[i,j] - y[j];
  end for;
  for j in 1:2 loop
    y[j] = j*sin(time);
    x[3,j] = j*cos(time);
  end for;
end exemplary;
```

Expected Results

1. y and $x[3,:]$ can be solved first, ordering does not matter.
2. $x[i+1,j]$ for $i >= 3$ has to be solved after 1) in ascending order for i.
3. $x[i,j]$ for $i < 3$ has to be solved after 1) in descending order for i.
4. Ordering for j does not matter. Nesting ordering of i and j does not matter.

Examplary Model

Block Lower Triangular Transformation (Unsolved)

```
BLOCK 1: Sliced Equation
________________________________________
### Variable:
        y[j]
### Equation:
        [FOR-] (2)($RES_SIM_1)
        [----] for j in 1:2 loop
        [----]   [SCAL] (1) y[j] = CAST(Real, j) * sin(time);
        [----] end for;
    with slices: {0, 1}
```

Examplary Model

Block Lower Triangular Transformation (Unsolved)

```
BLOCK 2: Sliced Equation
________________________________________
### Variable:
        x[3, j]
### Equation:
        [FOR-] (2)($RES_SIM_0)
        [----] for j in 1:2 loop
        [----]   [SCAL] (1) x[3, j] = CAST(Real, j) * cos(time);
        [----] end for;
    with slices: {0, 1}
```

Examplary Model

Block Lower Triangular Transformation (Unsolved)

```
BLOCK 3: Sliced Equation
________________________________________
### Variable:
        x[i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {i in 1:4, j in 1:2} loop
        [----]     [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {2, 0, 3, 1}
```

Examplary Model

Block Lower Triangular Transformation (Unsolved)

```
BLOCK 4: Sliced Equation
________________________________________
### Variable:
        x[1 + i, j]
### Equation:
        [FOR-] (8)($RES_SIM_2)
        [----] for {i in 1:4, j in 1:2} loop
        [----]    [SCAL] (1) x[i + 1, j] = x[i, j] - y[j];
        [----] end for;
    with slices: {4, 5, 6, 7}
```

Examplary Model

SimCode Equations (Solved)

```
for j in 1:2 loop
  y[j] := CAST(Real, j) * sin(time);
end for;

for j in 1:2 loop
  x[3, j] := CAST(Real, j) * cos(time);
end for;

for j in 1:1:2 loop
  for i in 2:(-1):1 loop
    x[i, j] := x[1 + i, j] + y[j];
  end for;
end for;

for i in 3:1:4 loop
  for j in 1:1:2 loop
    x[1 + i, j] := -(y[j] - x[i, j]);
  end for;
end for;
```

Examplary Model

SimCode Equations (Solved)

```
for j in 1:2 loop
  y[j] := CAST(Real, j) * sin(time);
end for;

for j in 1:2 loop
  x[3, j] := CAST(Real, j) * cos(time);
end for;

for j in 1:1:2 loop
  for i in 2:(-1):1 loop
    x[i, j] := x[1 + i, j] + y[j];
  end for;
end for;

for i in 3:1:4 loop
  for j in 1:1:2 loop
    x[1 + i, j] := -(y[j] - x[i, j]);
  end for;
end for;
```

Examplary Model

SimCode Equations (Solved)

```
for j in 1:2 loop
  y[j] := CAST(Real, j) * sin(time);
end for;

for j in 1:2 loop
  x[3, j] := CAST(Real, j) * cos(time);
end for;

for j in 1:1:2 loop
  for i in 2:(-1):1 loop
    x[i, j] := x[1 + i, j] + y[j];
  end for;
end for;

for i in 3:1:4 loop
  for j in 1:1:2 loop
    x[1 + i, j] := -(y[j] - x[i, j]);
  end for;
end for;
```

Examplary Model

SimCode Equations (Solved)

