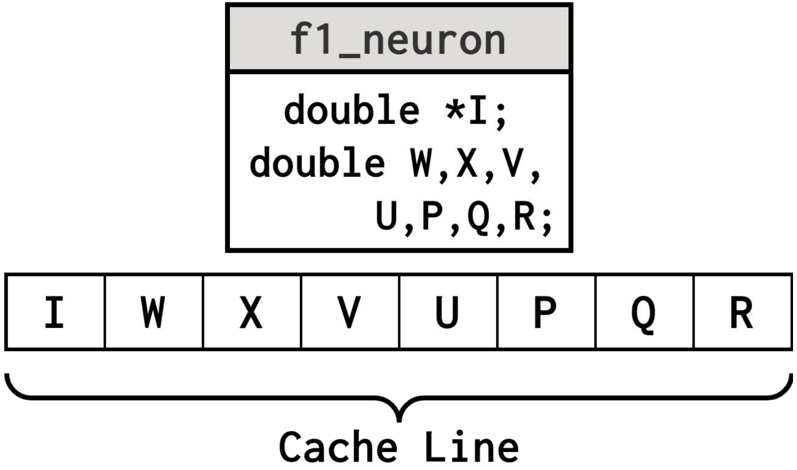
Region-Based Data Layout via Data Reuse Analysis

Caio S. Rohwedder
João P. L. De Carvalho - J. Nelson Amaral

Data Layout Transformations

Aggregate Types

```c
typedef struct {
    double *I; double W; double X; double V;
    double U; double P; double Q; double R;
} f1_neuron;
```



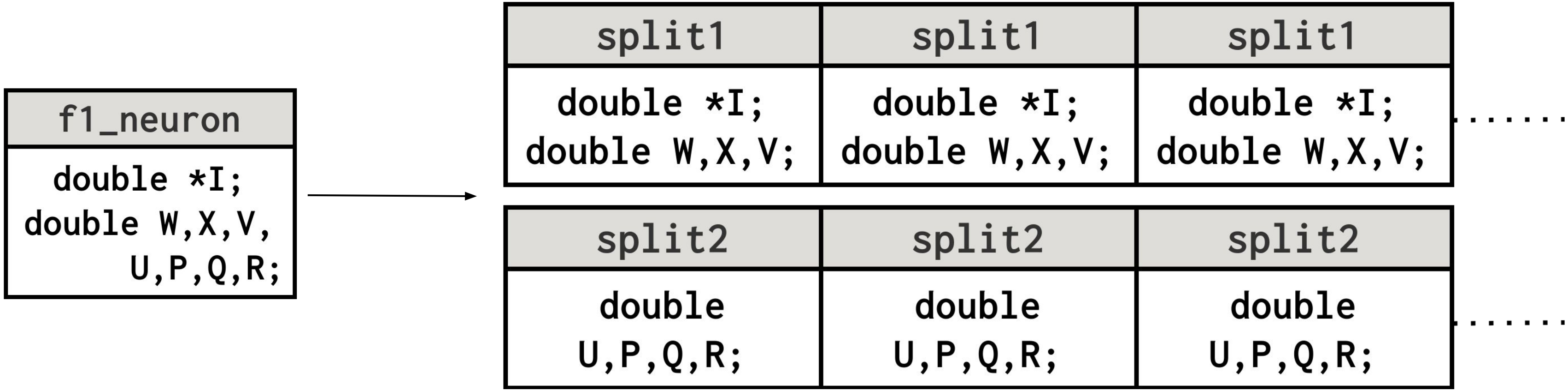
Array of Aggregate Types

f1_neuron	f1_neuron	f1_neuron	
double *I; double W,X,V, U,P,Q,R;	double *I; double W,X,V, U,P,Q,R;	double *I; double W,X,V, U,P,Q,R;	....

Array of Aggregate Types - Actual Use

f1_neuron	f1_neuron	f1_neuron
double *I; double W,X,V, U,P,Q,R;	double *I; double W,X,V, U,P,Q,R;	double *I; double W,X,V, U,P,Q,R;

Structure Splitting



Defining the Data Layout

- Observe the program
  - Profiling
  - Static analysis
- Use a heuristic

Defining the Data Layout

- Observe the program
  - Profiling
  - Static analysis
- Use a heuristic

Structure Splitting Heuristics

- Field affinity

group1	group1	group1
double *I;	double *I;	double *I;

group2	group2	group2
double W,X,V, U,P,Q,R;	double W,X,V, U,P,Q,R;	double W,X,V, U,P,Q,R;

There are problems.

There are problems:

- Whole-program scope
  - No single optimal layout
  - All mallocs and type references must be updated
    - Hard to enforce legality
      - C/C++ languages

There are problems:

- Whole-program scope
  - No single optimal layout
  - All mallocs and type references must be updated
    - Hard to enforce legality
      - C/C++ languages

There are problems:

- Whole-program scope
  - No single optimal layout
  - All mallocs and type references must be updated
    - Hard to enforce legality
      - C/C++ languages

There are problems:

- Require runtime knowledge
  - Expensive profiling

Mannarswamy et al. (2009) introduced the region-based approach

We propose:

Region-based data layout transformations in regions that present data reuse

We propose:

Region-based data layout transformations in regions that present data reuse

RebaseDL:
Region-Based Data Layout

RebaseDL:

- Defines layout statically
  - No profiling

RebaseDL:

- Simplifies enforcing legality
    - Only within regions

RebaseDL:

- Uses copying
  - Additional overhead
  - Offset by data reuse

RebaseDL:

- Also considers packing
    - Not limited to aggregate types

The Transformation

Snippet from 179.art

```
typedef struct {
    double *I; double W; double X; double V;
    double U; double P; double Q; double R;
} f1_neuron;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Candidates

Pairs → [Loop] - [Memory Range]

Candidates

Pairs → [Loop] - [Memory Range]

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Candidates

Pairs → [**Loop**] - [Memory Range]

- Single-entry single-exit (SESE) **region**

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Candidates

Pairs → [Loop] - [**Memory Range**]

- Base pointers
- Alias analysis
- Code versioning

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Candidates

Pairs → [Loop] - **[Memory Range]**

- Base pointers
- Alias analysis
- Code versioning

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Candidates

Pairs → [Loop] - **[Memory Range]**

- Base pointers
- Alias analysis
- Code versioning

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Candidates

Pairs → [Loop] - [**Memory Range**]

- Base pointers
- Alias analysis
- Code versioning

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

```
typedef struct {
    double *I; double W; double X; double V;
    double U; double P; double Q; double R;
} f1_neuron;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

```
typedef struct {
    double *I; double W; double X; double V;
    double U; double P; double Q; double R;
} f1_neuron;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
typedef struct {
  double P;
} f1_neuron_split;
```

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
typedef struct {
  double P;
} f1_neuron_split;
```

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
typedef struct {
  double P;
} f1_neuron_split;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
typedef struct {
  double P;
} f1_neuron_split;

f1_neuron_split *f1_layer_split = (f1_neuron_split *)
        malloc(numf1s * sizeof(f1_neuron_split));

for (int i = 0; i < numf1s; i++)
  f1_layer_split[i].P = f1_layer[i].P;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
typedef struct {
  double P;
} f1_neuron_split;

f1_neuron_split *f1_layer_split = (f1_neuron_split *)
        malloc(numf1s * sizeof(f1_neuron_split));

for (int i = 0; i < numf1s; i++)
  f1_layer_split[i].P = f1_layer[i].P;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
typedef struct {
  double P;
} f1_neuron_split;

f1_neuron_split *f1_layer_split = (f1_neuron_split *)
        malloc(numf1s * sizeof(f1_neuron_split));

for (int i = 0; i < numf1s; i++)
  f1_layer_split[i].P = f1_layer[i].P;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer_split[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
typedef struct {
  double P;
} f1_neuron_split;

f1_neuron_split *f1_layer_split = (f1_neuron_split *)
        malloc(numf1s * sizeof(f1_neuron_split));

for (int i = 0; i < numf1s; i++)
  f1_layer_split[i].P = f1_layer[i].P;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer_split[ti].P * bus[ti][tj];
}
```

