Optimistic Assumptions in Polyhedral Compilation

Johannes Doerfert [1] Tobias Grosser [2]

[1]Saarland University
Saarbrücken, Germany

[2]ETH Zürich
Zürich, Switzerland

October 29, 2015

SAARLAND UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE

Be Optimistic!

Programs might be nasty but *programmers* are not.

When static information are insufficient

Optimistic Assumptions & Speculative Versioning

When static information are insufficient

Optimistic Assumptions & Speculative Versioning

Optimistic Assumptions

When static information are insufficient

Optimistic Assumptions & Speculative Versioning

Optimistic Assumptions

1. Make optimistic assumptions to (better) optimize loops

When static information are insufficient

Optimistic Assumptions & Speculative Versioning

Optimistic Assumptions

1. Make optimistic assumptions to (better) optimize loops
2. Derive runtime conditions that imply these assumptions

When static information are insufficient

Optimistic Assumptions & Speculative Versioning

Optimistic Assumptions

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

When static information are insufficient

Optimistic Assumptions & Speculative Versioning

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Optimistic Assumptions

1. Make optimistic assumptions to (better) optimize loops
2. Derive 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 are insufficient

Runtime Conditions

Runtime Conditions

- Fast to derive (compile time)
- Fast to verify (runtime)
- High probability to be true
- *ToDo:* A feedback/profile driven approach

Optimistic Assumptions in Polly

SAARLAND UNIVERSITY
COMPUTER SCIENCE

(A) Applicability/Correctness

1. No Alias Assumption[1]
2. No Wrapping Assumption[2]
3. Finite Loops Assumption[2]
4. Array In-bounds Assumption[2]
5. Valid Multidimensional View Assumption (Delinearization)[3]

(B) Optimizations

1. Array In-bounds Check Hoisting[2]
2. Parametric Dependence Distances[4]
3. Possibly Invariant Loads

[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)

Optimistic Assumptions in Polly


```
void mem_copy(int N, float *A, float *B) {
  if ([redacted] || [redacted]) {

    #pramga vectorize
    for (i = 0; i < N; i++)
      A[i] = B[i+5];

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

Optimistic Assumptions in Polly

```
void mem_copy(int N, float *A, float *B) {
  if (&A[0] >= &B[N+5] || [redacted]) {

    #pramga vectorize
    for (i = 0; i < N; i++)
      A[i] = B[i+5];

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

Optimistic Assumptions in Polly

No Alias Assumptions

```
void mem_copy(int N, float *A, float *B) {
  if (&A[0] >= &B[N+5] || &A[N] <= &B[5]) {

    #pramga 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


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

  if ([redacted]) {

    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

```
void evn_odd(int N, int *Evn, int *Odd, int *A, int *B) {
  if ([redacted]) {
    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

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

  if ([redacted]) {

    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

Optimistic Assumptions in Polly

No Alias Assumptions

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

  if (N%2 ? ((&B[N+1] <= &Odd[0] || &Odd[(N+1)/2] <= &B[1]) &&
            (&A[N+1] <= &Odd[0] || &Odd[(N+1)/2] <= &A[1]))
          : ((&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
- Do not compare accesses to read-only arrays
- Use the iteration domain of the accesses

Optimistic Assumptions in Polly

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
void mem_shift(unsigned char N, float *A) {
  if ([redacted]) {

    #pramga vectorize
    for (unsigned char i = 0; i < N; i++)
      A[i] = A[N + i];

