Polly's Polyhedral Scheduling in the Presence of Reductions

Johannes Doerfert[⋆]
Kevin Streit[⋆]
Sebastian Hack[⋆]
Zino Benaissa[†]

[⋆] Saarland University
Saarbrücken, Germany

[†] Qualcomm Innovation Center
San Diego, USA

SAARLAND UNIVERSITY
COMPUTER SCIENCE

January 19, 2015

Reductions

```
for (i = 0; i < 4 * N; i++)
  sum += A[i];
```

P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In Proceedings of the 3rd International Conference on Supercomputing, ICS '89, pages 186–194, New York, NY, USA, 1989. ACM.

Reductions

```
tmp_sum[4] = {0,0,0,0}
for (i = 0; i < 4 * N; i+=4)
  tmp_sum[0:3] += A[i:i+3];

sum += tmp_sum[0] + tmp_sum[1];
     + tmp_sum[2] + tmp_sum[3];
```

P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In Proceedings of the 3rd International Conference on Supercomputing, ICS '89, pages 186–194, New York, NY, USA, 1989. ACM.

Reductions

```
for (i = 0; i < 4 * N; i++) {
  S(i);
  sum += A[i];
  P(i);
}
```

B. Pottenger and R. Eigenmann. Idiom recognition in the polaris parallelizing compiler. In Proceedings of the 9th International Conference on Supercomputing, ICS '95, pages 444–448, New York, NY, USA, 1995. ACM.

Reductions

```
tmp_sum[4] = {0,0,0,0}

for (i = 0; i < 4 * N; i+=4) {
  vecS(i:i+3);
  tmp_sum[0:3] += A[i:i+3];
  vecP(i:i+3);
}

sum += tmp_sum[0] + tmp_sum[1];
     + tmp_sum[2] + tmp_sum[3];
```

B. Pottenger and R. Eigenmann. Idiom recognition in the polaris parallelizing compiler. In Proceedings of the 9th International Conference on Supercomputing, ICS '95, pages 444–448, New York, NY, USA, 1995. ACM.

Reductions

```
for (i = 0; i < NX; i++) {
  for (j = 0; j < NY; j++) {
    q[i] = q[i] + A[i][j] * p[j];
    s[j] = s[j] + r[i] * A[i][j];
  }
}
```

X. Redon and P. Feautrier. Detection of recurrences in sequential programs with loops. In Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe, PARLE '93, pages 132–145, London, UK, UK, 1993.
X. Redon and P. Feautrier. Scheduling reductions. In Proceedings of the 8th International Conference on Supercomputing, ICS '94, pages 117–125, New York, NY, USA, 1994. ACM. X. Redon and P. Feautrier. Detection of scans in the

Reductions

```
for (i = 0; i <= N; i++)
  A[i] = i;
```

```
for (i = N; i >= 0; i--)
  sum += A[i];
```

G. Gupta, S. Rajopadhye, and P. Quinton. Scheduling reductions on realistic machines. In Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '02, pages 117–126, New York, NY, USA, 2002. ACM.

Reductions

```
for (i = 0; i <= N; i++)
  A[i] = i;

sums[N+1] = sum;
for (i = N; i >= 0; i--)
  sums[i] = sums[i+1] + A[i];
sum = sums[0];
```

G. Gupta, S. Rajopadhye, and P. Quinton. Scheduling reductions on realistic machines. In Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '02, pages 117–126, New York, NY, USA, 2002. ACM.

Reductions

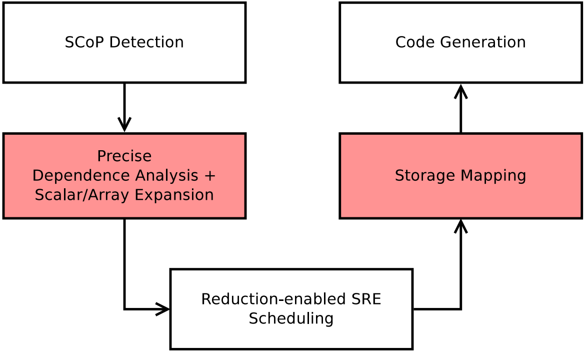
```
sums[N+1] = sum;
for (i = 0; i <= N; i++) {

  A[i] = i;
  sums[i] = sums[i+1] + A[i];

}
sum = sums[0];
```

G. Gupta, S. Rajopadhye, and P. Quinton. Scheduling reductions on realistic machines. In Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '02, pages 117–126, New York, NY, USA, 2002. ACM.