Transformation Example

Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
typedef struct {
  double P;
} f1_neuron_split;

f1_neuron_split *f1_layer_split = (f1_neuron_split *)
        malloc(numf1s * sizeof(f1_neuron_split));

for (int i = 0; i < numf1s; i++)
  f1_layer_split[i].P = f1_layer[i].P;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer_split[ti].P * bus[ti][tj];
}

free(f1_layer_split);
```

Transformation Example

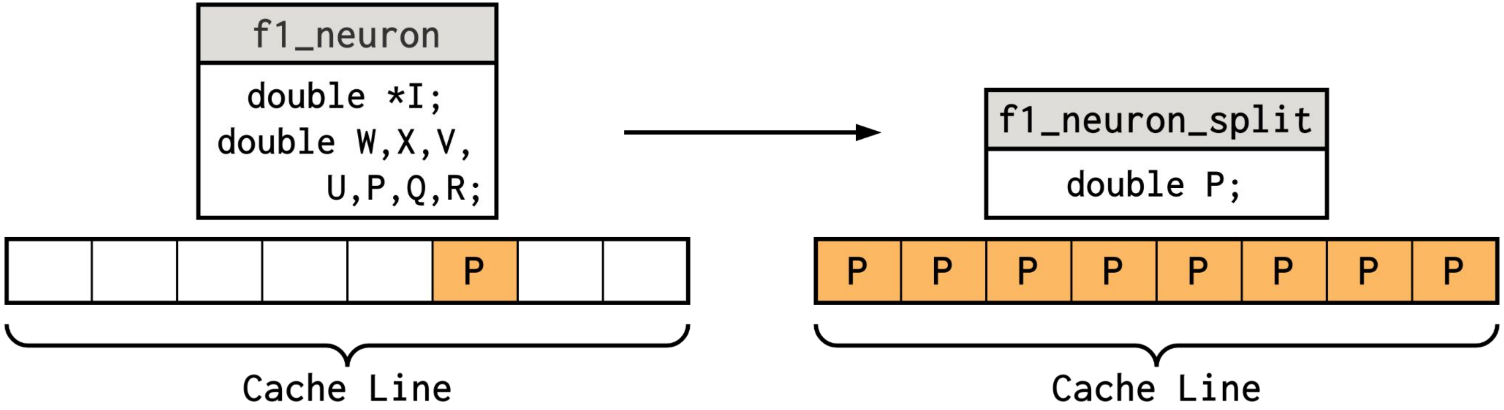
Transform **f1_layer** in **loop_tj**:

1. Create new type
2. Reorder fields
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
typedef struct {
  double P;
} f1_neuron_split;

f1_neuron_split *f1_layer_split = (f1_neuron_split *)
        malloc(numf1s * sizeof(f1_neuron_split));

for (int i = 0; i < numf1s; i++)
  f1_layer_split[i].P = f1_layer[i].P;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer_split[ti].P * bus[ti][tj];
}

free(f1_layer_split);
```

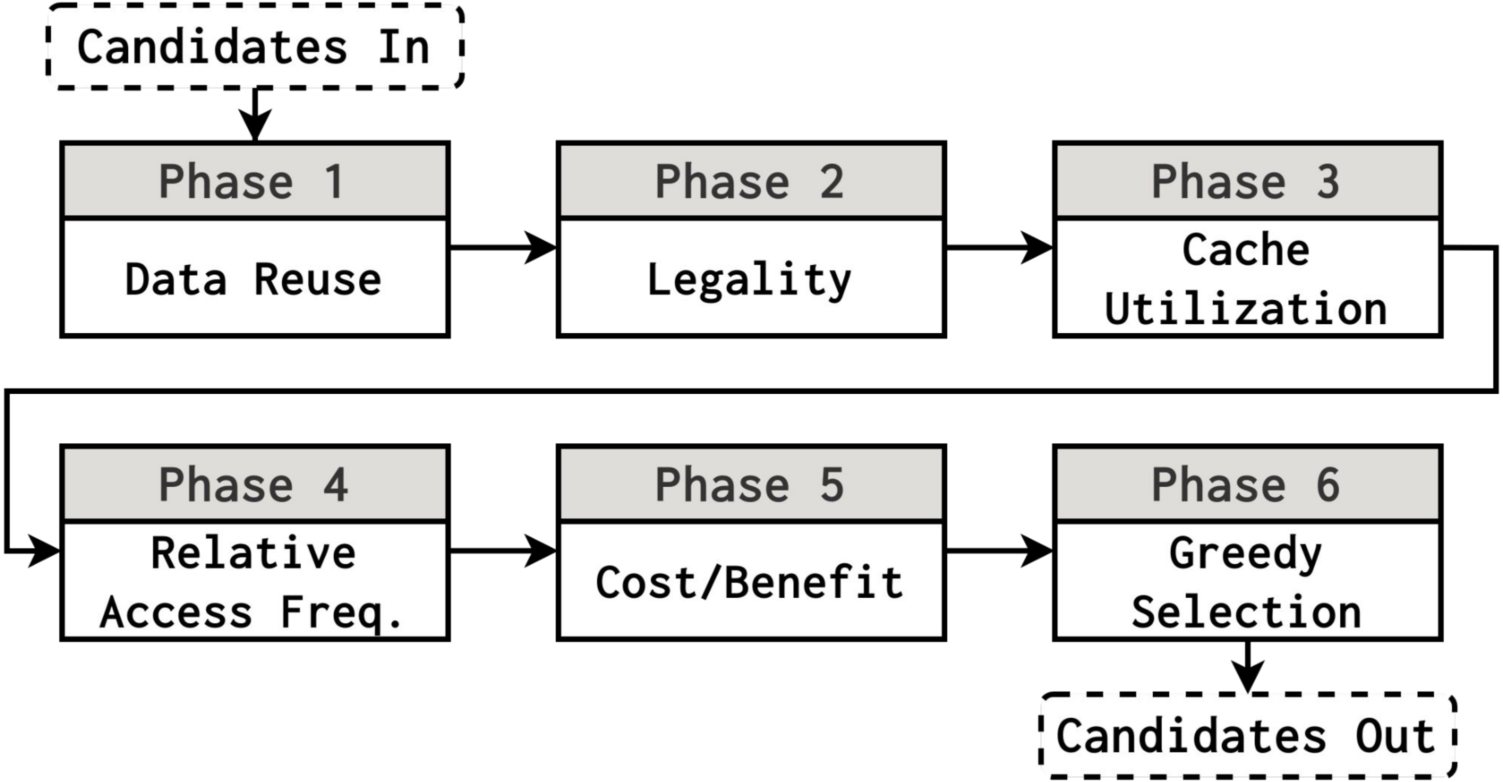
Transformation Example

As a result:



RebaseDL: The Analysis

Overview



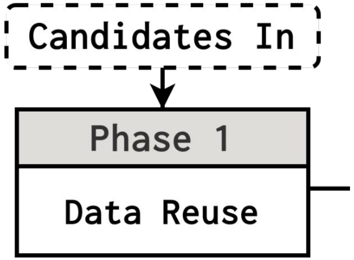
Candidates In
Phase 1
Data Reuse
Phase 2
Legality
Phase 3
Cache Utilization
Phase 4
Relative Access Freq.
Phase 5
Cost/Benefit
Phase 6
Greedy Selection
Candidates Out

Candidates In

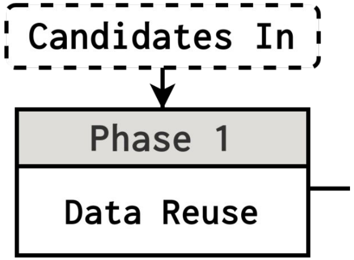
- For a loop nest
  - All loop - memory range pairs

Data Reuse

- Mitigates copy overhead
- Candidate must reuse their memory range
  - Otherwise, it is eliminated

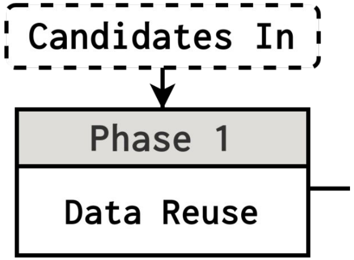


Data Reuse

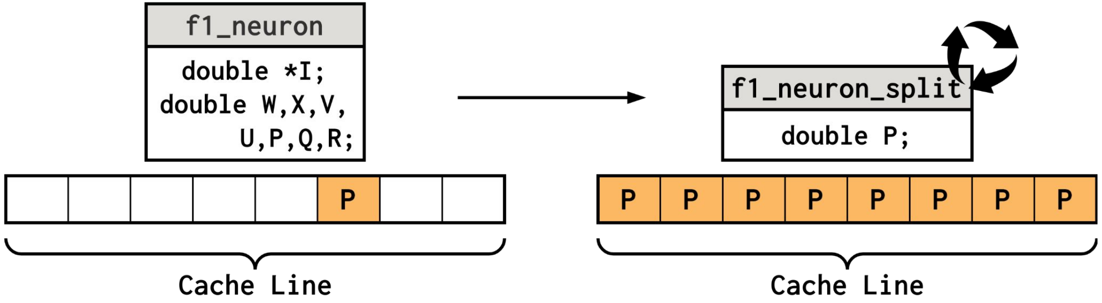


- Mitigates copy overhead
- Candidate must reuse their memory range
    - Otherwise, it is eliminated

