Assumption Tracking for Optimistic Optimizations

Johannes Doerfert ¹  Tobias Grosser ²  Sebastian Pop ³

¹Saarbrücken Graduate School of Computer Science
   Saarland University
   Saarbrücken, Germany

²Department of Computer Science
   ETH Zürich
   Zürich, Switzerland

³Samsung Austin R&D Center
   Austin, TX, USA

November 15, 2015
for (k = 1; k <= M; k++) {
    mc[k] = mpp[k - 1] + tpmm[k - 1];
    if ((sc = ip[k - 1] + tpim[k - 1]) > mc[k]) mc[k] = sc;
    if ((sc = dpp[k - 1] + tpdm[k - 1]) > mc[k]) mc[k] = sc;
    if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
    mc[k] += ms[k];
    if (mc[k] < -INFTY) mc[k] = -INFTY;
}

dc[k] = dc[k - 1] + tpdd[k - 1];
if ((sc = mc[k - 1] + tpmd[k - 1]) > dc[k]) dc[k] = sc;
if (dc[k] < -INFTY) dc[k] = -INFTY;

if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
}
}
#pragma clang loop vectorize(enable)

```c
for (k = 1; k <= M; k++) {
    mc[k] = mpp[k - 1] + tpmm[k - 1];
    if ((sc = ip[k - 1] + tpim[k - 1]) > mc[k]) mc[k] = sc;
    if ((sc = dpp[k - 1] + tpdm[k - 1]) > mc[k]) mc[k] = sc;
    if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
    mc[k] += ms[k];
    if (mc[k] < -INFTY) mc[k] = -INFTY;
}

for (k = 1; k <= M; k++) {
    dc[k] = dc[k - 1] + tpdd[k - 1];
    if ((sc = mc[k - 1] + tpmd[k - 1]) > dc[k]) dc[k] = sc;
    if (dc[k] < -INFTY) dc[k] = -INFTY;
    if (k < M) {
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
        ic[k] += is[k];
        if (ic[k] < -INFTY) ic[k] = -INFTY;
    }
}
```

+ up to 30% speedup
for (k = 1; k <= M; k++) {
    mc[k] = mpp[k - 1] + tpmm[k - 1];
    if ((sc = ip[k - 1] + tpim[k - 1]) > mc[k]) mc[k] = sc;
    if ((sc = dpp[k - 1] + tpdm[k - 1]) > mc[k]) mc[k] = sc;
    if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
    mc[k] += ms[k];
    if (mc[k] < -INFTY) mc[k] = -INFTY;
}
#pragma clang loop vectorize(enable)
for (k = 1; k <= M; k++) {
    if (k < M) {
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
        ic[k] += is[k];
        if (ic[k] < -INFTY) ic[k] = -INFTY;
    }
}
#pragma clang loop vectorize(enable)
for (k = 1; k <= M; k++) {
    dc[k] = dc[k - 1] + tpdd[k - 1];
    if ((sc = mc[k - 1] + tpmd[k - 1]) > dc[k]) dc[k] = sc;
    if (dc[k] < -INFTY) dc[k] = -INFTY;
}
+ up to 30% speedup
#pragma clang loop vectorize(enable)
for (k = 1; k <= M; k++) {
    mc[k] = mpp[k - 1] + tpmm[k - 1];
    if ((sc = ip[k - 1] + tpim[k - 1]) > mc[k]) mc[k] = sc;
    if ((sc = dpp[k - 1] + tpdm[k - 1]) > mc[k]) mc[k] = sc;
    if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
    mc[k] += ms[k];
    if (mc[k] < -INFTY) mc[k] = -INFTY;
}
for (k = 1; k <= M; k++) {
    dc[k] = dc[k - 1] + tpdd[k - 1];
    if ((sc = mc[k - 1] + tpmd[k - 1]) > dc[k]) dc[k] = sc;
    if (dc[k] < -INFTY) dc[k] = -INFTY;
}
#pragma clang loop vectorize(enable)
for (k = 1; k <= M; k++) {
    if (k < M) {
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
        ic[k] += is[k];
        if (ic[k] < -INFTY) ic[k] = -INFTY;
    }
}

+ up to 50% speedup
vectorized loop \Rightarrow up to 30% speedup

vectorized loops \Rightarrow up to 50% speedup

possible aliasing \Rightarrow runtime alias checks

possible dependences \Rightarrow static dependence analysis
1 vectorized loop \implies + up to 30\% speedup
1 vectorized loop $\implies$ + up to 30% speedup
2 vectorized loops $\implies$ + up to 50% speedup
1 vectorized loop $\implies$ + up to 30% speedup
2 vectorized loops $\implies$ + up to 50% speedup
possible aliasing $\implies$ - runtime alias checks
SPEC 2006 — hmmer — fast_algorithms.c

1 vectorized loop $\implies$ + up to 30% speedup
2 vectorized loops $\implies$ + up to 50% speedup
possible aliasing $\implies$ - runtime alias checks
possible dependences $\implies$ - static dependence analysis
PARSEC — blackscholes — blackscholes.c
float BlkSchlsEqEuroNoDiv(float sptprice, float strike, float rate, float volatility, float time, int otype) {
    float xD1, xD2, xDen, d1, d2, FutureValueX, NofXd1, NofXd2, NegNofXd1, NegNofXd2, Price;
    xD1 = rate + volatility * volatility; * 0.5;
    xD1 = xD1 * time;
    xD1 = xD1 + log( sptprice / strike );
    xDen = volatility * sqrt(time);
    xD1 = xD1 / xDen;
    xD2 = xD1 - xDen;
    d1 = xD1;
    d2 = xD2;
    NofXd1 = CNDF( d1 );
    NofXd2 = CNDF( d2 );
    FutureValueX = strike * ( exp( -(rate)*(time) ) );
    if (otype == 0) {
        Price = (sptprice * NofXd1) - (FutureValueX * NofXd2);
    } else {
        NegNofXd1 = (1.0 - NofXd1);
        NegNofXd2 = (1.0 - NofXd2);
        Price = (FutureValueX * NegNofXd2) - (sptprice * NegNofXd1);
    }
    return Price;
}
int bs_thread(void *tid_ptr) {
    int tid = *(int *)tid_ptr;
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);

