TORSTEN HOEFLER
DEPARTMENT OF COMPUTER SCIENCE, ETH ZÜRICH

A case for runtime recompilation in HPC
(or: MPI+X; X=LLVM)
Setting the stage

- We use LLVM, but not like you may think!

- Runtime Recompilation and Specialization
  - MPI optimizations [EuroMPI’13, LCPC’13]

- Automatic Performance Model Generation
  - Static and dynamic modeling [SPAA’14, PACT’14]

- Compilation for Heterogeneous Systems
  - Focused around Polyhedral techniques

- We only compile test-applications!
  - Mainly deal with IR and internal issues
Topics Today (ask anything!)

- We use LLVM, but not like you may think!

- Runtime Recompilation and Specialization
  - MPI optimizations [EuroMPI’13, LCPC’13]

- Automatic Performance Model Generation
  - Static and dynamic modeling [SPAA’14, PACT’14]

- Compilation for Heterogeneous Systems
  - Focused around Polyhedral techniques

- We only compile test-applications!
  - Mainly deal with IR and internal issues
What your vendor sold you

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
What your Applications get

10% of Ping-Pong performance

Schneider, Kjolstad, TH: MPI Datatype Processing using Runtime Compilation, EuroMPI’13
What your Applications get

Why?

10% of Ping-Pong performance

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
What your Applications get

How to measure?

Why?

Bandwidth [GB/s]

0 1 2 3 4 5

0K 200

Size [Byte]

Schneider, Kjolstad, TH: MPI Datatype Processing using Runtime Compilation, EuroMPI’13
What MPI offers

Manual packing

```c
sbuf = malloc(N*sizeof(double))
rbuf = malloc(N*sizeof(double))
for (i=1; i<N-1; ++i)
    sbuf[i]=data[i*N+N-1]
MPI_Isend(sbuf, ...)
MPI_Irecv(rbuf, ...)
MPI_Waitall(...)
for (i=1; i<N-1; ++i)
    data[i*N]=rbuf[i]
free(sbuf)
free(rbuf)
```

MPI Datatypes

```c
MPI_Datatype nt
MPI_Type_vector(N-2, 1, N,
    MPI_DOUBLE, &nt)
MPI_Type_commit(&nt)
MPI_Isend(&data[N+N-1], 1, nt, ...)
MPI_Irecv(&data[N], 1, nt, ...)
MPI_Waitall(...)
MPI_Type_free(&nt)
```

• No explicit copying
• Less code
• Often slower than manual packing (see [1])

[1] Schneider, Gerstenberger, TH: *Micro-Applications for Communication Data Access Patterns and MPI Datatypes*, EuroMPI'12
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```java
bt = Vector(2, 1, 2, MPI_BYTE)
nt = Vector(N, 1, 4, bt)
```

Internal Representation

```
If (dt.type == VECTOR)
    for (int i=0; i<dt.count; i++) {
        tin = inbuf; tout = outbuf
        for (b=0; b<dt.blklen; d++) {
            interpret(dt.basetype, tin, tout)
        }
        tin += dt.stride * dt.base.extent
        tout = dt.blklen * dt.base.size
    }
    inbuf += dt.extent
    outbuf += dt.size
```

Schneider, Kjolstad, TH: MPI Datatype Processing using Runtime Compilation, EuroMPI’13
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```cpp
bt = Vector(2, 1, 2, MPI_BYTE)
nt = Vector(N, 1, 4, bt)
```

If (dt.type == VECTOR)

```cpp
for (int i=0; i<dt.count; i++) {
    tin = inbuf; tout=outbuf;
    for (b=0; b<dt.blklen; d++) {
        interpret(dt.basetype, tin, tout)
    }
    tin += dt.stride * dt.base.extent
    tout = dt.blklen * dt.base.size
}
```

- None of these variables are known when this code is compiled
- Many nested loops and branches