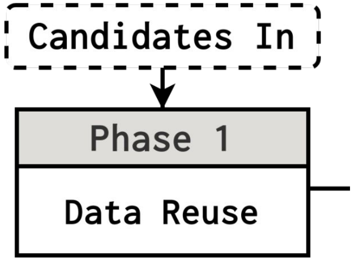
Data Reuse



- Mitigates copy overhead
- Candidate must reuse their memory range
  - Otherwise, it is eliminated



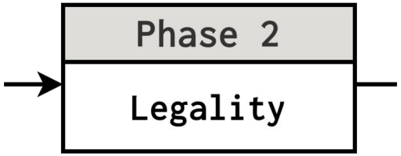
Data Reuse



- Mitigates copy overhead
- Candidate must reuse their memory range
    - Otherwise, it is eliminated

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Legality



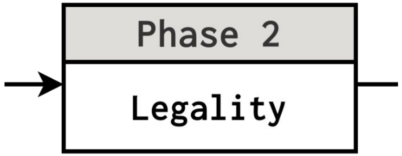
Inside the candidate's loop, verify:

- Intraprocedural
- No global references
- Known loop bounds
- Single base pointer

Legality

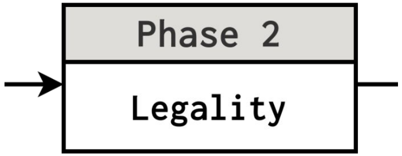
Inside the candidate's loop, verify:

- Intraprocedural
- No global references
- Known loop bounds
- Single base pointer




```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++) {
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
      foo(f1_layer);
    }
}
```

Legality

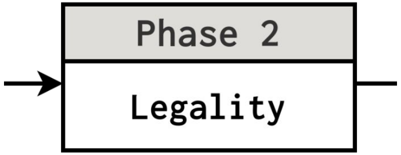


Inside the candidate's loop, verify:

- Intraprocedural
- No global references
- Known loop bounds
- Single base pointer

