Research Seminar

Prof. Robert van Engelen
Research Projects

Compiler research

- Loop-level code analysis, optimization, and code generation techniques
- Intermediate program representation in static single assignment (SSA) form
- Our research is used in GCC 4.x
Research Projects

- **SOAP/XML Web services**
  - XML message parsing and object serialization using schema-specific permutation phrase grammars and parsers
  - Protocols and algorithms for Web services security and discovery
  - Code generation for high-performance services
  - Our work on gSOAP is used by thousands of companies

`WS protocol stack`
Research Projects

- Parallel programming (e.g. multicore) research
  - Explore declarative programming style using domain-specific languages
  - Work is used by meteorological institutes to run parallel weather codes

Our 8 node PS3 cluster in the ACIS lab
Research Projects

- Statistical modeling with Bayesian networks
  - Probabilistic inference algorithms
  - Applications in Bioinformatics and Medicine
How to Plan Research?

Find a professor who will do the research for you

1. **State the problem:**
   pick a problem area to investigate

2. **Find related work:** (e.g. http://scholar.google.com http://citeseer.ist.psu.edu)
   how have others (tried) to solve it?

3. **Identify challenges:**
   what are the limitations of these solutions?

4. **Formulate the research hypothesis:**
   how can we solve it better?

5. **Conduct experiments to verify hypothesis:**
   implement, benchmark, collect data, apply statistics!

6. **Conclusions:**
   what have we learned?
Today’s Research Topic Presentation

- **State the problem**
  - Trading accuracy for speed (e.g. when drawing objects)
  - Motivating examples

- **Hypothesis: theoretical foundations of CR algebra**
  - Chains of recurrences (CR) algebra and domain mappings
  - A CR transformation to optimize math function evaluations

- **Challenges**
  - Naïve implementation does not give speedups!

- **Performance testing and tuning**
  - SSE vectorization performance results
  - Vector CR for modulo scheduling performance results
  - Generating auto-tuned code

- **Conclusions**
Trading Accuracy for Speed

- In most (numerical) applications the preferred floating point precision is ‘double’ or even higher
- However, we still suffer from …
  - Floating point round-off errors that generally occur in algorithms, i.e. ‘double’ is just a way to “limit” them
  - The inability of compilers to optimize fp operations
  - Lost performance on some platforms

- Maybe the right question to ask is:
  - Given $b$ bits of precision for the output to be desired, can we generate a (very) fast version of the algorithm that computes the output with $\geq b$ bits accuracy?
Example: Parametric Surfaces

\[
\begin{pmatrix}
x \\
y \\
z \\
\end{pmatrix} =
\begin{pmatrix}
1.2^u(1 + \cos v) \cos u \\
1.2^u(1 + \cos v) \sin u \\
1.2^u \sin v - 1.5 \cdot 1.2^u \\
\end{pmatrix}
\]

Conchoid

\[
\begin{pmatrix}
x \\
y \\
z \\
\end{pmatrix} =
\begin{pmatrix}
(4 + 3.8 \cos v) \cos u \\
(4 + 3.8 \cos v) \sin u \\
(\cos v + \sin v - 1)(1 + \sin v) \log(1 - \frac{\pi}{10} v) + 7.5 \sin v \\
\end{pmatrix}
\]

Apple

\[
\begin{pmatrix}
x \\
y \\
z \\
\end{pmatrix} =
\begin{pmatrix}
(|u| - 1)^2 \cos v \\
(|u| - 1)^2 \sin v \\
u \\
\end{pmatrix}
\]

Toupie

\[
\begin{pmatrix}
x \\
y \\
z \\
\end{pmatrix} =
\begin{pmatrix}
2 \sin(3u)/(2 + \cos v) \\
2(\sin u + 2 \sin(2u))/(2 + \cos(v + \frac{2}{3} \pi)) \\
(\cos u - 2 \cos(2u))(2 + \cos v)(2 + \cos(v + \frac{2}{3} \pi))/4 \\
\end{pmatrix}
\]

