On the Variety of Static Control Parts in Real-World Applications: from Affine via Multi-dimensional to Polynomial and Just-in-Time

Andreas Simbürger   Armin Größlinger

4th International Workshop on Polyhedral Compilation Techniques

Defining the Real World

Defining the Real World

- LLVM (`llvm.org`)

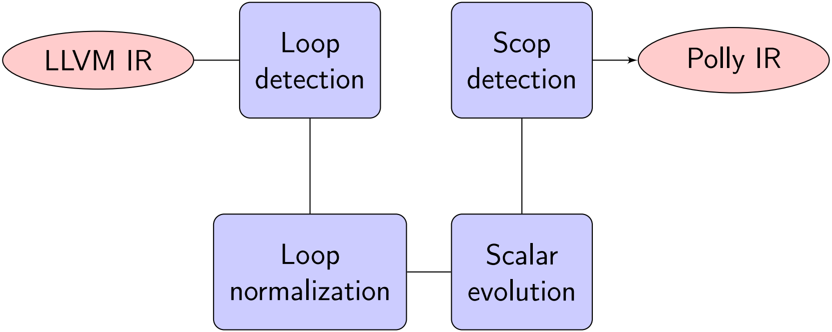
Defining the Real World

- LLVM (`llvm.org`)
- Polly (`polly.llvm.org`)

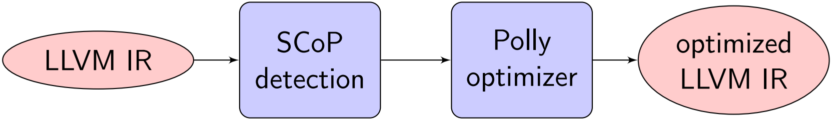
Defining the Real World

- LLVM (llvm.org)
- Polly (polly.llvm.org)
- PolyJIT (www.infosun.fim.uni-passau.de/cl/PolyJIT)

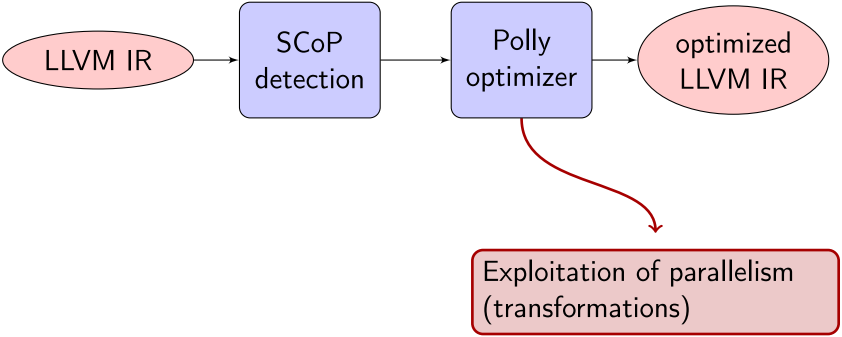
Automatic Detection of SCoPs in LLVM



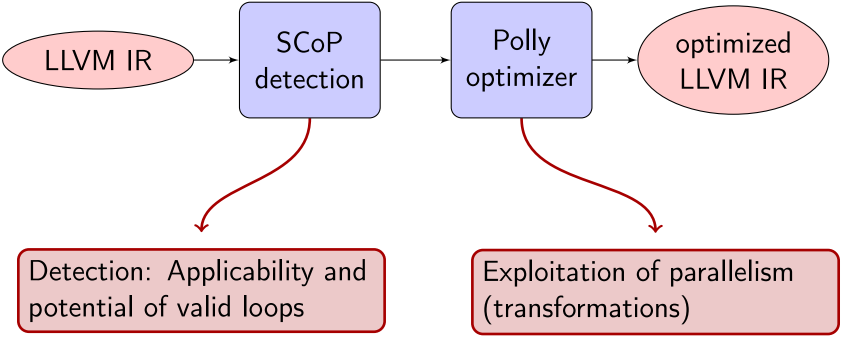
Effectiveness of Automatic Polyhedral Optimization