Reductions


SCoP Detection
Code Generation
Precise
Dependence Analysis +
Scalar/Array Expansion
Storage Mapping
Reduction-enabled SRE
Scheduling

Objectives & Challenges

Objectives & Challenges

Objectives

1) Detect general reduction computations
2) Parallelize/Vectorize reductions efficently
3) Interchange the order reductions are computed

Objectives & Challenges

Objectives

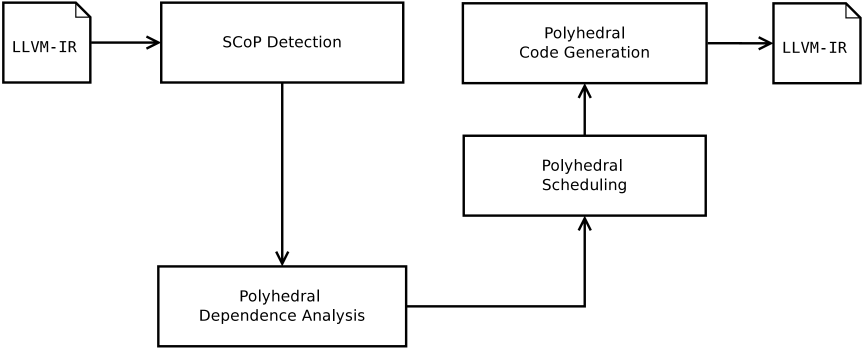
1) Detect general reduction computations
2) Parallelize/Vectorize reductions efficently
3) Interchange the order reductions are computed

Practical Considerations

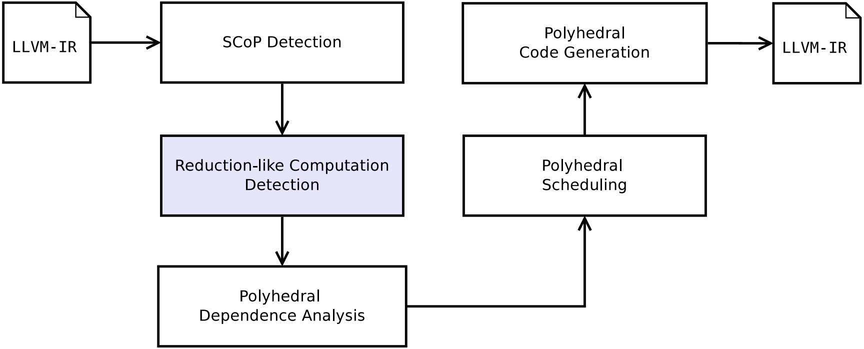
a) Avoid runtime regressions
b) Minimize memory overhead
c) Minimize compile time overhead

Overview — Polly in LLVM

Overview — Polly in LLVM



Overview — Polly in LLVM



Reduction-like Computations

Reduction-like Computations

- Updates on the same memory cells
- Associative & commutative computations
- Locally not observed or intervened

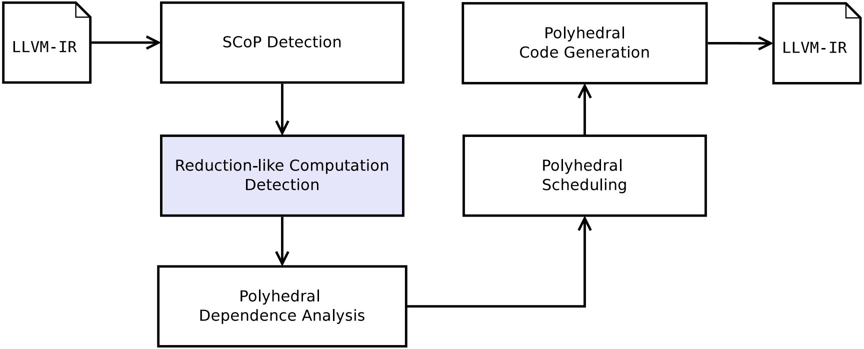
Reduction-like Computations

Reduction-like Computations

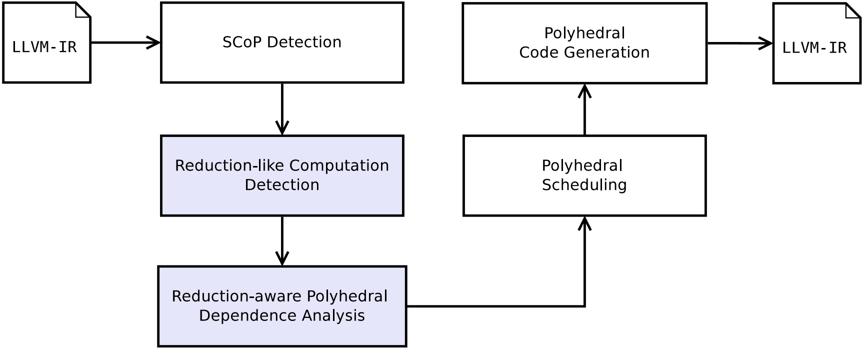
- Updates on the same memory cells
- Associative & commutative computations
- Locally not observed or intervened

Details are provided in the paper.