Trangluoid trefoil
Example: Parametric Surfaces from Industrial Design

Shape rendering using parametric descriptions
(Images courtesy Microsoft Research)
Example: Coordinate Transformations

- Calculate coordinates given another coordinate system using bijective transformations
  - Cartesian coordinates
  - Polar coordinates
  - Parabolic coordinates
  - Spherical coordinates
  - Cylindrical coordinates
  - Conical coordinates
  - …

\[
\begin{pmatrix}
  x \\
  y \\
  z
\end{pmatrix} =
\begin{pmatrix}
  r \sin \theta \cos \varphi \\
  r \sin \theta \sin \varphi \\
  r \cos \theta
\end{pmatrix}
\]

Mapping spherical to Cartesian coordinates
Example: Spherical Coordinates in Limited-Area Weather Forecasting

The physical horizontal domain is described in spherical coordinates:

\[ dX = a \cos \theta d\lambda = h_x dx \]
\[ dY = a d\theta = h_y dy \]

The computational grid is logically rectangular

(a = Earth radius)
Example: Signal Processing

- A look-up table (LUT) stores pre-computed values to speed up computations in time-critical program parts

- Considerations:
  - High accuracy is not always needed
  - Requires some storage space for the LUT
  - Additional memory loads

- Example
  - Transfer (aka “design”) functions of digital filters

```c
// initialize sine LUT
for (k = ...; k < ...; k++)
   LUTsin[k] = sin(2*pi*k/N);
...

// read LUT (with stride h)
for (i = ...; i < ...; i++)
{
   ...
   res = x[i]*LUTsin[i*h];
   ...
}
```
Example: Vectorization of Strength-Reduced Loops

- Concept: combine short-vector ops with strength reduction
  - Apply strength reduction at the vector level (integer only)
  - Using vector registers may lower register pressure of scalars
  - Simple to apply to linear and geometric functions
  - More difficult for polynomials and other nonlinear functions

for (i=0; i<n; i++)
a[i] = a[i] + k*i + 1;

SR

int j = 1;
for (i=0; i<n; i++)
{
    a[i] = a[i] + j;
    j = j + k;
}
Example: Vectorization of Strength-Reduced Loops

- Concept: combine short-vector ops with strength reduction
  - Apply strength reduction at the vector level (integer only)
  - Using vector registers may lower register pressure of scalars
  - Simple to apply to linear and geometric functions
  - More difficult for polynomials and other nonlinear functions

```c
for (i=0; i<n; i++)
a[i] = a[i] + k*i + 1;
```

```
__m128i *p, v, h;
v = _mm_set_epi32(3*k+1,2*k+1,k+1,1);
h = _mm_set1_epi32(4*k);
p = (__m128*)a;
for (i=0; i<n/4; i++)
{
    p[i] = _mm_add_epi32(p[i], v);
    v = _mm_add_epi32(v, h);
}
```

"Vector SR"

Strength-reduced vector ops (shown with SSE intrinsics)
Motivation

- The examples showed the use of function evaluations over a $d$-dimensional grid of data points:

$$x[i_1, \ldots, i_d] = f(i_1, \ldots, i_d) \text{ for all } i_1 = 0, \ldots, n_1 - 1$$
$$i_2 = 0, \ldots, n_2 - 1$$
$$\vdots$$
$$i_d = 0, \ldots, n_d - 1$$

- Function $f$ can be anything, but is usually composed of transcendental functions, polynomials, logarithms, and exponentials
Motivation

- Discussed so far
  - None of the example applications require very high floating-point accuracy
  - When developing highly-optimized function evaluation algorithms, we must guarantee some minimal threshold accuracy of the output

- How?
  - Use algebraic framework to automatically derive semantically equivalent codes (not numerically equivalent)
  - Use auto-tuning for performance and error analysis
Foundations: Chains of Recurrences (CR) Algebra

- When \( f(i_1, \ldots, i_d) \) is evaluated in consecutive order over the grid points \( i_1, \ldots, i_d \), then a (multivariate) “Chains of Recurrences” (CR) form can be constructed:

\[
\Phi = \text{symbolic-cr-form}(f, i_1, \ldots, i_d)
\]

see e.g. [E. Zima, 1992]