```
for j in 1:2 loop
  y[j] := CAST(Real, j) * sin(time);
end for;

for j in 1:2 loop
  x[3, j] := CAST(Real, j) * cos(time);
end for;

for j in 1:1:2 loop
  for i in 2:(-1):1 loop
    x[i, j] := x[1 + i, j] + y[j];
  end for;
end for;

for i in 3:1:4 loop
  for j in 1:1:2 loop
    x[1 + i, j] := -(y[j] - x[i, j]);
  end for;
end for;
```

4. Time Comparison

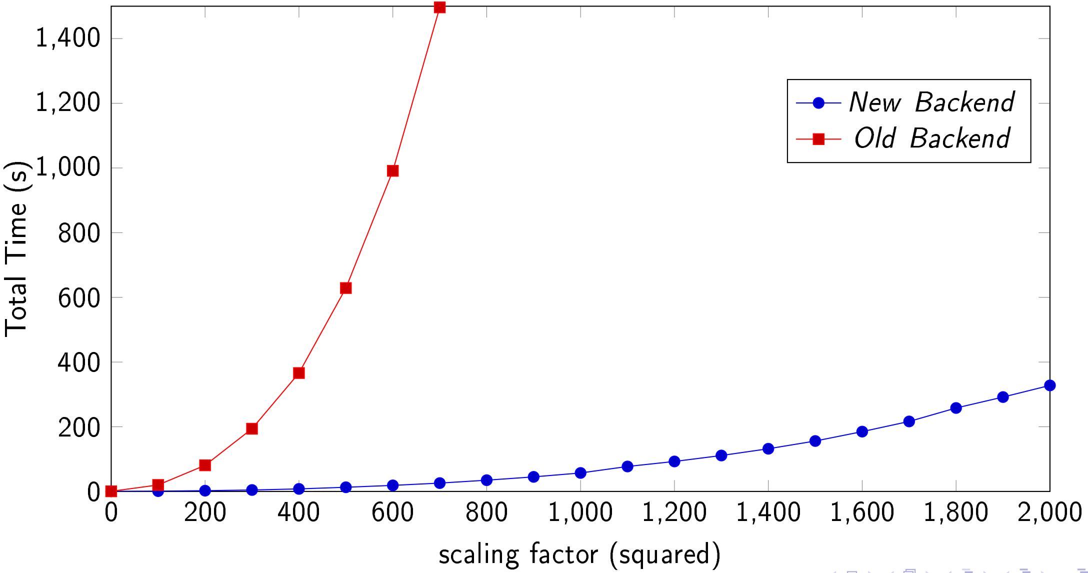
Scaled Examplary Model

Modelica Model

