Data-Flow/Dependence Profiling for Structured Transformations

Fabian Gruber [1] Manuel Selva [1] Diogo Sampaio [1]
Christophe Guillon [2] Antoine Moynault [2]
Louis-Noël Pouchet [3] Fabrice Rastello [1]

[1]Université Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

[2]STMicroelectronics

[3]Colorado State University

01.02.2019

Performance debugging

functional debugging

performance debugging

functional debugging

my program is crashing!

performance debugging

functional debugging

my program is crashing!

performance debugging

my program is slow!

functional debugging

my program is crashing!

where is my program crashing?

performance debugging

my program is slow!

functional debugging

my program is crashing!

where is my program crashing?

performance debugging

my program is slow!

where is my program slow?

functional debugging

my program is crashing!

where is my program crashing?

why is my program crashing?

performance debugging

my program is slow!

where is my program slow?

functional debugging

my program is crashing!

where is my program crashing?

why is my program crashing?

performance debugging

my program is slow!

where is my program slow?

why is my program slow?

functional debugging

my program is crashing!

where is my program crashing?

why is my program crashing?

. . .

performance debugging

my program is slow!

where is my program slow?

why is my program slow?

how can I make my program fast?

An example

Example: A simple stencil

```
float *A = ..., *B = ...;

for (t = 0; t < TSTEPS; t++) {


  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }

  swap(A, B);
}
```

Example: A possible transformation

```
float *A = ..., *B = ...;

for (t = 0; t < TSTEPS; t++) {
  #pragma omp parallel

  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }

  swap(A, B);
}
```

Example: A possible transformation ...

```
float *A = ..., *B = ...;

for (t = 0; t < TSTEPS; t++) {
  #pragma omp parallel
  #pragma omp simd
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }

  swap(A, B);
}
```

Example: The Real World

```
float *A = ..., *B = ...;

for (t = 0; t < TSTEPS; t++) {

  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {

  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

Example: The Real World ...

```
float *A = ..., *B = ...;

for (t = 0; t < TSTEPS; t++) {

  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

Example: The Real World ...

```
float *A = ..., *B = ...;

for (t = 0; t < TSTEPS; t++) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

Example: The Real World ...

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

Are the transformations still valid?

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

Problem 1: Interprocedural

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

Problem 3: Loop Bounds

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

Problem 2: Aliasing

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

Problem 4: Complex Dependencies

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

MICKEY

MICKEY

- a performance debugging & exploration tool
- data dependence profiling based
- interprocedural analysis
- using binary instrumentation

Goals

- find potential for structured transformations
  - tiling
  - parallelization
  - vectorization
  - ...
- works on general programs
- works on optimized binaries

MICKEY

- a performance debugging & exploration tool
- data dependence profiling based
- interprocedural analysis
- using binary instrumentation

Goals

- find potential for polyhedral transformations
  - tiling
  - parallelization
  - vectorization
  - ...
- works on general programs
- works on optimized binaries

MICKEY: challenges

- machine code is hard
    - basic blocks
    - function boundaries
    - control flow
    - call graph
    - data flow
    - iterators
    - . . .
- partially irregular programs
    - precisely model regular parts
    - approximate irregular parts
- interprocedural

Binary Instrumentation

Qemu

- (mostly) platform indenpendent IR
- instrument on the fly
  - trace branches, calls, ...
  - trace memory accesses
  - trace values

- detect data dependencies using shadow memory

Dynamic Dependence Graph (DDG)

```
for (t = 0; t < TSTEPS; t++) {
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }

  swap(A, B);
}
```

Dynamic Dependence Graph (DDG)

```
45 31 c9                  xor    %r9d,%r9d
85 ff                     test   %edi,%edi
7e 69                     jle    4007c0
44 8d 46 fd               lea    -0x3(%rsi),%r8d
f3 0f 10 0d a5 01 00      movss  0x1a5(%rip),%xmm1
00
49 83 c0 02               add    $0x2,%r8
66 0f 1f 84 00 00 00      nopw   0x0(%rax,%rax,1)
00 00
b8 01 00 00 00            mov    $0x1,%eax
83 fe 02                  cmp    $0x2,%esi
7e 29                     jle    4007a3
66 0f 1f 44 00 00         nopw   0x0(%rax,%rax,1)
f3 0f 10 44 82 fc         movss  -0x4(%rdx,%rax,4),%xmm0
f3 0f 58 04 82            addss  (%rdx,%rax,4),%xmm0
f3 0f 58 44 82 04         addss  0x4(%rdx,%rax,4),%xmm0
f3 0f 5e c1               divss  %xmm1,%xmm0
f3 0f 11 04 81            movss  %xmm0,(%rcx,%rax,4)
48 83 c0 01               add    $0x1,%rax
4c 39 c0                  cmp    %r8,%rax
75 dd                     jne    400780
[...]
```

Dynamic Dependence Graph (DDG)