- This form \( \Phi \) can be used to compute \( f \) for all points using fewer arithmetic operations by reusing values computed at previous points.
Foundations:
CR Form Construction and Evaluation

\[ f(i) = a + bi \overset{CR}{\Rightarrow} \{a, +, b\}_i \]
\[ p(i) = a + bi + ci^2 \overset{CR}{\Rightarrow} \{a, +, (b + c), +, 2c\}_i \]
\[ g(i) = a b^{ci} \overset{CR}{\Rightarrow} \{a, *, b^c\}_i \]

Efficient tabulation of value sequences of CR forms:

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(i) = {a, +, b} )</td>
<td>a</td>
<td>a+b</td>
<td>a+2b</td>
<td>a+3b</td>
</tr>
<tr>
<td>( p(i) = {a, +, (b+c), +, 2c} )</td>
<td>a</td>
<td>a+b+c</td>
<td>a+2b+4c</td>
<td>a+3b+9c</td>
</tr>
<tr>
<td>( g(i) = {a, *, b^c} )</td>
<td>a</td>
<td>ab^c</td>
<td>ab^{2c}</td>
<td>ab^{3c}</td>
</tr>
</tbody>
</table>
# Foundations: CR Algebra as a TRS

<table>
<thead>
<tr>
<th>#</th>
<th>LHS</th>
<th>RHS</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( { \phi_0, +, 0 }_i )</td>
<td>( \phi_0 )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( { \phi_0, *, 1 }_i )</td>
<td>( \phi_0 )</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( { 0, *, f_1 }_i )</td>
<td>( 0 )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( -{ \phi_0, +, f_1 }_i )</td>
<td>( { -\phi_0, +, -f_1 }_i )</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( -{ \phi_0, *, f_1 }_i )</td>
<td>( { -\phi_0, *, f_1 }_i )</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( { \phi_0, +, f_1 }_i \pm E )</td>
<td>( { \phi_0 \pm E, +, f_1 }_i )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>7</td>
<td>( { \phi_0, *, f_1 }_i \pm E )</td>
<td>( { \phi_0 \pm E, +, \phi_0 \star (f_1 - 1), *, f_1 }_i )</td>
<td>when ( E ) and ( f_1 ) are ( i )-loop invariant</td>
</tr>
<tr>
<td>8</td>
<td>( E \star { \psi_0, +, f_1 }_i )</td>
<td>( { E \star \psi_0, +, E \star f_1 }_i )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>9</td>
<td>( E \star { \phi_0, *, f_1 }_i )</td>
<td>( { E \star \phi_0, +, f_1 }_i )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>10</td>
<td>( E / { \phi_0, +, f_1 }_i )</td>
<td>( 1 / { \phi_0 / E, +, f_1 / E }_i )</td>
<td>when ( E \neq 1 ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>11</td>
<td>( E / { \phi_0, *, f_1 }_i )</td>
<td>( { E / \phi_0, *, 1 / f_1 }_i )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>12</td>
<td>( { \phi_0, +, f_1 }_i \pm { \psi_0, +, g_1 }_i )</td>
<td>( { \phi_0 \pm \psi_0, +, f_1 \pm g_1 }_i )</td>
<td>when ( f_1 ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>13</td>
<td>( { \phi_0, *, f_1 }_i \pm { \psi_0, +, g_1 }_i )</td>
<td>( { \phi_0 \pm \psi_0, +, \phi_0 \star (f_1 - 1), *, f_1 }_i \pm g_1 }</td>
<td>when ( f_1 ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>14</td>
<td>( { \phi_0, +, f_1 }_i \star { \psi_0, +, g_1 }_i )</td>