```
model exemplaryS
  parameter Integer s = 0;
  Real x[5+s,2+s];
  Real y[2+s];
equation
  for i in 1:4+s, j in 1:2+s loop
    x[i+1,j] = x[i,j] - y[j];
  end for;
  for j in 1:2+s loop
    y[j] = j*sin(time);
    x[3,j] = j*cos(time);
  end for;
end exemplaryS;
```

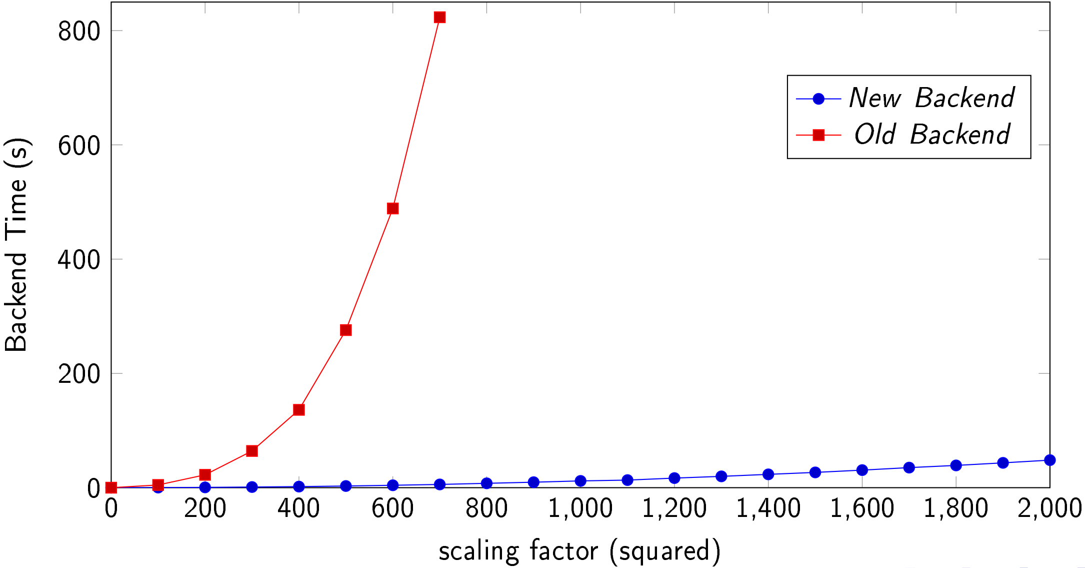
Scaled Examplary Model