Overview — Polly in LLVM



Overview — Polly in LLVM



Reduction Dependences

Reduction Dependences

- Loop carried self dependences
- Induced by reduction-like computations
- Inherit “associative” & “commutative” properties

W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. ACM Trans. Program. Lang. Syst., 16(4):1248–1278,

Reduction Dependences

```
    int f(int *A, int N) {
      int sum = 0;
      for (int i = 0; i < N; i++)
S:    {
        ...;
        sum += A[i];
        ...;
      }
      return sum;
    }
```

Dependence Analysis

- Performed on statement level
- Computes value-based dependences

Reduction Dependences

```
    int f(int *A, int N) {
      int sum = 0;
      for (int i = 0; i < N; i++)
      {
S:      ...;
R:      sum += A[i];
S:      ...;
      }
      return sum;
    }
```

Dependence Analysis

- Performed on statement level
- Computes value-based dependences

Reduction Dependence Analysis

- Isolates the load & store of reduction-like computations
- Performed both on access and statement level
- Identifies reuse of values by a reduction-like computation

Reduction Dependences

```
    int f(int *A, int N) {
      int sum = 0;
      for (int i = 0; i < N; i++)
S:      sum += A[i];
      return sum;
    }
```

Dependences

$\{\mathtt{Stmt\_S}[\mathtt{i0}] \to \mathtt{Stmt\_S}[1+\mathtt{i0}] : \mathtt{i0} >= 0 \text{ and } \mathtt{i0} <= \mathtt{N}-1\}$

Reduction Dependences

```
    int f(int *A, int N) {
      int sum = 0;
      for (int i = 0; i < N; i++)
R:      sum += A[i];
      return sum;
    }
```

Dependences

$\{ \quad \}$

Reduction Dependences

$\{\texttt{Stmt\_R}[\texttt{i0}] \rightarrow \texttt{Stmt\_R}[1 + \texttt{i0}] : \texttt{i0} >= 0 \text{ and } \texttt{i0} <= \texttt{N} - 1\}$

Reduction Dependences

```
    int f(int *A, int N) {
      int sum = 0;
      for (int i = 0; i < N; i++)
S:    {
        A[i] = A[i] + A[i - 1];
        sum += i;
        A[i - 1] = A[i] + A[i - 2];
      }
      return sum;
    }
```

Dependences

$\{\texttt{Stmt\_S[i0]} \rightarrow \texttt{Stmt\_S}[1+\texttt{i0}] : \texttt{i0} >= 0 \text{ and } \texttt{i0} <= \texttt{N}-1\}$

Reduction Dependences

```
    int f(int *A, int N) {
      int sum = 0;
      for (int i = 0; i < N; i++)
      {
S:      A[i] = A[i] + A[i - 1];
R:      sum += i;
S:      A[i - 1] = A[i] + A[i - 2];
      }
      return sum;
    }
```

Dependences

$\{\mathtt{Stmt\_S}[\mathtt{i0}] \to \mathtt{Stmt\_S}[1 + \mathtt{i0}] : \mathtt{i0} >= 0 \text{ and } \mathtt{i0} <= \mathtt{N} - 1\}$

Reduction Dependences

$\{\mathtt{Stmt\_R}[\mathtt{i0}] \to \mathtt{Stmt\_R}[1 + \mathtt{i0}] : \mathtt{i0} >= 0 \text{ and } \mathtt{i0} <= \mathtt{N} - 1\}$

Reduction Dependences

```
    void bicg(float q[NX], ...) {
      for (int i = 0; i < NX; i++) {
S:      q[i] = 0;
        for (int j = 0; j < NY; j++)
T:      {
          q[i] = q[i] + A[i][j] * p[j];
          s[j] = s[j] + r[i] * A[i][j];
        }
      }
    }
```

Dependences

{Stmt_S[i0] → Stmt_T[i0, 0] : . . . ;
Stmt_T[i0, i1] → Stmt_T[i0, 1 + i1] : . . . ;
Stmt_T[i0, i1] → Stmt_T[1 + i0, i1] : . . . }

Reduction Dependences

```
    void bicg(float q[NX], ...) {
      for (int i = 0; i < NX; i++) {
S:      q[i] = 0;
        for (int j = 0; j < NY; j++)
        {
R1:       q[i] = q[i] + A[i][j] * p[j];
R2:       s[j] = s[j] + r[i] * A[i][j];
        }
      }
    }
```