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++) {
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
      foo();
    }
}
```

Legality



Inside the candidate's loop, verify:

- Intraprocedural
- No global references
- Known loop bounds
- Single base pointer

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Legality

Phase 2

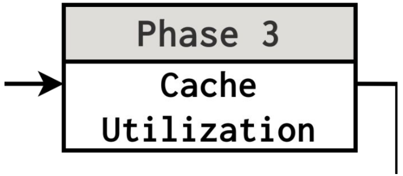
Legality

Inside the candidate's loop, verify:

- Intraprocedural
- No global references
- Known loop bounds
- Single base pointer

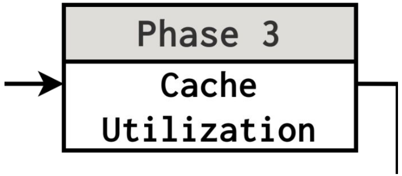
```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    f1_layer = f2_layer;
  for (ti = 0; ti < numf1s; ti++) {
    Y[tj].y += f1_layer[ti].P * bus[ti][tj];
  }
}
```

Cache Utilization



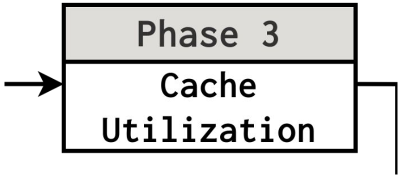
Amount of data used /
Amount of data brought to cache
by a candidate

Cache Utilization



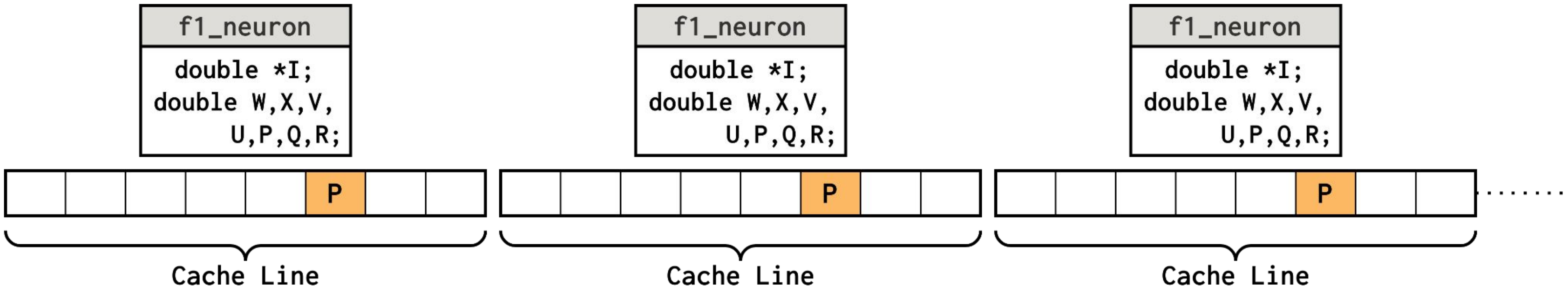
Amount of data used /
Amount of data brought to cache
by a candidate

Cache Utilization

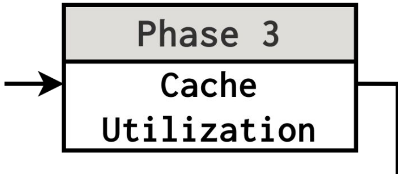


Amount of data used /
Amount of data brought to cache
by a candidate

- 8 bytes / 64 bytes = 0.125



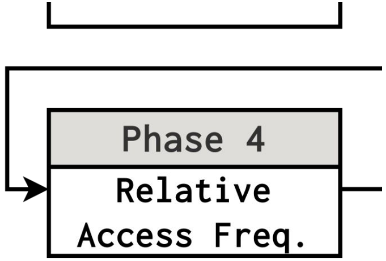
Cache Utilization



Amount of data used /
Amount of data brought to cache
by a candidate

- Candidates must have a low cache utilization

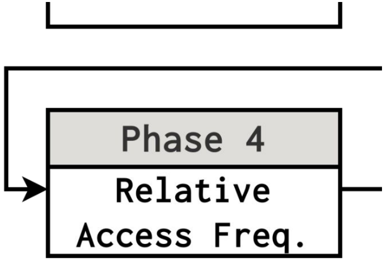
Relative Access Frequency



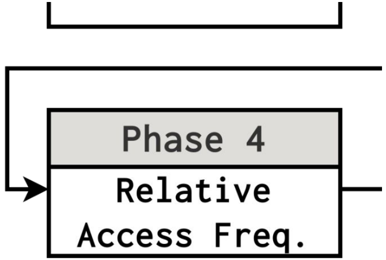
Freq. of candidate's accesses relative to freq. of the target loop

Relative Access Frequency

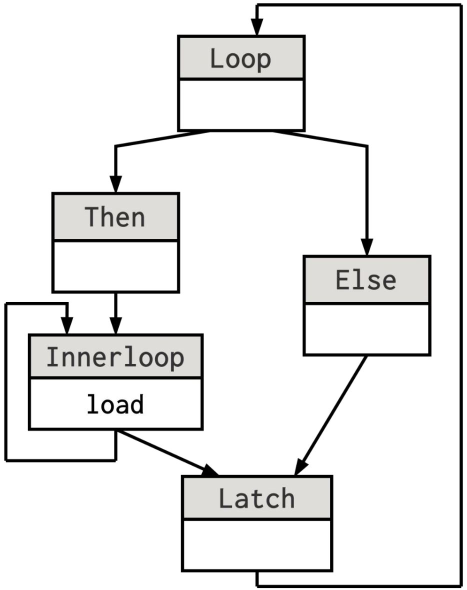
Freq. of candidate's accesses relative to freq. of the target loop



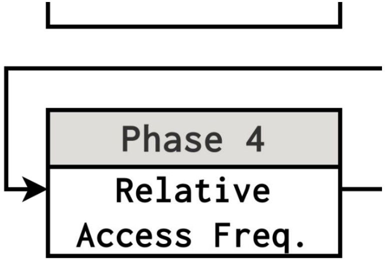
Relative Access Frequency



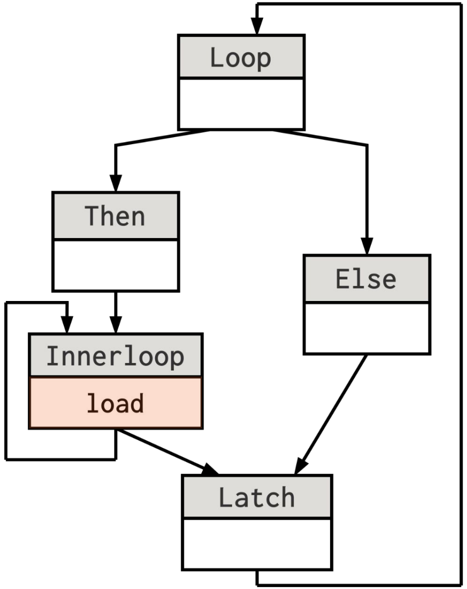
Freq. of candidate's accesses relative to freq. of the target loop



Relative Access Frequency

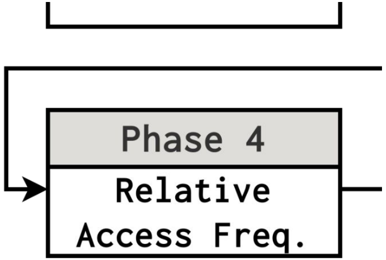


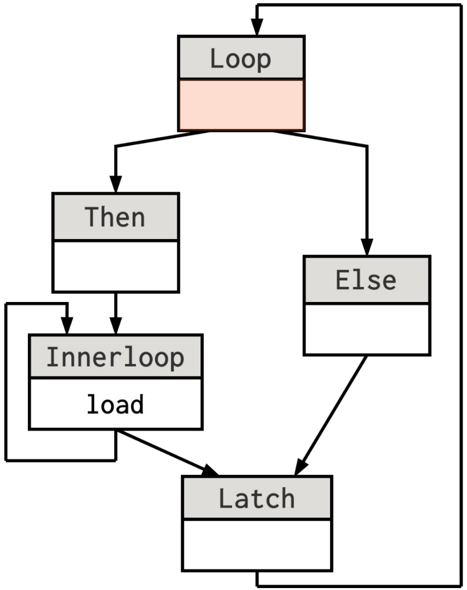
Freq. of candidate's **accesses**
relative to freq. of the target loop



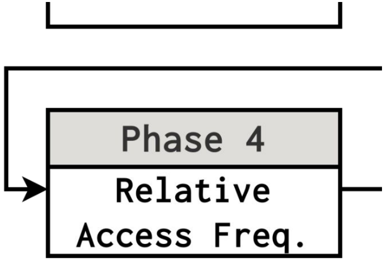
Relative Access Frequency

Freq. of candidate's accesses relative to freq. of the **target loop**



