Compiler technology:
Are we done yet?

David Padua
University of Illinois at Urbana-Champaign
Two lines of this talk

- Much has been accomplished
- The task ahead is daunting
In the beginning there was FORTRAN

• The compiler (ca. 1956) was a momentous accomplishment
• I learned about it at the turn of the century

• Accomplishment by John Backus and his small team even more impressive given how little was known then.
Subscript evaluation

• The address of array element $A(I, J, c3*K+6)$ is $base_A+I-1+(J-1)*DI+(c3*K+6-1)*DI*DJ$

• There was no strength reduction, induction variable analysis, nor data flow analysis.

• They used a pattern matching so that every time “$K$ is increased by $n$ (under the control of a $DO$), the index quantity is increased by $c3$ $DI$ $DJ$ $n$, giving the correct value” (Backus, Western Joint Computer Conference, 1957)
Induction variable analysis

• “... it was not practical to track down and identify linear changes in subscripts resulting from assignment statements. Thus, the sole criterion ...for efficient handling of array references was to be that the subscripts involved were being controlled by DO statements”
Operator precedence in Fortran I

• A big deal.

• “The lack of operator priority (often called precedence ...) in the IT language was the most frequent single cause of errors by the users of that compiler” Donald Knuth.

• The Fortran I algorithm:
  • Replace + and – with ))+(( and ))-(( respectively
  • Replace * and / with )*( and )/(, respectively
  • Add (( at the beginning of each expression and after each left parenthesis in the original expression.
  • Add )) at the end and before each right parenthesis

• “The resulting formula is properly parenthesized, believe it or not” D. Knuth
Register allocation

• Extremely complex
• Used to manage the three index registers of the 704
• “… much of it was carried along into Fortran II and still in use in the 705/9/90. In many programs it still contributes to the production of better code than can be achieved on the new Fortran IV compiler.” Saul Rosen
A difficult chore

• “… didn’t really work when it was delivered.”
• “At first people thought it would never be done.”
• “Then when it was in field test, with many bugs..., many thought it would never work. “
• “Fortran is now almost taken for granted, as if it were built into the computer hardware.”

Saul Rosen, 1967
The challenge then

• “It was our belief that if FORTRAN, during its first months, were to translate any reasonable “scientific” source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger.”

  John Backus

• How close they come to this goal? Hard to tell
• But we know they succeeded. Their work is one of the most influential in the history of computing
Back to the future

• A main goal of high level languages is to provide machine independence.

• This means that the same program should be able to run across multiple machines (different ISAs) without having to pay a ridiculous penalty. PORTABILITY CHALLENGE
Back to the future

• Additional challenges
  • Get as good performance as any expert programmer could starting from a simple program and targeting a class of machines.
  • Ditto for all classes of machines.
  • ABSOLUTE PERFORMANCE CHALLENGE
A great challenge

• Portability was, I believe, easier back in the 1950s.
• Machines today have a more complex memory, the CPU is incredibly more complex, and there is more variability in their organization.

• But to face this challenge we have powerful analysis and transformation techniques developed in the last four decades.

• How well are we doing in meeting these challenges?
• Let us consider exploitation of parallelism
1970s

• Program parallel machines using conventional languages.
  • Perhaps with some annotations (High Performance Fortran)
• The compiler identifies the parallelism and transforms the program to fit the target machine.
• Legacy codes could be ported to newer machines.
• No need for new languages
• Work started at Illinois during Illiac IV project
The technology

- Dependence analysis is the foundation.
- It computes relations between statement instances
- These relations can be used to transform programs
  - for locality (tiling),
  - parallelism (vectorization, parallelization),
  - communication (message aggregation),
  - reliability (automatic checkpoints),
  - power ...
The technology
Example of use of dependence

Consider the loop

```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j]=a[i][j-1]+a[i-1][j];
    }
}
```
The technology

Example of use of dependence

Compute dependences (part 1)

```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j] = a[i][j-1] + a[i-1][j];
    }
}
```

<table>
<thead>
<tr>
<th></th>
<th>i=1</th>
<th>i=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a[1][1] = a[1][0] + a[0][1]</td>
<td>a[2][1] = a[2][0] + a[1][1]</td>
</tr>
<tr>
<td>2</td>
<td>a[1][2] = a[1][1] + a[0][2]</td>
<td>a[2][2] = a[2][1] + a[1][2]</td>
</tr>
<tr>
<td>3</td>
<td>a[1][3] = a[1][2] + a[0][3]</td>
<td>a[2][3] = a[2][2] + a[1][3]</td>
</tr>
</tbody>
</table>
The technology
Example of use of dependence

Compute dependences (part 2)

```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j] = a[i][j-1] + a[i-1][j];
    }
}
```

for $i=1$ to $i=2$

- $j=1$: $a[1][1] = a[1][0] + a[0][1]$  
  $a[2][1] = a[2][0] + a[1][1]$

- $j=2$: $a[1][2] = a[1][1] + a[0][2]$  
  $a[2][2] = a[2][1] + a[1][2]$

- $j=3$: $a[1][3] = a[1][2] + a[0][3]$  
  $a[2][3] = a[2][2] + a[1][3]$

- $j=4$: $a[1][4] = a[1][3] + a[0][4]$  
The technology
Example of use of dependence

for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j] = a[i][j-1] + a[i-1][j];
    }
}

or
The technology Example of use of dependence.