Comparison - Total Time



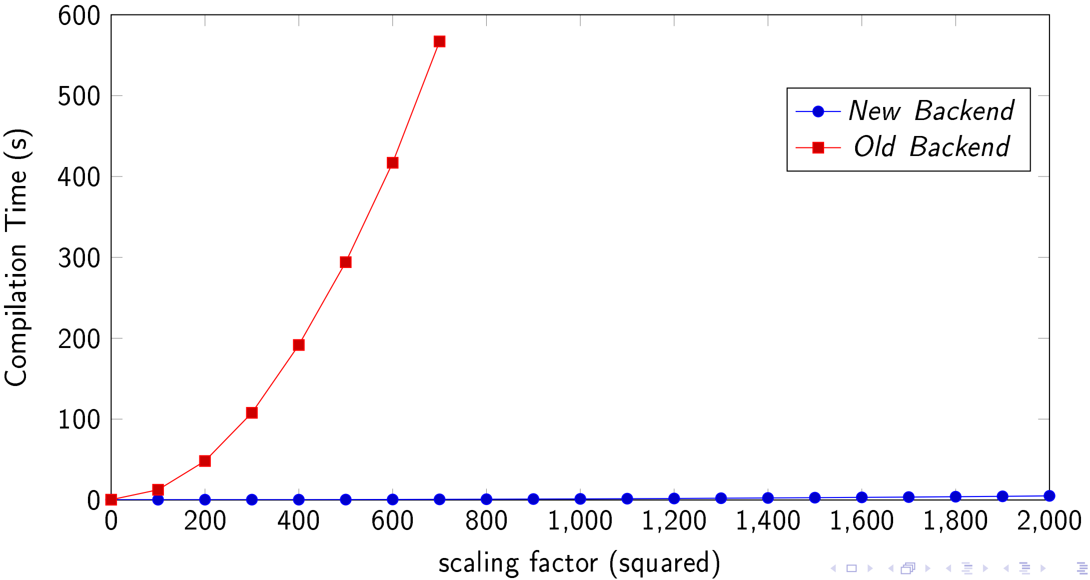
Scaled Examplary Model

Comparison - Backend Time



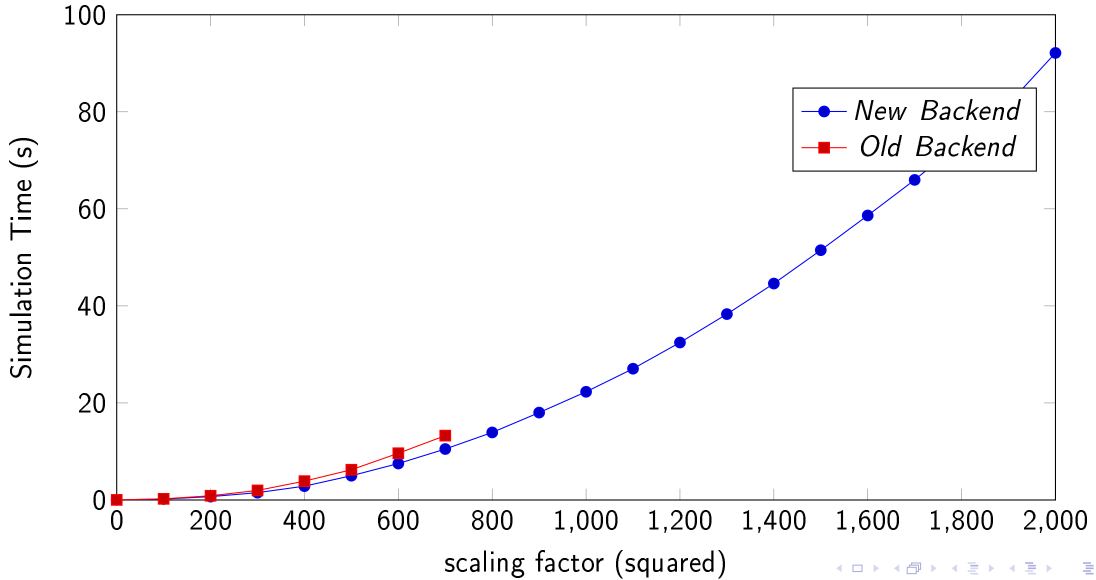
Scaled Examplary Model

Comparison - Compilation Time



Scaled Examplary Model

Comparison - Simulation Time



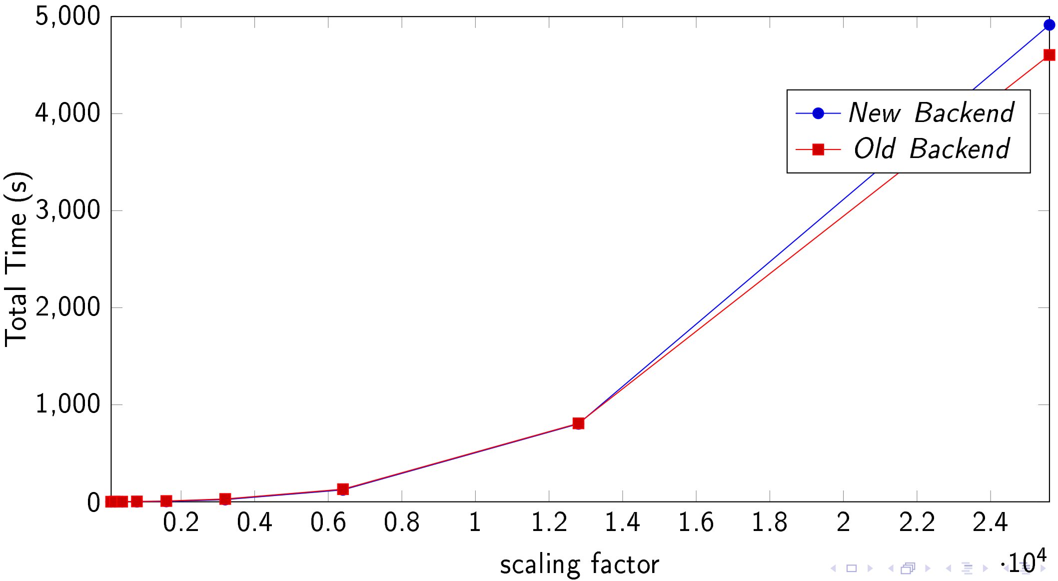
ScalableTestSuite

Modelica Model