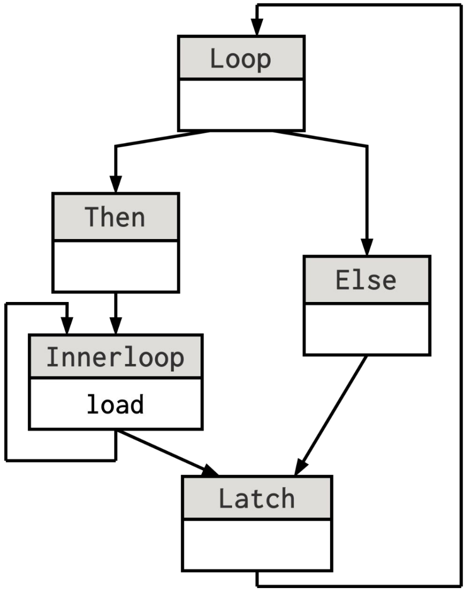


Relative Access Frequency

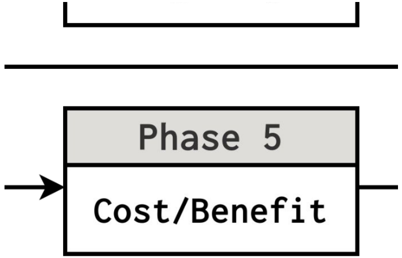


Freq. of candidate's accesses relative to freq. of the target loop

- Candidate must not have a low access frequency



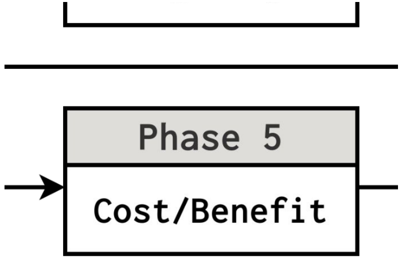
Cost/Benefit



A score for candidates.

- Cost/Benefit must be low

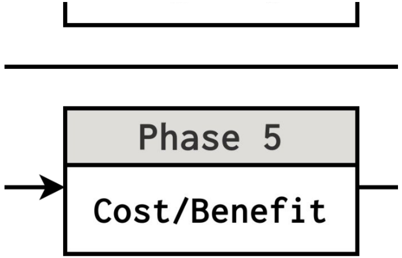
Cost/Benefit



Candidate benefit:

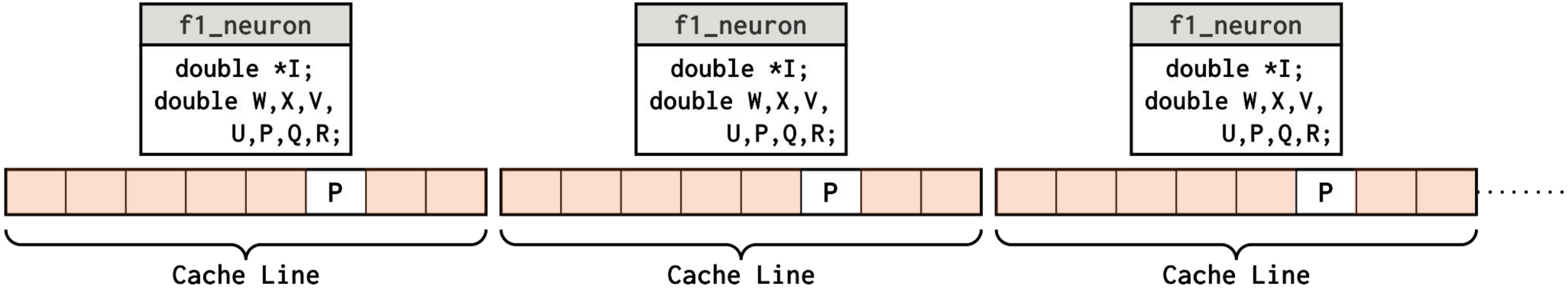
- Reduction in unused data loaded to cache
- Multiplied by the trip count of the target loop

Cost/Benefit

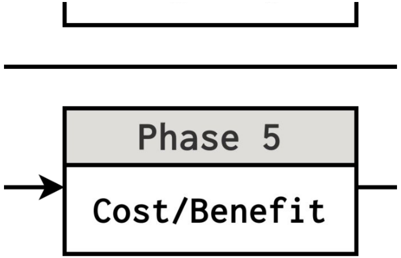


Candidate benefit:

- Reduction in unused data loaded to cache
- Multiplied by the trip count of the target loop



Cost/Benefit

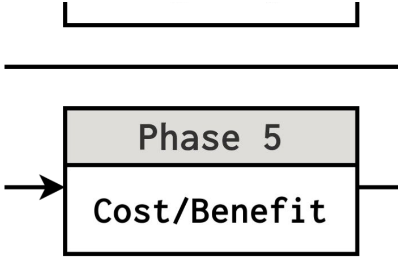


Candidate benefit:

- Reduction in unused data loaded to cache
- Multiplied by the trip count of the target loop

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

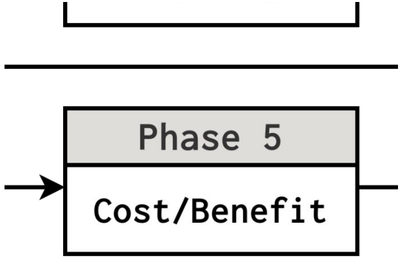
Cost/Benefit



Candidate cost:

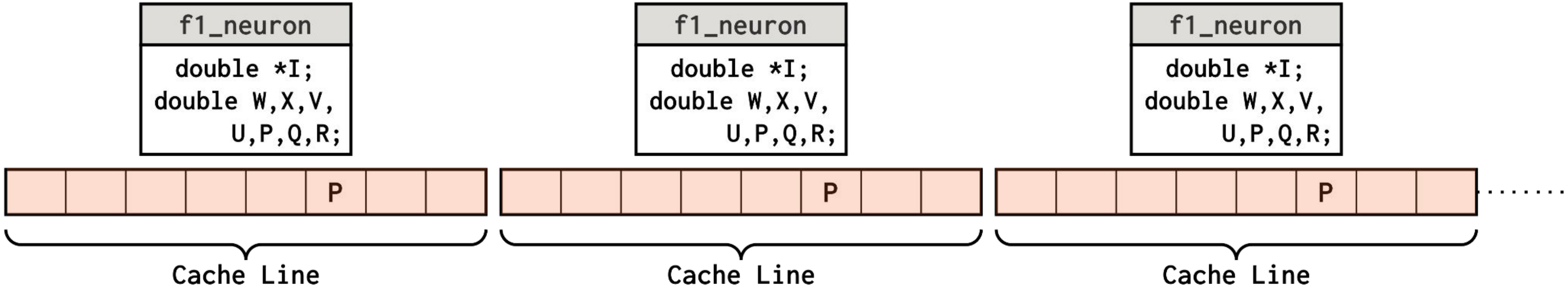
- Data brough to cache to create copy
- Multiplied by 2 if the copying back is needed

Cost/Benefit

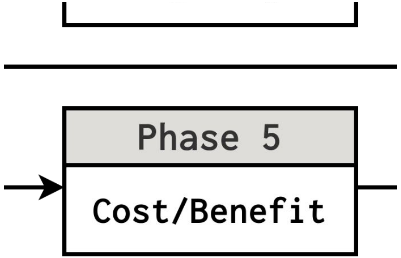


Candidate cost:

- Data brough to cache to create copy
- Multiplied by 2 if the copying back is needed

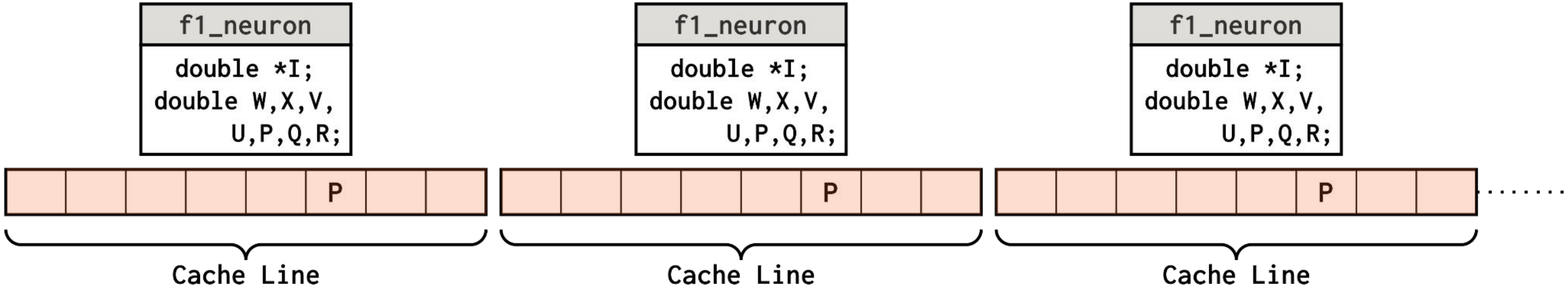


Cost/Benefit

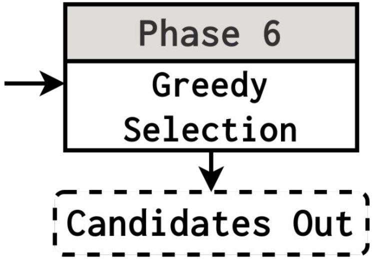


Candidate cost:

- Data brough to cache to create copy
- Multiplied by 2 if the copying back is needed

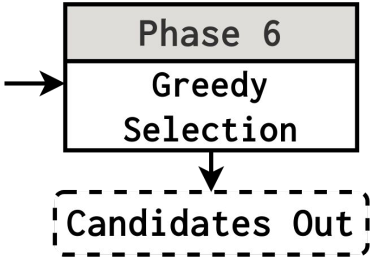


Greedy Selection



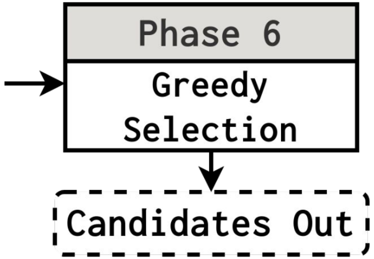
- Sorts candidates based on cost/benefit
  - Break ties with target loop depth
- Selects candidates → Candidates Out
  - Avoiding conflicting candidates

Greedy Selection



- Sorts candidates based on cost/benefit
  - Break ties with target loop depth
- Selects candidates → **Candidates Out**
  - Avoiding conflicting candidates

Greedy Selection



- Sorts candidates based on cost/benefit
  - Break ties with target loop depth
- Selects candidates → Candidates Out
  - Avoiding conflicting candidates