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
for (int i=0; i<N; ++i) {
    for(j=0;  j<2; ++j) {
        outbuf[j] = inbuf[j*2]
    }
    inbuf += 3*4
    outbuf += 2
}
```
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
for (int i=0; i<N; ++i) {
    for(j=0;  j<2; ++j) {
        outbuf[j] = inbuf[j*2]
    }
    inbuf += 3*4
    outbuf  += 2
}
```

- Loop unrolling
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
def (int i=0; i<N; ++i) {
    int j = 0
    outbuf[j] = inbuf[j*2]
    outbuf[j+1] = inbuf[(j+1)*2]
    inbuf += 3*4
    outbuf += 2
}
```

- Loop unrolling
- Constant Propagation

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
for (int i=0; i<N; ++i) {
    outbuf[0] = inbuf[0]
    inbuf += 12
    outbuf += 2
}
```

- Loop unrolling
- Constant Propagation
- Strength reduction

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
bound = outbuf + 2*N
while (outbuf<bound) {
  outbuf[0] = inbuf[0]
  inbuf += 12
  outbuf += 2
}
```

- Loop unrolling
- Constant Propagation
- Strength reduction

Schneider, Kjolstad, TH: MPI Datatype Processing using Runtime Compilation, EuroMPI’13
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
bound = (outbuf + 2*N)/2
while (outbuf<bound) {
    outbuf[0] = inbuf[0]
    inbuf += 24
    outbuf += 4
}
...
```

- Loop unrolling
- Constant Propagation
- Strength reduction
- Unrolling of outer loop

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled

```c
bound = (outbuf + 2*N)/2
while (outbuf<bound) {
    outbuf[0] = inbuf[0]
    inbuf += 24
    outbuf += 4
}
...
```

- Loop unrolling
- Constant Propagation
- Strength reduction
- Unrolling of outer loop
- SIMDization

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
Interpretation vs. Compilation

- MPI DDTs are interpreted at runtime, while manual pack loops are compiled
  - Loop unrolling
  - Constant Propagation
  - Strength reduction
  - Unrolling of outer loop
  - SIMDization

```c
for (int i=0; i<N; ++i) {
  for(j=0; j<2; ++j) {
    outbuf[j] = inbuf[j*2]
  }
  inbuf += 3*4
  outbuf += 2
}
```

```
bound = (outbuf + 2*N)/2
while (outbuf<bound) {
  outbuf[0] = inbuf[0]
  inbuf += 24
  outbuf += 4
}
...```

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
**Runtime-Compiled pack functions**

**Declare**

```c
MPI_Type_vector(cnt, blklen, ...)
```

**Optimize**

```c
MPI_Type_commit(new_ddt)
```

**Use**

```c
MPI_Send(cnt, buf, new_ddt,...)
```

---

Record arguments in internal representation (Tree of C++ objects)

Generate pack(*in, cnt, *out) function using LLVM IR. Compile to machine code. Store f-pointer.

```c
new_ddt.pack(buf, cnt tmpbuf)
PMPI_Send(...tmpbuf, MPI_BYTE)
```

---

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
Detour: Copying Data

- Basic elements of DDTs are always consecutive blocks
- If the size of the block is less the 256B we completely unroll the loop
- Otherwise: use fastest available instruction (SSE2 on our test system)

In-cache measurement on AMD Interlagos CPU (Blue Waters test system)
Detour: How to Copy Fast on x86?

- Lots of choice to move data!
  - > 36 ways on x86
- Restricted semantics allow for super-optimization [4]
  - Exhaustive search
  - Runs ~1 day
  - Generates close-to-optimal copy sequences

Overview of data movement and loop-forming instructions on x86-64.

Detour: Optimized Local Copy Sequence

Schneider, Gerstenberger, TH: Compiler Optimizations for Non-Contiguous Remote Data Movement, LCPC’13
Datatype Example (1): Packing Vectors

- Vector count and size and extent of subtype are always known
- Eliminate induction variables to reduce loop overhead
- Unroll loop for innermost loop 16 times

Datatype

Example (1): Packing Vectors

- HVector(2,1,6144) of Vector(8,8,32) of Contig(6) of MPI_FLOAT

This datatype is used by the Quantum-Chromodynamics code MILC [2]

Datatype Example (2): Irregular Data

Depending on index list length:

```
copy(inb+off[0], outb+..., len[0])
copy(inb+off[1], outb+..., len[1])
copy(inb+off[2], outb+..., len[2])
```

Inline indices into code

```
for (i=0; i<idx.len; i+=3) {
  inb0=load(idx[i+0])+inb
  inb1=load(idx[i+1])+inb
  inb2=load(idx[i+2])+inb
  // load oub and len
  copy(inb0, outb0, len0)
  copy(inb1, outb1, len1)
  copy(inb2, outb2, len2)
}
```

Minimize loop overhead by unrolling the loop over the index list

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
Datatype Example (2): Irregular Packing Performance

Hindexed DDT with random displacements

33% faster

Schneider, Kjolstad, TH: MPI Datatype Processing using Runtime Compilation, EuroMPI’13
What’s the catch?

- Emitting and compiling IR is (too?) expensive!
- Commit should tune the DDT, but we do not know how often it will be used – how much tuning is ok?
- Case study: MIMD Lattice Computation (thanks to Steve Gottlieb)

Most datatypes become seven times faster!

0-1 column is empty.
We don’t make anything slower than Cray MPI

But some need 30000 uses to amortize their costs at commit time

Most datatypes have to be reused 180-5000 times

Some even 38 times
Can we beat manual packing?

LAMMPS\_atomic \hspace{1cm} MILC\_su3\_zd \hspace{1cm} NAS\_LU\_y \hspace{1cm} SPECFEM3D\_oc \hspace{1cm} WRF\_x\_vec \hspace{1cm} WRF\_y\_vec

<table>
<thead>
<tr>
<th>Datasize [Byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>6K</td>
</tr>
<tr>
<td>8K</td>
</tr>
<tr>
<td>10K</td>
</tr>
<tr>
<td>25K</td>
</tr>
<tr>
<td>50K</td>
</tr>
<tr>
<td>75K</td>
</tr>
<tr>
<td>100K</td>
</tr>
<tr>
<td>1K</td>
</tr>
<tr>
<td>20K</td>
</tr>
<tr>
<td>30K</td>
</tr>
<tr>
<td>40K</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>60K</td>
</tr>
<tr>
<td>90K</td>
</tr>
<tr>
<td>120K</td>
</tr>
<tr>
<td>150K</td>
</tr>
<tr>
<td>50K</td>
</tr>
<tr>
<td>60K</td>
</tr>
<tr>
<td>70K</td>
</tr>
<tr>
<td>80K</td>
</tr>
</tbody>
</table>

Packing [% of Packing+Comm.]

Pack Method
- Cray MPI
- Manual Packing in C
- Our Implementation

Schneider, Kjolstad, TH: *MPI Datatype Processing using Runtime Compilation*, EuroMPI’13
The Runtime Recompilation for HPC Manifesto

- Example demonstrator: MPI Datatypes (works great!)
  - Has (limited) language interface
  - Missing information:
    - How often will the DDT be reused?
    - How will it be used (Send/Recv/Pack/Unpack)?
    - Will the buffer argument be always the same?
    - Will the data to pack be in cache or not?

- How can we generalize this?
  - Runtime-optimize everything!!
  - Two main problems:
    - What to runtime-recompile?
      - Idea: largest subgraph of CFG with constant variables (define “largest”!)
    - When to runtime-recompile?
      - Is it worth the recompilation overhead!?

http://spcl.inf.ethz.ch/Research/Parallel_Programming/MPI_Datatypes/libpack
Counting Loop Iterations!

- Structures that determine program runtime
  
  **LOOPS**

- Assumption: Other instructions do not influence it

- Example:
  
  ```
  for (x=0; x < n/p; x++)
      for (y=1; y < n; y=2*y )
          veryComplicatedOperation(x,y);
  ```
Related Work: Counting Loop Iterations

- In the Polyhedral model:

```plaintext
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      S1

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    S2
```

- piplib
- PolyLib
- PPL
- Polly
- ...
Related Work: Counting Loop Iterations

- When the polyhedral model fails