```
for (t = 0; t < TSTEPS; t++) {
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }

  swap(A, B);
}
```

Dynamic Dependence Graph (DDG)

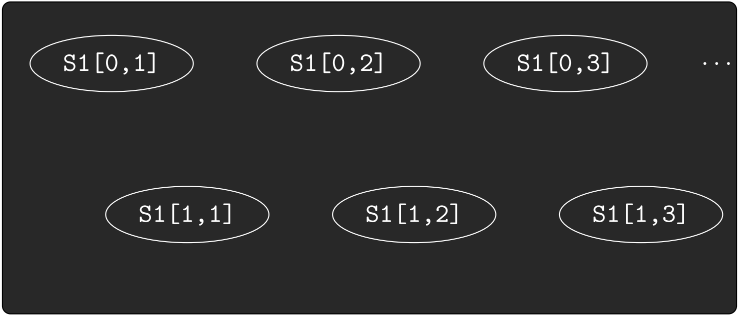
```
for (t = 0; t < TSTEPS; t++) {
  for (i = 1; i < N - 1; i++) {
    S1[t, i];
  }

  S2[t];
}
```

Dynamic Dependence Graph (DDG)

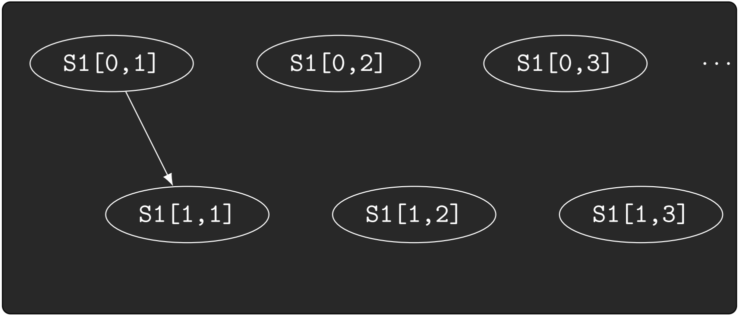
Dynamic Dependence Graph (DDG)

Dynamic Dependence Graph (DDG)

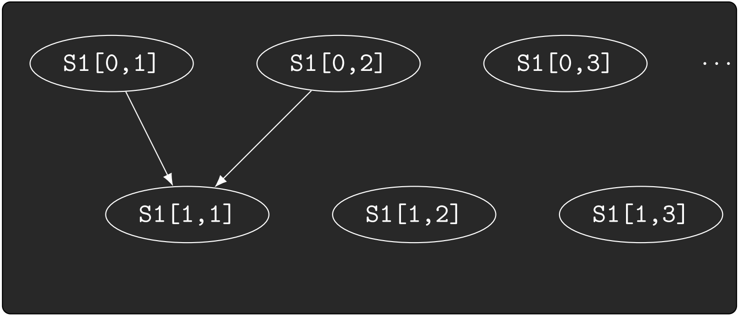
Dynamic Dependence Graph (DDG)



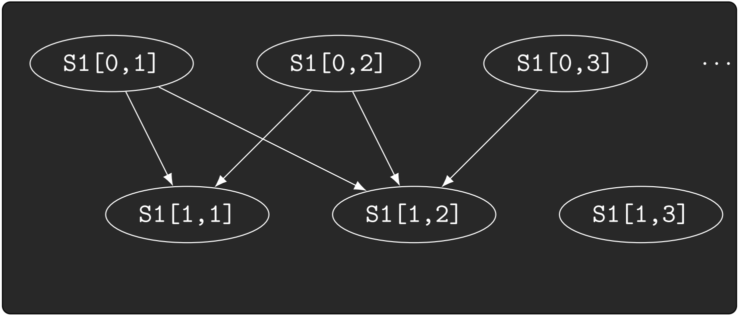
Dynamic Dependence Graph (DDG)



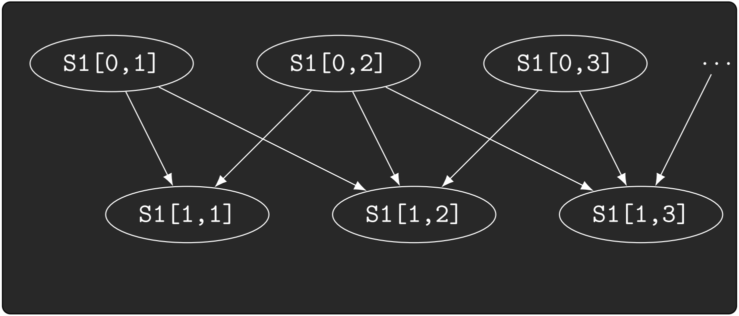
Dynamic Dependence Graph (DDG)



Dynamic Dependence Graph (DDG)



Dynamic Dependence Graph (DDG)



DDG folding ...

The DDG viewed as a stream of points & dependencies.