    for (int j = 0; j < NUM_RUNS; j++)
        for (int i = start; i < end; i++)
            prices[i] = BlkSchlsEqEuroNoDiv(sptprice[i], strike[i], rate[i],
                                            volatility[i], otime[i], otype[i]);

    return 0;
}
int bs_thread(void *tid_ptr) {
    int tid = *(int *)tid_ptr;
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);

    for (int j = 0; j < NUM_RUNS; j++)
        for (int i = start; i < end; i++)
            prices[i] = BlkSchlsEqEuroNoDiv(sptprice[i], strike[i], rate[i],
                                             volatility[i], otime[i], otype[i]);

    return 0;
}

+ 2.9× speedup for manual parallelization on a quad-core i7
int bs_thread(void *tid_ptr) {
    int tid = *(int *)tid_ptr;
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);

    for (int j = 0; j < NUM_RUNS; j++)
        for (int i = start; i < end; i++)
            prices[i] = BlkSchlsEqEuroNoDiv(sptprice[i], strike[i], rate[i],
                                          volatility[i], otime[i], otype[i]);

    return 0;
}

+ 2.9× speedup for manual parallelization on a quad-core i7
+ 2.8× speedup for automatic parallelization on a quad-core i7
int bs_thread(void *tid_ptr) {
    int tid = *(int *)tid_ptr;
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);

    for (int j = 0; j < NUM_RUNS; j++)
        for (int i = start; i < end; i++)
            prices[i] = BlkSchlsEqEuroNoDiv(sptprice[i], strike[i], rate[i],
                                             volatility[i], otime[i], otype[i]);

    return 0;
}

+ 2.9× speedup for manual parallelization on a quad-core i7
+ 2.8× speedup for automatic parallelization on a quad-core i7

- Possible aliasing arrays
int bs_thread(void *tid_ptr) {
    int tid = *(int *)tid_ptr;
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);

    for (int j = 0; j < NUM_RUNS; j++)
        for (int i = start; i < end; i++)
            prices[i] = BlkSchlsEqEuroNoDiv(sptprice[i], strike[i], rate[i],
                                             volatility[i], otime[i], otype[i]);
    return 0;
}

+ 2.9× speedup for manual parallelization on a quad-core i7
+ 2.8× speedup for automatic parallelization on a quad-core i7

- Possible aliasing arrays
- Possible execution of non-pure calls
int bs_thread(void *tid_ptr) {
    int tid = *(int *)tid_ptr;
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);

    for (int j = 0; j < NUM_RUNS; j++)
        for (int i = start; i < end; i++)
            prices[i] = BlkSchlsEqEuroNoDiv(sptprice[i], strike[i], rate[i],
                                            volatility[i], otime[i], otype[i]);

    return 0;
}

+ 2.9× speedup for manual parallelization on a quad-core i7
+ 2.8× speedup for automatic parallelization on a quad-core i7

- Possible aliasing arrays
- Possible execution of non-pure calls
- Possible execution of dead-iterations (0 ≤ j < NUM_RUNS - 1)
int bs_thread(void *tid_ptr) {
    int tid = *(int *)tid_ptr;
    int start = tid * (numOptions / nThreads);
    int end = start + (numOptions / nThreads);

    for (int j = 0; j < NUM_RUNS; j++)
        for (int i = start; i < end; i++)
            prices[i] = BlkSchlsEqEuroNoDiv(sptprice[i], strike[i], rate[i],
                                             volatility[i], otime[i], otype[i]);

    return 0;
}

+ 2.9× speedup for manual parallelization on a quad-core i7
+ 2.8× speedup for automatic parallelization on a quad-core i7
+ 6.5× speedup for sequential execution (native input)
- Possible aliasing arrays
- Possible execution of non-pure calls
- Possible execution of dead-iterations (0 <= j < NUM_RUNS - 1)
NAS Parallel Benchmarks — BT — rhs.c
void compute_rhs() {
    int i, j, k, m;
    double rho_inv, uijk, up1, um1, vijk, vp1, vm1, wijk, wp1, wm1;

    if (timeron) timer_start(t_rhs);

    for (k = 0; k <= grid_points[2]-1; k++) {
        for (j = 0; j <= grid_points[1]-1; j++) {
            for (i = 0; i <= grid_points[0]-1; i++) {
                rho_inv = 1.0/u[k][j][i][0];
                rho_i[k][j][i] = rho_inv;
                us[k][j][i] = u[k][j][i][1] * rho_inv;
                vs[k][j][i] = u[k][j][i][2] * rho_inv;
                ws[k][j][i] = u[k][j][i][3] * rho_inv;
                square[k][j][i] = 0.5* (u[k][j][i][1]*u[k][j][i][1] +
                                       u[k][j][i][2]*u[k][j][i][2] +
                                       u[k][j][i][3]*u[k][j][i][3]) * rho_inv;
                qs[k][j][i] = square[k][j][i] * rho_inv;
            }
        }
    }
}
NAS Parallel Benchmarks — BT — rhs.c

```c
for (k = 0; k <= grid_points[2] - 1; k++) {
    for (j = 0; j <= grid_points[1] - 1; j++) {
        for (i = 0; i <= grid_points[0] - 1; i++) {
            for (m = 0; m < 5; m++) {
                rhs[k][j][i][m] = forcing[k][j][i][m];
            }
        }
    }
}

if (timeron) timer_start(t_rhsx);

for (k = 1; k <= grid_points[2] - 2; k++) {
    for (j = 1; j <= grid_points[1] - 2; j++) {
        for (i = 1; i <= grid_points[0] - 2; i++) {
            uijk = us[k][j][i];
            up1 = us[k][j][i+1];
            um1 = us[k][j][i-1];

            rhs[k][j][i][0] = rhs[k][j][i][0] + dx1tx1 * 
            (u[k][j][i+1][0] - 2.0*u[k][j][i][0] + 
            u[k][j][i-1][0]) - 
            tx2 * (u[k][j][i+1][1] - u[k][j][i-1][1]);
        }
    }
}
```
rh[k][j][i][1] = rhs[k][j][i][1] + dx2tx1 * 
(u[k][j][i+1][1] - 2.0*u[k][j][i][1] + 
  u[k][j][i-1][1]) + 
xxcon2*con43 * (up1 - 2.0*uijk + um1) - 
  tx2 * (u[k][j][i+1][1]*up1 - 
   u[k][j][i-1][1]*um1 + 
   (u[k][j][i+1][4]- square[k][j][i+1]-
   u[k][j][i-1][4]+ square[k][j][i-1])* c2);