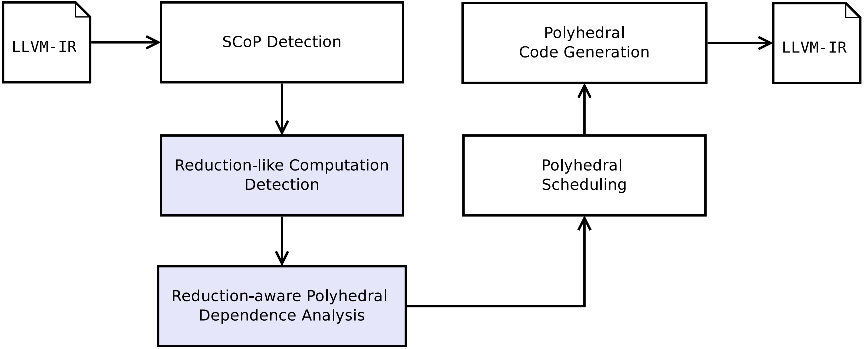
Dependences

{Stmt_S[i0] → Stmt_R1[i0, 0] : . . . }

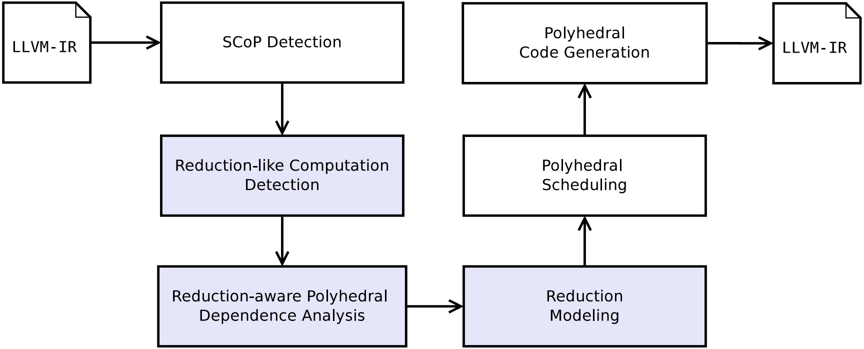
Reduction Dependences

{Stmt_R1[i0, i1] → Stmt_R1[i0, 1 + i1] : . . . ;
Stmt_R2[i0, i1] → Stmt_R2[1 + i0, i1] : . . . }

Overview — Polly in LLVM



Overview — Polly in LLVM



Reduction Modeling

Reduction Modeling

Reduction-enabled Code Generation

- Keep the polyhedral representation
- Perform parallelism check *with* and *without* reduction dependences

Reduction Modeling

Reduction-enabled Code Generation

- Keep the polyhedral representation
- Perform parallelism check *with* and *without* reduction dependences

Reduction-enabled Scheduling

- Ignore reduction dependences during the scheduling
- May need additional *privatization dependences*

Reduction Modeling

Reduction-enabled Code Generation

- Keep the polyhedral representation
- Perform parallelism check *with* and *without* reduction dependences

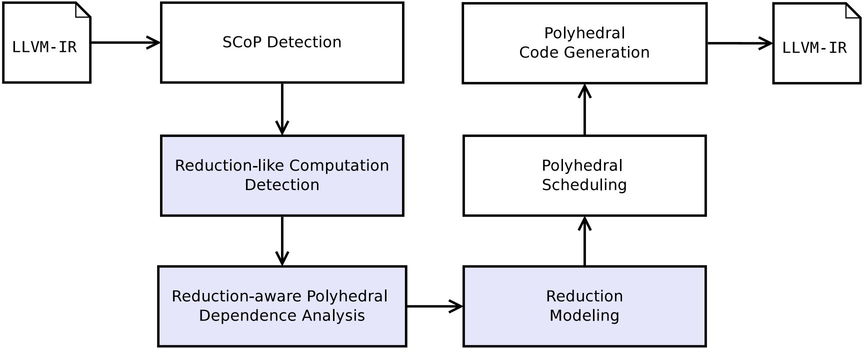
Reduction-enabled Scheduling

- Ignore reduction dependences during the scheduling
- May need additional *privatization dependences*

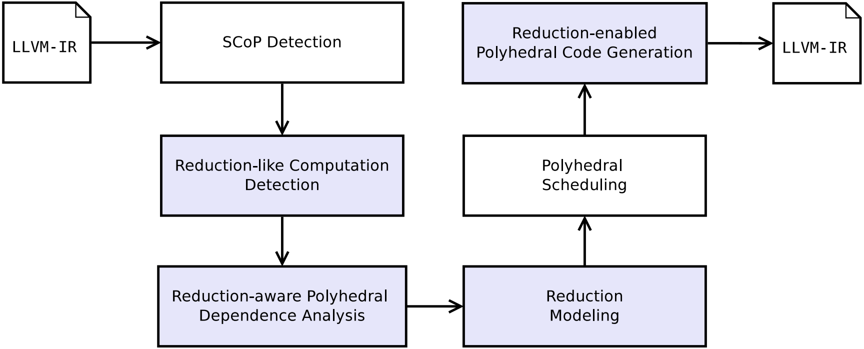
Reduction-aware Scheduling

- Let the scheduler make the parallelization decision based on the environment and the potential cost of privatization

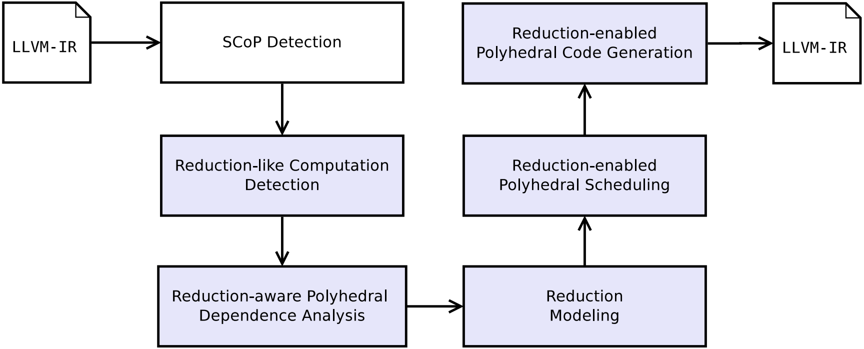
Overview — Polly in LLVM



Overview — Polly in LLVM



Overview — Polly in LLVM



Reduction-enabled Scheduling

```
    void bicg(float q[NX], ...) {
      for (int i = 0; i < NX; i++) {
S:      q[i] = 0;
        for (int j = 0; j < NY; j++)
        {
R1:       q[i] = q[i] + A[i][j] * p[j];
R2:       s[j] = s[j] + r[i] * A[i][j];
        }
      }
    }
```