```
S1[0, 1];
S1[0, 2];
S1[0, 3];
S1[1, 1];       S1[0, 1] -> S1[1, 1];
...
```

The folded DDG

```
# statements
{ S1[t, i]: 0 <= t < TSTEPS and 1 <= N < N - 1 }

# dependencies, f(t_dst, i_dst) -> (t_src, i_src)
{ S1[t_src, i_src] -> S1[t_dst, i_dst]: t_src = t_dst-1 and
                                        i_src = i_dst ... }
```

Interprocedural DDG

```
for (t = 0; t < TSTEPS; t++) {
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

Interprocedural DDG ...

```
for (t = 0; t < TSTEPS; t++) {
  solve_one_step(A, B);
  S2[t];
}

void solve_one_step(float *A, float *B) {
  for (i = 1; i < N - 1; i++) {
    S1[t, i];
  }
}
```

Interprocedural DDG ...

```
L1: for (t = 0; t < TSTEPS; t++) {
          solve_one_step(A, B);
          S2[t];
    }

F1: void solve_one_step(float *A, float *B) {
L2:       for (i = 1; i < N - 1; i++) {
            S1[t, i];
          }
    }
```

Interprocedural DDG ...

```
L1: for (t = 0; t < TSTEPS; t++) {
          solve_one_step(A, B);
          S2[t];
      }

F1: void solve_one_step(float *A, float *B) {
L2:     for (i = 1; i < N - 1; i++) {
            S1[t, i];
        }
      }

# statements?


```

Interprocedural DDG ...

```
L1: for (t = 0; t < TSTEPS; t++) {
      solve_one_step(A, B);
      S2[t];
    }

F1: void solve_one_step(float *A, float *B) {
L2:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# statements?
L1/S2[t]
L1/F1/L2/S1[t, i]
```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

#path

```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# path
F1
```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# path
F1/F2
```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# path
F1/F2/L1
```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# path
F1/F2
```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# path
F1
```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# path
F1/F1
```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# path
F1/F1/F2
```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# path
F1/F1/F2/L1
```

Recursion?

```
F1: void solve(int t) {
      if (t < TSTEPS) {
        solve_one_step(A, B);
        S2[t];
        solve(t+1);
      }
    }

F2: void solve_one_step(float *A, float *B) {
L1:   for (i = 1; i < N - 1; i++) {
        S1[t, i];
      }
    }

# path
F1/F1/F1/F2/L1
```

Recursion!

- Treat recursion like loops
- `F1/F1/F1/F2/L1` $\rightarrow$ $\mathtt{L}_{F1}$`/F2/L1`

- Important differences:
  - recursive loops defined on call graph
  - returns have semantics!

  - you can't exit a recursive loop with a call
  - only with a return
  - a return can also iterates a recursive loop

MICKEY & our example

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

MICKEY: interprocedural

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

MICKEY: canonical iterators

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

MICKEY: data dependence driven

```
float *A = ..., *B = ...;

while (t++ != params.TSTEPS) {
  omp_set_num_threads(8);
  solve_one_step(A, B);
  swap(A, B);
}

void solve_one_step(float *A, float *B) {
  if (debug) print("TSTEP", t);
  for (i = 1; i < N - 1; i++) {
    B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
  }
}
```

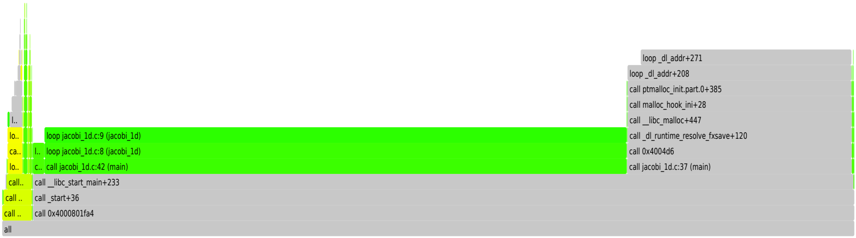
Analysis backend

- Uses a polyhedral optimizer (PoCC) as backend
- Input: folded DDG + annotations
- Finds loop transformations valid for a given execution

User Feedback

- Visualize regularity of regions with a flame graph
- Feedback on potential for
  - tiling
  - parallelization
  - vectorization
  - ...

User Feedback: Flame Graph



User Feedback: Annotated AST