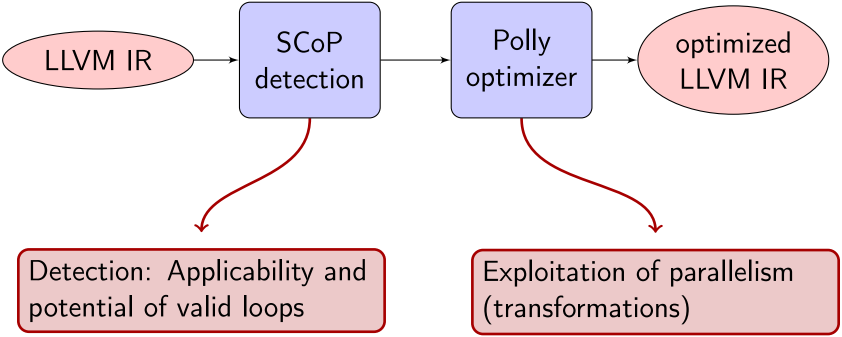
Effectiveness of Automatic Polyhedral Optimization



Effectiveness of Automatic Polyhedral Optimization



Effectiveness of Automatic Polyhedral Optimization



The detection process lacks thorough empirical evaluation!

PolyJIT: pprof

- Set of 50 programs commonly used in various domains.
- 8 domains (Multimedia, Scientific, Simulation, Encryption, Compilation, Compression, Databases, Verification).
- Extract run time and compile time statistics.

Measuring a SCoP's fraction of the total run time

What fraction of a program's total run time is spent inside SCoPs?

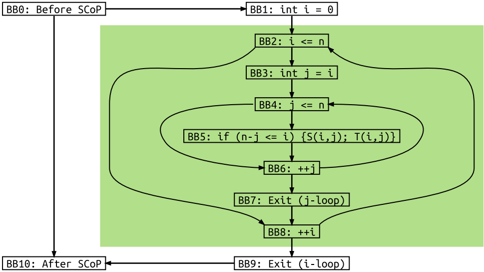
```
for (int i=0; i<=n; ++i)
  for (int j=i; j<=n; ++j)
    if (i >= n-j) {
S: A[i+n][j+i] = B[n+2*i-1][j];
T: B[i+n][j-i] = A[n-2*i+1][j];
    }
```

Definition (Execution SCoP coverage)

$$\text{ExecCov} = \frac{\text{Time spent inside SCoPs}}{\text{Total program run time}}$$

Measuring a SCoP's fraction of the total run time

What fraction of a program's total run time is spent inside SCoPs?



Definition (Execution SCoP coverage)

$$\text{ExecCov} = \frac{\text{Time spent inside SCoPs}}{\text{Total program run time}}$$

Static Control Parts: Class Static

Detection at compile time

```
for (int i=0; i<=n; ++i)
  for (int j=i; j<=n; ++j)
    if (i >= n-j) {
S: A[i+n][j+i] = B[n+2*i-1][j];
T: B[i+n][j-i] = A[n-2*i+1][j];
    }
```

Static Control Parts: Class Static

Detection at compile time

```
for (int i=0; i<=n; ++i)
  for (int j=i; j<=n; ++j)
    if (i >= n-j) {
S: A[i+n][j+i] = B[n+2*i-1][j];
T: B[i+n][j-i] = A[n-2*i+1][j];
    }
```

1. Affine expressions in
   - Loop bounds
   - Conditions
   - Memory accesses

Static Control Parts: Class Static

Detection at compile time

```
for (int i=0; i<=n; ++i)
  for (int j=i; j<=n; ++j)
    if (i >= n-j) {
S: A[i+n][j+i] = B[n+2*i-1][j];
T: B[i+n][j-i] = A[n-2*i+1][j];
    }
```