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

Optimistic Assumptions in Polly

No Wrapping Assumption


```
void mem_shift(unsigned char N, float *A) {
  if (N <= 128) {

    #pramga 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

$$\underline{i} * \underline{c_0} + \underline{p} * \underline{c_1} \equiv_p (\underline{i} * \underline{c_0} + \underline{p} * \underline{c_1}) \;\; \texttt{mod} \;\; 2^k$$

Optimistic Assumptions in Polly

No Wrapping Assumption

$$\underline{i} * \underline{c_0} + \underline{p} * \underline{c_1} \equiv_p (\underline{i} * \underline{c_0} + \underline{p} * \underline{c_1}) \texttt{ mod } 2^k$$

$$\texttt{i} \in \texttt{[0, N-1]} \wedge \texttt{N} \in \texttt{[0, } 2^8\texttt{]}$$

$$\texttt{(N + i)} \equiv_p \texttt{(N + i) mod } 2^8$$

Optimistic Assumptions in Polly

No Wrapping Assumption

$$\underline{i} * \underline{c_0} + \underline{p} * \underline{c_1} \equiv_p (\underline{i} * \underline{c_0} + \underline{p} * \underline{c_1}) \text{ mod } 2^k$$

$$\texttt{i} \in \texttt{[0, N-1]} \wedge \texttt{N} \in \texttt{[0, } 2^8\texttt{]}$$

$$\texttt{(N + i)} \equiv_p \texttt{(N + i) mod } 2^8$$

$$\Longrightarrow \texttt{(N + i)} \leq_p \texttt{255}$$

Optimistic Assumptions in Polly

No Wrapping Assumption

$$\underline{i} * \underline{c_0} + \underline{p} * \underline{c_1} \equiv_p (\underline{i} * \underline{c_0} + \underline{p} * \underline{c_1}) \texttt{ mod } 2^k$$

$$\begin{aligned} & \texttt{i} \in \texttt{[0, N-1]} \wedge \texttt{N} \in \texttt{[0, } 2^8\texttt{]} \\ & \texttt{(N + i)} \equiv_p \texttt{(N + i) mod } 2^8 \\ \Longrightarrow \quad & \texttt{(N + i)} \leq_p \texttt{255} \\ \Longrightarrow \quad & \texttt{N} \leq \texttt{128} \end{aligned}$$

Optimistic Assumptions in Polly

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
void mem_shift(unsigned N, float *A) {
  if ([redacted]) {

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

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

Optimistic Assumptions in Polly

Finite Loops Assumption

```
void mem_shift(unsigned N, float *A) {
  if (N % 2 == 0) {

    #pramga 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

Optimistic Assumptions in Polly

```
void stencil(int N, int M, float A[128][128]) {
    if ([redacted]) {

      #pragma loop interchange
      for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
          A[2*j][i] += A[2*j+1][i];

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

Optimistic Assumptions in Polly

Array In-bounds Assumptions

```
void stencil(int N, int M, float A[128][128]) {
    if ([redacted]) {

      #pragma loop interchange
      for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
          A[2*j][i] += A[2*j+1][i];

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

Optimistic Assumptions in Polly

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
void stencil(int N, int M, float A[128][128]) {
    if (N <= 128) {

      #pragma loop interchange
      for (int i = 0; i < N; i++)
        for (int j = 0; j < M; j++)
          A[2*j][i] += A[2*j+1][i];

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

- Out-of-bound accesses introduce multiple addresses for one memory location (e.g., `&A[1][0] == &A[0][128]`)

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 ([redacted]) {

    #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 ([redacted]) {

    #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

Valid Multidimensional View Assumption


```c
#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 multi-dimensional view of a linearized (one-dimensional) array
- Derive conditions that accesses are in-bounds for each dimension

Optimistic Assumptions in Polly


```
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 */
  }
```

Optimistic Assumptions in Polly


```
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 */
  }
```

Optimistic Assumptions in Polly

Array In-bounds Check Hoisting


```
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 */
  }
```

Optimistic Assumptions in Polly

Array In-bounds Check Hoisting

```
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) {
    if ([redacted]) {

      for (int i = 0; i < N; i++)
        for (int j = 0; j < i/2; j++)
          A[i+j] = 1;

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

Optimistic Assumptions in Polly

Array In-bounds Check Hoisting


```
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) {
    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

Optimistic Assumptions in Polly

Check Hoisting

```
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)
      }

    }
  }
```

Optimistic Assumptions in Polly

```
void vectorize(int N, double *A) {
    if ([redacted]) {

      #pragma vectorize width(4)
      for (int i = c; i < N+c; i++)
        A[i-c] += A[i];

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

Optimistic Assumptions in Polly

Parametric Dependence Distances

```
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

Optimistic Assumptions in Polly

```
void may_load(int *size0, int *size1) {
    for (int i = 0; i < *size0; i++)
      for (int j = 0; j < *size1; j++)
        ...
  }
```

Optimistic Assumptions in Polly

Possibly Invariant Loads

```
void may_load(int *size0, int *size1) {
  int size0val = *size0;
  int size1val = 1;

  [redacted]
    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

```
void may_load(int *size0, int *size1) {
  int size0val = *size0;
  int size1val = 1;

  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

Optimistic Assumptions in Polly

Possibly Invariant Loads

```
void may_load(int *size0, int *size1) {
  int size0val = *size0;
  int size1val = 1;

  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

Ongoing Work

- Use profiling to minimize non-beneficial code duplication
- Derive more powerful checks (e.g., generate inspector loops)
- Find more opportunities to speculatively optimize using runtime conditions

Thank You.