<td>( { \phi_0 \star \psi_0, +, { \phi_0, +, f_1 }_i \star g_1 + { \psi_0, +, g_1 }_i \star f_1 + f_1 \star g_1 }_i )</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>( { \phi_0, *, f_1 }_i \star { \psi_0, *, g_1 }_i )</td>
<td>( { \phi_0 \star \psi_0, *, { \phi_0, +, f_1 }_i \star g_1 } \star f_1 + { \psi_0, *, g_1 }_i \star f_1 + f_1 \star g_1 }_i )</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>( { \phi_0, *, f_1 }^E_i )</td>
<td>( { \phi_0 \star E, +, f_1 }^E_i )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>17</td>
<td>( { \phi_0, *, f_1 }^{\psi_0, +, g_1} }</td>
<td>( { \phi_0 \star \psi_0, *, { \phi_0, +, f_1 }_i \star g_1 + f_1 \star { \psi_0, +, g_1 }_i \star f_1 + f_1 \star g_1 }_i )</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>( E / { \phi_0, +, f_1 }_i )</td>
<td>( { E / \phi_0, +, f_1 }_i )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>19</td>
<td>( { \phi_0, +, f_1 }_i )</td>
<td>( n { \phi_0, +, f_1 }_i \star { \phi_0, +, f_1 }_i \star n ) if ( n \in \mathbb{Z}, n + 1 )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>20</td>
<td>( { \phi_0, +, f_1 }_i )</td>
<td>( 1 / { \phi_0, +, f_1 }_i \star n ) if ( n \in \mathbb{Z}, n &lt; 0 )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>21</td>
<td>( { \phi_0, +, \phi_1, *, f_2 }_i )</td>
<td>( { \phi_0, +, f_2 }_i )</td>
<td>when ( \phi_0 = f_2 + 1 )</td>
</tr>
<tr>
<td>22</td>
<td>( { \phi_0, #, f_1 }_i )</td>
<td>( f_1 )</td>
<td>when ( \phi_0 ) is ( \mathcal{B} )</td>
</tr>
<tr>
<td>23</td>
<td>( -{ \phi_0, #, f_1 }_i )</td>
<td>( -{ \phi_0, #, -f_1 }_i )</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>( { \phi_0, #, f_1 }_i \pm E )</td>
<td>( { \phi_0 \pm E, #, f_1 \pm E }_i )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>25</td>
<td>( E \star { \phi_0, #, f_1 }_i )</td>
<td>( { E \star \phi_0, #, E \star f_1 }_i )</td>
<td>when ( E ) is ( i )-loop invariant</td>
</tr>
<tr>
<td>26</td>
<td>( { \phi_0, #, f_1 }_i \pm { \psi_0, #, g_1 }_i )</td>
<td>( { \phi_0 \pm \psi_0, #, f_1 \pm g_1 }_i )</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>( { \phi_0, #, f_1 }_i \star { \psi_0, #, g_1 }_i )</td>
<td>( { \phi_0 \star \psi_0, #, f_1 \star g_1 }_i )</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>( { \phi_0, #, f_1 }_i \pm { \psi_0, +, g_1 }_i )</td>
<td>( { \phi_0 \pm \psi_0, #, f_1 \pm \mathcal{F} { \psi_0, +, g_1 }_i } )</td>
<td>(see Appendix A.1 for ( \mathcal{F} ))</td>
</tr>
<tr>
<td>29</td>
<td>( { \phi_0, #, f_1 }_i \star { \psi_0, +, g_1 }_i )</td>
<td>( { \phi_0 \star \psi_0, #, f_1 \star \mathcal{F} { \psi_0, +, g_1 }_i } )</td>
<td>(see Appendix A.1 for ( \mathcal{F} ))</td>
</tr>
<tr>
<td>30</td>
<td>( { \phi_0, +, f_1 }_i )</td>
<td>( { \phi_0, +, f_2 }_i )</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>( { \phi_0, *, f_1 } )</td>
<td>( { \phi_0, *, f_2 }_i )</td>
<td></td>
</tr>
</tbody>
</table>
CR Loop Generation

\[ p(i) = a + bi + ci^2 + di^3 \]

\[ = \{a, +, (b + c + d), +, (2c + 6d), +, (6d)\} \]