1. Affine expressions in
   - Loop bounds
   - Conditions
   - Memory accesses
2. Static control flow

Static Control Parts: Class Static

Detection at compile time

```
for (int i=0; i<=n; ++i)
  for (int j=i; j<=n; ++j)
    if (i >= n-j) {
S: A[i+n][j+i] = B[n+2*i-1][j];
T: B[i+n][j-i] = A[n-2*i+1][j];
    }
```

1. Affine expressions in
   - Loop bounds
   - Conditions
   - Memory accesses
2. Static control flow
3. Side-effect known function calls

Static Control Parts: Class Static

Detection at compile time

```
for (int i=0; i<=n; ++i)
  for (int j=i; j<=n; ++j)
    if (i >= n-j) {
S: A[i+n][j+i] = B[n+2*i-1][j];
T: B[i+n][j-i] = A[n-2*i+1][j];
    }
```

1. Affine expressions in
   - Loop bounds
   - Conditions
   - Memory accesses
2. Static control flow
3. Side-effect known function calls

What can we do, if it is not a static (affine) SCoP?

Problem 1: Multi-dimensional array accesses

Contiguous

```
A[i][j];
```

Problem 1: Multi-dimensional array accesses

Contiguous

clang -O0

```
%0 = mul nsw i32 %i, %n
%idx = getelementptr float* %A, i32 %0
%idx1 = getelementptr float* %idx, i32 %j
```

`A[i][j]`

Problem 1: Multi-dimensional array accesses

Contiguous

clang -O1

```
%0 = mul nsw i32 %i, %n
%idx.s = add i32 %0, %j
%idx1 = getelementptr float* %A, i32 %idx.s
```

`A[n*i+j]`

Problem 1: Multi-dimensional array accesses

Contiguous

clang -O1

```
%0 = mul nsw i32 %i, %n
%idx.s = add i32 %0, %j
%idx1 = getelementptr float* %A, i32 %idx.s
```

```
A[n*i+j]
```

Delinearization of array accesses

```
A[n*i+i+j]
```

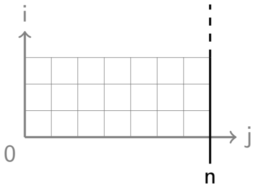
$$n * i + i + j = (n + 1) * i + j$$

Delinearization of array accesses

```
A[n*i+i+j]
```

$$n * i + i + j = (n + 1) * i + j$$

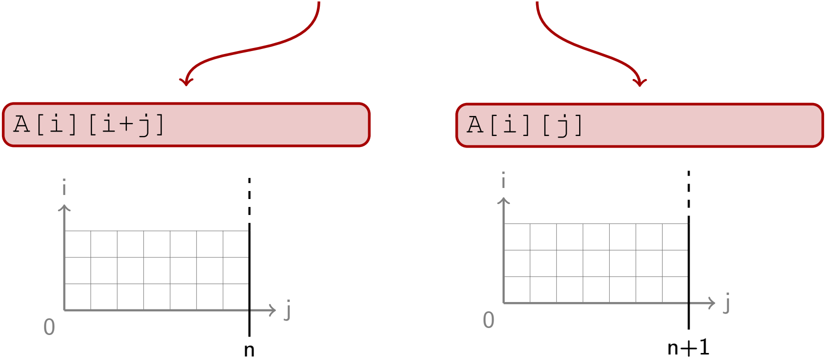
```
A[i][i+j]
```



Delinearization of array accesses

```
A[n*i+i+j]
```

$$n * i + i + j = (n + 1) * i + j$$



Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

```
A[(n+2+m)*i]                A[i'+2*m+2*n]
```

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

`A[(n+2+m)*i]`	`A[i'+2*m+2*n]`
$a = a'$	$ni + 2i + mi = i' + 2m + 2n$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

`A[(n+2+m)*i]` `A[i'+2*m+2*n]`

$a = a'$ $ni + 2i + mi = i' + 2m + 2n$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

```
A[(n+2+m)*i]                    A[i'+2*m+2*n]
```