Dependences

$\{\texttt{Stmt\_S}[i0] \rightarrow \texttt{Stmt\_R1}[i0, 0] : i0 >= 0 \text{ and } i0 <= NX\}$

Reduction Dependences

$\{\texttt{Stmt\_R1}[i0, i1] \rightarrow \texttt{Stmt\_R1}[i0, 1 + i1] : \ldots\}$
$\{\texttt{Stmt\_R2}[i0, i1] \rightarrow \texttt{Stmt\_R2}[1 + i0, i1] : \ldots\}$

Reduction-enabled Scheduling

Privatization Dependences

- Transitive extension along reduction dependences
- Already contained in memory based dependences
- Order reduction computations and others on the same memory cells

Reduction-enabled Scheduling

```
    void bicg(float q[NX], ...) {
      for (int i = 0; i < NX; i++) {
S:      q[i] = 0;
        for (int j = 0; j < NY; j++)
        {
R1:       q[i] = q[i] + A[i][j] * p[j];
R2:       s[j] = s[j] + r[i] * A[i][j];
        }
      }
    }
```

Dependences

$\{\mathtt{Stmt\_S}[\mathtt{i0}] \rightarrow \mathtt{Stmt\_R1}[\mathtt{i0}, 0] : \mathtt{i0} >= 0 \text{ and } \mathtt{i0} <= \mathtt{NX}\}$

Reduction Dependences

$\{\mathtt{Stmt\_R1}[\mathtt{i0}, \mathtt{i1}] \rightarrow \mathtt{Stmt\_R1}[\mathtt{i0}, 1 + \mathtt{i1}] : \ldots\}$
$\{\mathtt{Stmt\_R2}[\mathtt{i0}, \mathtt{i1}] \rightarrow \mathtt{Stmt\_R2}[1 + \mathtt{i0}, \mathtt{i1}] : \ldots\}$

Reduction-enabled Scheduling

```
    void bicg(float q[NX], ...) {
      for (int i = 0; i < NX; i++) {
S:      q[i] = 0;
        for (int j = 0; j < NY; j++)
        {
R1:       q[i] = q[i] + A[i][j] * p[j];
R2:       s[j] = s[j] + r[i] * A[i][j];
        }
      }
    }
```

Dependences

$\{\texttt{Stmt\_S}[\texttt{i0}] \to \texttt{Stmt\_R1}[\texttt{i0}, 0] : \texttt{i0} >= 0 \text{ and } \texttt{i0} <= \texttt{NX}\}$

Reduction Dependences

$\{\texttt{Stmt\_R1}[\texttt{i0}, \texttt{i1}] \to \texttt{Stmt\_R1}[\texttt{i0}, 1 + \texttt{i1}] : \ldots\}$
$\{\texttt{Stmt\_R2}[\texttt{i0}, \texttt{i1}] \to \texttt{Stmt\_R2}[1 + \texttt{i0}, \texttt{i1}] : \ldots\}$

Privatization Dependences

$\{\texttt{Stmt\_S}[\texttt{i0}] \to \texttt{Stmt\_R1}[\texttt{i0}, \texttt{o0}] : \texttt{o0} >= 1 \text{ and } \texttt{o0} <= \texttt{NY} - 1 \text{ and } \texttt{i0} >= 0 \text{ and } \texttt{i0} <= \texttt{NX}\}$