rh[k][j][i][2] = rhs[k][j][i][2] + dx3tx1 * 
(u[k][j][i+1][2] - 2.0*u[k][j][i][2] + 
  u[k][j][i-1][2]) + 
xxcon2 * (vs[k][j][i+1] - 2.0*vs[k][j][i] + 
  vs[k][j][i-1]) - 
  tx2 * (u[k][j][i+1][2]*up1 - u[k][j][i-1][2]*um1);

rh[k][j][i][3] = rhs[k][j][i][3] + dx4tx1 * 
(u[k][j][i+1][3] - 2.0*u[k][j][i][3] + 
  u[k][j][i-1][3]) + 
xxcon2 * (ws[k][j][i+1] - 2.0*ws[k][j][i] + 
  ws[k][j][i-1]) - 
  tx2 * (u[k][j][i+1][3]*up1 - u[k][j][i-1][3]*um1);

/* ≈300 more lines of similar code */
NAS Parallel Benchmarks — BT — rhs.c

```c
for (k = 0; k <= grid_points[2]-1; k++)
    for (j = 0; j <= grid_points[1]-1; j++)
        for (i = 0; i <= grid_points[0]-1; i++)
            for (m = 0; m < 5; m++)
                rhs[k][j][i][m] = forcing[k][j][i][m];

if (timeron) timer_start(t_rhsx);

for (k = 1; k <= grid_points[2]-2; k++) {
    for (j = 1; j <= grid_points[1]-2; j++) {
        for (i = 1; i <= grid_points[0]-2; i++) {
            /* ... */
```

\textsuperscript{a}Sanyam and Yew, PLDI 15
for (k = 0; k <= grid_points[2]-1; k++)
  for (j = 0; j <= grid_points[1]-1; j++)
    for (i = 0; i <= grid_points[0]-1; i++)
      for (m = 0; m < 5; m++)
        rhs[k][j][i][m] = forcing[k][j][i][m];

if (timeron) timer_start(t_rhsx);

for (k = 1; k <= grid_points[2]-2; k++) {
  for (j = 1; j <= grid_points[1]-2; j++) {
    for (i = 1; i <= grid_points[0]-2; i++) {
      /* ... */

+ 6× speedup for 8 threads/cores \(^a\)

\(^a\)Sanyam and Yew, PLDI 15
NASDAQ Parallel Benchmarks — BT — rhs.c

```c
for (k = 0; k <= grid_points[2]-1; k++)
    for (j = 0; j <= grid_points[1]-1; j++)
        for (i = 0; i <= grid_points[0]-1; i++)
            for (m = 0; m < 5; m++)
                rhs[k][j][i][m] = forcing[k][j][i][m];

if (timeron) timer_start(t_rhsx);

for (k = 1; k <= grid_points[2]-2; k++) {
    for (j = 1; j <= grid_points[1]-2; j++) {
        for (i = 1; i <= grid_points[0]-2; i++) {
            /* ... */

+ 6× speedup for 8 threads/cores
- Possible variant loop bounds
```

\(^a\)Sanyam and Yew, PLDI 15
NAS Parallel Benchmarks — BT — rhs.c

```c
for (k = 0; k <= grid_points[2]-1; k++)
    for (j = 0; j <= grid_points[1]-1; j++)
        for (i = 0; i <= grid_points[0]-1; i++)
            for (m = 0; m < 5; m++)
                rhs[k][j][i][m] = forcing[k][j][i][m];

if (timeron) timer_start(t_rhsx);

for (k = 1; k <= grid_points[2]-2; k++) {
    for (j = 1; j <= grid_points[1]-2; j++) {
        for (i = 1; i <= grid_points[0]-2; i++) {
            /* ... */
```

+ 6× speedup for 8 threads/cores
- Possible variant loop bounds
- Possible out-of-bound accesses

---

*Sanyam and Yew, PLDI 15*
NAS Parallel Benchmarks — BT — rhs.c

```c
for (k = 0; k <= grid_points[2]-1; k++)
    for (j = 0; j <= grid_points[1]-1; j++)
        for (i = 0; i <= grid_points[0]-1; i++)
            for (m = 0; m < 5; m++)
                rhs[k][j][i][m] = forcing[k][j][i][m];

if (timeron) timer_start(t_rhsx);

for (k = 1; k <= grid_points[2]-2; k++) {
    for (j = 1; j <= grid_points[1]-2; j++) {
        for (i = 1; i <= grid_points[0]-2; i++) {
            /* ... */

+ 6× speedup for 8 threads/cores\(^a\)
- Possible variant loop bounds
- Possible out-of-bound accesses
- Possible execution of non-pure calls

\(^a\)Sanyam and Yew, PLDI 15
NAS Parallel Benchmarks — BT — rhs.c

```c
for (k = 0; k <= grid_points[2] - 1; k++)
for (j = 0; j <= grid_points[1] - 1; j++)
for (i = 0; i <= grid_points[0] - 1; i++)
for (m = 0; m < 5; m++)
  rhs[k][j][i][m] = forcing[k][j][i][m];

if (timeron) timer_start(t_rhsx);

for (k = 1; k <= grid_points[2] - 2; k++) {
  for (j = 1; j <= grid_points[1] - 2; j++) {
    for (i = 1; i <= grid_points[0] - 2; i++) {
      /* ... */