```plaintext
for (j = 1; j <= n; j = j*2) 
    for (k = j; k <= n; k = k++)
        veryComplicatedOperation(j,k);
```

\[ j \in [1, n] \]

\[ k \in [j, n] \]

\[ N = (n + 1) \log_2 n - n + 2 \]

\[ N = \frac{n(n + 1)}{2} \]

Related Work: Counting Loop Iterations

- When the polyhedral model cannot handle it

```
j=10;
k=10;
while (j>0){
j=j+k;
k--;
}
```
Counting Arbitrary Affine Loop Nests

- Affine loops

\[
\begin{align*}
x &= x_0; & \quad & \text{Initial assignment} \\
\text{while}(c^T x < g) & \quad & \text{Loop guard} \\
& \quad \quad x = Ax + b; & \quad & \text{Loop update}
\end{align*}
\]

- Perfectly nested affine loops

\[
\begin{align*}
\text{while}(c_1^T x < g_1) \{ \\
& \quad x = A_1 x + b_1; \\
\text{while}(c_2^T x < g_2) \{ \\
& \quad \quad \ldots \\
& \quad \quad x = A_{k-1}x + b_{k-1}; \\
& \quad \quad \text{while}(c_k^T x < g_k) \{ \\
& \quad \quad \quad \ldots \\
& \quad \quad \quad x = A_kx + b_k; \\
& \quad \quad \quad \text{while}(c_{k+1}^T x < g_{k+1}) \{ \ldots \} \\
& \quad \quad \quad x = U_kx + v_k; \} \\
& \quad \quad \quad x = U_{k-1}x + v_{k-1}; \}
& \quad \quad \ldots \} \\
& \quad \quad x = U_1x + v_1; \}
\end{align*}
\]

\[A_k, U_k \in \mathbb{R}^{m \times m}, \quad b_k, v_k, c_k \in \mathbb{R}^m, \quad g_k \in \mathbb{R} \text{ and } k = 1 \ldots r.\]
Counting Arbitrary Affine Loop Nests

- Example

  for (j=1; j < n/p + 1; j= j*2) 
    for (k=j; k < m; k = k + j )
      veryComplicatedOperation(j,k);
Counting Arbitrary Affine Loop Nests

- **Example**

  ```
  for (j=1; j < n/p + 1; j= j*2) 
    for (k=j; k < m; k = k + j )
      veryComplicatedOperation(j,k);
  ```