Evaluation — Compile Time

Evaluation — Compile Time

Statement-wise Dependence Analysis

- Standard value-based dependence analysis in Polly

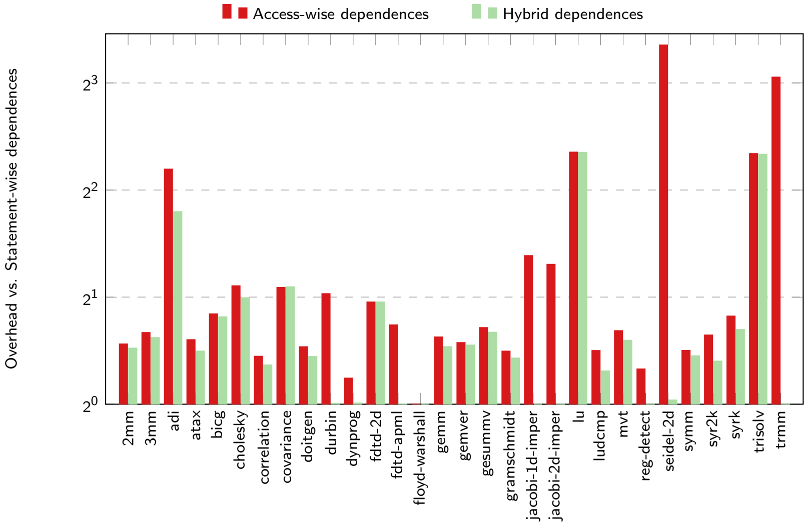
Hybrid Dependence Analysis

- Adds 85% in average — takes up to 5× as long

Access-wise Dependence Analysis

- Adds ∼ 170% in average — takes up to 10× as long

Evaluation — Compile Time



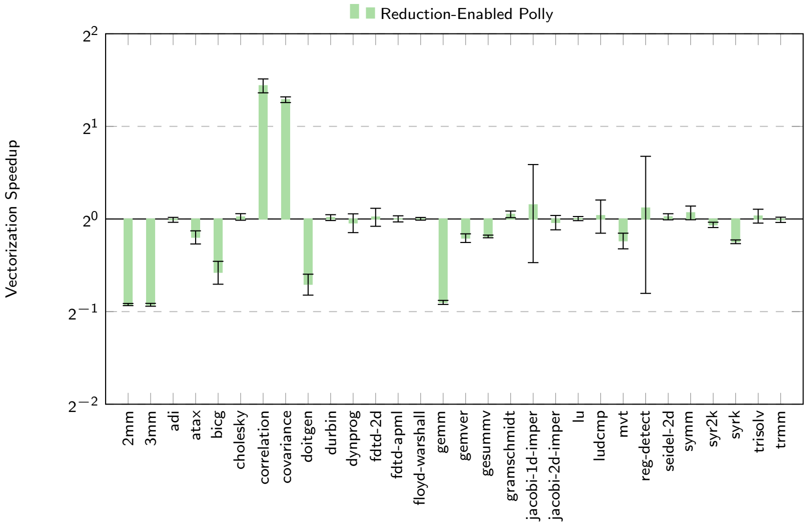
Evaluation — Runtime

Evaluation — Runtime

Runtime Evaluation Notes

- Polly's heuristic to choose a vector dimension is underdeveloped
- The LLVM vectorizer can treat simple innermost (scalar) reductions
- Polybench is highly parallel $\rightarrow$ reduction parallelism is almost never needed

Evaluation — Runtime



Conclusion

Conclusion

```
    void bicg(float q[NX], ...) {
      for (int i = 0; i < NX; i++) {
S:      q[i] = 0;
        for (int j = 0; j < NY; j++)
        {
R1:       q[i] = q[i] + A[i][j] * p[j];
R2:       s[j] = s[j] + r[i] * A[i][j];
        }
      }
    }
```

Dependences:

$\{\texttt{Stmt\_S}[\texttt{i0}] \rightarrow \texttt{Stmt\_R1}[\texttt{i0}, 0] : \texttt{i0} >= 0 \text{ and } \texttt{i0} <= \texttt{NX}\}$

Reduction Dependences:

$\{\texttt{Stmt\_R1}[\texttt{i0}, \texttt{i1}] \rightarrow \texttt{Stmt\_R1}[\texttt{i0}, 1 + \texttt{i1}] : \ldots\}$
$\{\texttt{Stmt\_R2}[\texttt{i0}, \texttt{i1}] \rightarrow \texttt{Stmt\_R2}[1 + \texttt{i0}, \texttt{i1}] : \ldots\}$

Privatization Dependences:

$\{\texttt{Stmt\_S}[\texttt{i0}] \rightarrow \texttt{Stmt\_R1}[\texttt{i0}, \texttt{o0}] : \texttt{o0} >= 1 \text{ and } \texttt{o0} <= \texttt{NY} - 1 \text{ and } \texttt{i0} >= 0 \text{ and } \texttt{i0} <= \texttt{NX}\}$

Conclusion

```
      void bicg(float q[NX], ...) {
        for (int i = 0; i < NX; i++) {
S:        q[i] = 0;
          for (int j = 0; j < NY; j++)
          {
R1:         q[i] = q[i] + A[i][j] * p[j];
R2:         s[j] = s[j] + r[i] * A[i][j];
          }
        }
      }
```

Dependences:

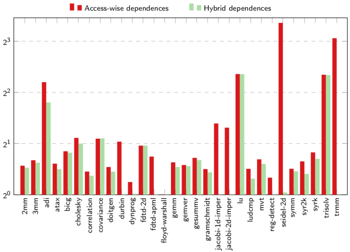
{Stmt_S[i0] → Stmt_R1[i0, 0] : i0 >= 0 and i0 <= NX}

Reduction Dependences:

{Stmt_R1[i0, i1] → Stmt_R1[i0, 1 + i1] : . . . }
{Stmt_R2[i0, i1] → Stmt_R2[1 + i0, i1] : . . . }

Privatization Dependences:

{Stmt_S[i0] → Stmt_R1[i0, o0] : o0 >= 1 and o0 <= NY − 1 and i0 >= 0 and i0 <= NX}



Conclusion

```
     void bicg(float q[NX], ...) {
       for (int i = 0; i < NX; i++) {
S:       q[i] = 0;
         for (int j = 0; j < NY; j++)
         {
R1:        q[i] = q[i] + A[i][j] * p[j];
R2:        s[j] = s[j] + r[i] * A[i][j];
         }
       }
     }
```

Dependences:

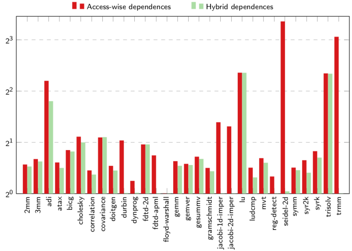
{Stmt_S[i0] → Stmt_R1[i0, 0] : i0 >= 0 and i0 <= NX}