+ 6× speedup for 8 threads/cores
- Possible variant loop bounds
- Possible out-of-bound accesses
- Possible execution of non-pure calls
- Possible integer under/overflows complicate loop bounds
```

\(^a\)Sanyam and Yew, PLDI 15
Be Optimistic!

Programs might be nasty but programmers are not.
When static information is insufficient

Optimistic Assumptions & Speculative Versioning

1. Take optimistic assumptions to (better) optimize loops
2. Derive simple runtime conditions that imply these assumptions
3. Version the code based on the assumptions made and conditions derived.

Speculative Versioning

if (/* Runtime Conditions */) /* Optimized Loop Nest */
else /* Original Loop Nest */
When static information is insufficient

Optimistic Assumptions & Speculative Versioning

### Optimistic Assumptions

1. Take optimistic assumptions to (better) optimize loops
When static information is insufficient

Optimistic Assumptions & Speculative Versioning

Optimistic Assumptions

1. Take optimistic assumptions to (better) optimize loops
2. Derive simple runtime conditions that imply these assumptions
When static information is insufficient
Optimistic Assumptions & Speculative Versioning

Optimistic Assumptions

1. Take optimistic assumptions to (better) optimize loops
2. Derive simple runtime conditions that imply these assumptions
3. Version the code based on the assumptions made and conditions derived.
When static information is insufficient
Optimistic Assumptions & Speculative Versioning

Optimistic Assumptions

1. Take optimistic assumptions to (better) optimize loops
2. Derive simple runtime conditions that imply these assumptions
3. Version the code based on the assumptions made and conditions derived.

Speculative Versioning

```plaintext
if (/* Runtime Conditions */)
  /* Optimized Loop Nest */
else
  /* Original Loop Nest */
```
When static information is insufficient

Runtime Conditions

- Fast to derive (compile time)
- Fast to verify (runtime)
- High probability to be true
Polly
The polyhedral loop optimizer in LLVM
Polly
The polyhedral loop optimizer in LLVM

### Polyhedral Optimizer

- Loop Nest Optimizer
- Precise compute model (affine constraints)
- Combines many classical loop optimizations
  - Tiling, Interchange, Fusion, Fission, ...
- Examples: Pluto, ppcg
Polly
The polyhedral loop optimizer in LLVM

Polly — Advantages

- **Automatic & Semi-automatic Mode**
- **Robust & Widely Applicable**
- **Embedded in LLVM**
  - Source & Target Independent
  - Information flow between Polly and other passes
void loop_bounds(int *size0, int *size1) {
    for (int i = 0; i < *size0; i++)
        for (int j = 0; j < *size1; j++)
            ...
}
void loop_bounds(int *size0, int *size1) {
    int size0val = *size0;
    int size1val = *size1;

    for (int i = 0; i < size0val; i++)
        for (int j = 0; j < size1val; j++)
            ...
}

- Hoist invariant loads
Optimistic Assumptions in Polly
Possibly Invariant Loads

```c
void loop_bounds(int *size0, int *size1) {
    int size0val = *size0;
    int size1val = 0;

    if (size0val > 0)
        size1val = *size1;