```c
while(\(c_1^T x < g_1\)) {
  x = A_1 x + b_1;
  while(\(c_2^T x < g_2\)) {
    ...
    x = A_{k-1} x + b_{k-1};
    while(\(c_k^T x < g_k\)) {
        x = A_k x + b_k;
        while(\(c_{k+1}^T x < g_{k+1}\)) {... }
        x = U_k x + v_k; }
    x = U_{k-1} x + v_{k-1};
    ...
  } 
  x = U_1 x + v_1;}
```
Counting Arbitrary Affine Loop Nests

Example

for (j=1; j < n/p + 1; j= j*2)
  for (k=j; k < m; k = k + j )
    veryComplicatedOperation(j,k);

\[
\begin{pmatrix}
  j \\
  k
\end{pmatrix} =
\begin{pmatrix}
  0 & 0 \\
  0 & 1
\end{pmatrix}
\begin{pmatrix}
  j \\
  k
\end{pmatrix} +
\begin{pmatrix}
  1 \\
  0
\end{pmatrix};
\]

while(c_1^T x < g_1) {
  x = A_1 x + b_1;
  while(c_2^T x < g_2) {
    ...
    x = A_{k-1} x + b_{k-1};
    while(c_k^T x < g_k) {
      x = A_k x + b_k;
      while(c_{k+1}^T x < g_{k+1}) {... }
      x = U_k x + v_k; }
    x = U_{k-1} x + v_{k-1};
    ...
  }
  x = U_1 x + v_1; }

TH, Kwasniewski: *Automatic Complexity Analysis of Explicitly Parallel Programs*, Symp. on Parallelism in Algorithms and Architectures, SPAA’14
Counting Arbitrary Affine Loop Nests

- Example

\[
\text{for } (j=1; \ j < n/p + 1; \ j= j*2) \\
\quad \text{for } (k=j; \ k < m; \ k = k + j) \\
\quad \quad \text{veryComplicatedOperation}(j,k);
\]

\[
\begin{pmatrix}
  j \\
  k
\end{pmatrix} = \begin{pmatrix}
  0 & 0 \\
  0 & 1
\end{pmatrix} \begin{pmatrix}
  j \\
  k
\end{pmatrix} + \begin{pmatrix}
  1 \\
  0
\end{pmatrix};
\]

\[
\text{while}(\begin{pmatrix}
  1 & 0
\end{pmatrix} \begin{pmatrix}
  j \\
  k
\end{pmatrix} \leq \frac{n}{p} + 1)\{
\]

\[
\text{while}(c_1^T x < g_1) \{ \\
\quad x = A_1 x + b_1;
\}
\]

\[
\text{while}(c_2^T x < g_2) \{ \\
\quad \ldots \\
\quad x = A_{k-1} x + b_{k-1};
\quad \text{while}(c_k^T x < g_k) \{ \\
\quad \quad x = A_k x + b_k;
\quad \quad \text{while}(c_{k+1}^T x < g_{k+1}) \{ \ldots \} \\
\quad \quad x = U_k x + v_k; \\
\quad \} \\
\quad x = U_{k-1} x + v_{k-1}; \\
\quad \ldots \}
\]

\[
\quad x = U_1 x + v_1; \}
\]

TH, Kwasniewski: Automatic Complexity Analysis of Explicitly Parallel Programs, Symp. on Parallelism in Algorithms and Architectures, SPAA’14
Counting Arbitrary Affine Loop Nests

- Example

```
for (j=1; j < n/p + 1; j= j*2)
    for (k=j; k < m; k = k + j )
        veryComplicatedOperation(j,k);
```

```
while(c_1^T x < g_1)
    x = A_1 x + b_1;
while(c_2^T x < g_2)
    ...
    x = A_{k-1} x + b_{k-1};
while(c_{k}^T x < g_{k})
    x = A_{k} x + b_{k};
    while(c_{k+1}^T x < g_{k+1})
        ...
        x = U_{k} x + v_{k};
    x = U_{k-1} x + v_{k-1};
...
x = U_{1} x + v_{1};
```

\[
\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix};