1. Apply CR algebra TRS

2. Construct the CR evaluation loop template

   \[
   \begin{align*}
   cr0 &= a; \\
   cr1 &= b + c + d; \\
   cr2 &= 2c + 6d; \\
   cr3 &= 6d;
   \end{align*}
   \]

   for (i=0; i<n; i++)
   {
   \[
   \begin{align*}
   x[i] &= cr0; \\
   cr0 &= cr0 + cr1; \\
   cr1 &= cr1 + cr2; \\
   cr2 &= cr2 + cr3;
   \end{align*}
   \}

3. Execute
CR Loop Equivalences

for (i=0; i<n; i++)
{
    r = r0;
    r0 = r0 + r1;
    r1 = r1 + r2;
    s = s0;
    s0 = s0 + s1;
    x[i] = r * s;
}

t0 = r0*s0;
t1 = r0*s1+r1*s0+r1*s1;
t2 = 2*r1*s1+2*r2*s1+r2*s0;
t3 = 3*r2*s1;
for (i=0; i<n; i++)
{
    x[i] = t0;
    t0 = t0 + t1;
    t1 = t1 + t2;
    t2 = t2 + t3;
}

{r0, +, r1, +, r2} * {s0, +, s1} = {r0s0, +, r0s1+r1s0+r1s1, +, 2r1s1+2r2s2+r2s0, +, 3r1s1}
Horner’s Rule Versus CR Forms

**Horner’s rule**

```
for (i = 0; i < n; i++)
{
  x[i] = a+i*(b+i*(c+i*d));
}
```

Cost per iteration: 3 additions and 3 multiplications
Total = 6n fp ops

**CR form evaluation code**

```
cr0 = a;
cr1 = b + c + d;
cr2 = 2*c + 6*d;
cr3 = 6*d;
for (i = 0; i < n; i++)
{
  x[i] = cr0;
  cr0 = cr0 + cr1;
  cr1 = cr1 + cr2;
  cr2 = cr2 + cr3;
}
```

Cost per iteration: 3 additions
Total = 3n fp ops

**Pros:**
- Good ILP (in-order/out-of-order)
- Automatically vectorizable

**Cons:**
- Not a high level of ILP (in-order)
- Not automatically vectorizable
- May increase register pressure
SIMD/Vectorization Issues

- Closed-form functions have no point-to-point dependences:
  - Easy to vectorize
  - Intel compiler with SVML auto-vectorizes loops containing polynomials, transcendental functions, exponentials, etc…

- CR forms use point-to-point recurrences, which inhibits loop vectorization

![Graph showing comparison between SVML of Horner's rule and CR form with Grid size (# data points) on x-axis and Better performance on y-axis.](image)
SIMD/Vectorization Issues

- Closed-form functions have no point-to-point dependences:
  - Easy to vectorize
  - Intel compiler with SVML auto-vectorizes loops containing polynomials, transcendental functions, exponentials, etc ...

- CR forms use point-to-point recurrences, which inhibits loop vectorization

- What if we compute 4 CR forms in parallel using SSE operations?
Domain Mapping

- Bijective mappings can be used to increase index space dimensionality
  - For example:
    \[ x[i] = f(i) = f(j + d \times i') \]
    such that \( i' = i / d \) and \( j = i \mod d \)
  - Therefore, we can easily obtain vector operations on d-wide vectors to speed up the computation:
    \[ x[i', i'+1, \ldots, i'+d-1] = [f(i'), f(i'+1), \ldots, f(i'+d-1)] \]
  - This is essentially what an auto-vectorizer does

- Can we do this in a “CR-like syntax” to obtain vector operations on CR forms?
Example

Compute the sequence $x[i] = \sin(h \times i)$ using 4 decoupled functions

- $f_0(i) = \sin(h \times 4 \times i)$
- $f_1(i) = \sin(h \times (4 \times i + 1))$
- $f_2(i) = \sin(h \times (4 \times i + 2))$
- $f_3(i) = \sin(h \times (4 \times i + 3))$
Vector CR (VCR) Construction