    for (int i = 0; i < size0val; i++)
        for (int j = 0; j < size1val; j++)
            ...
}
```

- Hoist invariant loads
- Keep conditions for conditionally executed loads
void loop_bounds(int *size0, int *size1) {
    int size0val = *size0;
    int size1val = 0;

    if (size0val > 0)
        size1val = *size1;

    for (int i = 0; i < size0val; i++)
        for (int j = 0; j < size1val; j++)
            ...
}

- Hoist invariant loads
- Keep conditions for conditionally executed loads
- Powerful in combination with alias checks
Loop Optimizations Today and with Polly
## Loop Optimizations Today and with Polly

<table>
<thead>
<tr>
<th>Today in LLVM</th>
<th>Polly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
<td>Inner Loops</td>
</tr>
<tr>
<td></td>
<td>(Unstructured) Loops Nests</td>
</tr>
</tbody>
</table>
## Loop Optimizations Today and with Polly

<table>
<thead>
<tr>
<th>Today in LLVM</th>
<th>Polly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
<td>Inner Loops</td>
</tr>
<tr>
<td><strong>Approach</strong></td>
<td>Seq. of Specialized Passes</td>
</tr>
</tbody>
</table>

Orthogonal strengths and weaknesses

Johannes Doerfert, Tobias Grosser, Sebastian Pop
## Loop Optimizations Today and with Polly

<table>
<thead>
<tr>
<th>Today in LLVM</th>
<th>Polly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
<td>Inner Loops</td>
</tr>
<tr>
<td><strong>Approach</strong></td>
<td>Seq. of Specialized Passes</td>
</tr>
<tr>
<td><strong>Modeling</strong></td>
<td>Individual Expressions</td>
</tr>
</tbody>
</table>

Johannes Doerfert, Tobias Grosser, Sebastian Pop
## Loop Optimizations Today and with Polly

<table>
<thead>
<tr>
<th>Today in LLVM</th>
<th>Polly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
<td>(Unstructured) Loops Nests</td>
</tr>
<tr>
<td>Inner Loops</td>
<td>Combined Modeling</td>
</tr>
<tr>
<td><strong>Approach</strong></td>
<td></td>
</tr>
<tr>
<td>Seq. of Specialized Passes</td>
<td></td>
</tr>
<tr>
<td><strong>Modeling</strong></td>
<td>Sets of Constraints</td>
</tr>
<tr>
<td>Individual Expressions</td>
<td></td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>Precise Flow-Dependences</td>
</tr>
<tr>
<td>Heuristic-based Dependences</td>
<td>Optimistic Assumptions</td>
</tr>
<tr>
<td>Instruction Cost Model</td>
<td>Memory Access Analysis</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Loop Optimizations Today and with Polly

<table>
<thead>
<tr>
<th>Today in LLVM</th>
<th>Polly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
<td>Inner Loops</td>
</tr>
<tr>
<td><strong>Approach</strong></td>
<td>Seq. of Specialized Passes</td>
</tr>
<tr>
<td><strong>Modeling</strong></td>
<td>Individual Expressions</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>Heuristic-based Dependences Instruction Cost Model</td>
</tr>
<tr>
<td><strong>Transforms</strong></td>
<td>Unroll/Reroll/Rotate/LICM Inner-loop vectorization</td>
</tr>
</tbody>
</table>
# Loop Optimizations Today and with Polly

<table>
<thead>
<tr>
<th></th>
<th><strong>Today in LLVM</strong></th>
<th><strong>Polly</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target</strong></td>
<td>Inner Loops</td>
<td>(Unstructured) Loops Nests</td>
</tr>
<tr>
<td><strong>Approach</strong></td>
<td>Seq. of Specialized Passes</td>
<td>Combined Modeling</td>
</tr>
<tr>
<td><strong>Modeling</strong></td>
<td>Individual Expressions</td>
<td>Sets of Constraints</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td>Heuristic-based Dependences</td>
<td>Precise Flow-Dependences</td>
</tr>
<tr>
<td></td>
<td>Instruction Cost Model</td>
<td>Optimistic Assumptions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory Access Analysis</td>
</tr>
<tr>
<td><strong>Transforms</strong></td>
<td>Unroll/Reroll/Rotate/LICM</td>
<td>Tiling/Fusion/Fission/Skewing</td>
</tr>
<tr>
<td></td>
<td>Inner-loop vectorization</td>
<td>Outer-loop vectorization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Auto-parallelization</td>
</tr>
</tbody>
</table>
Optimistic Assumptions in Polly

(A) Applicability/Correctness

1. No Alias Assumption
2. No Wrapping Assumption
3. Finite Loops Assumption
4. Array In-bounds Assumption
5. Valid Multidimensional View Assumption (Delinearization)
6. Possibly Invariant Loads

(B) Optimizations

1. Array In-bounds Check Hoisting
2. Parametric Dependence Distances

1 Joint work Fabrice Rastello (INRIA Grenoble) & others. [OOPSLA’15]
2 Joint work with Tobias Grosser (ETH)
3 Tobias Grosser & Sebastian Pop (Samsung) [ICS’15]
4 Joint work with Zino Benaissa (Qualcomm)
Ongoing Work

- **Polly Mainline**
  - Improved optimization choices
  - Profile guided optimization
  - More powerful checks (e.g., generate inspector loops)
  - User feedback, register tiling, user provided assumptions
  - ...
Ongoing Work

- **Polly Mainline**
  - Improved optimization choices
  - Profile guided optimization
  - More powerful checks (e.g., generate inspector loops)
  - User feedback, register tiling, user provided assumptions
  - ...
  - *Integrate Polly into LLVM mainline*
Ongoing Work

- **Polly Mainline**
  - Improved optimization choices
  - Profile guided optimization
  - More powerful checks (e.g., generate inspector loops)
  - User feedback, register tiling, user provided assumptions
  - ...
  - *Integrate Polly into LLVM mainline*

- **Independent Projects using Polly**
  - Heterogeneous Compute (OpenCL)
  - High-level Synthesis
  - Dynamic Compilation (JITs)
Ongoing Work

■ Polly Mainline
  ▶ Improved optimization choices
  ▶ Profile guided optimization
  ▶ More powerful checks (e.g., generate inspector loops)
  ▶ User feedback, register tiling, user provided assumptions
  ▶ …
  ▶ *Integrate Polly into LLVM mainline*

■ Independent Projects using Polly
  ▶ Heterogeneous Compute (OpenCL)
  ▶ High-level Synthesis
  ▶ Dynamic Compilation (JITs)

Thank You.
Polly

Incomplete feature list

- **Loops & Conditions**
  - **Styles**
    - *for/while/do/goto*
  - **Arbitrary Presburger Expressions**
    - ```
      for (i = 0; i < 22 && i < mod(i + b, 13); i += 2)
      if (5 * i + b <= 13 || 12 > b)
    ```
  - **Multiple back-edges/exit-edges & unstructured control flow**
    - *break; continue; goto; switch*
  - **Data-dependent control flow**
    - ```
      if (B[i]) A[i] = A[i] / B[i];
    ```

- **Accesses**
  - **Multi-dimensionality**: `A[][n][m] / A[][10][100]`
  - **Non-affine**: `A[i * j]`
  - **Scalar**: `x = A[i]; ...; B[i] = x;`

- **User Annotations**
  - `__builtin_assume(M > 8)`
  - `__builtin_assume(__polly_profitable == 1)`
NAS Parallel Benchmarks — BT — rhs.c

```c
clang -Rpass-analysis=polly-scops -O3 -polly rhs.c
```

```c
rhs.c:47:3: remark: SCoP begins here. [-Rpass-analysis=polly-scops]
    for (k = 0; k <= grid_points[2]-1; k++) {
^  
/* ... */
```

```c
rhs.c:418:16: remark: SCoP ends here. [-Rpass-analysis=polly-scops]
    if (timeron) timer_stop(t_rhs);
^  
```
NAS Parallel Benchmarks — BT — rhs.c

```sh
clang -Rpass-analysis=polly-scops -O3 -polly rhs.c
```

```c
rhs.c:79:16: remark: No-error assumption: [grid_points, grid_points', timeron] ->
  { : timeron = 0 } [-Rpass-analysis=polly-scops]
if (timeron) timer_start(t_rhsx);
```