$a = a'$ $\qquad ni + 2i + mi = i' + 2m + 2n$

$a - a' = 0$ $\qquad ni + 2i + mi - i' - 2m - 2n = 0$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

`A[(n+2+m)*i]`	`A[i'+2*m+2*n]`
$a = a'$	$ni + 2i + mi = i' + 2m + 2n$
$a - a' = 0$	$ni + 2i + mi - i' - 2m - 2n = 0$
Split into terms	$ni, 2i, mi, -i', -2m, -2n$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

`A[(n+2+m)*i]`	`A[i'+2*m+2*n]`
$a = a'$	$ni + 2i + mi = i' + 2m + 2n$
$a - a' = 0$	$ni + 2i + mi - i' - 2m - 2n = 0$
Split into terms	$ni, 2i, mi, -i', -2m, -2n$
Group by parameters	$n(i-2) + m(i-2) + 1(2i - i')$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

`A[(n+2+m)*i]`	`A[i'+2*m+2*n]`
$a = a'$	$ni + 2i + mi = i' + 2m + 2n$
$a - a' = 0$	$ni + 2i + mi - i' - 2m - 2n = 0$
Split into terms	$ni, 2i, mi, -i', -2m, -2n$
Group by parameters	$n(i-2) + m(i-2) + 1(2i - i')$
Factor out common expressions	$(n+m)(i-2) + (1)(2i - i')$

Static Control Parts: Class Multi

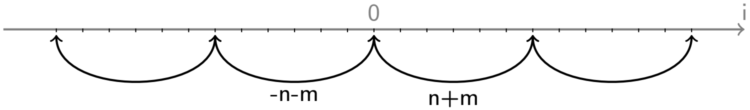
Let's allow delinearizeable accesses!

`A[(n+2+m)*i]`	`A[i'+2*m+2*n]`
$a = a'$	$ni + 2i + mi = i' + 2m + 2n$
$a - a' = 0$	$ni + 2i + mi - i' - 2m - 2n = 0$
Split into terms	$ni, 2i, mi, -i', -2m, -2n$
Group by parameters	$n(i-2) + m(i-2) + 1(2i - i')$
Factor out common expressions	$(n+m)(i-2) + (1)(2i - i')$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

`A[(n+2+m)*i]`	`A[i'+2*m+2*n]`
$a = a'$	$ni + 2i + mi = i' + 2m + 2n$
$a - a' = 0$	$ni + 2i + mi - i' - 2m - 2n = 0$
Split into terms	$ni, 2i, mi, -i', -2m, -2n$
Group by parameters	$n(i-2) + m(i-2) + 1(2i - i')$
Factor out common expressions	$(n+m)(i-2) + (1)(2i - i')$

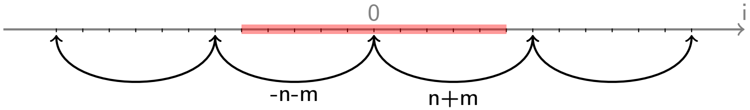


Bounds check	$\lvert 1(2i - i') \rvert \leq \lvert n + m \rvert - 1$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

`A[(n+2+m)*i]`	`A[i'+2*m+2*n]`
$a = a'$	$ni + 2i + mi = i' + 2m + 2n$
$a - a' = 0$	$ni + 2i + mi - i' - 2m - 2n = 0$
Split into terms	$ni, 2i, mi, -i', -2m, -2n$
Group by parameters	$n(i-2) + m(i-2) + 1(2i - i')$
Factor out common expressions	$(n+m)(i-2) + (1)(2i - i')$

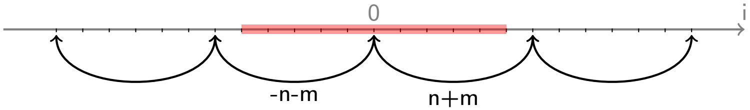


Bounds check	$\lvert 1(2i - i')\rvert \leq \lvert n+m\rvert - 1$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