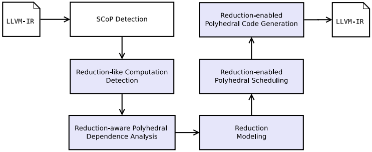
Reduction Dependences:

{Stmt_R1[i0, i1] → Stmt_R1[i0, 1 + i1] : . . . }
{Stmt_R2[i0, i1] → Stmt_R2[1 + i0, i1] : . . . }

Privatization Dependences:

{Stmt_S[i0] → Stmt_R1[i0, o0] : o0 >= 1 and o0 <= NY − 1 and i0 >= 0 and i0 <= NX}





Conclusion

```
    void bicg(float q[NX], ...) {
      for (int i = 0; i < NX; i++) {
S:      q[i] = 0;
        for (int j = 0; j < NY; j++)
        {
R1:       q[i] = q[i] + A[i][j] * p[j];
R2:       s[j] = s[j] + r[i] * A[i][j];
        }
      }
    }
```

Dependences:

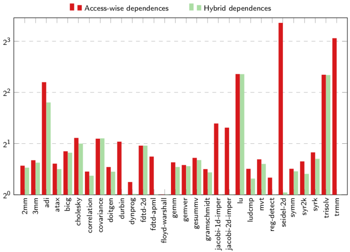
{Stmt_S[i0] → Stmt_R1[i0, 0] : i0 >= 0 and i0 <= NX}

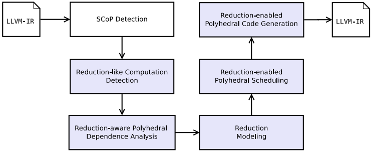
Reduction Dependences:

{Stmt_R1[i0, i1] → Stmt_R1[i0, 1 + i1] : . . . }
{Stmt_R2[i0, i1] → Stmt_R2[1 + i0, i1] : . . . }

Privatization Dependences:

{Stmt_S[i0] → Stmt_R1[i0, o0] : o0 >= 1 and o0 <= NY − 1
and i0 >= 0 and i0 <= NX}





Thank You.

Extensions

```
for (i = 0; i < N; i++) {
  S(i);
  last = f(i);
}
```

Unary Reductions

- Induce only WAW dependences
- Can be reordered or parallelized
- Only the last value needs to be recovered

Extensions

```
for (i = 0; i < N; i++) {
  sum += A[i];
  S(i);
  sum += B[i];
}
```

Multiple Statement Reductions

- Allowed between "compatible" reductions
- Induce dependence cycles, no self dependences
- Complicate efficent code generation/privatization

Extensions

```
for (i = 0; i < N; i++)
  A[i] = A[i] + A[i-1];
```

Scans/Recurrences

- Induce only RAW dependences
- Cannot be reordered but parallelized
- Different code generation than reductions

Reduction-like Computation Detection

```c
int f(int *A, int N) {
  int sum = 0;
  for (int i = 0; i < N; i++)

    sum += A[i];


  return sum;
}
```

```llvm
define i32 @f(i32* %A, i32 %N) {
entry:
  %sum = alloca i32
  store i32* %sum, i32 0
  br label %for.cond

for.cond:
  %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
  %cmp = icmp slt i32 %iv, %N
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %idx = getelementptr inbounds i32* %A, i32 %iv
  %tmp1 = load i32* %idx, align 4
  %sum.reload = load i32* %sum
  %add = add nsw i32 %sum.reload, %tmp1
  store i32* %sum, i32 %add


  %iv.next = add nuw nsw i32 %iv, 1
  br label %for.cond

for.end:
  %sum.reload3 = load i32* %sum
  ret i32 %sum.reload3
}
```

Reduction-like Computation Detection

```c
int f(int *A, int N) {
  int sum = 0;
  for (int i = 0; i < N; i++) {

    sum += A[i];
    A[i] = sum;
  }
  return sum;
}
```

```llvm
define i32 @f(i32* %A, i32 %N) {
entry:
  %sum = alloca i32
  store i32* %sum, i32 0
  br label %for.cond

for.cond:
  %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
  %cmp = icmp slt i32 %iv, %N
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %idx = getelementptr inbounds i32* %A, i32 %iv
  %tmp1 = load i32* %idx, align 4
  %sum.reload = load i32* %sum
  %add = add nsw i32 %sum.reload, %tmp1
  store i32* %sum, i32 %add

  store i32* %idx, i32 %add
  %iv.next = add nuw nsw i32 %iv, 1
  br label %for.cond

for.end:
  %sum.reload3 = load i32* %sum
  ret i32 %sum.reload3
}
```