```c
~
```
NAS Parallel Benchmarks — BT — rhs.c

clang -Rpass-analysis=polly-scops -O3 -polly rhs.c

rhs.c:50:23: remark: Inbounds assumption: [grid_points, grid_points’, grid_points’’] ->
{ : grid_points <= 0 or (grid_points >= 1 and grid_points’ <= 0) or (grid_points >= 1 and grid_points’ >= 104 and grid_points’’ <= 0) or (grid_points >= 1 and grid_points’ <= 103 and grid_points’’ >= 103) } [-Rpass-analysis=polly-scops]
    rho_inv = 1.0/u[k][j][i][0];

rhs.c:144:27: remark: Inbounds assumption: [grid_points, grid_points’, timeron, grid_points’’] ->
{ : grid_points <= 2 or (grid_points >= 3 and grid_points’ <= 104) } [-Rpass-analysis=polly-scops]
    rhs[k][j][i][m] = rhs[k][j][i][m] - dssp *

rhs.c:171:27: remark: Inbounds assumption: [grid_points, grid_points’, timeron, grid_points’’] ->
{ : grid_points <= 2 or (grid_points >= 3 and grid_points’ <= 2) or (grid_points >= 3 and grid_points’ <= 104 and grid_points’’ >= 3 and grid_points’’ <= 105 and grid_points’’ >= 3) } [-Rpass-analysis=polly-scops]
    rhs[k][j][i][m] = rhs[k][j][i][m] - dssp *
NAS Parallel Benchmarks — BT — rhs.c

clang -Rpass-analysis=polly-scops -O3 -polly rhs.c

rhs.c:419:1: remark: No-overflows assumption: [grid_points, grid_points’, grid_points’’, timeron] ->
{ (grid_points >= 3 and grid_points’ >= 3 and grid_points’’ >= -2147483643) or (grid_points >= 3 and 
grid_points’’ <= 2 and grid_points’ >= -2147483643 and grid_points’’ >= -2147483646) or 
(grid_points <= 2 and grid_points >= -2147483644 and grid_points’ <= 2 and grid_points’’ >= -2147483646) or 
(grid_points’ = -2147483644 and grid_points >= 3 and grid_points’’ <= 2 and grid_points’’ >= -2147483646) or 
(grid_points = -2147483644 and grid_points’ = -2147483643 and grid_points’’ <= 2 and grid_points’’ >= -2147483646) }

__builtin_assume(grid_points[0] >= -2147483643 &&
grid_points[1] >= -2147483643 &&
grid_points[2] >= -2147483643);
NAS Parallel Benchmarks — BT — rhs.c

```
clang -Rpass-analysis=polly-scops -O3 -polly rhs.c
```

```
    [-Rpass-analysis=polly-scops]
    rho_inv = 1.0/u[k][j][i][0];
  ^

    [-Rpass-analysis=polly-scops]
    u[k][j][i][1]*u[k][j][i][1] +
  ^
```
Optimistic Assumptions in Polly

Array In-bounds Assumptions

```c
void stencil(int N, int M, float A[128][128]) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
      A[j][i] += A[2*j+1][i];
}
```
void stencil(int N, int M, float A[128][128]) {
    if (N < 128) {
        for (int j = 0; j < M; j++)
            for (int i = 0; i < N; i++)
                A[j][i] += A[2*j+1][i];
    } else {
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                A[j][i] += A[2*j+1][i];
    }
}
void stencil(int N, int M, float A[128][128]) {
    if (N < 128) {
        for (int j = 0; j < M; j++)
            for (int i = 0; i < N; i++)
                A[j][i] += A[2*j+1][i];
    } else {
        for (int i = 0; i < N; i++)
            for (int j = 0; j < M; j++)
                A[j][i] += A[2*j+1][i];
    }
}

- Out-of-bound accesses introduce multiple addresses for one memory location (e.g., &A[1][0] == &A[0][128])
void mem_copy(int N, float *A, float *B) {
    #pragma vectorize
    for (i = 0; i < N; i++)
      A[i] = B[i+5];
  } else {
    /* original code */
  }
}
void mem_copy(int N, float *A, float *B) {
        #pragma vectorize
        for (i = 0; i < N; i++)
            A[i] = B[i+5];
    } else {
        /* original code */
    }
}
void mem_copy(int N, float *A, float *B) {
        #pragma vectorize
        for (i = 0; i < N; i++)
            A[i] = B[i+5];
    } else {
        /* original code */
    }
}

- Compare minimal/maximal accesses to possible aliasing arrays
Optimistic Assumptions in Polly

No Alias Assumptions

```c
void evn_odd(int N, int *Evn, int *Odd, int *A, int *B) {

                  ((&B[N] <= &Evn[0] || &Evn[(N+1)/2] <= &B[0]) &&
                   (&A[N] <= &Evn[0] || &Evn[(N+1)/2] <= &A[0]))) {

        for (int i = 0; i < N; i += 2)
            if (N % 2)
                Odd[i/2] = A[i+1] - B[i+1];
            else
                Evn[i/2] = A[i] + B[i];

    } else {
        /* original code */
    }
}
```

- Compare minimal/maximal accesses to possible aliasing arrays
Optimistic Assumptions in Polly

No Alias Assumptions

```c
void evn_odd(int N, int *Evn, int *Odd, int *A, int *B) {
    if (N%2 

    for (int i = 0; i < N; i += 2)
        if (N % 2)
            Odd[i/2] = A[i+1] - B[i+1];
        else
            Evn[i/2] = A[i] + B[i];
    } else {
        /* original code */
    }
}
```

- Compare minimal/maximal accesses to possible aliasing arrays
- Do not compare accesses to read-only arrays
Optimistic Assumptions in Polly

No Alias Assumptions