`A[(n+2+m)*i]`	`A[i'+2*m+2*n]`
$a = a'$	$ni + 2i + mi = i' + 2m + 2n$
$a - a' = 0$	$ni + 2i + mi - i' - 2m - 2n = 0$
Split into terms	$ni, 2i, mi, -i', -2m, -2n$
Group by parameters	$n(i-2) + m(i-2) + 1(2i - i')$
Factor out common expressions	$(n+m)(i-2) + (1)(2i - i')$



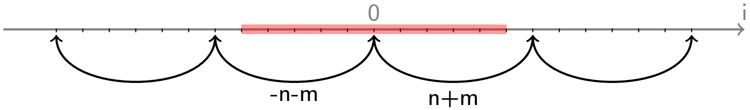
Bounds check $|1(2i - i')| \leq |n+m| - 1$

$$a - a' = 0 \Leftrightarrow i - 2 = 0 \wedge 2i - i' = 0$$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

`A[(n+2+m)*i]`	`A[i'+2*m+2*n]`
$a = a'$	$ni + 2i + mi = i' + 2m + 2n$
$a - a' = 0$	$ni + 2i + mi - i' - 2m - 2n = 0$
Split into terms	$ni, 2i, mi, -i', -2m, -2n$
Group by parameters	$n(i-2) + m(i-2) + 1(2i - i')$
Factor out common expressions	$(n+m)(i-2) + (1)(2i - i')$



Bounds check $\quad |1(2i - i')| \leq |n+m| - 1$

$$a - a' = 0 \Leftrightarrow i - 2 = 0 \wedge 2i - i' = 0$$

$$i = 2 \text{ and } i' = 4$$

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

$$a - a' = \sum_{x=1}^{k} \pi_x \gamma_x \tag{1}$$

Where π_x are polynomials in the parameters and γ_x are affine expressions in the iterators.

$$\forall \vec{i} \in D : |\pi_x \gamma_x| \leq |\pi_{x+1}| - 1 \text{ for } 1 \leq x < k \tag{2}$$

When (1) and (2) hold, $a - a' = 0$ is equivalent to

$$\gamma_1 = 0 \wedge \cdots \wedge \gamma_k = 0$$

Static Control Parts: Class Algebraic

Let's allow polynomials!

```
for (int i=0; i<=n; i++) {
  A[m*i+n] = __;
  __ = A[m*(i-1)+n];
}
```

Static Control Parts: Class Algebraic

Let's allow polynomials!

```
for (int i=0; i<=n; i++) {
  A[m*i+n] = __;
  __ = A[m*(i-1)+n];
}
```

Accept arbitrary polynomials in

- Loop bounds
- Array subscripts
- Unsupported: Products in the iterators (`i*i`)

Static Control Parts: Class Algebraic

Let's allow polynomials!

```
for (int i=0; i<=n; i++) {
  A[m*i+n] = __;
  __ = A[m*(i-1)+n];
}
```

Accept arbitrary polynomials in

- Loop bounds
- Array subscripts
- Unsupported: Products in the iterators (`i*i`)

Multi is a subset of *Algebraic*

Problem 2: Multi-dimensional array accesses

Non-Contiguous

```
float **A;
A[i][j] = A[i-1][j-1];
```

Problem 2: Multi-dimensional array accesses

Non-Contiguous

```
%i = load i64* %i.addr
%j = load i64* %j.addr
%outer = load float*** %A
%arrayidx3 = getelementptr inbounds float**
            %outer, i64 %i
%inner = load float** %arrayidx3
%arrayidx4 = getelementptr inbounds float*
            %inner, i64 %j
```

Static Control Parts: Class Pointer to Pointer

Let's allow pointers to pointers!

```
float **A;
for (int i=0; i<=n; ++i)
  for (int j=i; j<=n; ++j)
    A[i][j] = A[i-1][j-1];
```

Static Control Parts: Class Pointer to Pointer

Let's allow pointers to pointers!

```
float **A;
for (int i=0; i<=n; ++i)
  for (int j=i; j<=n; ++j)
    A[i][j] = A[i-1][j-1];
```

- No aliasing between inner dimensions.