```
  [...]
|= AST=(0,0,i)
|-- Loop i => parallel/tilable loop, 2097203 operations (30.77%)
    +-- alignment: 3 stride-0 (524322) 2 stride-1 (1048576) 0 stride-N (0)
|-- contains: S64 S67 S62 S65 S57 S54 S55
|-- Stmts full names:
                + 4011800000000101_4011bf(c0,c1); // @LOAD@
                + 4011800000000101_4011cb(c0,c1); // @LOAD@
               [...]
     |= AST=(0,0,i,j)
     |-- Loop j => parallel/tilable loop, 2097203 operations (30.77%)
         +-- alignment: 1 stride-0 (524288) 0 stride-1 (0) 4 stride-N (1048610)
     |-- contains: S64 S67 S62 S65 S57 S54 S55
     |-- Stmts full names:
                    + 4011800000000101_4011bf(c0,c1); // @LOAD@
                    + 4011800000000101_4011cb(c0,c1); // @LOAD@
                    [...]
|= AST=(0,6,i)
|-- Loop i => parallel/tilable loop, 1048576 operations (15.38%)
    +-- alignment: 2 stride-0 (1048576) 0 stride-1 (0) 0 stride-N (0)
|-- contains: S3 S4
  [...]
```

User Feedback: Pseudo-Code

```
[...]
parfor (jTile = 0; jTile <= floor(32767/32); jTile++) {
  for (i = 0; i <= 15; i++) {
    parfor (j = (32 * jTile); j <= min(32767, ((32 * jTile) + 31)); j++) {
      4012280000000101_401249(i,j); /* @LOAD@ */;
      4012280000000101_40125e(i,j); /* @FLOAT@ */;
      4012280000000101_401265(i,j); /* @FLOAT@ */;
      4012280000000101_401275(i,j); /* @FLOAT@ */;
      4012280000000101_40127d(i,j); /* @FLOAT@ */;
      4012280000000101_401289(i,j); /* @STORE@ */;
      4012280000000101_401281(i,j); /* @FLOAT@ */;
      4012280000000101_401285(i,j); /* @STORE@ */;
    }
  }
}
[...]
```

Evaluation

Evaluation

- use a parallel benchmark suite (Rodinia)
- run single-threaded
- see what optimizations MICKEY proposes
- do the same with a static polyhedral optimizer

Evaluation: MICKEY & Rodinia

Benchmark	Optim.	Benchmark	Optim.	Benchmark	Optim.
`backprop`	T 2D, P, V	`kmeans`	T 4D, P, V	`particlefilter`	T 2D, P, V
`bfs`	T 2D, P	`lavaMD`	T 3D, P	`pathfinder`	T 2D, P
`b+tree`	T 3D, P, V	`leukocyte`	T 3D, P, V	`srad_v1`	T 2D, P
`cfd`	T 3D, P, V	`lud`	T 3D, P	`srad_v2`	T 2D, P
`heartwall`	T 5D, P	`myocyte`	T 1D, P, V	`streamcluster`	-
`hotspot`	T 2D, P	`nn`	T 1D, P		
`hotspot3D`	T 3D, P	`nw`	T 2D, P, V		

T *n*D. *n* dimensional tiling
P. thread level parallelism
V. vectorization

Evaluation: LLVM Polly & Rodinia

In all benchmarks Polly[1] fails to model the entire parallel loop

Benchmark	Reasons	Benchmark	Reasons	Benchmark	Reasons
backprop	A	kmeans	RFA	particlefilter	CF
bfs	BF	lavaMD	BF	pathfinder	BP
b+tree	BF	leukocyte	RCBFAP	srad_v1	RF
cfd	F	lud	BF	srad_v2	RF
heartwall	RCBF	myocyte	CBA	streamcluster	RCBFAP
hotspot	B	nn	RF		
hotspot3D	BF	nw	RF		

R. unhandled function call

C. complex CFG (`break`/`return`)

B. non-affine loop bound/conditional

F. non-affine access function

A. unhandled pointer aliasing

P. base pointer not loop invariant

[1]Polly 7.0.1; interprocedural kernels force inlined; `-ffast-math`

Things left out

- Approximation of quasi-affine dependencies
- Detection of induction variables
- Tail/Sibling calls, exceptions
- Statement coalescing in the backend
- ...

Conclusion

MICKEY

- a performance debugging & exploration tool
- data dependence based
- finds potential for structured transformations
  - tiling
  - parallelization
  - vectorization
  - . . .
- works on general programs
- works on optimized binaries

Thank you for your attention