```c
void evn_odd(int N, int *Evn, int *Odd, int *A, int *B) {
        (&B[N] <= &Evn[0] || &Evn[(N+1)/2] <= &B[0]) &&
        for (int i = 0; i < N; i += 2)
            if (N % 2)
                Odd[i/2] = A[i+1] - B[i+1];
            else
                Evn[i/2] = A[i] + B[i];
    } else {
        /* original code */
    }
}
```

- Compare minimal/maximal accesses to possible aliasing arrays
- Do not compare accesses to read-only arrays
- Use the iteration domain of the accesses
void evn_odd(int N, int *Evn, int *Odd, int *A, int *B) {

        ((&B[N] <= &Evn[0] || &Evn[(N+1)/2] <= &B[0]) &&

        for (int i = 0; i < N; i += 2)
            if (N % 2)
                Odd[i/2] = A[i+1] - B[i+1];
            else
                Evn[i/2] = A[i] + B[i];

    } else {
        /* original code */
    }
}

- Compare minimal/maximal accesses to possible aliasing arrays
- Do not compare accesses to read-only arrays
- Use the iteration domain of the accesses
void mem_shift(unsigned char N, float *A) {
    if (N <= 128) {
        #pragma vectorize
        for (unsigned char i = 0; i < N; i++)
            A[i] = A[N + i];
    } else {
        /* original code */
    }
}
void mem_shift(unsigned char N, float *A) {
    if (N <= 128) {
        #pragma vectorize
        for (unsigned char i = 0; i < N; i++)
            A[i] = A[N + i];
    } else {
        /* original code */
    }
}

- Finite bit width can cause integer expressions to “wrap around”
- Wrapping causes multiple addresses for one memory location
Optimistic Assumptions in Polly

No Wrapping Assumption

\[ i \cdot c_0 + p \cdot c_1 \equiv_p (i \cdot c_0 + p \cdot c_1) \mod 2^k \]
Optimistic Assumptions in Polly

No Wrapping Assumption

\[ i \cdot c_0 + p \cdot c_1 \equiv_p (i \cdot c_0 + p \cdot c_1) \mod 2^k \]

\[ i \in [0, N-1] \land N \in [0, 2^8] \]

\[ (N + i) \equiv_p (N + i) \mod 2^8 \]
Optimistic Assumptions in Polly

No Wrapping Assumption

\[ i \cdot c_0 + p \cdot c_1 \equiv_p (i \cdot c_0 + p \cdot c_1) \mod 2^k \]

\[ i \in [0, N-1] \land N \in [0, 2^8] \]

\[ (N + i) \equiv_p (N + i) \mod 2^8 \]

\[ \Rightarrow (N + i) \leq_p 255 \]
Optimistic Assumptions in Polly

No Wrapping Assumption

\[ i \cdot c_0 + p \cdot c_1 \equiv_p (i \cdot c_0 + p \cdot c_1) \mod 2^k \]

\[ i \in [0, N-1] \land N \in [0, 2^8] \]

\[ (N + i) \equiv_p (N + i) \mod 2^8 \]

\[ \implies (N + i) \leq_p 255 \]

\[ \implies N \leq 128 \]
void mem_shift(unsigned N, float *A) {
    if (N % 2 == 0) {
        #pragma vectorize
        for (unsigned i = 0; i != N; i+=2)
            A[i+4] = A[i];
    } else {
        /* original code */
    }
}
void mem_shift(unsigned N, float *A) {
    if (N % 2 == 0) {

        #pragma vectorize
        for (unsigned i = 0; i != N; i+=2)
            A[i+4] = A[i];

    } else {
        /* original code */
    }
}

- Allows to provide other LLVM passes real loop bounds
- Infinite loops create unbounded optimization problems
#define A(x, y) A[n1 * x + y]
void set_subarray(float *A, int o0, int o1, int s0, int s1, int n0, int n1) {
    if (/* fill in bounds check */) {
        #pragma parallel
        for (int i = 0; i < s0; i++)
            for (int j = 0; j < s1; j++)
                A(o0 + i, o1 + j) = 1;
    } else {
        /* original code */
    }
}
Optimistic Assumptions in Polly

#define A(x, y) A[n1 * x + y]

void set_subarray(float *A, int o0, int o1, int s0, int s1, int n0, int n1) {
    if (o1 + s1 <= n1) {
        #pragma parallel
        for (int i = 0; i < s0; i++)
            for (int j = 0; j < s1; j++)
                A(o0 + i, o1 + j) = 1;
    }
    else {
        /* original code */
    }
}
#define A(x, y) A[n1 * x + y]

```c
void set_subarray(float *A, int o0, int o1, int s0,
                  int s1, int n0, int n1) {

    if (o1 + s1 <= n1) {
        #pragma parallel
        for (int i = 0; i < s0; i++)
            for (int j = 0; j < s1; j++)
                A(o0 + i, o1 + j) = 1;

    } else {
        /* original code */
    }
}
```

- Define multi-dimensional view of a linearized (one-dimensional) array
- Derive conditions that accesses are in-bounds for each dimension
**Optimistic Assumptions in Polly**

```c
struct SafeArray { int Size, int *Array };

inline void set(SafeArray A, int idx, int val) {
    if (idx < 0 || A.Size <= idx)
        throw OutOfBounds;
    A.Array[idx] = val;
}