Static Control Parts: Class Pointer to Pointer

Let's allow pointers to pointers!

```
float **A;
for (int i=0; i<=n; ++i)
  for (int j=i; j<=n; ++j)
    A[i][j] = A[i-1][j-1];
```

- No aliasing between inner dimensions.
- No aliasing of the outer dimension with other pointers/arrays.

Static Control Parts: Class Dynamic

Let's be lazy and do everything at run time

```
for (int i=0; i<=n; i++) {
  A[m*i+n] = __;
  __ = A[m*(i-1)+n];
}
```

Static Control Parts: Class Dynamic

Let's be lazy and do everything at run time

```
for (int i=0; i<=n; i++) {
  A[m*i+n] = __;
  __ = A[m*(i-1)+n];
}
```

- Run time specialization for

Static Control Parts: Class Dynamic

Let's be lazy and do everything at run time

```
for (int i=0; i<=n; i++) {
  A[42*i+n] = __;
  __ = A[42*(i-1)+n];
}
```

- Run time specialization for
  - Known parameter values

Static Control Parts: Class Dynamic

Let's be lazy and do everything at run time

```
for (int i=0; i<=n; i++) {
  A[42*i+n] = __;
  __ = A[42*(i-1)+n];
}
```

- Run time specialization for
  - Known parameter values
  - Known aliasing

Static Control Parts: Class Dynamic

Let's be lazy and do everything at run time

```
for (int i=0; i<=n; i++) {
  A[42*i+n] = __;
  __ = B[42*(i-1)+n];
}
```

- Run time specialization for
  - Known parameter values
  - Known aliasing

Static Control Parts: Class Dynamic

Let's be lazy and do everything at run time

```
for (int i=0; i<=n; i++) {
  A[42*i+n] = __;
  __ = B[42*(i-1)+n];
}
```

- Run time specialization for
  - Known parameter values
  - Known aliasing
- Function calls to other SCoPs

Expectations

Compile time

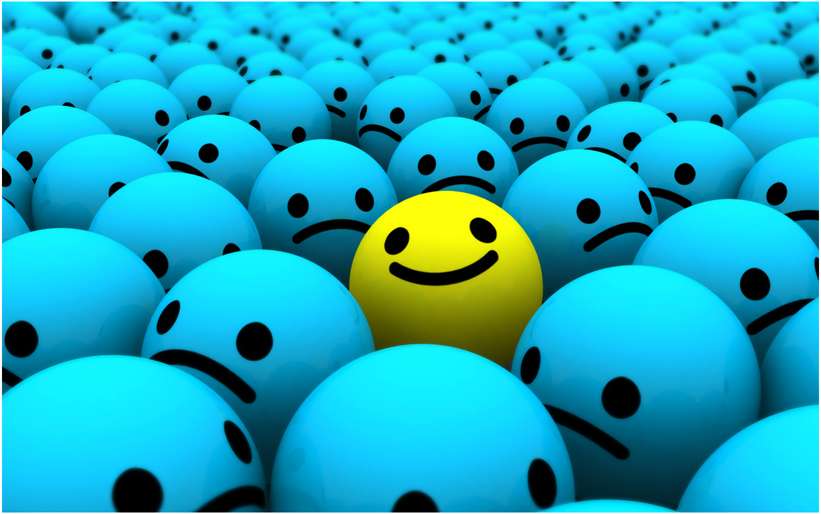
- Multi-dimensional array accesses are used often, so *Multi* (*Algebraic*) should contain a lot more SCoPs than *Static*.
- *Pointer to Pointer* should cover a few more SCoPs than *Static*.