• Find parallelism

```c
for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
        a[i][j]=a[i][j-1]+a[i-1][j];
    }
}
```
The technology
Example of use of dependence

• Transform the code

```c
for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
    a[i][j]=a[i][j-1]+a[i-1][j];
  }
}
```

for k=4; k<2*n; k++) forall (i=max(2,k-n):min(n,k-2)) a[i][k-i]=...
Successes

• The technology has become part of practically all compilers.
• Widely used for targeting vector instructions.
  • Blue waters programming model: MPI+vector extensions
• Much research in this area during the last four decades and still an active area of research.
How well does it work?

- Depends on three factors:
  1. The accuracy of the dependence analysis
  2. The set of transformations
  3. The sequence in which transformations are applied
How well does it work? Vectorization

- Vectorization important:
  - Vector extensions are of great importance. Easy parallelism. Will continue to evolve
    - SSE
    - AltiVec
  - Longest experience
  - Most widely used (parallelization less popular)
  - Easier than parallelization/localization
  - Convenient strategy to access vector extensions in a portable manner
    - Alternatives: assembly language or machine-specific macros
How well does it work?

Vectorizers - 2005

How well does it work?

Vectorizers - 2010

<table>
<thead>
<tr>
<th>Loops</th>
<th>Compiler</th>
<th>XLC</th>
<th>ICC</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td></td>
<td>159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vectorized</td>
<td></td>
<td>74</td>
<td>75</td>
<td>32</td>
</tr>
<tr>
<td>Not vectorized</td>
<td></td>
<td>85</td>
<td>84</td>
<td>127</td>
</tr>
<tr>
<td>Average Speed Up</td>
<td></td>
<td>1.73</td>
<td>1.85</td>
<td>1.30</td>
</tr>
</tbody>
</table>

How well does it work?
Parallelization - 1998
Results at the end of the project
Other transformations
Locality

Matrix-matrix multiplication on Intel Xeon

Is Search Really Necessary to Generate High-Performance BLAS?
And the problem is not confined to parallelism
The spiral experience

J. Xiong, J. Johnson, and D Padua. SPL: A Language and Compiler for DSP Algorithms. PLDI 2001
What is left to do?

Improvements needed

• Obvious conclusion from these observations is that there is much room for improvement.
• Compilers cannot be trusted to deliver the best performance possible.
• Much user intervention is needed today to deliver good performance.
• The result is typically not portable.
• And compilers provide low quality feedback.
What is left to do?

Attitudes

- Some claim that the compiling problem is solved
- Others that some of the challenges can never be met
- Others that automatic optimization is not an important problem.

- Instead, I’ll say that we face a great challenge
- And this challenge will only grow as machine parallelism and heterogeneity increases
- Are we up to the task?
What is left to do?
Our community

• Is small
• Fragmented.
• Few groups doing actual compiler research
• No massive, systematic, gradual effort in any direction
• And the emphasis of many of our conferences on great innovation does not help
What is left to do?
Directions 1/4

• Need an evaluation methodology

• How can we have a science otherwise?

• We should be able to objectively measure progress and setbacks, in overall compiler technology

• We should try to compare to an upper bound, to the optimum. Not just show that new techniques are a little better.
What is left to do?
Directions 2/4

• **Need better optimization technology**
  • We understand analysis and calculus of program transformation better that how to organize the transformations
  • Need to understand better the process of optimization and develop stable methods.
  • Should not be that
    • increasing optimization level decreases performance or that
    • performance can be improved significantly by (blindly) flipping compiler switches.
What is left to do?
Directions 3/4

• **Need better understanding of relation between notation and program optimization**

• Language/compiler co-design

• What is the right level of abstraction for the compiler to manipulate for portable optimization?
High level notation
fft