```
model CascadedFirstOrder
  "N cascaded first order systems, approximating a pure delay"
  parameter Integer N = 10 "Order of the system";
  parameter Modelica.SIunits.Time T = 1 "System delay";
  final parameter Modelica.SIunits.Time tau = T/N "Individual time
     constant";
  Real x[N] (each start = 0, each fixed = true);
equation
  tau*der(x[1]) = 1 - x[1];
  for i in 2:N loop
    tau*der(x[i]) = x[i-1] - x[i];
  end for;
end CascadedFirstOrder;
```

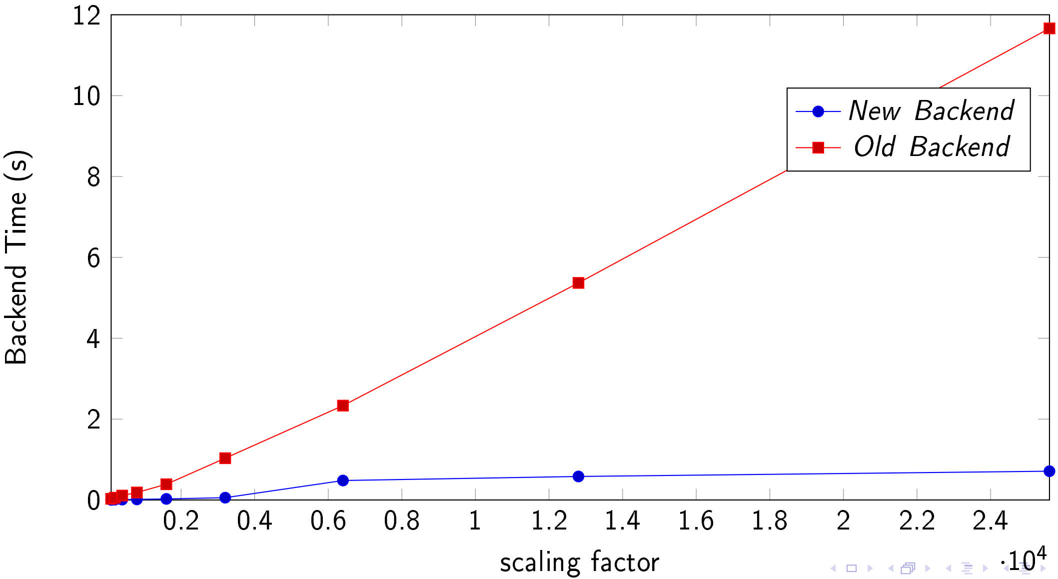
ScalableTestSuite

Comparison - Total Time



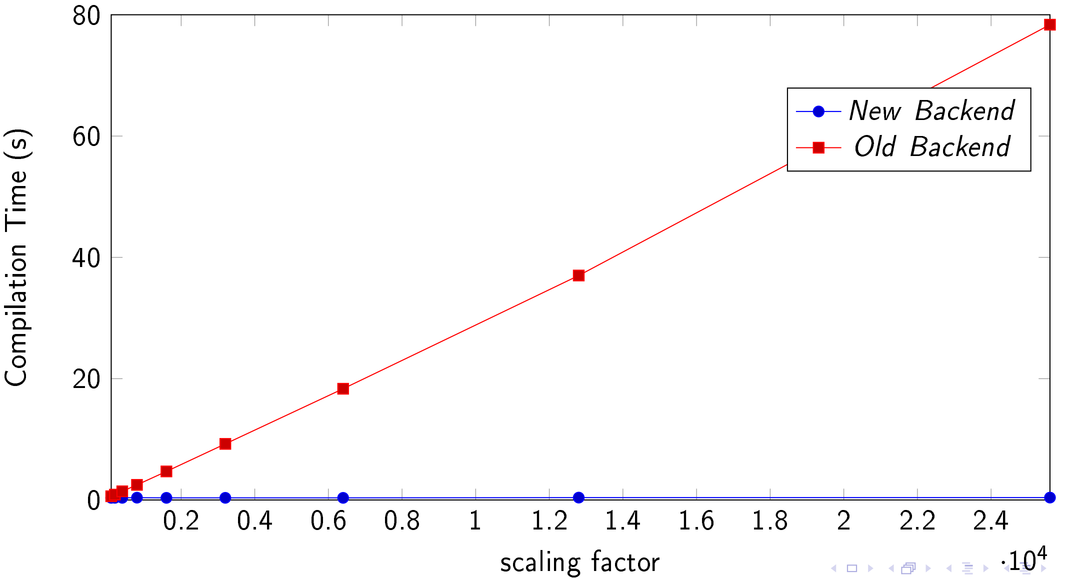
ScalableTestSuite

Comparison - Backend Time



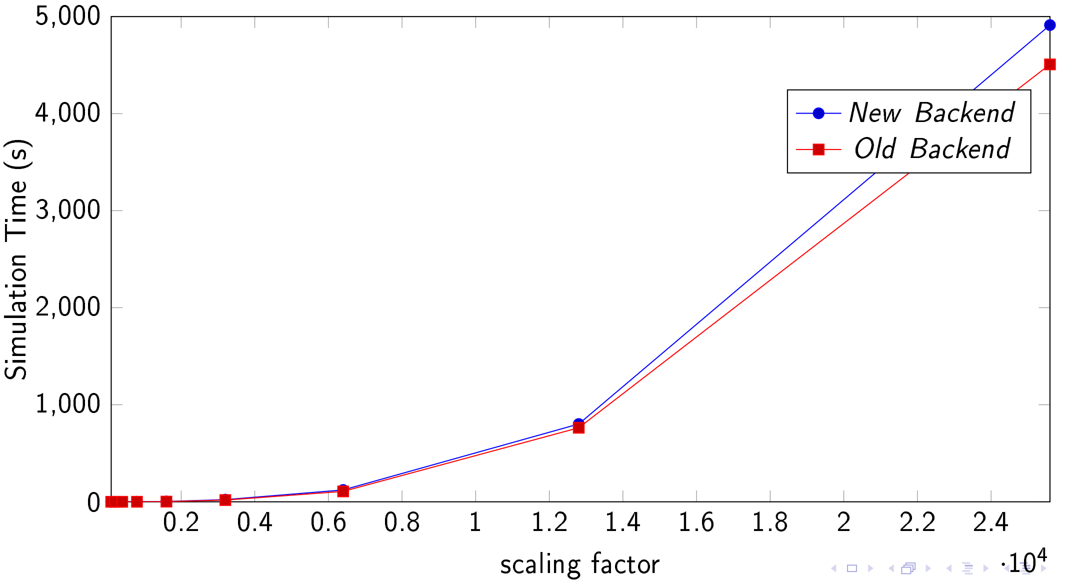
ScalableTestSuite

Comparison - Compilation Time



ScalableTestSuite

Comparison - Simulation Time



5. Summary

Summary

Recent Development

- Pseodo-Array matching and sorting for basic slices,
- Pseodo-Array handling for essential backend modules.

Current Development

- Pseudo-Array jacobians and sparsity pattern,
- Pseodo-Array handling for optimization modules,
- Mixed-kind variables,
- Better memory management.

Upcoming Plans

- Algorithm handling and function inlining,
- Pseudo-Array Index Reduction.

Summary

Recent Development

- Pseodo-Array matching and sorting for basic slices,
- Pseodo-Array handling for essential backend modules.

Current Development

- Pseudo-Array jacobians and sparsity pattern,
- Pseodo-Array handling for optimization modules,
- Mixed-kind variables,
- Better memory management.

Upcoming Plans

- Algorithm handling and function inlining,
- Pseudo-Array Index Reduction.

Summary

Recent Development

- Pseodo-Array matching and sorting for basic slices,
- Pseodo-Array handling for essential backend modules.

Current Development

- Pseudo-Array jacobians and sparsity pattern,
- Pseodo-Array handling for optimization modules,
- Mixed-kind variables,
- Better memory management.

Upcoming Plans

- Algorithm handling and function inlining,
- Pseudo-Array Index Reduction.

Summary

Thank you for your attention!