Run time

$$\textit{Static} \subseteq \textit{Multi} \subseteq \textit{Algebraic} \subseteq \textit{Dynamic}$$

$$\textit{Static} \subseteq \textit{Pointer to Pointer} \subseteq \textit{Pointer to Pointer (No Alias)}$$

Reality



Reality

Name	Stat | Alg | Dyn	S_{st}
Compilation		
js		120
python		84
ruby		100
tcc		18
Compression		
7za		100
bzip2		30
gzip		23
xz		23
Multimedia		
avconv		950
povray		110
x264		55
Scientific		
linpack		9
xeigtstc		630
xeigtstd		610
xeigtsts		610
xlintstd		690
xlintstds		160
xlintstrfc		170
xlintstrfd		150
xlintstrfs		150
xlintstrfz		170
xlintsts		530
xlintstzc		190

Name	Stat | Alg | Dyn	S_{st}
Encryption		
blowfish		0
bn		24
cast		2
ccrypt		3
des		9
dsa		24
ecdsa		24
hmac		24
mcrypt-aes		38
mcrypt-cip		39
md5		24
openssl		67
rc4		1
rsa		24
sha1		24
sha256		24
sha512		24
ssl		32
Simulation		
crafty		57
lammps		330
lulesh-omp		16
lulesh		14

Findings

Class *Algebraic* and *Multi*

Findings

Class *Algebraic* and *Multi*

- Very low increment in SCoPs (between 2 and 37) over *Static*.

Findings

Class *Algebraic* and *Multi*

- Very low increment in SCoPs (between 2 and 37) over *Static*.
- Only 10 out of 50 experiments show a small number of non-affine expressions

Findings

Class *Algebraic* and *Multi*

- Very low increment in SCoPs (between 2 and 37) over *Static*.
- Only 10 out of 50 experiments show a small number of non-affine expressions
- Povray shows an $ExecCov_{Algebraic}$ of 41% as compared to an $ExecCov_{Static}$ of 8.5%.

Findings

Class *Algebraic* and *Multi*

- Very low increment in SCoPs (between 2 and 37) over *Static*.
- Only 10 out of 50 experiments show a small number of non-affine expressions
- Povray shows an $ExecCov_{Algebraic}$ of 41% as compared to an $ExecCov_{Static}$ of 8.5%.
- No notable increase in $ExecCov_{Algebraic}$ in the other experiments.

Findings

Name	S_{st}	S_{pp}	S_{ppa}
Compilation			
js	120	36	95
python	84	6	33
ruby	100	19	40
tcc	18	0	0
Compression			
7za	100	9	24
bzip2	30	0	0
gzip	23	0	0
xz	23	1	1
Multimedia			
avconv	950	11	310
povray	110	3	71
x264	55	14	44
Scientific			
linpack	9	0	0
xeigtstc	630	5	5
xeigtstd	610	0	0
xeigtsts	610	0	0
xlintstd	690	0	0
xlintstds	160	0	0
xlintstrfc	170	0	0
xlintstrfd	150	0	0
xlintstrfs	150	0	0
xlintstrfz	170	0	0
xlintsts	530	0	0
xlintstzc	190	0	0

Name	S_{st}	S_{pp}	S_{ppa}
Encryption			
blowfish	0	0	0
bn	24	1	10
cast	2	0	0
ccrypt	3	0	0
des	9	0	7
dsa	24	1	10
ecdsa	24	1	10
hmac	24	1	10
mcrypt-aes	38	0	0
mcrypt-cip	39	0	0
md5	24	1	10
openssl	67	1	18
rc4	1	0	0
rsa	24	1	10
sha1	24	1	10
sha256	24	1	10
sha512	24	1	10
ssl	32	1	12
Simulation			
crafty	57	0	9
lammps	330	45	470
lulesh-omp	16	0	0
lulesh	14	0	0

Findings

Class *Pointer to Pointer*

Findings

Class *Pointer to Pointer*

- Low increment in SCoPs compared to *Static* (between 1 and 45). 22 out of 50 experiments show an increment in SCoP count.