- In the **Spiral project** we looked into divide an conquer for fft.
- Work is based on the recurrence:
  \[
  F_{rs} = (F_r \otimes I_s)T_s^{rs}(I_r \otimes F_s)L_r^{rs}
  \]
- This creates a space of possibilities
Rules = Breakdown Strategies

\[ DCT_{2}^{(H)} \rightarrow \text{diag} \left( 1, 1 / \sqrt{2} \right) \cdot F_2 \]

\[ DCT_{n}^{(H)} \rightarrow P \cdot (DCT_{n/2}^{(H)} \oplus DCT_{n/2}^{(W)}) \cdot (I_{n/2} \otimes F_2) \]

\[ DCT_{n}^{(W)} \rightarrow S \cdot DCT_{n}^{(H)} \cdot D \]

\[ DCT_{n}^{(I)} \rightarrow M_1 \cdots M_r \]

\[ DFT_n \rightarrow \text{CosDFT}_n + j \cdot \text{SinDFT}_n \]

\[ DFT_n \rightarrow B \cdot (DCT_{n/2}^{(I)} \oplus DST_{n/2}^{(I)}) \cdot C \]

\[ DFT_{nm} \rightarrow (DFT_n \otimes I_m) \cdot D \cdot (I_n \otimes DFT_m) \cdot P \]

\[ \text{CosDFT}_n \rightarrow \cdots \text{CosDFT}_{n/2} \cdots DCT_{n/4}^{(H)} \cdots \]

\[ \text{SinDFT}_n \rightarrow \cdots \text{SinDFT}_{n/2} \cdots DCT_{n/4}^{(H)} \cdots \]

\[ \text{WHT}_{2^n} \rightarrow \prod_{i=1}^{n} (I_{2^{n-1}+\ldots+1} \otimes \text{WHT}_{2^n} \otimes I_{2^{n-1}+\ldots+1}) \]

\[ \text{MDCT}_{n \times 2^n} \rightarrow S \cdot DCT_n^{(W)} \cdot P \]
High level notation fft

Complex DFT (Intel Core i7, 2.66 GHz, 4 cores, double precision)
Performance [Gflop/s] vs. input size

Spiral generated

Intel IPP 6.0

Spiral generated (1 thread)

FFT W 3.2


High level notation

A similar divide and conquer approach can be followed for sorting.

Work is based on the fact that:
- Data can be partitioned by pivot (quicksort)
- Or by digit (radix sort)
- Or by location (merge sort)

And that different sorting mechanisms can be used at different levels.
High level notation sorting

Partition by pivot

Partition by pivot

Partition by digit

...
High level notation sorting

Classifier Sort

IBM ESSL

C++ STL

Performance (keys per cycle)

Standard Deviation

C++ STL

XSort

IBM ESSL

Brian A. Garber, Daniel Hoeflinger, Xiaoming Li, María Jesús Garzarán, David A. Padua: Automatic generation of a parallel sorting algorithm. IPDPS 2008: 1-5
High level notation
Hydra system

Alexandre Duchateau, David A. Padua, Denis Barthou: Hydra: Automatic algorithm exploration from linear algebra equations. CGO 2013: 1-10
- All operands
- (Type inference)

- Status
  - Known, Unknown
- Shape
  - Triangular, diagonal
- Type
  - Matrix, (vector, scalar)
- Modifiers (transpose)
- (Sizes)
- (Sparsity)
\[
\begin{align*}
T(0,0) &= A(0,0) \times X(0,0) + A(0,1) \times X(1,0) \\
T(0,1) &= A(0,0) \times X(0,1) + A(0,1) \times X(1,1) \\
T(1,0) &= A(1,0) \times X(0,0) + A(1,1) \times X(1,0) \\
T(1,1) &= A(1,0) \times X(0,1) + A(1,1) \times X(1,1)
\end{align*}
\]
Equation Dependence Graph
Derivation
Set Equations Dependence Graph
Identification
Dependence Graph
Termination
Final Graph

$L \times X \times U - X = C$

$\emptyset$

$L11 \times X11 \times U11 - X11 = T$

1: $L_{00,00} \times X_{00,00} \times U_{00,00} - X_{00,00} = C_{00}$
2: $L_{10,00} \times X_{00,00} + L_{11,00} \times X_{10,00} - X_{10,00} = C_{10}$
3: $L_{00,00} \times X_{00,01} + L_{00,01} \times X_{01,00} - X_{01,00} = C_{01}$
4: $L_{10,00} \times X_{00,01} + ... + L_{11,11} \times X_{11,11} - X_{11,11} = C_{11}$

1: $L_{00,00} \times X_{00,00} \times U_{00,00} - X_{00,00} = T_{00}$
2: $L_{10,00} \times X_{00,00} \times U_{00,00} + L_{11,00} \times X_{10,00} - X_{10,00} = T_{10}$
3: $L_{00,00} \times X_{00,01} \times U_{00,01} + L_{00,01} \times X_{01,01} - X_{01,01} = T_{01}$
4: $L_{10,00} \times X_{00,01} \times U_{00,01} + ... + L_{11,11} \times X_{11,11} \times U_{11,11} - X_{11,11} = T_{11}$
What is left to do?
Directions 4/4

- Need better tools for rapid development of effective compilers
- Great opportunities in machine design / special purpose hardware
Conclusion

• Much left to do, but it CAN be done
Inventor, futurist predicts dawn of total artificial intelligence

Brooklyn, New York (VBS.TV) -- ...Computers will be able to improve their own source codes ... in ways we ... could never conceive.
We should not underestimate computers