- Step 1 \( f(i) = f(\{j, +, d\}_i') \xrightarrow{CR} \Phi_{i'}(j) \)
  \[
  \Phi_{i'}(j) = \{ \varphi_0(j), \odot_1, \varphi_1(j), \odot_2, \cdots, \odot_k, \varphi_k(j) \}_{i'}
  \]

- Step 2 \( \tilde{\varphi}_0(j) = \begin{pmatrix} \varphi_0(j) \\ \varphi_0(j+1) \\ \vdots \\ \varphi_0(j+d-1) \end{pmatrix} \); \( \cdots \)
  \( \tilde{\varphi}_k(j) = \begin{pmatrix} \varphi_k(j) \\ \varphi_k(j+1) \\ \vdots \\ \varphi_k(j+d-1) \end{pmatrix} \)

- Step 3: Pack the VCR coefficients into vector registers and generate vector loop code
Example VCR Code

\[ \Phi_i^2 = \Re(\{ (\frac{-1}{2} \frac{I}{\gamma})^i, (\alpha + \beta) \}) + \Re(\{ (\frac{1}{2} \frac{I}{\delta})^i, (\alpha - \beta) \}) \]

\text{for } (i = 0; i < n/4; i++)
{
    xv[i] = _mm_add_ps(crv0i, crv1i);
    vtmp1 = _mm_mul_ps(crv0r, crv1r);
    vtmp2 = _mm_mul_ps(crv0i, crv1i);
    crv0rt = _mm_sub_ps(vtmp1, vtmp2);
    vtmp1 = _mm_mul_ps(crv0r, crv2r);
    vtmp2 = _mm_mul_ps(crv0i, crv2i);
    crv0i = _mm_add_ps(vtmp1, vtmp2);
    crv0r = crv0rt;
    crv1r = crv1rt;
}

ch = cos(h); sh = sin(h);
c2h = 2*ch*ch-1; s2h = 2*sh*ch;
c4h = 2*c2h*c2h-1; s4h=2*s2h*c2h;
c_4h = c4h; s_4h = -s4h;
crv0r = {0.0, 0.5*sh, sh*ch, sh*ch*ch+0.5*sh*s2h}
crv1r = {0.0, 0.5*sh, sh*ch, sh*ch*ch+0.5*sh*s2h}
crv2r = {c4h, c4h, c4h, c4h}
crv3r = {c_4h, c_4h, c_4h, c_4h}
crv0i = {-0.5, -0.5*ch, -0.5*c2h, -0.5*ch*c2h+sh*ch*ch}
crv1i = {0.5, 0.5*ch, 0.5*c2h, 0.5*ch*c2h-sh*ch*ch}
crv2i = {s4h, s4h, s4h, s4h}
crv3i = {s_4h, s_4h, s_4h, s_4h}

 Computes 4 data points for \( \sin(i \ h) \) per iteration
Domain Mappings to Increase Vector Length

- Bijective projections to reduce dimensionality and increase vector length
  - Example: $x[i,j] = \sin i + \cos j = \sin(i' \mod m) + \cos(i'/m)$ such that $i' = i + m \times j$

- Not practical for CR forms (open question)
- Alternative is to use multivariate CR forms

September 16, 2008
Research Seminar 2008
Other Approaches to Speed Up CR Evaluations

- Related work: subdomain splitting
  - Compute CR forms in subdomains (=blocks) to speed up CR form evaluation on parallel processors
  - Uses thread-level parallelism, see [E. Zima, 1992]
Performance Testing: Benchmark Functions and Code Classifications

- Benchmark functions:
  - *Poly3*: a random 3rd order polynomial
  - *Spline*: a random bicubic spline
  - *Bin15*: a 15th order polynomial
  - *Sine*: $x[i] = \sin(h*i)$
  - *Sinh*: $x[i] = \sinh(h*i)$
  - *Exp*: $x[i] = \exp(h*i*(i+1)/2)$

- Test code classification:
  - **Scalar**: original closed-form function in loop
  - **SVML**: Intel compiler SVML auto-vectorized code of original closed-form function
  - **Scalar CR**: simple scalar CR form
  - **VCRx.y**: x-way y-wide vector CR form (using SSE or y scalars)
Performance Results with SSE
(Xeon 5160 3GHz - Intel Compiler - VCR w/ SSE2 Intrinsics)