\]

\[
\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
\]

\[
\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} j \\ k \end{pmatrix} < \left\lfloor \frac{n}{p} + 1 \right\rfloor
\]

\[
\begin{pmatrix} j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} j \\ k \end{pmatrix} < m
\]
Counting Arbitrary Affine Loop Nests

Example

\[
\begin{align*}
&\text{for } (j=1; j < n/p + 1; j = j*2) \\
&\quad \text{for } (k=j; k < m; k = k + j) \\
&\quad \text{veryComplicatedOperation}(j, k);
\end{align*}
\]

\[
\begin{align*}
\begin{bmatrix} j \\ k \end{bmatrix} &= \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} j \\ k \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \\
\end{align*}
\]

\[
\begin{align*}
\text{while}(c_1^T x < g_1) \{ \\
&x = A_1 x + b_1; \\
&\text{while}(c_2^T x < g_2) \{ \\
&\quad \ldots \\
&\quad x = A_{k-1} x + b_{k-1}; \\
&\quad \text{while}(c_k^T x < g_k) \{ \\
&\quad \quad x = A_k x + b_k; \\
&\quad \quad \text{while}(c_{k+1}^T x < g_{k+1}) \{ \\
&\quad \quad \quad \ldots \} \\
&\quad \quad x = U_k x + v_k; \\
&\quad \quad \ldots \} \\
&\quad x = U_{k-1} x + v_{k-1}; \\
&\quad \ldots \} \\
&x = U_1 x + v_1; \}
\end{align*}
\]
Counting Arbitrary Affine Loop Nests

- **Example**

  \[
  \text{for (} j=1; j < n/p + 1; j= j*2) \\
  \text{for (} k=j; k < m; k = k + j) \\
  \text{veryComplicatedOperation}(j,k);
  \]

  

  \[
  \begin{align*}
  \text{while}(c_1^T x < g_1) \{ \\
  & x = A_1 x + b_1; \\
  & \text{while}(c_2^T x < g_2) \{ \\
  & \quad \ldots \\
  & \quad x = A_{k-1} x + b_{k-1}; \\
  & \quad \text{while}(c_k^T x < g_k) \{ \\
  & \quad \quad \ldots \\
  & \quad \quad x = A_k x + b_k; \\
  & \quad \quad \text{while}(c_{k+1}^T x < g_{k+1}) \{ \\
  & \quad \quad \quad \ldots \} \\
  & \quad \quad x = U_k x + v_k; \} \\
  & \quad x = U_{k-1} x + v_{k-1}; \\
  & \ldots \} \\
  & x = U_1 x + v_1; \}
  \end{align*}
  \]

  \[
  x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};
  \]

  \[
  \text{while}(\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x < \frac{n}{p} + 1)\{ \\
  & x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \\
  & \text{while}(\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} x < m)\{ \\
  & \quad x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
  & \quad \} x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \\
  & \} \\
  \]

  \[
  \text{where} \quad x = \begin{pmatrix} j \\ k \end{pmatrix}
  \]

TH, Kwasniewski: *Automatic Complexity Analysis of Explicitly Parallel Programs*, Symp. on Parallelism in Algorithms and Architectures, SPAA’14
The Workflow

Parallel program