```
for (tk = 0; tk < numf3s; tk++) {
  for (tj = 0; tj < numf2s; tj++) {
    Y[tj].y = 0;
    if (!Y[tj].reset)
      for (ti = 0; ti < numf1s; ti++)
        Y[tj].y += f1_layer[ti].P * bus[ti][tj];
  }
}
```

RebaseDL in LLVM

```
➜ opt -load-pass-plugin libRebaseDLPass.so -passes=rebasedl input.ll
```

Artifact Available

https://doi.org/10.5281/zenodo.10457086

Evaluation

Analysis Results

Total of 71 candidates in SPEC CPU benchmark suites

Benchmark	Candidates
175.vpr	1
179.art	2
445.gobmk	1
447.dealII	18
464.h264ref	1

Benchmark	Candidates
502.gcc_r	3
510.parest_r	42
525.x264_r	1
526.blender_r	1
538.imagick_r	1

Data Reuse Impact Evaluation

- Data reuse affects the transformation’s performance?

Data Reuse Impact Evaluation

- Data reuse affects the transformation's performance?

Best performing region

Benchmark	Candidates
175.vpr	1
179.art	2
445.gobmk	1
447.dealII	18
464.h264ref	1

Benchmark	Candidates
502.gcc_r	3
510.parest_r	42
525.x264_r	1
526.blender_r	1
538.imagick_r	1

Data Reuse Impact Evaluation

- Data reuse affects the transformation's performance?

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Data Reuse Impact Evaluation

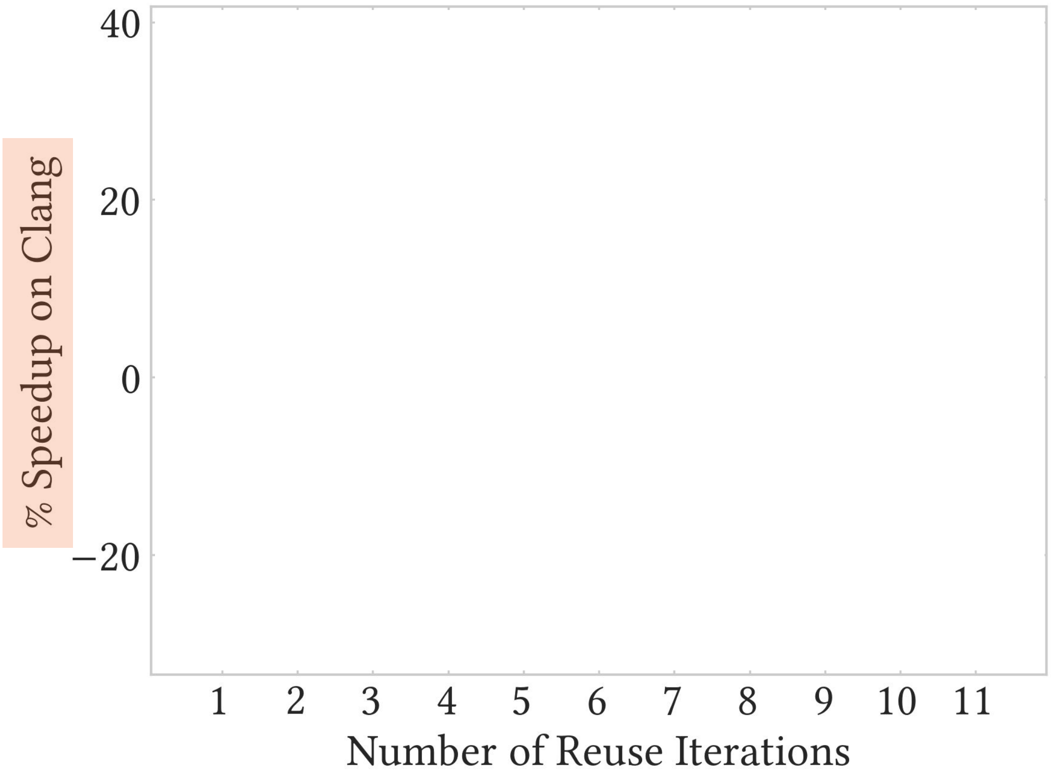
- Data reuse affects the transformation's performance?

- Keep the same behaviour
- Change `numf2s`

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

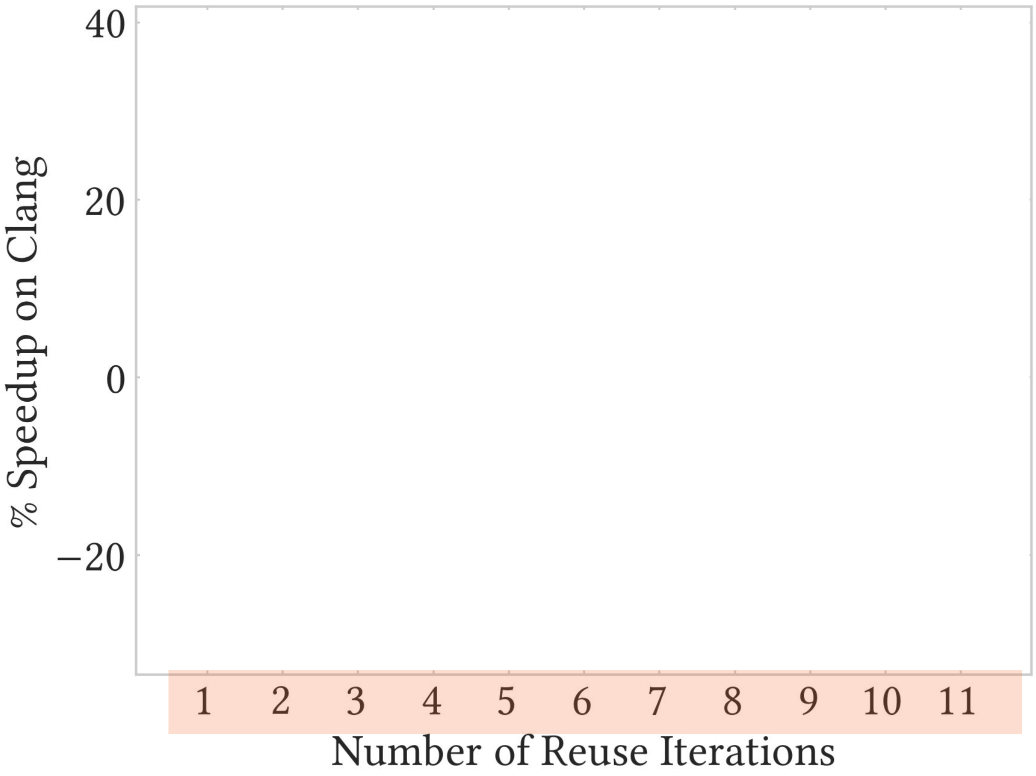
Data Reuse Impact Evaluation

- Data reuse affects the transformation's performance?



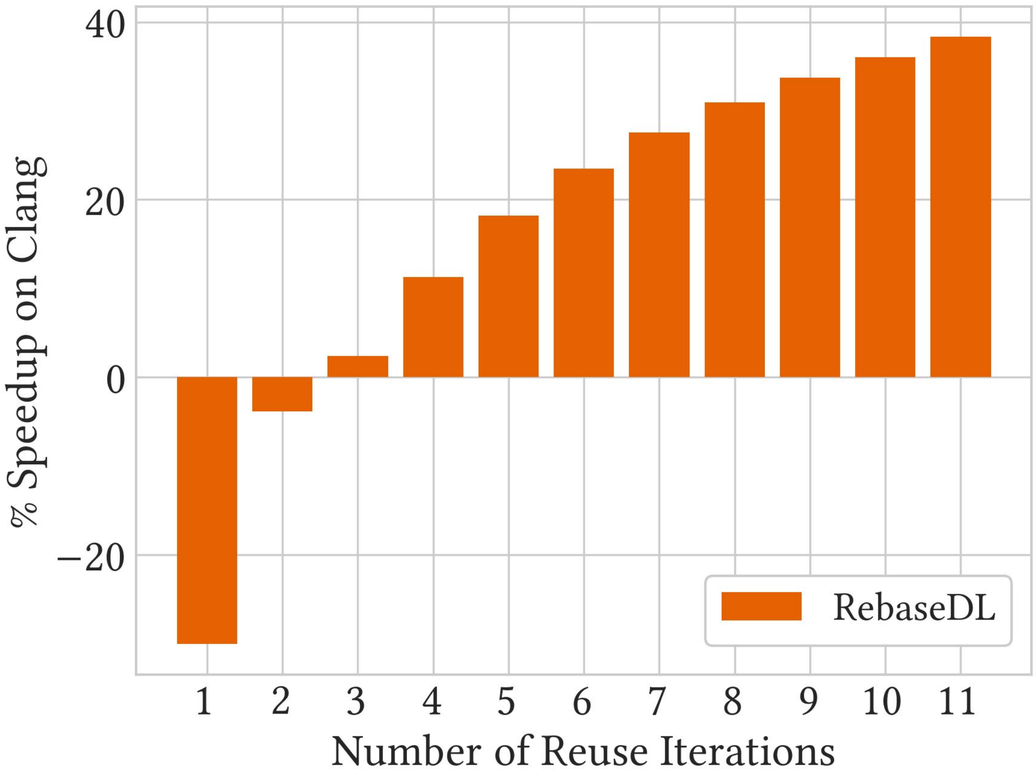
Data Reuse Impact Evaluation

- Data reuse affects the transformation's performance?



Data Reuse Impact Evaluation

- Data reuse affects the transformation’s performance?



Transformation Evaluation

- Transformation performance?

Transformation Evaluation

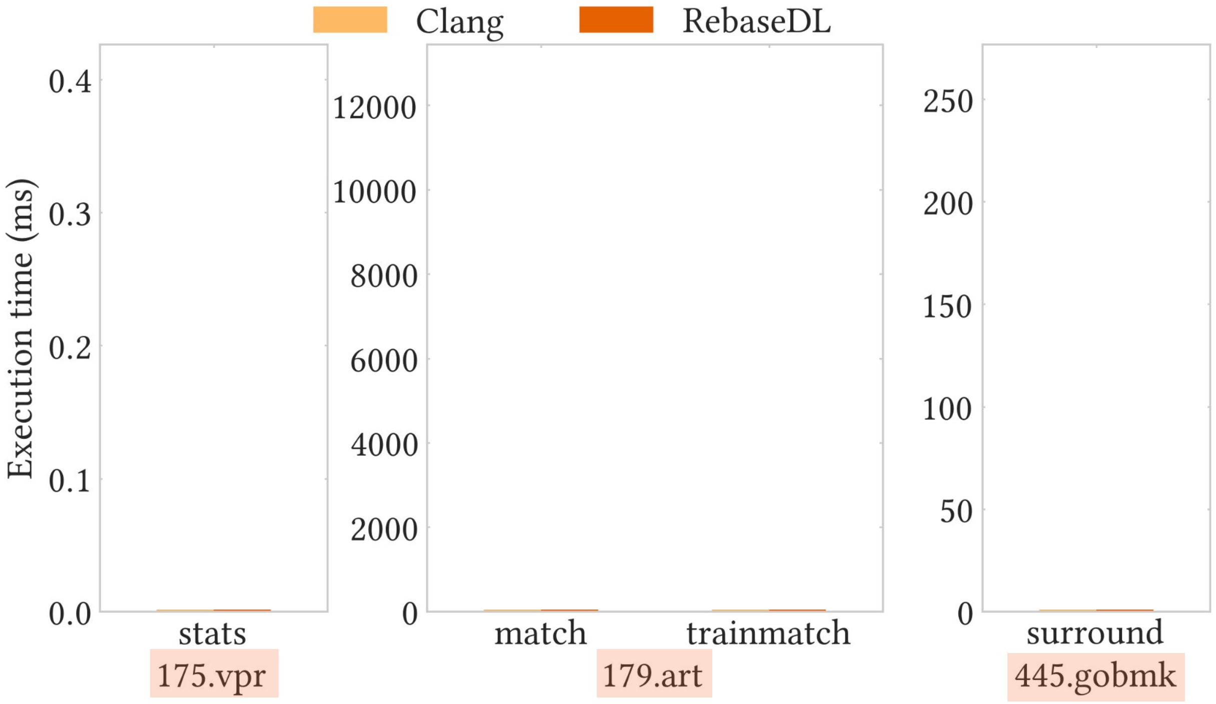
- Transformation performance?

- Not all regions execute with SPEC inputs