Reduction-like Computation Detection

```c
int f(int *A, int N) {
  int sum = 0;
  for (int i = 0; i < N; i++) {
    int tmp = sum;
    sum += A[i];
    A[i] = tmp;
  }
  return sum;
}
```

```llvm
define i32 @f(i32* %A, i32 %N) {
entry:
  %sum = alloca i32
  store i32* %sum, i32 0
  br label %for.cond

for.cond:
  %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
  %cmp = icmp slt i32 %iv, %N
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %idx = getelementptr inbounds i32* %A, i32 %iv
  %tmp1 = load i32* %idx, align 4
  %sum.reload = load i32* %sum
  %add = add nsw i32 %sum.reload, %tmp1
  store i32* %sum, i32 %add

  store i32* %idx, i32 %sum.reload
  %iv.next = add nuw nsw i32 %iv, 1
  br label %for.cond

for.end:
  %sum.reload3 = load i32* %sum
  ret i32 %sum.reload3
}
```

Reduction-like Computation Detection

```c
int f(int *A, int N) {
  int sum = 0;
  for (int i = 0; i < N; i++) {

    sum += A[i];
    A[i] = sum;
  }
  return sum;
}
```

```llvm
define i32 @f(i32* %A, i32 %N) {
entry:
  %sum = alloca i32
  store i32* %sum, i32 0
  br label %for.cond

for.cond:
  %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
  %cmp = icmp slt i32 %iv, %N
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %idx = getelementptr inbounds i32* %A, i32 %iv
  %tmp1 = load i32* %idx, align 4
  %sum.reload = load i32* %sum
  %add = add nsw i32 %sum.reload, %tmp1
  store i32* %sum, i32 %add
  %sum.reload2 = load i32* %sum
  store i32* %idx, i32 %sum.reload2
  %iv.next = add nuw nsw i32 %iv, 1
  br label %for.cond

for.end:
  %sum.reload3 = load i32* %sum
  ret i32 %sum.reload3
}
```

Reduction-like Computation vs. Reduction Dependences

```c
     int f(int *A, int N) {
       int sum = 0;
       for (int i = 0; i < N ; i++) {
R1:      sum = sum * 3;
         S(i);
R2:      sum = sum + A[i];
       }
       return sum;
     }
```

```llvm
define i32 @f(i32* %A, i32 %N) {
entry:
  %sum = alloca i32
  store i32* %sum, i32 0
  br label %for.cond

for.cond:
  %iv = phi i32 [ 0, %entry ], [ %iv.next, %for.body ]
  %cmp = icmp slt i32 %iv, %N
  br i1 %cmp, label %Stmt.R1, label %for.end

Stmt.R1:
  %sum.reload = load i32* %sum
  %mul = mul nsw i32 %sum.reload, 3
  store i32* %sum, i32 %mul
  br label %Stmt.S

Stmt.S:
  ...
  br label %Stmt.R2

Stmt.R2:
  %idx = getelementptr inbounds i32* %A, i32 %iv
  %tmp1 = load i32* %idx, align 4
  %sum.reload2 = load i32* %sum
  %add = add nsw i32 %sum.reload2, %tmp1
  store i32* %sum, i32 %add
  %iv.next = add nuw nsw i32 %iv, 1
  br label %for.cond

for.end:
  %sum.reload3 = load i32* %sum
  ret i32 %sum.reload3
}
```

Reduction-aware Scheduling by Hand

```
     void f(int *A, long n) {
       for (long i = 0; i < 2 * n; i++)
S0:      A[0] += i;
       for (long i = 0; i < 2 * n; i++)
S1:      A[i + 1] = 1;
     }
```

Reduction-aware Scheduling by Hand

```
      void f(int *A, long n) {
        for (long i = 0; i < 2 * n; i++)
S0:       A[0] += i;
        for (long i = 0; i < 2 * n; i++)
S1:       A[i + 1] = 1;
      }
```

Schedule:

$$[n] \rightarrow \{Stmt\_S0[i0] \rightarrow scattering[0, -i0, 0] : i0\%2 = 0;$$
$$Stmt\_S0[i0] \rightarrow scattering[2, \quad i0, 0] : i0\%2 = 1\};$$
$$[n] \rightarrow \{Stmt\_S1[i0] \rightarrow scattering[1, \quad i0, 0]\}$$

Reduction-aware Scheduling by Hand

```
      void f(int *A, long n) {
        for (long i = 0; i < 2 * n; i++)
S0:       A[0] += i;
        for (long i = 0; i < 2 * n; i++)
S1:       A[i + 1] = 1;
      }
```

Schedule:

$$[n] \rightarrow \{Stmt\_S0[i0] \rightarrow scattering[0, -i0, 0] : i0\%2 = 0; \; Stmt\_S0[i0] \rightarrow scattering[2, i0, 0] : i0\%2 = 1\};$$
$$[n] \rightarrow \{Stmt\_S1[i0] \rightarrow scattering[1, i0, 0]\}$$

```
#pragma known-parallel reduction
for (int c0 = 0; c0 <= 2; c0 += 1) {
  if (c0 == 2) {
    #pragma simd reduction
    for (int c1 = 1; c1 < 2 * n; c1 += 2)
      Stmt_S0(c1);
  } else if (c0 == 1) {
    #pragma simd
    for (int c1 = 0; c1 < 2 * n; c1 += 1)
      Stmt_S1(c1);
  } else
    #pragma simd reduction
    for (int c1 = -2 * n + 2; c1 <= 0; c1 += 2)
      Stmt_S0(-c1);
}
```