void set_safe_array(int N, SafeArray A) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < i/2; j++)
            set(A, i+j, 1); /* Throws out-of-bounds */
}
```
struct SafeArray { int Size, int *Array };  

inline void set(SafeArray A, int idx, int val) {
    if (idx < 0 || A.Size <= idx)
        throw OutOfBounds;
    A.Array[idx] = val;
}

void set_safe_array(int N, SafeArray A) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < i/2; j++)
            set(A, i+j, 1); /* Throws out-of-bounds */
}
struct SafeArray { int Size, int *Array; }

inline void set(SafeArray A, int idx, int val) {
    if (idx < 0 || A.Size <= idx)
        throwOutOfBounds;
    A.Array[idx] = val;
}

void set_safe_array(int N, SafeArray A) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < i/2; j++)
            set(A, i+j, 1); /* Throws out-of-bounds */
}
void set_safe_array(int N, SafeArray A) {
    if ((3*N)/2 <= A.Size) {
        for (int i = 0; i < N; i++)
            for (int j = 0; j < i/2; j++)
                A[i+j] = 1;
    } else {
        /* original code */
    }
}
void set_safe_array(int N, SafeArray A) {
    if ((3*N)/2 <= A.Size) {
        for (int i = 0; i < N; i++)
            for (int j = 0; j < i/2; j++)
                A[i+j] = 1;
    } else {
        /* original code */
    }
}

Hoist in-bounds access conditions out of the loop nest
void copy(int N, double A[N][N], double B[N][N]) {
    if (DebugLevel <= 5) {
        #pragma parallel
        for (int i = 0; i < N; i++)
            #pragma simd
            for (int j = 0; j < N; j++)
                A[i][j] = B[i][j];
    } else {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                A[i][j] = B[i][j];
            if (DebugLevel > 5)
                printf("Column\%d\ copied\n", i)
        }
    }
}
void vectorize(int N, double *A) {
    if (c >= 4) {
        #pragma vectorize width(4)
        for (int i = c; i < N+c; i++)
            A[i-c] += A[i];
    } else {
        /* original code */
    }
}
void vectorize(int N, double *A) {
    if (c >= 4) {
        #pragma vectorize width(4)
        for (int i = c; i < N+c; i++)
            A[i-c] += A[i];
    } else {
        /* original code */
    }
}

- Assume large enough dependence distance, e.g., for vectorization
Motivation
Motivation

- Modern off-the-shelf processors are complex and powerful
  - low overhead vector units
  - multiple cores and levels of cache
Motivation

- Modern off-the-shelf processors are complex and powerful
  - low overhead vector units
  - multiple cores and levels of cache
- Programs do not exploit this power
  - not cache aware
  - written for single threaded execution
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];
    }
}

Johannes Doerfert, Tobias Grosser, Sebastian Pop
if ((ny >= 1)) {
    ub1 = floord((nx + -1), 256);
    #pragma omp parallel for private(c2, c3, c4, c5, c6) firstprivate(ub1)
    for (c1 = 0; c1 <= ub1; c1++)
        for (c2 = 0; c2 <= floord((ny + -1), 256); c2++)
            for (c3 = (8 * c1); c3 <= min(floord((nx + -1), 32), ((8 * c1) + 7)); c3++)
                for (c4 = (8 * c2); c4 <= min(floord((ny + -1), 32), ((8 * c2) + 7)); c4++)
                    for (c5 = (32 * c4); c5 <= min(((32 * c4) + 31), (ny + -1)); c5++)
                        # pragma ivdep
                        # pragma vector always
                        # pragma simd
                        for (c6 = (32 * c3); c6 <= min(((32 * c3) + 31), (nx + -1)); c6++)
                            q[c6]=q[c6]+A[c6][c5]*p[c5];
}

if ((nx >= 1)) {
    ub1 = floord((ny + -1), 256);
    #pragma omp parallel for private(c2, c3, c4, c5, c6) firstprivate(ub1)
    for (c1 = 0; c1 <= ub1; c1++)
        for (c2 = 0; c2 <= floord((nx + -1), 256); c2++)
            for (c3 = (8 * c1); c3 <= min(floord((nx + -1), 32), ((8 * c1) + 7)); c3++)
                for (c4 = (8 * c2); c4 <= min(floord((nx + -1), 32), ((8 * c2) + 7)); c4++)
                    for (c5 = (32 * c4); c5 <= min(((32 * c4) + 31), (nx + -1)); c5++)
                        # pragma ivdep
                        # pragma vector always
                        # pragma simd
                        for (c6 = (32 * c3); c6 <= min(((32 * c3) + 31), (ny + -1)); c6++)
                            s[c6]=s[c6]+r[c5]*A[c5][c6];
}
The Polyhedral Model

Input Loop Nest

for (i = 0; i <= 1024; i++) {
    for (j = 0; j <= 1024; j++) {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}

Polyhedral Abstraction
The Polyhedral Model

Input Loop Nest

```
for (i = 0; i <= 1024; i++) {
    for (j = 0; j <= 1024; j++) {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}
```

Polyhedral Abstraction
The Polyhedral Model

Input Loop Nest

for (i = 0; i <= 1024; i++) {
    for (j = 0; j <= 1024; j++) {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}

Polyhedral Abstraction
The Polyhedral Model

Input Loop Nest

```plaintext
for (i = 0; i <= 1024; i++) {
    for (j = 0; j <= 1024; j++) {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}
```

Polyhedral Abstraction
The Polyhedral Model

Input Loop Nest

for (i = 0; i <= 1024; i++) {
    for (j = 0; j <= 1024; j++) {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}

Polyhedral Abstraction
The Polyhedral Model

Original
The Polyhedral Model

Original

Transformed

Motivation
The Polyhedral Model

Original

Transformed
The Polyhedral Model

Original

Transformed
The Polyhedral Model

Original

Transformed

for (j = 0; j <= 2048; j++) {
    parfor (i = max(j - 1024, 0); i <= min(j, 1024); i++) {
        s[j - i] = s[j - i] + r[i] * A[i][j - i];
        q[i] = q[i] + A[i][j - i] * p[j - i];
    }
}
for (j = 0; j <= 2048; j++) {
    parfor (i = max(j - 1024, 0); i <= min(j, 1024); i++) {
        s[j - i] = s[j - i] + r[i] * A[i][j - i];
        q[i] = q[i] + A[i][j - i] * p[j - i];
    }
}

Johannes Doerfert, Tobias Grosser, Sebastian Pop
The Polyhedral Model

```
for (i = 0; i <= 1024; i++) {
    for (j = 0; j <= 1024; j++) {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}
```

Input Loop Nest
The Polyhedral Model

```
for (i = 0; i <= 1024; i++) {
    for (j = 0; j <= 1024; j++) {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}
```

Input Loop Nest

Static Analysis

Polyhedral Abstraction
The Polyhedral Model

for (i = 0; i <= 1024; i++) {
    for (j = 0; j <= 1024; j++) {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}

Input Loop Nest

Static Analysis

Polyhedral Abstraction

Scheduler

Schedule
The Polyhedral Model

```plaintext
for (i = 0; i <= 1024; i++) {
    for (j = 0; j <= 1024; j++) {
        s[j] = s[j] + r[i] * A[i][j];
        q[i] = q[i] + A[i][j] * p[j];
    }
}
for (j = 0; j <= 2048; j++) {
    parfor (i = max(j - 1024, 0); i <= min(j, 1024); i++) {
        s[j - i] = s[j - i] + r[i] * A[i][j - i];
        q[i] = q[i] + A[i][j - i] * p[j - i];
    }
}
```

Input Loop Nest

Static Analysis

Polyhedral Abstraction

Scheduler

Schedule

Output Loop Nest

Code Generation

Johannes Doerfert, Tobias Grosser, Sebastian Pop