```c
do i = 1, procCols
    call mpi_irecv(buff, ..., dp_type, reduce_each_proc(i), i, mpi_comm_world, request, ierr)
    call mpi_send(buff2, ..., dp_type, reduce_each_proc(i), A, mpi_comm_world, ierr)
    call mpi_send(request, status, ierr)
enddo

do i = id*n/p, (id+1)*n/p
    do j = 1, nsize
        call compute
    enddo
```

Closed form representation

\[ x(i_1, ..., i_r) = A_{\text{final}}(i_1, ..., i_r) \cdot x_0 + b_{\text{final}}(i_1, ..., i_r) \]

with

\[ i_r = 0, ..., n_r(x_{0,k}), k = 1, ..., r \]

Affine loop synthesis

```c
while(c^1_x < g_1) {
    x = A^1_x + b^1;
    while(c^2_x < g_2) {
        x = A^2_x + b^2;
        while(c^3_x < g_3) {
            ...
        }
    }
    x = U_{k} x + v_{k};
    x = U_{k-1} x + v_{k-1};
    ...
    x = U_{2} x + v_{2};
    x = U_{1} x + v_{1};
}
```

Loop extraction

Number of iterations

\[ N = \sum_{i_1=0}^{n_1(x_{0,1})} \sum_{i_2=0}^{n_2(x_{0,2})} ... \sum_{i_r-1=0}^{n_r-1(x_{0,r-1})} n_r(x_{0,r}) \]

Program analysis

\[ W = N \bigg|_{p=1} \]

\[ D = N \bigg|_{p \to \infty} \]

TH, Kwasniewski: Automatic Complexity Analysis of Explicitly Parallel Programs, Symp. on Parallelism in Algorithms and Architectures, SPAA'14
Algorithm details

Closed form representation of a loop

- Single affine statement
  \[ x = Lx + p \]

- Counting function
  \[ n(x_0) \]

- Example
  \[ x(i, x_0) = L(i) \cdot x_0 + p(i) \]
  \[ x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]
  \[ n(x_{\text{while}}(g)) \leftarrow \text{arg min} \ (c^T \cdot x(i, x)) \leq g) \]
  \[ x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]

\[ x = x_0; \]
\[ \text{while } (c^T x < g) \]
\[ x = Ax + b; \]
\[ x(i, x_0) = A^i x_0 + \sum_{j=0}^{i-1} A^j \cdot b \]
\[ x(i, x_0) = \left( \begin{array}{cc} 1 & 0 \\ 1 & 1 \end{array} \right) x_0 + \sum_{j=0}^{i-1} \left( \begin{array}{cc} 1 & 0 \\ 1 & 0 \end{array} \right)^j \left( \begin{array}{c} 0 \\ 0 \end{array} \right) = \left( \begin{array}{cc} 1 & 0 \\ 1 & 1 \end{array} \right) x_0 + \left( \begin{array}{c} 0 \\ 0 \end{array} \right) \]
\[ n(x_0) = \left[ \frac{m - k_0}{j_0} \right] \]
Algorithm in details

Folding the loops

\[ x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \]

while (0 < \frac{n}{p}){
    \[ x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]
    while (1 < m) {
        \[ x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]
    } \]
    \[ x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]
Algorithm in details

Folding the loops

\[
x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};
\]

while ( \( 0 < \frac{x}{p} \) ){

\[
x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
\]

while ( \( 1 < m \) ){

\[
x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
\]
}

\[
x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix};
\]

while ( \( 0 < \frac{x}{p} \) ){

\[
x = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
\]

\[
x = \begin{pmatrix} 1 & 0 \\ i & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
\]

\[
x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix};
\]
}

Algorithm in details

Folding the loops

\[ x = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix}; \]

\[
\text{while (} 0 \leq x < \frac{n}{p} \text{)} \{
\]
\[ x = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]

\[
\text{while (} 0 \leq x < m \text{)} \{
\]
\[ x = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]

\[
\} x = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} x + \begin{pmatrix} 0 \\ 0 \end{pmatrix}; \]

TH, Kwasniewski: *Automatic Complexity Analysis of Explicitly Parallel Programs*, Symp. on Parallelism in Algorithms and Architectures, SPAA’14
Algorithm in details

Starting conditions

\[ x_{0,1} \rightarrow x = x_0; \]
\[ \text{while} \ (c^T_1 x < g_1) \{ \]
\[ x = A_1 x + b_1; \]
\[ \text{while} \ (c^T_2 x < g_2) \{ \]
\[ x = A_2 x + b_2; \]
\[ \text{while} \ (c^T_3 x < g_3) \{ \]
\[ x = A_3 x + b_3; \]
\[ x = U_2 x + v_2; \]
\[ x = U_1 x + v_1; \]
\} \]

TH, Kwasniewski: *Automatic Complexity Analysis of Explicitly Parallel Programs*, Symp. on Parallelism in Algorithms and Architectures, SPAA’14
Algorithm in details

Counting the number of iterations

We have:
Algorithm in details

Counting the number of iterations

We have:

- The closed form for each loop:
  - Single affine statement
  - Counting function
- Starting condition for each loop
Algorithm in details

Counting the number of iterations

We have:

- The closed form for each loop:
  - *Single affine statement*
  - *Counting function*
- Starting condition for each loop

Number of iterations:

\[
N = \sum_{i_1=0}^{n_1(x_0,1)} \sum_{i_2=0}^{n_2(x_0,2)} ... \sum_{i_{r-1}=0}^{n_{r-1}(x_0,r-1)} n_r(x_0,r).
\]
Algorithm in details

Counting the number of iterations

- The equation gives precise number of iterations

\[ N = \sum_{i_1=0}^{n_1(x_0,1)} \sum_{i_2=0}^{n_2(x_0,2)} \ldots \sum_{i_{r-1}=0}^{n_{r-1}(x_0,r-1)} n_r(x_0,r). \]
Algorithm in details

Counting the number of iterations

- The equation gives precise number of iterations
  \[ N = \sum_{i_1=0}^{n_1(x_0,1)} \sum_{i_2=0}^{n_2(x_0,2)} \ldots \sum_{i_{r-1}=0}^{n_{r-1}(x_0,r-1)} n_r(x_0,r). \]

- But simplification may fail \(\rightarrow\) Sum approximation
  
  - *Approximate sums by integrals*
  
  \(\rightarrow\) lower and upper bounds
Solving more general problems

- Multipath loops
- Conditional statements
- Non-affine loops