Benchmark	Candidates	Benchmark	Candidates
175.vpr	1	502.gcc_r	3
179.art	2	510.parest_r	42
445.gobmk	1	525.x264_r	1
447.dealII	18	526.blender_r	1
464.h264ref	1	538.imagick_r	1

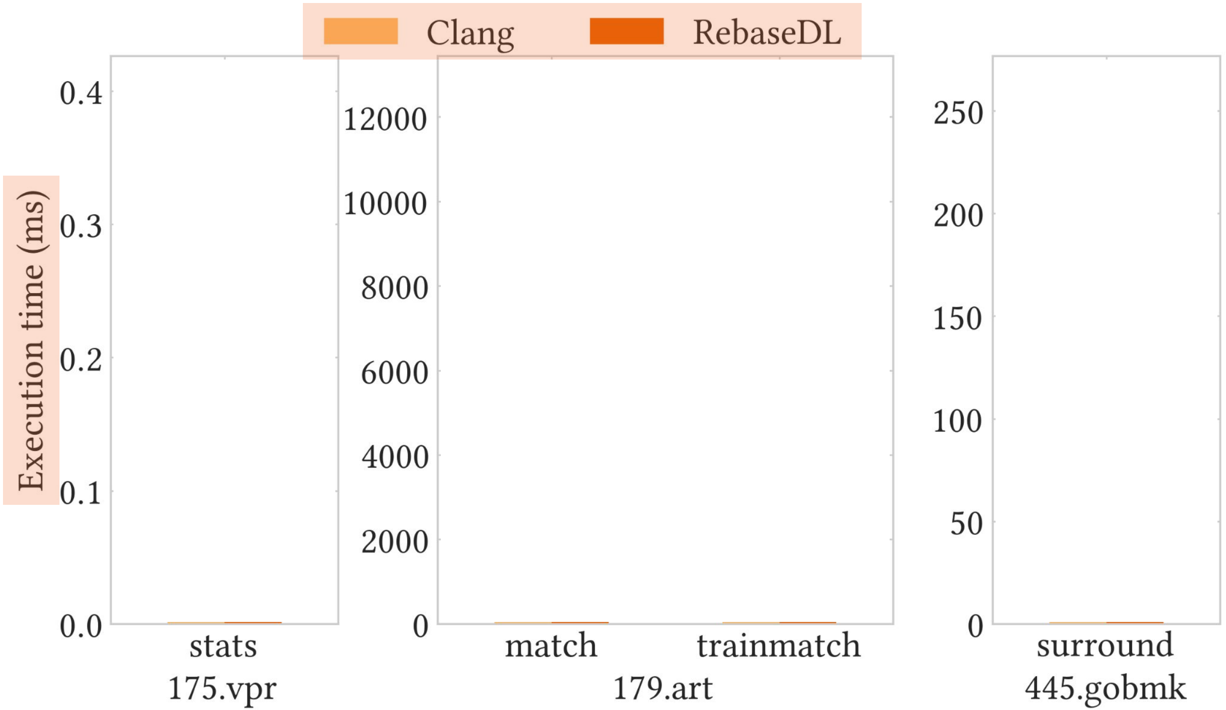
Transformation Evaluation

- Transformation performance?



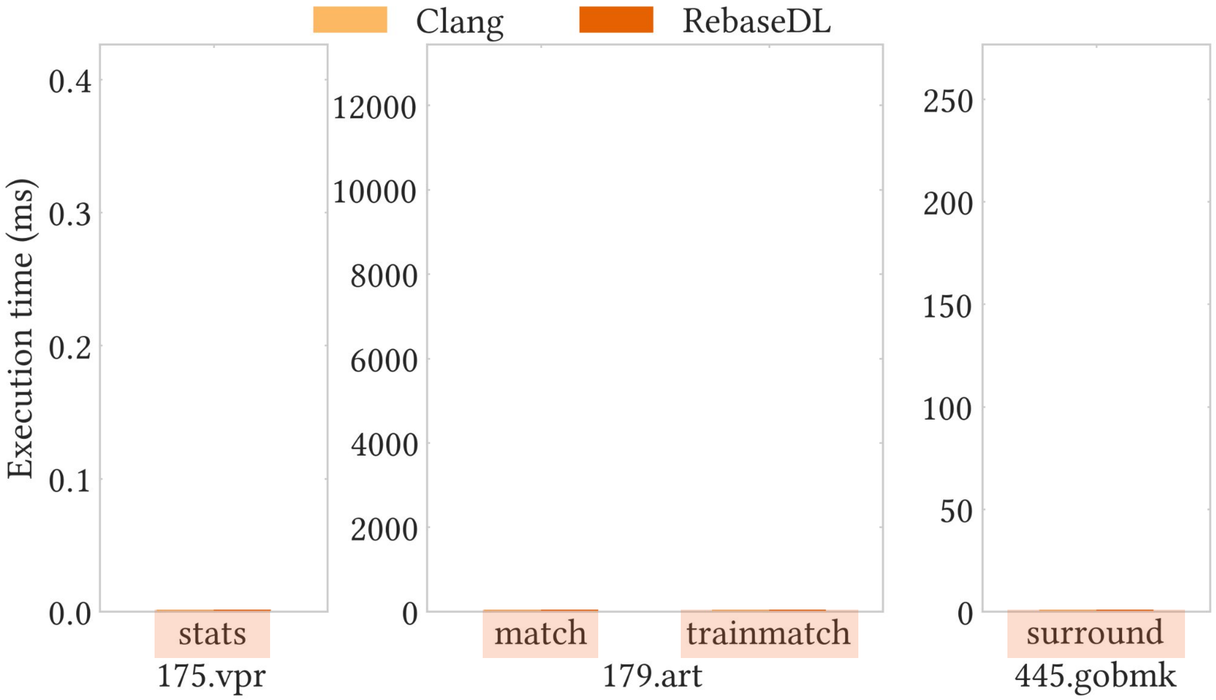
Transformation Evaluation

- Transformation performance?



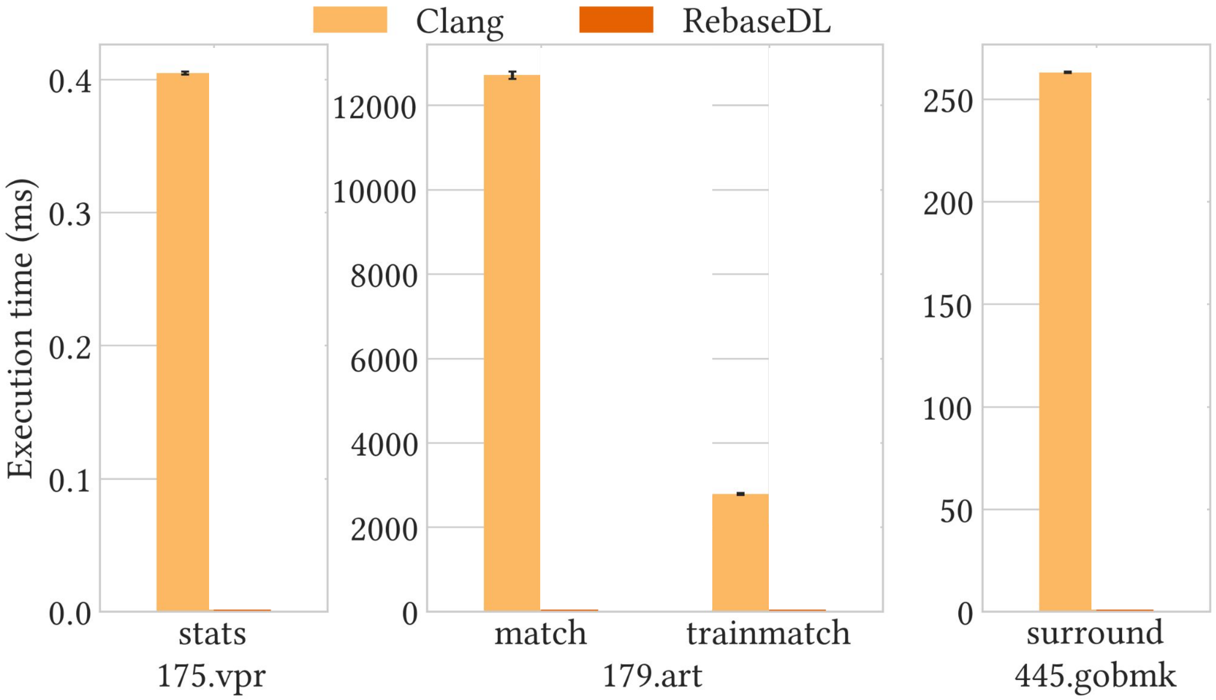
Transformation Evaluation

- Transformation performance?



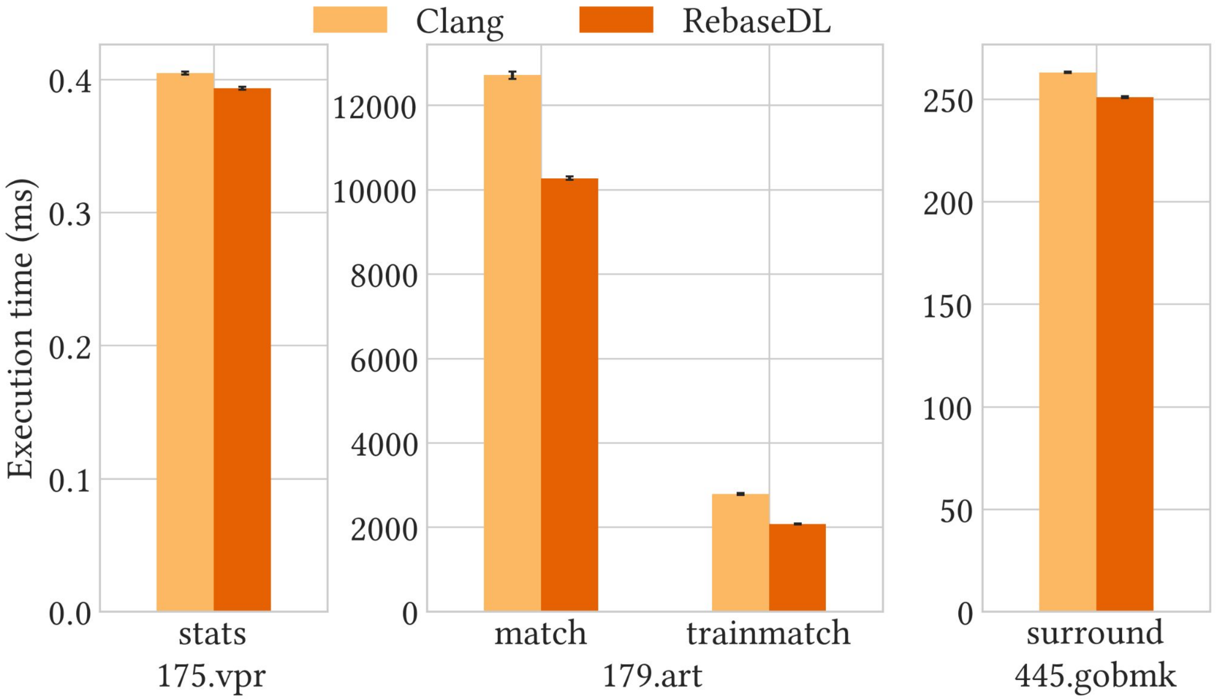
Transformation Evaluation

- Transformation performance?



Transformation Evaluation

- Transformation performance?



Transformation Evaluation

	Region
	estimate2
	estimate3
	estimate4
447.dealII	estimate5
	estimate6
	estimate7
	fe_q1
464.h264ref	macroblock
	dominators
502.gcc_r	ira_init
	omega
525.x264_r	encoder

Transformation Evaluation

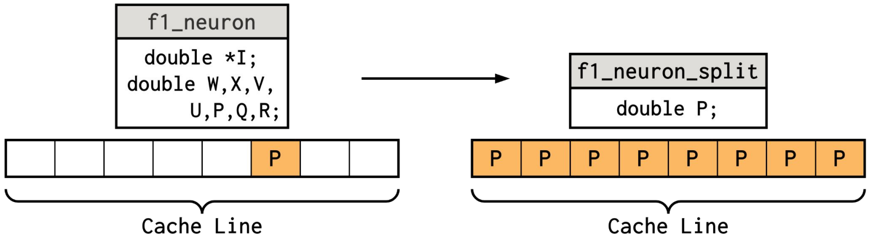
	Region	Clang Execution time (ms)	RebaseDL Execution time (ms)	Speedup over Clang
447.dealII	estimate2	7892.03 ± 18.22	7945.52 ± 12.3	0.993
	estimate3	2161.01 ± 11.73	2191.46 ± 7.13	0.986
	estimate4	2094.93 ± 8.33	2100.48 ± 6.30	0.997
	estimate5	651.33 ± 4.65	670.28 ± 2.90	0.972
	estimate6	653.57 ± 3.87	668.61 ± 2.33	0.978
	estimate7	641.21 ± 3.11	648.06 ± 1.88	0.989
	fe_q1	110.18 ± 0.09	108.96 ± 0.06	1.011
464.h264ref	macroblock	2747.48 ± 8.67	2747.54 ± 10.83	1.000
502.gcc_r	dominators	69.09 ± 0.15	69.66 ± 0.21	0.992
	ira_init	0.0422 ± 0.0003	0.0456 ± 0.0002	0.926
	omega	3.954 ± 0.029	4.050 ± 0.026	0.976
525.x264_r	encoder	662.78 ± 2.00	664.58 ± 2.36	0.997

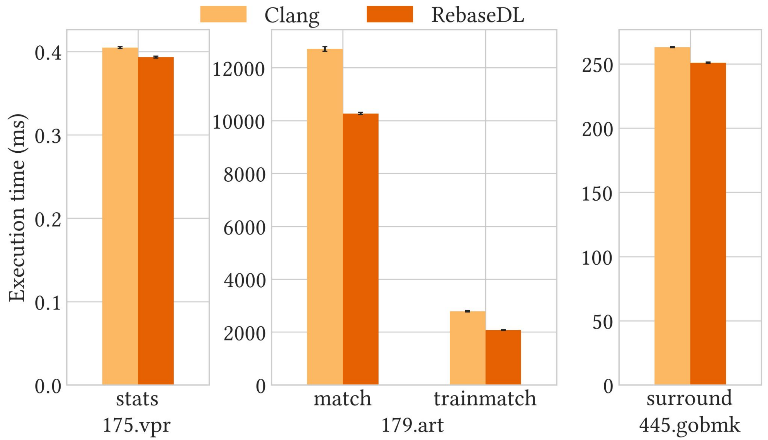
Transformation Evaluation

		Clang	RebaseDL		
	Region	Execution time (ms)	Execution time (ms)	Speedup over Clang	Reuse Iterations
447.dealII	estimate2	7892.03 ± 18.22	7945.52 ± 12.3	0.993	1
	estimate3	2161.01 ± 11.73	2191.46 ± 7.13	0.986	1
	estimate4	2094.93 ± 8.33	2100.48 ± 6.30	0.997	1
	estimate5	651.33 ± 4.65	670.28 ± 2.90	0.972	1