Findings

Class *Pointer to Pointer*

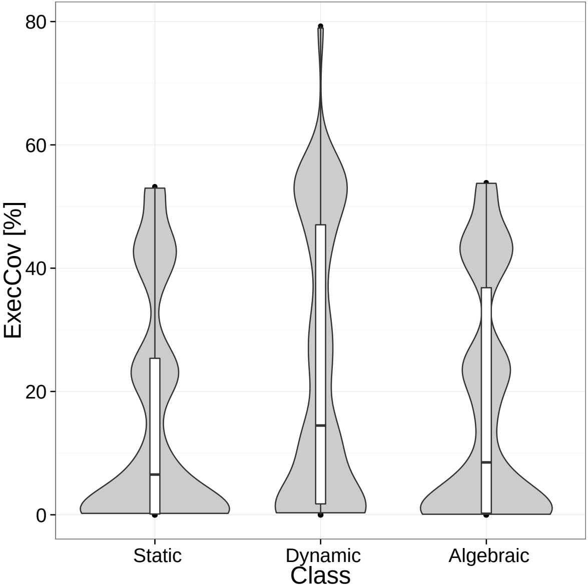
- Low increment in SCoPs compared to *Static* (between 1 and 45). 22 out of 50 experiments show an increment in SCoP count.
- The SCoP count can be (optimistically) increased by disabling alias checks.

Findings

Class *Pointer to Pointer*

- Low increment in SCoPs compared to *Static* (between 1 and 45). 22 out of 50 experiments show an increment in SCoP count.
- The SCoP count can be (optimistically) increased by disabling alias checks.
- No $\mathit{ExecCov}_{\mathit{Pointer\ to\ Pointer}}$ information available yet.

Run-time findings



Threats to Validity

Threats to Validity

1. Construct Validity: Timing causes overhead.

Threats to Validity

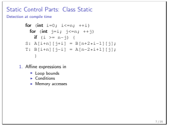
1. Construct Validity: Timing causes overhead.
2. External Validity: Generalizability depends on the sample size.

Threats to Validity

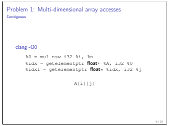
1. Construct Validity: Timing causes overhead.
2. External Validity: Generalizability depends on the sample size.
3. Internal Validity: Quality of the input data. Relying on developer's testing.

Questions?

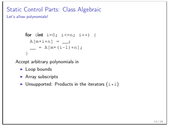
The Real World



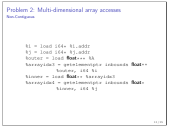
SCoPs: Static



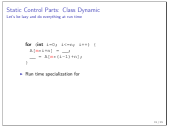
SCoPs: Multi



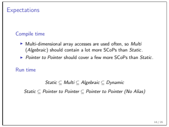
SCoPs: Algebraic



SCoPs: Pointer to Pointer



SCoPs: Dynamic



Expectations

Reality

Static Control Parts: Class Multi

Let's allow delinearizeable accesses!

1. Split $a - a'$ into its terms, i.e., $a - a' = \sum_{x=1}^{l} t_x$ where each t_x is a product of iterators and parameters (and a constant).
2. Group terms by their parameters, i.e., $a - a' = \sum_{x=1}^{m} \rho_x \gamma_x$ where each ρ_x is a product of parameters (or a constant).
3. Factor out common γ_x, i.e., $a - a' = \sum_{x=1}^{k} \pi_x \gamma_x$
4. Check the (total) ordering criterion using quantifier elimination, i.e.,

$$\forall \vec{i}, \vec{i'}, \vec{p} (\vec{i} \in D \wedge \vec{i'} \in D' \rightarrow |\pi_x \gamma_x| \leq |\pi_y| - 1)$$