```fortran
do j=1, lastrow-firstrow+1
  sum = 0.d0
  do k=rowstr(j), rowstr(j+1)-1
    sum = sum + a(k)*p(colidx(k))
  enddo
  w(j) = sum
enddo
```

\[ N = \frac{\text{na} \cdot \text{u}}{\text{nprows}} \]

TH, Kwasniewski: *Automatic Complexity Analysis of Explicitly Parallel Programs*, Symp. on Parallelism in Algorithms and Architectures, SPAA’14
Case studies

- NAS Parallel Benchmarks: EP

\[ N(m, p) = \left\lfloor \frac{2^{m-16} \cdot (u + 2^{16})}{p} \right\rfloor \]

```plaintext
u:
do i=1,100
   ik = kk/2
   if (ik .eq. 0) goto 130
   kk = ik
   continue
```

TH, Kwasniewski: *Automatic Complexity Analysis of Explicitly Parallel Programs*, Symp. on Parallelism in Algorithms and Architectures, SPAA’14
Case studies

- NAS Parallel Benchmarks: EP

\[ N(m, p) = \left\lfloor \frac{2^{m-16} \cdot (u + 2^{16})}{p} \right\rfloor \]

```
  u:  do i=1,100
      ik =kk/2
      if (ik .eq. 0) goto 130
      kk=ik
      continue
```

\[
W = T_1 \approx 2^m \\
D = T_\infty \approx 1
\]

\[
E_P = \frac{2^m}{p \left\lfloor \frac{2^m}{p} \right\rfloor}
\]

\[ E_P \approx 1 \text{ if } p \leq 2^m \]

\[ E_P \approx 2^m / p \text{ if } p > 2^m \]
Case studies

CG – conjugate gradient

\[ N \approx k_1 \left\lfloor \frac{m}{p} \right\rfloor + k_2 \sqrt{\left\lfloor \frac{m}{p} \right\rfloor} + k_3 \log_2 \sqrt{p} \]

\[ D = T_\infty \approx c_n \left( 3 \Theta + t \pm 2 \left\lfloor \frac{m}{p} \right\rfloor + p + u_1 + u_2 \right) \]

\[ E_p = \frac{D = T_\infty = \infty}{k_4} \left( k_1 \left\lfloor \frac{m}{p} \right\rfloor + k_2 \sqrt{\left\lfloor \frac{m}{p} \right\rfloor} + k_3 \log_2 \sqrt{p} \right) \]

IS – integer sort

15 applications (NAS/Mantevo/Mibench):

- 100% of loops were treated (with unknowns)
- 9-45% of loops were predicted exact

**Geometric mean: 18%, median: 18%**
What problems are remaining?

- Well, what about non-affine loops?
  - More general abstract interpretation (next step)
    - Any ideas for a more general algebra?
  - In general not solvable
    → will always have undefined terms

- Ad-hoc (partial) solution: online machine learning – PEMOGEN
  - Replace cross-validation with LASSO (regression with $L_1$ regularizer)
    - Much cheaper! (some issues with accuracy – RIP?)
  - Replace LASSO with online LASSO [1]
    - No traces! $O(1)$ memory overhead!

$$N = \frac{na \cdot u}{nprows}$$
PEMOGEN – static+dynamic analysis

- Also integrated into LLVM compiler
  - Automatic kernel detection and instrumentation (Loop Call Graph)
  - Static dataflow analysis reduces parameter space for each kernel

Quality: NAS UA and Mantevo MiniFE

Overhead: Mantevo

A. Bhattacharyya, TH: *PEMOGEN: Automatic Adaptive Performance Modeling during Program Runtime*, PACT’14
The Dragon’s Wishlist

- **Faster online JIT support**
  - Optimize LLVM itself
  - Performance expectations for passes
  - Analyze benefits of passes (and orders)

- **Superoptimization**
  - Would be nice, works well, offline!
  - Some approaches exist

- **Specific passes**
  - Better alias analysis
  - Abstract interpretation (cf., PAGAI, e.g., for MPI matching)
  - More in tomorrow’s LLVM BoF!