	estimate6	653.57 ± 3.87	668.61 ± 2.33	0.978	1
	estimate7	641.21 ± 3.11	648.06 ± 1.88	0.989	1
	fe_q1	110.18 ± 0.09	108.96 ± 0.06	1.011	125, 27, or 8
464.h264ref	macroblock	2747.48 ± 8.67	2747.54 ± 10.83	1.000	2
502.gcc_r	dominators	69.09 ± 0.15	69.66 ± 0.21	0.992	2
	ira_init	0.0422 ± 0.0003	0.0456 ± 0.0002	0.926	2
	omega	3.954 ± 0.029	4.050 ± 0.026	0.976	1 or 2
525.x264_r	encoder	662.78 ± 2.00	664.58 ± 2.36	0.997	0

Region-Based Data Layout via Data Reuse Analysis

Caio Salvador Rohwedder
csalvado@ualberta.ca





Extra Slides

Transformation Example - Packing

Previous example:

- Region-based
  - structure splitting
  - field reordering

Transformation Example - Packing

```
typedef struct {
    double *I; double W; double X; double V;
    double U; double P; double Q; double R;
} f1_neuron;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example - Packing

- Struct → Array
- Keep access pattern

```
typedef struct {
    double *I; double W; double X; double V;
    double U; double P; double Q; double R;
} f1_neuron;
```

```
for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example - Packing

- Struct → Array
- Keep access pattern

```
double *f1_layer;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example - Packing

- Struct → Array
- Keep access pattern

```
double *f1_layer;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Transformation Example - Packing

- Struct → Array
- Keep access pattern

```
double *f1_layer;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[5+ti*8]  * bus[ti][tj];
}
```

Transformation Example - Packing

Transform **f1_layer** in **loop_tj**:

1. ~~Create new type~~
2. ~~Reorder fields~~
3. Allocate and copy
4. Replace uses
5. Copy back and deallocate

```
double *f1_layer;

for (tj = 0; tj < numf2s; tj++) {
  Y[tj].y = 0;
  if (!Y[tj].reset)
    for (ti = 0; ti < numf1s; ti++)
      Y[tj].y += f1_layer[5+ti*8]  * bus[ti][tj];
}
```