Performance Results (n = 1,000)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Scalar</th>
<th>SVML</th>
<th>Scalar CR</th>
<th>VCR4.4</th>
<th>VCR8.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spline</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bin15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sinh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance Results with SSE

(Xeon 5160 3GHz - Intel Compiler - VCR w/ SSE2 Intrinsics)

Performance Results (n = 100,000)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Scalar</th>
<th>SVML</th>
<th>Scalar CR</th>
<th>VCR4.4</th>
<th>VCR8.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spline</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bin15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sinh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Best
Performance Results with SSE
(Xeon 5160 3GHz – Intel Compiler – VCR with SSE2 Intrinsics)

Single prec. benchmark:
\[ x[i] = \sin(i \cdot h) \text{ for } i=0,\ldots,n-1 \]
32-bit float performance

Double prec. benchmark:
\[ x[i] = \sin(i \cdot h) \text{ for } i=0,\ldots,n-1 \]
64-bit double performance
Modulo Scheduling Results
(Sun UltraSPARC IIIi 1.2 GHz – Sun Studio 12 compiler)

Performance Results (n = 1,000)

- Scalar
- Scalar CR
- VCR4.1

Best

Reg spill
Modulo Scheduling Results
(Sun UltraSPARC IIIi 1.2 GHz – Sun Studio 12 compiler)

Performance Results (n = 100,000)

- Scalar
- Scalar CR
- VCR4.1

Best

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly3</td>
<td></td>
</tr>
<tr>
<td>Spline</td>
<td></td>
</tr>
<tr>
<td>Bin15</td>
<td></td>
</tr>
<tr>
<td>Sine</td>
<td></td>
</tr>
<tr>
<td>Sinh</td>
<td></td>
</tr>
<tr>
<td>Exp</td>
<td></td>
</tr>
</tbody>
</table>
Poly3 VCR Modulo Sched. Benefits
(Sun UltraSPARC IIIi 1.2 GHz – Sun Studio 12 compiler)

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Scalar</th>
<th>Scalar CR</th>
<th>VCR4.1</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unroll factor</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Steady-state cycle count</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>(= 12 / 4 points)</td>
</tr>
<tr>
<td>FP ops / grid point</td>
<td>3 FMA + 2 adds</td>
<td>3 adds</td>
<td>3 adds</td>
<td></td>
</tr>
</tbody>
</table>
Auto-Tuning

- The performance of VCRx.y is superior, but the best possible performance is not predictable
- Note that the vector width y is fixed
  - $y = 4$ for single prec. SSE ops
  - $y = 2$ for double prec. SSE ops
  - $y = 1$ for scalar ops
- An auto-tuning strategy is needed, because …
  - Low x value wastes potential parallelism
  - High x value causes register spill
  - Thus, we run codes with $x = k \times y$ ($k$ is integer) to determine optimal $x$
  - We run code versions to determine largest block size $b$ that bounds the floating point rounding error
  - Tuning for memory load/store effects
Conclusions

- Vectorization results:
  - VCR code is overall faster than best vectorizers

- Modulo scheduling results:
  - VCR code has fewer operations and fewer data dependences, thus higher ILP and lower steady-state cycle counts (smaller kernels)

- An auto-tuning strategy is proposed
  - Generates and tunes VCR codes to search for best performance and highest accuracy

- Future work
  - Loop strength reduction optimization at the level of vector instructions possibly in combination with scalars to reduce register pressure
Graduate Students and Support

- Current group of 4 PhD students
- **Before** students can join our group …
  - They have to show excellence in grades and productivity
- **During** their research students are …
  - Supported as RAs (full or part RA/TA)
  - Attend conferences and present papers
  - Present research at group meetings
  - Take a 3 month paid internship at Intel, IBM, Avaya, …
  - … and are expected to (help) write research papers!
- **After** graduation …
  - Former MS students (20) hired at companies, such as Dell and Microsoft
  - Former PhD student hired as assistant professor
Thanks